Lecture Notes in Computer Science 2030

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

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

Furio Honsell Marino Miculan (Eds.)

Foundations of Software Science and Computation Structures

4th International Conference, FOSSACS 2001
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2001
Genova, Italy, April 2-6, 2001
Proceedings

Springer

Volume Editors

Furio Honsell
Marino Miculan
Università di Udine, Dipartimento di Matematica e Informatica
Via delle Scienze 206,33100 Udine, Italy
E-mail: {honsell/miculan}@dimi.uniud.it

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Foundations of software science and computation structures : 4th international conference ; proceedings / FOSSACS 2001, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2 - 6, 2001. Furio Honsell ; Marino Miculan (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2030)
ISBN 3-540-41864-4

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

ISSN 0302-9743
ISBN 3-540-41864-4 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 PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10782468 06/3142 5 4 3 2 1 0

Foreword

ETAPS 2001 was the fourth 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), ten satellite workshops (CMCS, ETI Day, JOSES, LDTA, MMAABS, PFM, RelMiS, UNIGRA, WADT, WTUML), seven invited lectures, a debate, and ten tutorials.

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 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 2001 was hosted by the Dipartimento di Informatica e Scienze dell'Informazione (DISI) of the Università di Genova and was organized by the following team:

Egidio Astesiano (General Chair)
Eugenio Moggi (Organization Chair)
Maura Cerioli (Satellite Events Chair)
Gianna Reggio (Publicity Chair)
Davide Ancona
Giorgio Delzanno
Maurizio Martelli

with the assistance of Convention Bureau Genova. Tutorials were organized by Bernhard Rumpe (TU München). Overall planning for ETAPS conferences is the responsibility of the ETAPS Steering Committee, whose current membership is:

Egidio Astesiano (Genova), Ed Brinksma (Enschede), Pierpaolo Degano (Pisa), Hartmut Ehrig (Berlin), José Fiadeiro (Lisbon), Marie-Claude Gaudel (Paris), Susanne Graf (Grenoble), Furio Honsell (Udine), Nigel

Horspool (Victoria), Heinrich Hußmann (Dresden), Paul Klint (Amsterdam), Daniel Le M'etayer (Rennes), Tom Maibaum (London), Tiziana Margaria (Dortmund), Ugo Montanari (Pisa), Mogens Nielsen (Aarhus), Hanne Riis Nielson (Aarhus), Fernando Orejas (Barcelona), Andreas Podelski (Saarbrücken), David Sands (Göteborg), Don Sannella (Edinburgh), Perdita Stevens (Edinburgh), Jerzy Tiuryn (Warsaw), David Watt (Glasgow), Herbert Weber (Berlin), Reinhard Wilhelm (Saarbrücken)

ETAPS 2001 was organized in cooperation with

the Association for Computing Machinery
the European Association for Programming Languages and Systems
the European Association of Software Science and Technology
the European Association for Theoretical Computer Science

and received generous sponsorship from:

ELSAG
Fondazione Cassa di Risparmio di Genova e Imperia
INDAM – Gruppo Nazionale per l'Informatica Matematica (GNIM)
Marconi
Microsoft Research
Telecom Italia
TXT e-solutions
Università di Genova

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 finally Springer-Verlag for agreeing to publish the ETAPS proceedings.

January 2001

Donald Sannella
ETAPS Steering Committee chairman

Preface

The present volume contains the proceedings of the international conference *Foundations of Software Science and Computation Structures* (FOSSACS 2001), held in Genova, Italy, April 2-4, 2001. FOSSACS is a constituent event of the *Joint European Conferences on Theory and Practice of Software* (ETAPS). This was the fourth meeting of ETAPS. The previous three meetings took place in Lisbon (1998), Amsterdam (1999), and Berlin (2000).

FOSSACS seeks papers significant to software sciences, which offer progress in foundational research. Central objects of interest are the algebraic, categorical, logical, and geometric theories, models, and methods which support the specification, synthesis, verification, analysis, and transformation of sequential, concurrent, distributed, and mobile programs and software systems. This volume deals with a wide spectrum of topics within this scope, many of which are motivated by recent trends and problems in the practice of software and information technology.

There are 27 papers in these proceedings. The first one accompanies the invited lecture *Adequacy for algebraic effects*, delivered by Gordon Plotkin (Edinburgh University) at FOSSACS 2001. The last one concerns a tool presentation. The other 25 are contributed papers, selected out of a total of 63 submissions. Each paper was thoroughly evaluated by the PC members. The selection procedure ended with a virtual meeting of the Program Committee which lasted over a week. I would like to sincerely thank all of the PC members for the excellent work they did during this difficult process. The tool presentation was selected by a special committee appointed by Donald Sannella the ETAPS Chairman.

Very special thanks go to the co-editor of these proceedings Marino Miculan, whose assistance has been invaluable in organizing the conference web-page, the electronic submission, reviewing, and notification processes, as well as in preparing the files for the publisher. Thanks to Richard van de Stadt for producing and allowing us to utilize the *CyberChair* software. And thanks also to all the members of the Organizing Committee, chaired by Egidio Astesiano, Eugenio Moggi, and Gianna Reggio. Finally I would like to thank the Steering Committee of ETAPS, and especially its Chairman Donald Sannella, for the precious advice and excellent coordination in all the various activities leading up to this FOSSACS conference.

January 2001

Furio Honsell
FOSSACS 2001 Program Committee Chair

FOSSACS 2001 Program Committee

Luca Cardelli (Cambridge, UK)
Philippe Darondeau (Rennes, France)
Mariangiola Dezani (Torino, Italy)
Abbas Edalat (London, UK)
Javier Esparza (München, Germany)
Bart Jacobs (Nijmegen, The Netherlands)
Matthew Hennessy (Sussex, UK)
Martin Hofmann (Edinburgh, UK)
Furio Honsell (Udine, Italy, chair)
Pierre Lescanne (Lyon, France)
Ugo Montanari (Pisa, Italy)
Mogens Nielsen (Århus, Denmark)
Mitsu Okada (Tokyo, Japan)
Carolyn Talcott (Stanford, USA)
Pawel Urzyczyn (Warsaw, Poland)
Igor Walukiewicz (Warsaw, Poland)

List of Referees

Fabio Alessi
Roberto Amadio
Andre Arnold
Paolo Baldan
Stefano Berardi
Marco Bernardo
Richard Blute
Chiara Bodei
Mikolaj Bojanczyk
Marcello Bonsangue
Ahmed Bouajjani
Jérémie Bourdon
Julian Bradfield
Roberto Bruni
Nadia Busi
Benoit Caillaud
Olivier Carton
Paul Caspi
Ilaria Castellani
Jan Cederquist
Jacek Chrzaszcz
Paolo Ciancarini
David Clark
Hubert Comon
Adriana Compagnoni
Mario Coppo
Andrea Corradini
Rocco De Nicola
Philippe Devienne
Fer-Jan de Vries
Pietro Di Gianantonio
Krzysztof Diks
Catalin Dima
Dan Dougherty
Gilles Dowek
Hartmut Ehrig
Thomas Engel
Alessandro Fantechi
Maribel Fernandez
Paolo Ferragina
Andrzej Filinski
Marcelo Fiore
Cedric Fournet
Massimo Franceschet
Carsten Führmann
Maurizio Gabbrielli
Philippa Gardner
Paul Gastin
Stephane Gaubert
Paola Giannini
Juergen Giesl
Stefania Gnesi
Andrew D. Gordon
Bernd Grobauer
Josef Gruska
Stefano Guerrini
Jesper Gulmann Henriksen
Vineet Gupta
Masahito Hasegawa
Reiko Heckel
Reinhold Heckmann
Keijo Heljanko
Hugo Herbelin
Holger Hermanns

Daniel Hirschkoff
Ian Hodkinson
Markus Holzer
Thomas Hune
Graham Hutton
Katsushi Inoue
Radha Jagadeesan
David Janin
Marcin Jurdzinski
Richard Kennaway
Delia Kesner
Astrid Kiehn
Dilsun Kirli
Teodor Knapik
Barbara Koenig
Antonin Kucera
Ralf Kuesters
Alexander Kurz
Dietrich Kuske
Marta Kwiatkowska
Slawomir Lasota
Marina Lenisa
Francesca Levi
Luigi Liquori
Gerald Luettgen
Angelika Mader
Pasquale Malacaria
Narciso Marti-Oliet
Michael Marz
Richard Mayr
Guy McCusker
Massimo Merro
Marino Miculan
Angelo Montanari
César A. Muñoz H.
Anca Muscholl
Margherita Napoli
Mirabelle Nebut
Mitsuhiro Okada
Joachim Parrow
Christine Paulin-Mohring
Larry C. Paulson
Wieslaw Pawlowski
Adriano Peron
Carla Piazza
Sophie Pinchinat
Erik Poll
John Power
K. V. S. Prasad
Rosario Pugliese
R. Ramanujam
Jean-Xavier Rampon
Julian Rathke
Bernhard Reus
Eike Ritter
Simona Ronchi Della Rocca
Francesca Rossi
Luca Roversi
Ivano Salvo
Don Sannella
Luigi Santocanale
Vladimiro Sassone
Ivan Scagnetto
Aleksy Schubert
Francesca Scozzari
Peter Selinger
Paula Severi
Jiří Srba
Alin Stefanescu
Perdita Stevens
Makoto Takeyama
Francis Tang
Hendrik Tews
Wolfgang Thomas
Simone Tini
Yoshihito Toyama
Daniele Turi
Tomasz Urbanski
Frank Valencia
Andrea Valente
Daniele Varacca
Betti Venneri
Marisa Venturini Zilli
Rene Vestergaard
Alicia Villanueva
Daria Walukiewicz
Freek Wiedijk
Nobuko Yoshida
Hans Zantema
Pascal Zimmer
Jan Zwanenburg

Table of Contents

Invited Paper

Contributed Papers

Tool Presentation

Adequacy for Algebraic Effects*

Gordon Plotkin and John Power

Division of Informatics, University of Edinburgh, King's Buildings,
Edinburgh EH9 3JZ, Scotland

Abstract. Moggi proposed a monadic account of computational effects. He also presented the computational λ-calculus, λ_c, a core call-by-value functional programming language for effects; the effects are obtained by adding appropriate operations. The question arises as to whether one can give a corresponding treatment of operational semantics. We do this in the case of algebraic effects where the operations are given by a single-sorted algebraic signature, and their semantics is supported by the monad, in a certain sense. We consider call-by-value PCF with—and without—recursion, an extension of λ_c with arithmetic. We prove general adequacy theorems, and illustrate these with two examples: nondeterminism and probabilistic nondeterminism.

1 Introduction

Moggi introduced the idea of a general account of computational effects, proposing encapsulating them via monads $T : \mathbf{C} \to \mathbf{C}$; the main idea is that $T(x)$ is the type of computations of elements of x. He also presented the computational λ-calculus λ_c as a core call-by-value functional programming language for effects [21]. The effects themselves are obtained by adding appropriate operations, specified by a signature Σ. Moggi introduced the consideration of these operations in the context of his metalanguage ML(Σ) whose purpose is to give the semantics of programming languages [22,23], but which is not itself thought of as a programming language.

In our view any complete account of computation should incorporate a treatment of operational semantics; this has been lacking for the monadic view. To progress, one has to deal with the operations as they are the source of the effects. In this paper we give such a treatment in the case of *algebraic* effects where the operations are given by a single-sorted algebraic signature Σ; semantically such an n-ary operation f is taken to denote a family of morphisms

$$f_x : T(x)^n \longrightarrow T(x)$$

parametrically natural with respect to morphisms in the Kleisli category $\mathbf{C}_T$; T is then said to *support* the family f_x. (In [22] only naturality with respect to morphisms in $\mathbf{C}$ is considered; we use the stronger assumption.) Note that

* This work has been done with the support of EPSRC grant GR/M56333: The Structure of Programming Languages: Syntax and Semantics.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 1–24, 2001.

there is no assumption that the monads at hand are commutative. For $\mathbf{C} = \mathbf{Set}$, examples are the finite powerset monad and binary choice operations; the monad for probabilistic nondeterminism and probabilistic choice operations; and the monad for printing and the printing operations (these are noncommutative). As will be discussed below, there are natural analogues of these examples in the domain-theoretic context where $\mathbf{C} = \mathbf{Dcppo}$, the category of dcppos and continuous functions. Generally, suppose we are given a category $\mathbf{C}$ with finite products and a finitary equational theory over a signature Σ. Assuming free Σ-algebras exist, let T be the associated monad. Then every operation symbol yields such a family, in an evident way. In the case $\mathbf{C} = \mathbf{Set}$ a converse holds, that every parametrically natural family arises as a composition of such families, as follows, e.g., from a remark in Section 3 below.

On the other hand, for example, the exceptions monad does not support its exception handling operation: only the weaker naturality holds there. This monad is a free algebra functor for an equational theory, viz the one that has a constant for each exception and no equations; however the exception handling operation is not definable: only the exception raising operations are. Other standard monads present further difficulties. So while our account of operational semantics is quite general, it certainly does not cover all cases; it remains to be seen if it can be further extended.

To give an account of operational semantics we need a programming language based on the computational λ-calculus with some basic datatypes and functions in order to permit computation. We take as the test of our account whether a useful general adequacy theorem can be proved. So we consider a call-by-value PCF with algebraic effects, an extension of the computational λ-calculus with operations, arithmetic and recursion (see, e.g., [34,32] for versions of call-by-value PCF). We begin by treating the sublanguage without recursion. Section 2 presents both a *small step* and a *(collecting) big step* operational semantics; there is also an associated *evaluation function*. Section 3 considers denotational semantics and gives an adequacy theorem. The semantics is given axiomatically in terms of a suitable class of categorical structures appropriately extending the usual monadic view of the computational λ-calculus. This could as well have been based on closed Freyd categories [30], and [2] is a treatment of nondeterminism along such lines. Section 4 considers two examples: nondeterminism and probabilistic nondeterminism.

We consider the full language with recursion in Section 5. Small step semantics is straightforward, but big step semantics presents some difficulties as evaluation naturally yields infinite values since programs may not terminate. We also consider an intermediate *medium step* semantics which is big step as regards effect-free computation and small step as regards effects. For the semantics we assume a suitable order-enrichment [16] in order to give a least fixed-point treatment of recursion. This then yields an adequacy theorem, which is the main result of the paper. One wonders if a more general treatment of recursion is possible within synthetic or axiomatic domain theory, cf. [32]. In Section 6 we revisit the examples, but with recursion now present. Finally, in Section 7 we present

some ideas for further progress. While all definitions are given in Section 5, some previous knowledge of domain theory is really assumed; for Section 6 this is essential as both ordinary and probabilistic powerdomains are considered.

Our treatment of operational semantics might well be seen as rather formal and does not immediately specialise to the usual accounts for the examples at hand. In a way this has to be so: it is hard to imagine a theory which yields the natural operational semantics for any possible computational effect. On the other hand we can prove adequacy (with and without recursion) for our formal approach and then easily recover adequacy results for the standard operational semantics of the various examples.

2 PCF without Recursion: Operational Semantics

We begin with the syntax of our language. Its types are given by

$$\sigma \ ::= \ \iota \mid o \mid 1 \mid \sigma \times \sigma \mid \sigma \to \sigma$$

where ι is the type of the natural numbers and o is that of the booleans. For the terms, we assume we are given a single-sorted algebraic signature Σ and a countably infinite set of variables x; the signature provides a set of operation symbols f and associates an arity $\mathrm{ar}_f \geq 0$ to each. The terms are given by

$$\begin{array}{rcl} M & ::= & \underline{0} \mid \mathrm{succ}(M) \mid \mathrm{zero}(M) \mid \mathrm{pred}(M) \mid \\ & & \sharp\!\!\sharp \mid \flat\!\!\flat \mid \textbf{if } M \textbf{ then } M \textbf{ else } M \mid \\ & & * \mid <M, M> \mid \pi_1(M) \mid \pi_2(M) \mid \\ & & x \mid \lambda x : \sigma . M \mid MM \mid \\ & & f(M_1, \ldots, M_n) \end{array}$$

where, in the last clause, f is an operation symbol of arity n. Substitution $M[N/x]$ is defined as usual, and we identify terms up to α-equivalence.

As regards comparison with λ_c, we have fixed a particular set of base types, viz ι and o, and function symbols, viz $\underline{0}$, succ, zero, pred, $\sharp\!\!\sharp$ and $\flat\!\!\flat$. We also have a conditional construct **if** and the operation symbols f from Σ. We do not have a type constructor $T(\sigma)$ as it may be defined to be $1 \to \sigma$. Nor do we have a **let** constructor or constructions $[M]$ or $\mu(M)$ as we consider **let** $x : \sigma = M$ **in** N as syntactic sugar for $(\lambda x : \sigma . N)M$ (preferring explicit declaration of types in binding contexts), and $[M]$ as syntactic sugar for $\lambda x : 1 . M$, where x is fresh, and $\mu(M)$ as syntactic sugar for $M*$.

The typing rules specify the well formed sequents

$$\Gamma \vdash M : \sigma$$

The rules will all be evident, except perhaps, that for the last construct, which is

$$\frac{\Gamma \vdash M_i : \sigma \ (\text{for } i = 1, n)}{\Gamma \vdash f(M_1, \ldots, M_n) : \sigma}$$

(where $\mathrm{ar}_f = n$). Thus effects can occur at any type. We write $M : \sigma$ for $\vdash M : \sigma$ and say then that the (closed) term M is *well-typed*.

We give both a small step and a collecting big step operational semantics. These are given without using any further information on the operations; it is in that sense that they are purely formal. One might instead have introduced equations corresponding to the intended effects, e.g., semilattice equations for finite nondeterminism, as in [12] and then worked with operational semantics up to provable equality.

The small step semantics proceeds by means of two kinds of transitions between closed terms, an unlabelled one

$$M \to M'$$

and a labelled one

$$M \xrightarrow{f_i} M'$$

where f is an operation symbol of arity $n > 0$ and $1 \leq i \leq n$, and there is also a predicate on closed terms

$$M \downarrow_a$$

for operations a of arity 0 i.e., *constants*. The unlabelled transition relation corresponds to effect-free computation; the labelled one corresponds to an effect, which is mirrored syntactically by "entering" the ith argument place of an operation symbol f of strictly positive arity. Constants yield a kind of exceptional termination.

Transitions terminate at *values*, given by

$$V \ ::= \ \underline{0} \mid \mathrm{succ}(V) \mid \mathit{tt} \mid \mathit{ff} \mid * \mid <V, V> \mid \lambda x : \sigma.M$$

where we restrict $\lambda x : \sigma.M$ to be closed. We write $\underline{n}$ for $\mathrm{succ}^n(\underline{0})$.

It is convenient to use Felleisen's idea [7] of specifying transitions via evaluation contexts and redexes. The evaluation contexts here are given by:

$$E \ ::= \ [\,] \mid \mathrm{succ}(E) \mid \mathrm{zero}(E) \mid \mathrm{pred}(E) \mid \textbf{if } E \textbf{ then } M \textbf{ else } M \mid \\ <E, M> \mid <V, E> \mid \pi_1(E) \mid \pi_2(E) \mid EM \mid VE$$

where the terms appearing are restricted to be closed.

Redexes and their transitions are given by:

$$\mathrm{zero}(\underline{0}) \to \mathit{tt} \quad \mathrm{zero}(\underline{n+1}) \to \mathit{ff}$$

$$\mathrm{pred}(\underline{0}) \to \underline{0} \quad \mathrm{pred}(\underline{n+1}) \to \underline{n}$$

$$\textbf{if } \mathit{tt} \textbf{ then } M \textbf{ else } N \to M \quad \textbf{if } \mathit{ff} \textbf{ then } M \textbf{ else } N \to N$$

$$\pi_1(<V, V'>) \to V \quad \pi_2(<V, V'>) \to V'$$

$$(\lambda x : \sigma.M)V \to M[V/x]$$

$$f(M_1, \ldots, M_n) \xrightarrow{f_i} M_i$$

where, in the last clause, $n = \mathrm{ar}_f > 0$, and $1 \leq i \leq n$, and where we restrict the redexes (the left hand sides) to be closed. Noting that $\to$ is deterministic here, we will find it useful to write R^+ for the unique N, if any, such that $R \to N$. For any closed well-typed term one of three mutually exclusive possibilities holds: it is a value; it has the form $E[R]$ for a unique E and R; or it has the form $E[a()]$ for a unique E and a.

We now define the transition relations by the two rules:

$$\frac{R \to N}{E[R] \to E[N]}$$

$$\frac{R \xrightarrow{f_i} N}{E[R] \xrightarrow{f_i} E[N]}$$

and the predicate by the rule

$$E[a()] \downarrow_a$$

For any closed well-typed term M which is not a value, exactly one of three mutually exclusive possibilities hold:

- $M \to N$ for some N; in this case N is determined and of the same type as M.
- $M \xrightarrow{f_i} N_i$ for some f and N_i $(1 \leq i \leq \mathrm{ar}_f)$; in this case f and the N_i are determined and the latter are of the same type as M.
- $M \downarrow_a$ for some a; in this case a is determined.

The big step operational semantics

$$M \Rightarrow t$$

evaluates closed terms to *effect values*, which are the terms given by

$$t \ ::= \ V \ \mid \ f(t_1, \ldots, t_n)$$

The idea is that these terms "collect" together all the possible effects of a computation. The big step semantics is then defined by the following rules

$$V \Rightarrow V$$

$$\frac{R \to N \quad E[N] \Rightarrow t}{E[R] \Rightarrow t}$$

$$\frac{R \xrightarrow{f_i} N_i \quad E[N_i] \Rightarrow t_i \quad (i = 1, n)}{E[R] \Rightarrow f(t_1, \ldots, t_n)} \ (\text{where } n = \mathrm{ar}_f)$$

$$E[a()] \Rightarrow a()$$

Note that a closed term evaluates to at most one effect value; also if that term is well-typed, so is the effect value and it has the same type as the term. Small step and big step operational semantics can both be presented structurally. Example

$$\frac{M \to M'}{MN \to M'N} \qquad \frac{M \xrightarrow{f_i} M'}{MN \xrightarrow{f_i} M'N} \qquad \frac{M \downarrow_a}{MN \downarrow_a}$$

$$\frac{N \to N'}{VN \to VN'} \qquad \frac{N \xrightarrow{f_i} N'}{VN \xrightarrow{f_i} VN'} \qquad \frac{N \downarrow_a}{VN \downarrow_a}$$

$$(\lambda x : \sigma.M)V \to M[V/x]$$

Fig. 1. Small Step Rules for Function Application.

small step rules for function application are given in Figure 1. The other rules for small step operational semantics can easily be given in the same (somewhat tedious) style.

The rules for big step semantics are not quite so obvious. First we need *effect contexts*; these are given by

$$\epsilon \ ::= \ [\,] \ \mid \ f(\epsilon_1, \ldots, \epsilon_n)$$

Any effect term t can be written uniquely in the form $\epsilon[V_1, \ldots, V_k]$. The rules are given in Figure 2. In reading these, if a term M^+ appears in a rule, the rule only applies if M^+ exists.

To connect the two semantics, we associate an effect value $\mid M \mid$ with any closed term M which is *terminating* in the sense that there is no infinite chain of (small step) transitions from M:

$$\mid M \mid = \begin{cases} \mid N \mid & (\text{if } M \to N) \\ f(\mid N_1 \mid, \ldots, \mid N_n \mid) & (\text{if } M \xrightarrow{f_i} N_i, \text{ for } i = 1, n, \text{ where } n = \mathrm{ar}_f) \\ a() & (\text{if } M \downarrow_a) \end{cases}$$

Proposition 1. *The following are equivalent for any closed well-typed M and t*

1. *M is terminating, and $\mid M \mid = t$*
2. *$M \Rightarrow t$*

Thus we have two independent characterisations of the *evaluation function* $\mid \cdot \mid$ on closed well-typed terms. One could also give a direct recursive definition of this function, but one is then faced with interpreting the recursion and relating this to the above rule-based definitions. While the effort does not seem worthwhile here, it may be so for PCF with recursion, as will be seen.

As the reader may have gathered, there is no possibility of nontermination:

Theorem 1. *Every closed well-typed term terminates.*

Proof. This can be proved by a computability argument. We content ourselves with defining the computability predicates. At the types ι, o and 1 all values are

$$\underline{0} \Rightarrow \underline{0} \qquad \frac{M \Rightarrow \epsilon[V_1, \ldots, V_k]}{\mathrm{succ}(M) \Rightarrow \epsilon[\mathrm{succ}(V_1), \ldots, \mathrm{succ}(V_k)]}$$

$$\frac{M \Rightarrow \epsilon[V_1, \ldots, V_k]}{\mathrm{pred}(M) \Rightarrow \epsilon[\mathrm{pred}(V_1)^+, \ldots, \mathrm{pred}(V_k)^+]} \qquad \frac{M \Rightarrow \epsilon[V_1, \ldots, V_k]}{\mathrm{zero}(M) \Rightarrow \epsilon[\mathrm{zero}(V_1)^+, \ldots, \mathrm{zero}(V_k)^+]}$$

$$\underline{tt} \Rightarrow \underline{tt} \qquad \underline{ff} \Rightarrow \underline{ff}$$

$$\frac{L \Rightarrow \epsilon[V_1, \ldots, V_k] \quad (\textbf{if } V_i \textbf{ then } M \textbf{ else } N)^+ \Rightarrow t_i \text{ (for } i = 1, k)}{\textbf{if } L \textbf{ then } M \textbf{ else } N \Rightarrow \epsilon[t_1, \ldots, t_k]}$$

$$* \Rightarrow *$$

$$\frac{M \Rightarrow \epsilon_1[V_1, \ldots, V_k] \quad N \Rightarrow \epsilon_2[W_1, \ldots, W_l]}{< M, N > \Rightarrow \epsilon_1[\epsilon_2[< V_1, W_1 >, \ldots, < V_1, W_l >], \ldots, \epsilon_2[< V_k, W_1 >, \ldots, < V_k, W_l >]]}$$

$$\frac{M \Rightarrow \epsilon[V_1, \ldots, V_k]}{\pi_i(M) \Rightarrow \epsilon[\pi_i(V_1)^+, \ldots, \pi_i(V_k)^+]} \ (i = 1, 2)$$

$$\lambda x : \sigma.M \Rightarrow \lambda x : \sigma.M$$

$$\frac{M \Rightarrow \epsilon_1[V_1, \ldots, V_k] \quad N \Rightarrow \epsilon_2[W_1, \ldots, W_l] \quad (V_i W_j)^+ \Rightarrow t_{ij} \text{ (i=1,k; j= 1, l)}}{MN \Rightarrow \epsilon_1[\epsilon_2[t_{11}, \ldots, t_{1l}], \ldots, \epsilon_2[t_{k1}, \ldots, t_{kl}]]}$$

$$\frac{M_i \Rightarrow \epsilon_i[V_{i1}, \ldots, V_{ik_i}] \ (i = 1, n)}{f(M_1, \ldots, M_n) \Rightarrow f(\epsilon_1[V_{11}, \ldots, V_{1k_1}], \ldots, \epsilon_n[V_{n1}, \ldots, V_{nk_n}])}$$

Fig. 2. Big Step Operational Semantics.

computable. A value of product type is computable if both its components are. A value $\lambda x : \sigma.M$ is computable if, and only if, $M[V/x]$ is for every computable value $V : \sigma$, where we say that a closed term is computable if, and only if, every transition sequence from it terminates in a computable value.

There is a natural equational theory, including "β-equations" and commutation equations for operation symbols. This establishes judgements of the form

$$\Gamma \vdash M = N : \sigma$$

where it is assumed that $\Gamma \vdash M : \sigma$ and $\Gamma \vdash N : \sigma$. There are evident rules for equality including closure under the term-forming operations. The axioms are given in Figure 3 where they are presented as equations, or equational schemas $M = N$; these should be interpreted as judgements $\Gamma \vdash M = N : \sigma$. The commutation schema for operations is equivalent to a collection of equations for the individual language constructs. It would be better to allow open values and

contexts (in a fairly evident sense) but the more restricted version presented here suffices for our purposes. The next proposition makes it easy to verify that the denotational semantics models the operational semantics.

Proposition 2. *Suppose that M is a closed term such that $M : \sigma$. Then if $M \Rightarrow t$ it follows that $\vdash M = t : \sigma$*

$$\mathrm{zero}(\underline{0}) = tt \qquad \mathrm{zero}(\underline{n+1}) = ff$$
$$\mathrm{pred}(\underline{0}) = \underline{0} \quad \mathrm{pred}(\underline{n+1}) = \underline{n}$$
$$\textbf{if } tt \textbf{ then } M \textbf{ else } N = M \quad \textbf{if } ff \textbf{ then } M \textbf{ else } N = N$$
$$\pi_1(<V, V'>) = V \quad \pi_2(<V, V'>) = V'$$
$$(\lambda x : \sigma.M)V = M[V/x]$$
$$E[f(M_1, \ldots, M_n)] = f(E[M_1], \ldots, E[M_n])$$

Fig. 3. Equations.

3 PCF without Recursion: Adequacy

We begin by defining the categorical structures that provide models of our language, building on the sound and complete class of models for the λ_c-calculus provided by Moggi in [21]. A *model* of λ_c consists of a category $\mathbf{C}$ with finite products, together with a strong monad $< T, \eta, \mu, \mathrm{st} >$ on $\mathbf{C}$, such that T has Kleisli exponentials. The latter means that for each pair of objects x and y, the functor $\mathbf{C}(- \times x, Ty) : \mathbf{C}^{op} \longrightarrow Set$ is representable; in other words, there exists an object $x \Rightarrow y$ and a natural isomorphism

$$(\lambda_T)_z : \mathbf{C}(z \times x, Ty) \cong \mathbf{C}(z, x \Rightarrow y)$$

We write $g^\dagger : z \times T(x) \to T(y)$ for the parametrised lift of $g : z \times x \to T(y)$ to $z \times T(x)$ (and we use the same notation for the ordinary unparametrised lift where $g : x \to T(y)$ as that is essentially the subcase where z is the terminal object).

We need to extend this with structure for the operations, and for arithmetic and booleans. For the former, as already stated, we assume for each n-ary operation f a family

$$f_x : T(x)^n \longrightarrow T(x)$$

parametrically natural with respect to morphisms in the Kleisli category $\mathbf{C}_T$; this means that for every map $g : z \times x \to T(y)$, the diagram

$$\begin{array}{ccc} z \times T(x)^n & \xrightarrow{(g^\dagger \cdot (z \times \pi_i))_{i=1}^n} & T(y)^n \\ {\scriptstyle z \times f_x}\downarrow & & \downarrow{\scriptstyle f_y} \\ z \times T(x) & \xrightarrow[g^\dagger]{} & T(y) \end{array}$$

commutes (cf. [27]). Equivalently we can ask that the family be natural with respect to morphisms in the category $\mathbf{C}$ and that it respect the monad multiplication and the strength. Assuming that the n-fold coproduct of 1 with itself exists in $\mathbf{C}$, one can also show that there is a natural 1-1 correspondence between such families and global elements of $T(n)$. Finally, if 1 is a generator then parametric and ordinary naturality (with respect to Kleisli morphisms) coincide.

For arithmetic we assume $\mathbf{C}$ has a natural numbers object

$$1 \xrightarrow{0} \mathbf{N} \xrightarrow{s} \mathbf{N}$$

and for the booleans we assume that the sum $\mathbf{T} =_{\text{def}} 1 + 1$ exists in $\mathbf{C}$. We write $\text{Iter}_x(a, f)$ for the unique morphism from $\mathbf{N}$ to x corresponding to a pair of morphisms $1 \xrightarrow{a} x \xrightarrow{f} x$.

This gives us a $\mathbf{C}$ object as the denotation $[\![\sigma]\!]$ of each type σ, following [21] and taking $[\![\iota]\!] = \mathbf{N}$ and $[\![o]\!] = \mathbf{T}$. The denotation $[\![\Gamma]\!]$ of an environment Γ of the form $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ is then the product of the denotations of the σ_i, as usual, and we now have to find the denotations

$$[\![M]\!] : [\![\Gamma]\!] \to T([\![\sigma]\!])$$

of terms of type σ in the environment Γ. These are defined as in [21] for the λ_c part of our language, once we settle the interpretations of the function symbols $\underline{0}$, succ, zero, pred, tt and ff. We then have to consider the conditional and the effect operations. For the former, it is enough to specify a morphism in $\mathbf{C}$ of appropriate type. For $\underline{0}$ and succ we take $1 \xrightarrow{0} \mathbf{N}$ and $\mathbf{N} \xrightarrow{s} \mathbf{N}$ respectively; for zero and pred we take $\mathbf{N} \xrightarrow{z} \mathbf{T}$ and $\mathbf{N} \xrightarrow{p} \mathbf{N}$, where $z = \text{Iter}_{\mathbf{T}}(\text{inl}, \text{inr} \circ t)$ and $p = \pi_1 \circ \text{Iter}_{\mathbf{N}\times\mathbf{N}}(< 0, 0 >, < s \circ \pi_1, \pi_1 >)$ respectively; and for tt and ff we take $1 \xrightarrow{inl} \mathbf{T}$ and $1 \xrightarrow{inr} \mathbf{T}$.

In order to give the semantics of conditionals, we note the isomorphisms

$$\begin{aligned} \mathbf{C}_T(y, z)^2 &\cong \mathbf{C}(1, y \Rightarrow z)^2 \\ &\cong \mathbf{C}(\mathbf{T}, y \Rightarrow z) \\ &\cong \mathbf{C}_T(\mathbf{T} \times y, z) \\ &\cong \mathbf{C}_T(y \times \mathbf{T}, z) \end{aligned}$$

and take $\mathrm{cond}_z : T(z)^2 \times \mathbf{T} \longrightarrow z$ to be the $\mathbf{C}_T$ morphism corresponding to the pair π_1, π_2 of $\mathbf{C}_T$ morphisms from $T(z)^2$ to z. Now, for a term $M$of the form **if** L **then** M **else** N, where $\Gamma \vdash M : \sigma$, we define

$$[\![\textbf{if } L \textbf{ then } M \textbf{ else } N]\!] = \mathrm{cond}^{\dagger}_{[\![\sigma]\!]^{\circ}} << [\![M]\!], [\![N]\!] >, [\![L]\!] >$$

Finally, for a term M of type σ of the form $f(M_1, \ldots, M_n)$ we define

$$[\![f(M_1, \ldots, M_n)]\!] = f_{[\![\sigma]\!]^{\circ}} < [\![M_1]\!], \ldots, [\![M_n]\!] >$$

The next two lemmas say that the semantics of a value is effect free (it *exists* in the sense of [21]) and that the above equations are sound for the semantics.

Lemma 1. *Suppose* $V : \sigma$. *Then* $[\![V]\!] : 1 \to [\![\sigma]\!]$ *factors through* $\eta_{[\![\sigma]\!]}$.

Lemma 2. *If* $\Gamma \vdash M = N : \sigma$ *then* $[\![M]\!] = [\![N]\!]$

The naturality condition for operations is used here to establish the soundness of the commutation schema. In fact, only naturality with respect to Kleisli morphisms is used (rather than parametric naturality); the latter would be needed for open contexts.

When reading the following adequacy theorem, recall that, by Theorem 1, all computations terminate.

Theorem 2. Adequacy *Suppose that* $M : \sigma$. *Then* $[\![M]\!] = [\![\,|\, M \,|\,]\!]$

Proof. This is an immediate consequence of Proposition 2 and Lemma 2

This result is very much in the spirit of Mezei and Wright [19], and Theorem 4.26 of [8] is a similar result for recursive program schemes. Such results say that the denotational semantics of a program is that of the result of a preliminary symbolic computation. Our result may not seem to the reader to be the expected statement of adequacy, but it does imply that the semantics of a term determines its operational result (at least up to its meaning). Furthermore, as we shall see in the next section, it readily yields the adequacy theorem one would expect in concrete cases.

4 Examples

We take $\mathbf{C}$ to be **Set** in our examples, and consider two monads, one for nondeterminism and the other for probabilistic nondeterminism.

Nondeterminism

Here T is $\mathcal{F}^+$ the nonempty finite powerset functor, equipped with the evident strong monad structure (and recall that every monad on **Set** has a unique strength). We take Σ to have one binary (infix) operator **or**, and $\mathbf{or}_X$ to be binary union. Note that $\mathcal{F}^+(X)$ is the free semilattice over X (meaning the

structure with an associative, commutative and absorptive binary operator); this algebraic view of nondeterminism was emphasised in [12]. is $\{0, 1\}$.

A small step structural operational semantics can be be defined much like our general one, except that there is no reason to record whether a "left choice" or a "right choice" is made. So the definition of the transition relation $\rightarrow_n$ is exactly as the general one except that one puts

$$M_1 \ \mathbf{or} M_2 \rightarrow_n M_i \ \text{(i=1,2)}$$

This $\rightarrow_n$ is then the union of the general $\rightarrow$ and the $\stackrel{\mathbf{or}_i}{\rightarrow}$.

For big step semantics of nondeterminism one normally defines a nondeterministic transition relation between closed terms and values; for example the rule for function application is

$$\frac{M \Rightarrow_n \lambda x : \sigma.M' \quad N \Rightarrow_n V \quad M'[V/x] \Rightarrow_n V'}{MN \Rightarrow_n V'}$$

However, there is another possibility. This is to define a collecting big step transition relation $M \Rightarrow_n u$ between closed terms and nonempty finite sets of values. It can be given a structural definition by very similar rules to those for the general collecting big step semantics, such as

$$\frac{M \Rightarrow_n \{\lambda x : \sigma.M_i\} \quad N \Rightarrow_n \{V_j\} \quad M_i[V_j/x] \Rightarrow_n u_{ij} \ \text{(for all } i,j)}{MN \Rightarrow_n \bigcup_{i,j} u_{ij}}$$

Its relation to the normal nondeterministic big step transition relation is that for any $M : \sigma$,

$$M \Rightarrow_n u \text{ iff } u = \{V \mid M \Rightarrow_n V\}$$

Now we can make the relationship between operational semantics for nondeterminism and the corresponding case of the general operational semantics for algebraic effects explicit. First to every effect term t assign a nonempty finite set of values $h(t)$ by

$$h(V) = \{V\}$$
$$h(t \ \mathbf{or} \, t') = h(t) \cup h(t')$$

Then one has that for any $M : \sigma$,

$$M \Rightarrow_n u \text{ iff } \exists t.M \Rightarrow t \wedge u = h(t)$$

as will be evident from the form of the rules for the collecting big step transition relation.

One has for any effect term $t : \sigma$ that

$$[\![t]\!](*) = \bigcup_{V \in h(t)} [\![V]\!](*)$$

and with this one can prove an adequacy theorem for nondeterminism using our general adequacy theorem.

Theorem 3. *For any closed term $M : \sigma$,*

$$[\![M]\!](*) = \bigcup_{M \Rightarrow_n V} [\![V]\!](*)$$

Proof.

$$\begin{aligned} [\![M]\!](*) &= [\![\,|\,M\,|\,]\!](*) \text{ (by Theorem 2)} \\ &= \bigcup_{V \in h(|M|)} [\![V]\!](*) \text{ (by above remark)} \\ &= \bigcup_{M \Rightarrow_n V} [\![V]\!](*) \text{ (by above remark)} \end{aligned}$$

As stated this theorem is rather abstract because of the higher types. For $\sigma = \iota$ it takes the form that for any closed term $M : \iota$

$$m \in [\![M]\!](*) \text{ iff } M \Rightarrow_n \underline{m}$$

Probabilistic Nondeterminism

Here things are, perhaps, not quite so simple. We take $T(X)$ to be $\mathcal{D}_\omega(X)$ the set of finite probability distributions over X. The unit η sends an element of X to the corresponding point distribution. Every finite distribution can be represented (though not uniquely) as an affine combination $\sum_{i=1,n} p_i \eta(x_i)$ of point distributions (meaning that $p_i \geq 0$ and $\sum_{i=1,n} p_i = 1$). The multiplication is given by:

$$\mu(\sum_{i=1,n} p_i \eta(\nu_i)) = \sum_{i=1,n} p_i \nu_i$$

and the (unique) strength by

$$\mathrm{st}(<x, \sum_{i=1,n} p_i \eta(y_i)>) = \sum_{i=1,n} p_i \eta(<x, y_i>)$$

We take one operation, a binary "fair-coin" probabilistic choice $+$ whose semantics is given by

$$\nu +_X \nu' = 1/2\nu + 1/2\nu'$$

Note the use of infix notation. A point worth noting is that while $\mathcal{D}$ supports this family, $\mathcal{D}_\omega(X)$ is not the free algebra over X corresponding to the equations true of $+_X$ as that only generates binary distributions.

Giving small step operational semantics is a little awkward. One might imagine using a relation $M \xrightarrow{p} N$ where p is 1 if no probabilistic choice is involved and $1/2$ otherwise. However consideration of the example $\underline{0} + \underline{0}$ shows that some information needed to find the distribution of final values is lost this way. If one tries a big step semantics $M \stackrel{p}{\Rightarrow} N$ the problem becomes more acute: consider the two terms $(\underline{0} + \underline{1}) + (\underline{1} + \underline{1})$ and $(\underline{1} + \underline{0}) + (\underline{0} + \underline{0})$. One standard solution for the small step semantics is to record enough information on the path taken

to resolve any ambiguities; in that case the small step semantics is essentially identical to the general one, where one would have both $M \xrightarrow{+_1} \underline{0}$ and $M \xrightarrow{+_2} \underline{0}$ for $M = \underline{0} + \underline{0}$.

There is also a collecting big step transition relation $M \Rightarrow_p \nu$ where M is closed and ν is a distribution over values. Here is an example rule (we now omit writing η)

$$\frac{M \Rightarrow_p \sum_i p_i(\lambda x : \sigma.M_i) \quad N \Rightarrow_p \sum_j q_j V_j \quad M_i[V_j/x] \Rightarrow_p \nu_{ij} \text{ (for all } i,j)}{MN \Rightarrow_p \sum_{ij} p_i q_j \nu_{ij}}$$

We can relate this to our general collecting big step operational semantics much as in the case of ordinary nondeterminism, defining a distribution $h(t)$ over values for every effect term t by

$$h(V) = V$$
$$h(t + t') = 1/2h(t) + 1/2h(t')$$

and one has for any $M : \sigma$ that $M \Rightarrow_p \nu$ if, and only if, $\exists t.M \Rightarrow t \wedge \nu = h(t)$. Proceeding as before we now note that for any effect term $t : \sigma$

$$[\![t]\!](*) = \sum p_i [\![V_i]\!](*)$$

where $h(t) = \sum p_i V_i$, which yields an adequacy theorem.

Theorem 4. *Suppose that $M : \sigma$ and set $\sum p_i x_i = [\![M]\!](*)$. Then there are V_i such that $x_i = [\![V_i]\!](*)$ and $M \Rightarrow_p \sum p_i V_i$.*

Again, this takes a clearer form for any terms M of type ι:

$$[\![M]\!](*) = \sum p_i m_i \text{ iff } M \Rightarrow_p \sum p_i \underline{m}_i$$

5 PCF with Recursion

We add recursion to our language by a binding operator:

$$\mathrm{Rec}\,(f : \sigma \to \tau, x : \sigma.M)$$

with the typing rule

$$\frac{\Gamma, f : \sigma \to \tau, x : \sigma \vdash M : \tau}{\Gamma \vdash \mathrm{Rec}\,(f : \sigma \to \tau, x : \sigma.M) : \sigma \to \tau}$$

For the operational semantics, we regard this as a new kind of redex and add the rule

$$\mathrm{Rec}\,(f : \sigma \to \tau, x : \sigma.M) \to \lambda x : \sigma.M[\mathrm{Rec}\,(f : \sigma \to \tau, x : \sigma.M)/f]$$

This yields the small step operational semantics as before, with the analogous definitions of values and evaluation contexts (and with the analogous unique

analysis of closed well-typed terms into one of the forms $E[R]$ or $E[a()]$). The possible transitions of a closed well-typed term can again be analysed into one of three mutually exclusive possibilities.

What differs from the previous situation is that terms need not terminate and so the small step semantics yields a tree of possibly infinite depth, branching finitely at labelled transitions but deterministic at unlabelled ones. So it is natural to consider *infinitary* effect values, that is, infinitary Σ-terms. The right tool for these is $CT_\Sigma(X)$, the free continuous Σ-algebra over a set X; it contains both finite partial and total elements as well as infinitary ones. (A *continuous Σ-algebra* is a dcppo equipped with continuous functions of appropriate arity for each operation symbol of Σ; a morphism of such algebras is a strict continuous function preserving the operations; CT_Σ is the left adjoint to the forgetful functor from the category of continuous Σ-algebras to that of sets—see below for the definitions of dcppo, etc.)

We may think of elements of this algebra as finite or infinite Σ-terms, with elements of X acting as extra constants. The finite ones are given by the grammar:

$$t \ ::= \ x \ \mid \ f(t_1, \ldots, t_n) \ \mid \ \Omega$$

with x ranging over X and with least element Ω. Every element t is is the limit of its finite approximants $t^{(k)}$ of level k, defined by

$$t^{(0)} = \Omega$$

$$t^{(k+1)} = \begin{cases} \Omega & (\text{if } t = \Omega) \\ x & (\text{if } t = x \in X) \\ f(t_1^{(k)}, \ldots, t_n^{(k)}) & (\text{if } t = f(t_1, \ldots, t_n) \text{ where } n = \mathrm{ar}_f) \end{cases}$$

We therefore take the (possibly infinitary) effect values of type σ to be the elements of $CT_\Sigma(Val_\sigma)$ where Val_σ is the set of values of type σ, and wish to associate to every term $M : \sigma$ such a value $|\, M \,|$.

To this end we need to "factor out" the $\to$ moves, which we do by defining a medium step operational semantics for closed terms, by

$$M \Rightarrow V \text{ iff } M \to^* V$$

$$M \stackrel{f_i}{\Rightarrow} N \text{ iff } \exists L. M \to^* L \stackrel{f_i}{\to} N$$

$$M \Downarrow_a \text{ iff } \exists L. M \to^* L \downarrow_a$$

$M \Uparrow$ iff there is an infinite sequence $M = M_0 \to M_1 \to \ldots \to M_n \to \ldots$

The approximants of level k of $|\, M \,|$ (where $M : \sigma$) are now defined by $|\, M \,|^{(0)} = \Omega$ and

$$|\, M \,|^{(k+1)} = \begin{cases} V & (\text{if } M \Rightarrow V) \\ f(|\, M_1 \,|^{(k)}, \ldots, |\, M_n \,|^{(k)}) & (\text{if } M \stackrel{f_i}{\Rightarrow} M_i \text{ for } i = 1, n, \text{ where } n = \mathrm{ar}_f) \\ a() & (\text{if } M \Downarrow_a) \\ \Omega & (\text{if } M \Uparrow) \end{cases}$$

Lemma 3. *Suppose that $M : \sigma$. Then $|\,M\,|$ satisfies the following equation.*

$$|\,M\,| = \begin{cases} V & (\text{if } M = V) \\ |\,N\,| & (\text{if } M \to N) \\ f(|\,N_1\,|, \ldots, |\,N_n\,|) & (\text{if } M \xrightarrow{f_i} N_i \text{ for } i = 1, n, \text{ where } n = ar_f) \\ a() & (\text{if } M = E[a()]) \end{cases}$$

This lemma can be strengthened to show that $|\cdot|$ is the least such function, under the pointwise ordering; indeed that could be taken as an alternative definition of $|\cdot|$.

For a collecting big step semantics one would naturally wish to give a system of rules defining the relation $M \Rightarrow t$ between closed well-typed terms of type σ and effect values in $CT_\Sigma(Val_\sigma)$ where

$$M \Rightarrow t \text{ iff } t = |\,M\,|$$

However it is not immediately clear how to think of a system of finitary rules as generating such a relation, let alone the precise form such rules should take. A related approach would be to define the evaluation function $|\cdot|$ as the least solution to a recursive equation defined by cases following the structure of M. Another idea is to define a nondeterministic relation between closed terms and finite effect values in $CT_\Sigma(Val_\sigma)$ such that the set of such values is directed and has lub the evaluation of the term; one can accomplish this by adding the axiom $M \Rightarrow \Omega$ and the rule

$$\mathrm{Rec}\,(f : \sigma \to \tau, x : \sigma.M) \Rightarrow \lambda x : \sigma.M[\mathrm{Rec}\,(f : \sigma \to \tau, x : \sigma.M)/f]$$

to the other rules; this idea appears in [13]. We do not enter further into these issues here. turn

As before there is a natural equational theory. The axioms are as before (but for the extended language) together with

$$\mathrm{Rec}\,(f : \sigma \to \tau, x : \sigma.M) = \lambda x : \sigma.M[\mathrm{Rec}\,(f : \sigma \to \tau, x : \sigma.M)/f]$$

This goes well with the medium step semantics

Lemma 4. *Suppose $M : \sigma$. Then*

1. *If $M \Rightarrow V$ then $\vdash M = V : \sigma$.*
2. *If $M \xrightarrow{f_i} N_i$ for $i = 1, n$ (where $n = ar_f$) then $\vdash M = f(N_1, \ldots, N_n) : \sigma$*
3. *If $M \Downarrow_a$ then $\vdash M = a() : \sigma$*

This lemma is an immediate consequence of the evident corresponding lemma for small step operational semantics.

We now turn to denotational semantics, assuming the same categorical structure as before, but enriched with a suitable ordering structure to accommodate recursion. We use the cartesian closed category **Dcpo** of dcpos and continuous functions, and the closed category **Dcppo** of dcppos and strict continuous functions. (A *dcpo (complete partial order)* is a partial order with lubs of directed

sets; a *continuous* function $f : P \to Q$ between such dcpos is a monotone function which preserves the lubs. A *dcppo (complete pointed partial order)* is a dcpo with a least element; a function between such dcppos is *strict* if it preserves the least elements.)

So we assume that $\mathbf{C}$ is **Dcpo**-enriched and so are T, $\mathbf{C}$-products and the Kleisli exponentials. We also assume that $\mathbf{C}_T$ is **Dcppo**-enriched and that the strength is strict, in the sense that, $\mathrm{st}_{x,y^\circ} < f, \perp_{z,T(y)} > = \perp_{z,T(x\times y)}$ holds for any $f : z \to x$.

With this we can define $[\![\sigma]\!]$ as before and also the denotations

$$[\![M]\!] : [\![\Gamma]\!] \to T([\![\sigma]\!])$$

of terms $\Gamma \vdash M : \sigma$, except for the recursion construct. For this for a term $\Gamma, f : \sigma \to \tau, x : \sigma \vdash M : \tau$ we set

$$[\![\mathrm{Rec}\,(f : \sigma \to \tau, x : \sigma.M)]\!] = \mathrm{Y}(\lambda g : \mathbf{C}([\![\Gamma]\!], [\![\sigma \to \tau]\!]).\lambda_T([\![M]\!])^\circ < id_{[\![\Gamma]\!]}, g >)$$

where Y is the usual least fixed-point operator, $\mathrm{Y}(G) = \bigvee_{n\geq 0} G^n(\perp)$.

As before, the semantics is sound for the equational theory; combining this with Lemma 4 we obtain

Lemma 5. *Let $M : \sigma$ be a closed term. Then*

1. *If $M \Rightarrow V$ then $[\![M]\!] = [\![V]\!]$*
2. *If $M \stackrel{f_i}{\Rightarrow} N_i$ for $i = 1, n$ (where $n = ar_f$) then $[\![M]\!] = [\![f(N_1, \ldots, N_n)]\!]$*
3. *If $M \Downarrow_a$ then $[\![M]\!] = [\![a()]\!]$*

Now, since every $V : \sigma$ has a denotation $[\![V]\!]$ in the dcppo $\mathbf{C}_T(1, [\![\sigma]\!])$ and since, for every n-ary operation f, we have an n-ary continuous function on this dcppo induced by $f_{[\![\sigma]\!]}$, there is a unique continuous, strict Σ-homomorphism

$$[\![\ ]\!] : CT_\Sigma(Val_\sigma) \to \mathbf{C}_T(1, [\![\sigma]\!])$$

lifting $[\![\ ]\!] : Val_\sigma \to \mathbf{C}_T(1, [\![\sigma]\!])$. We are now in a position to state the main theorem of the paper:

Theorem 5. Adequacy for recursion. *For any term $M : \sigma$, $[\![M]\!] = [\![\,|\,M\,|\,]\!]$*

It is straightforward to prove half of this theorem, that $[\![M]\!] \geq [\![\,|\,M\,|\,]\!]$. That is an immediate consequence of the inequality

$$[\![M]\!] \geq [\![\,|\,M\,|^{(k)}]\!]$$

which can be proved by induction on k using Lemma 5. To prove the other half of the theorem we introduce a new language $\mathcal{A}$ in order to talk about approximants to terms of PCF with recursion. (It would be desirable to find an alternate proof using logical relations, e.g., as in [32].) The language $\mathcal{A}$ is obtained by adding two new families of constructs Ω_σ and $\mathrm{Rec}_n(f : \sigma \to \tau, x : \sigma.M)$ to the language of Section 2 with the typing rules:

$$\Gamma \vdash \Omega_\sigma : \sigma$$

and

$$\frac{\Gamma, f : \sigma \to \tau, x : \sigma \vdash M : \tau}{\Gamma \vdash \mathrm{Rec}_n(f : \sigma \to \tau, x : \sigma.M) : \sigma \to \tau}$$

The redexes and their transitions are defined as in Section 2, but with the addition of:

$$\mathrm{Rec}_{n+1}(f : \sigma \to \tau, x : \sigma.M) \to \lambda x : \sigma.M[\mathrm{Rec}_n(f : \sigma \to \tau, x : \sigma.M)/f]$$

and

$$\mathrm{Rec}_0(f : \sigma \to \tau, x : \sigma.M) \to \lambda x : \sigma.\Omega_\tau$$

Values, evaluation contexts and small step transitions are defined analogously to before. Every closed well-typed term is either a value or else can be analysed uniquely into one of the forms $E[R]$, $E[a()]$ or $E[\Omega_\tau]$; this yields an evident corresponding analysis of the transition possibilities, and we have

Lemma 6. *Every transition sequence terminates, either in a value or in a term of the form $E[\Omega_\tau]$.*

Proof. This can again be proved using a computability argument. The computability predicates are defined as in Theorem 1 except that we say that a closed term is computable iff every transition sequence from it terminates, either in a computable value or else in a term of the form $E[\Omega_\tau]$.

This allows us to define evaluation $|\, M \,|$. For any $M : \sigma$ we define $|\, M \,|$ in $CT_\Sigma(Val_\sigma)$ by

$$|\, M \,| = \begin{cases} V & (\text{if } M = V) \\ |\, N \,| & (\text{if } M \to N) \\ f(|\, N_1 \,|, \ldots, |\, N_n \,|) & (\text{if } M \xrightarrow{f_i} N_i, \text{ for } i = 1, n, \text{ where } n = \mathrm{ar}_f) \\ \Omega_\sigma & (\text{if } M = E[\Omega_\tau]) \end{cases}$$

We again have an equational theory. This is as in Section 2, together with

$$\mathrm{Rec}_{n+1}(f : \sigma \to \tau, x : \sigma.M) = \lambda x : \sigma.M[\mathrm{Rec}_n(f : \sigma \to \tau, x : \sigma.M)/f]$$

$$\mathrm{Rec}_0(f : \sigma \to \tau, x : \sigma.M) = \lambda x : \sigma.\Omega_\tau$$

$$E[\Omega_\tau] = \Omega_\sigma$$

As before $\vdash M =| M |: \sigma$ (if $M : \sigma$). For the semantics we take the same structure as that for PCF with recursion, defined as for the language of Section 2 and putting for the new constructs

$$[\![\mathrm{Rec}_n(f : \sigma \to \tau, x : \sigma.M)]\!] = \mathrm{Y}^{(n)}(\lambda g : \mathbf{C}([\![\Gamma]\!], [\![\sigma \to \tau]\!]).\lambda_T([\![M]\!]) \circ < id_{[\![\Gamma]\!]}, g >)$$

where $\mathrm{Y}^{(n)}(G) = G^n(\bot)$, and

$$[\![\Omega_\sigma]\!] = \bot$$

The equations are sound for this semantics and so we have that $[\![M]\!] = [\![\,|\,M\,|\,]\!]$, for $M : \sigma$.

With all this in hand we can now turn to the central relation

$$\overline{M} \prec M$$

of *approximation* between $\mathcal{A}$-terms and terms of PCF with recursion. This is the least relation closed under the common term-forming constructs such that $x \prec x$, $\Omega_\sigma \prec M$, and $\mathrm{Rec}_n(f : \sigma \to \tau, x : \sigma.\overline{M}) \prec \mathrm{Rec}\,(f : \sigma \to \tau, x : \sigma.M)$ if $\overline{M} \prec M$. It is straightforward to prove that this relation is closed under substitution in the sense that if $\overline{M} \prec M$ and $\overline{N} \prec N$ then $\overline{M}[\overline{N}/x] \prec M[N/x]$; it is equally straightforward to show that if $\overline{M} \prec M$ then $[\![\overline{M}]\!] \leq [\![M]\!]$ (where $\Gamma \vdash \overline{M} : \sigma$ and $\Gamma \vdash M : \sigma$).

The *nth-approximant* $M^{(n)}$ of M is obtained by replacing every occurrence of Rec by one of Rec_n. Clearly $M^{(n)} \prec M$ and if $\Gamma \vdash M : \sigma$ then $\Gamma \vdash M^{(n)} : \sigma$; further, for any term $\Gamma \vdash M : \sigma$, $[\![M^{(n)}]\!]$ is increasing and has lub $[\![M]\!]$ (as can be shown by a straightforward induction on n). For the next lemma we extend the definition of the $\prec$ relation to contexts, taking $[\,] \prec [\,]$.

Lemma 7. *Suppose that $\overline{M} \prec M$ where $\overline{M} : \sigma$ and $M : \sigma$. Then:*

1. *If $\overline{M}$ is a value, so is M.*
2. *If $\overline{M}$ has the form $\overline{E}[\overline{R}]$ then M has the form $E[R]$ where $\overline{E} \prec E$ and $\overline{R} \prec R$.*
3. *If $\overline{M}$ has the form $\overline{E}[a()]$ then M has the form $E[a()]$ where $\overline{E} \prec E$.*

Lemma 8. *Suppose that $\overline{R} \prec R$ where $\overline{R} : \sigma$ and $R : \sigma$. Then:*

1. *If $\overline{R} \to \overline{N}$ then, for some $N \succ \overline{N}$, $R \to N$.*
2. *If $\overline{R} \xrightarrow{f_i} \overline{N_i}$, for $i = 1, ar_f$ then for some $N_i \succ \overline{N}_i$, $R \xrightarrow{f_i} N_i$.*

Proposition 3. *Suppose that $\overline{M} \prec M$ where $\overline{M} : \sigma$ and $M : \sigma$. Then we have that $[\![\,|\,\overline{M}\,|\,]\!] \leq [\![\,|\,M\,|\,]\!]$.*

Proof. The proof proceeds by well-founded induction on the (union of) the transitions from $\overline{M}$ and cases on its form.

Suppose that it is a value. Then so is M, by Lemma 7 and so we have that $|\,\overline{M}\,| = \overline{M}$, $|\,M\,| = M$ and $[\![\overline{M}]\!] \leq [\![M]\!]$ (as $\overline{M} \prec M$). Taking these facts together yields the required conclusion.

Next, suppose that $\overline{M}$ has the form $\overline{E}[\overline{R}]$. Then by Lemma 7, M has the form $E[R]$ where $\overline{E} \prec E$ and $\overline{R} \prec R$. There are two subcases. In the first $\overline{R} \to \overline{N}$ and so, by Lemma 8, for some $N \succ \overline{N}$, $R \to N$. But then we have:

$$\begin{aligned}
[\![\,|\,\overline{M}\,|\,]\!] &= [\![\,|\,\overline{E}[\overline{N}]\,|\,]\!] \text{ (by lemma 3)} \\
&\leq [\![\,|\,E[N]\,|\,]\!] \text{ (by induction hypothesis, as } \overline{E}[\overline{N}] \prec E[N]) \\
&= [\![\,|\,M\,|\,]\!] \text{ (by lemma 3)}
\end{aligned}$$

The second subcase is where $\overline{R} \xrightarrow{f_i} \overline{N_i}$, for $i = 1, \mathrm{ar}_f$, and this is dealt with similarly.

Finally, the case where $\overline{M}$ has the form $\overline{E}[a()]$ is straightforward.

This proposition immediately yields the other half of Theorem 5 as we have

$$\begin{aligned}[\![M]\!] &= \bigvee_{n\geq 0} [\![M^{(n)}]\!] \\ &= \bigvee_{n\geq 0} [\![\,|\,M^{(n)}\,|\,]\!] \\ &\leq [\![\,|\,M\,|\,]\!] \text{ (by the lemma)}\end{aligned}$$

6 Examples and Recursion

We now take **C** to be **Dcpo** in our examples, and again consider two monads, for nondeterminism and probabilistic nondeterminism.

Nondeterminism

The original powerdomain construction was not defined for all dcppos; the free continuous algebra approach of [12,1] yields a usable definition for all dcppos. For dcpos we take $T(P)$ to be the free dcppo over P which is also a continuous semilattice. This can be usefully analysed further. Let $\mathcal{S}(P)$ be the free continuous semilattice over a dcpo P; the existence of such algebras follows from the adjoint functor theorem, as in, e.g., [12,1]. We write $\cup$ for the semilattice operation. This is a **Dcpo**-enriched monad and so has a unique continuous strength. One can show that if, in fact, P is a dcppo then so is $\mathcal{S}(P)$, and, indeed, it is the free continuous semilattice over P in **Dcppo**; thus it is the powerdomain in the sense of [12]. Now let $P_\perp$ be the free dcppo over a dcpo P. Then one has that $T(P)$ is $\mathcal{S}(P_\perp)$.

For a set X—considered as a flat dcpo—$T(X)$ is easy to describe directly. It consists of all subsets of $X_\perp$ which are either finite or are countable and contain $\perp$. Such sets are ordered by the Egli-Milner ordering $u \leq v$ if, and only if, either 1) $\perp \in X$ and $u \setminus \perp \subset v$ or 2) $\perp \notin X$ and $u = v$. A lub $\bigvee u_k$ of an increasing sequence contains $x \in X$ if, and only if, some u_k does, and $\perp$ if, and only if, every u_k does. Finally, $\cup_X$ is set-theoretic union, and $\eta_X(x)$ is the singleton $\{x\}$.

The small step semantics $\to_n$ is, as before, the union of the general $\to$ and the $\overset{\cup_i}{\to}$. We write $M \downarrow_n$ if there is no infinite transition sequence from a term $M : \sigma$; since $\to_n$ is finitary there is then, by König's lemma, a bound on the length of such transition sequences. Linking up to the evaluation $|\,M\,|$ one finds that $M \to_n^* V$ if, and only if, V occurs in $|\,M\,|$ (meaning that it occurs in some approximant) and $M \downarrow_n$ if, and only if, $|\,M\,|$ is finite and total (meaning that it contains no occurrence of any Ω_τ).

Here is an adequacy theorem for ground types (ι or o).

Theorem 6. *Let σ be a ground type and suppose that $M : \sigma$. Then $x \in [\![M]\!](*)$ if, and only if, one of the following hold*

1. *$M \to_n^* V$ and $x = [\![V]\!](*)$*
2. *$M \not\downarrow_n$ and $x = \perp$*

Proof. We have that $[\![M]\!](*) = [\![|\, M \,|]\!](*) = \bigvee_{n\geq 0} [\![|\, M \,|^{(n)}]\!](*)$

Suppose first that $x \neq \perp$. Then $x \in [\![|\, M \,|^{(n)}]\!](*)$ for some n and it follows that $\{x\} = [\![V]\!](*)$ for some V that occurs in $|\, M \,|^{(n)}$. For such a V we have that $M \rightarrow_n^* V$.

Suppose instead that $x = \perp$. Then $\perp$ is in every $[\![|\, M \,|^{(n)}]\!](*)$. Since no $[\![V]\!](*)$ is $\{\perp\}$ at ground types (actually at all types), it follows that no $|\, M \,|^{(n)}$ is total and hence that $M \not\downarrow_n$.

Essentially this result is in [31] and a stronger one for both call-by-name and call-by-value is in [10]; see [3,11,9] for work on call-by-name and nondeterminism. One can presumably also prove an adequacy theorem at other types, but the statement is more complex due to the need for closure operators in the set-theoretic representation of powerdomains, and there is some work to be done in extending the theory of bifinite domains to predomains (a dcpo P should be bifinite if, and only if, $P_\perp$ is); see, e.g., [1,24,25] for details on powerdomains. However part of the conjectured theorem is simple to state, that at any type σ, if $M : \sigma$ then $\perp \in [\![M]\!](*)$ if, and only if, $M \downarrow_n$. As well as the above *convex* powerdomain, one can also prove appropriate adequacy theorems for the *upper* (Smyth) and *lower* (Hoare) powerdomains.

Probabilistic Nondeterminism

We take T to be $\mathcal{V}$, where $\mathcal{V}(P)$ is the dcpo of all evaluations on P. This functor and its strong monad structure is discussed in [14], and see, e.g., [15]; we just note that $\eta_P(x)$ is the singleton evaluation. The semantics of probabilistic choice $+$ is again an affine combination, this time of evaluations

$$\nu +_X \nu' = 1/2\nu + 1/2\nu'$$

$\mathcal{V}(P)$ is a dcppo which is a continuous Σ-algebra, and for any $f : P \to \mathcal{V}(Q)$, $f^\dagger$ is a strict, continuous Σ-algebra morphism.

As discussed before we can take the small step operational semantics to be the general one (for this case), and the question is rather one of interpretation. Let us write w to range over words in the alphabet $\{+_1, +_2\}$. We write $M \stackrel{w}{\Rightarrow} N$ to mean that w describes a sequence of medium step transitions from M to N. Now for a term $M : \sigma$ and value $V : \sigma$ set

$$\mathrm{Prob}(M, V) = \sum \{2^{-|w|} \mid M \stackrel{w}{\Rightarrow} V\}$$

Define a function θ from closed terms of type σ to the closed interval $[0, 1]$ by $\theta(M) = \mathrm{Prob}(M, V)$. Then θ obeys the equation

$$\theta(M) = \begin{cases} 1 & (\text{if } M \Rightarrow V) \\ 0 & (\text{if } M \Rightarrow V' \neq V) \\ 1/2\theta(N_1) + 1/2\theta(N_2) & (\text{if } M \stackrel{+_i}{\Rightarrow} N_i \ (i = 1, 2)) \\ 0 & (\text{if } M \not\downarrow) \end{cases} \tag{1}$$

This equation determines θ uniquely (use the Banach fixed-point theorem with the metric $d(\theta, \theta') = \sup_M |\theta(M) - \theta'(M)|$).

The next lemma relates the small step semantics of a term to its evaluation. Let h be the unique strict continuous Σ-algebra morphism from $CT_\Sigma(Val_\sigma)$ to $\mathcal{V}(Val_\sigma)$. The statement of the following lemma makes implicit use of the fact there there are at most countably infinitely many values of a given type, and that weighted countably infinite sums of evaluations exist (being defined pointwise).

Lemma 9. *Suppose that $M : \sigma$. Then*

$$h(|M|) = \sum_{V:\sigma} Prob(M, V)\eta(V)$$

Proof. It is straightforward to see that $h(|\cdot|)$ obeys the following equation (use the equation given for $|\cdot|$ in the previous section)

$$h(|M|) = \begin{cases} \eta(V') & (\text{if } M \Rightarrow V') \\ 1/2h(|N_1|) + 1/2h(|N_2|) & (\text{if } M \stackrel{+_i}{\Rightarrow} N_i \ (i = 1, 2)) \\ \bot & (\text{if } M \not\Downarrow) \end{cases}$$

It follows that, for any $V : \sigma$, $h(|\ \cdot\ |)(V)$ satisfies equation 1, and so as this equation has a unique solution, viz $\mathrm{Prob}(\cdot, V)$, the conclusion follows.

This enables us to prove an adequacy theorem at all types.

Theorem 7. *Suppose that $M : \sigma$. Then*

$$[\![M]\!](*) = \sum_{V:\sigma} Prob(M, V)[\![V]\!](*)$$

Proof. Both $[\![\cdot]\!](*)$ and $([\![\cdot]\!](*))^\dagger \circ h$ are strict continuous Σ-homomorphisms from $CT_\Sigma(Val_\sigma)$ to $\mathcal{V}([\![\sigma]\!])$, and both extend $[\![\cdot]\!](*) : Val_\sigma \to \mathcal{V}([\![\sigma]\!])$. They are therefore equal. We may then calculate:

$$\begin{aligned} [\![M]\!] &= [\![|M|]\!] \text{ (by Theorem 5)} \\ &= ([\![\cdot]\!](*))^\dagger(h(|M|)) \\ &= ([\![\cdot]\!](*))^\dagger(\sum_{V:\sigma} \mathrm{Prob}(M, V)\eta(V)) \text{ (by Lemma 9)} \\ &= \sum_{V:\sigma} \mathrm{Prob}(M, V)([\![\cdot]\!](*))^\dagger)(\eta(V)) \\ &= \sum_{V:\sigma} \mathrm{Prob}(M, V)[\![V]\!](*) \end{aligned}$$

An adequacy theorem for FPC with probabilistic choice was already proved in [13] (FPC can be viewed as an extension of our PCF with recursive types); for work on call-by-name and probabilistic nondeterminism see [6].

7 Conclusions

It is interesting to work out other examples. Printing provides one example, where one has a unary operation print_a for each symbol a of an alphabet A, and for **Set** one can take the (noncommutative) monad $T(X) = A^* \times X$, which is, in fact, the free Σ-algebra over X; for **Dcpo** one would also allow the possibility of infinite printing, and use $CT_\Sigma(P)$, the free continuous Σ-algebra over P. Here the general operational semantics is very much the same as what one would write anyway and it is straightforward to read off adequacy results for printing from the general theorems. An example worth some investigation is the combination $\mathcal{F}^+(\mathcal{D}_\omega(X))$ of probabilistic and ordinary nondeterminism; there is natural distributive law $\lambda : \mathcal{D}_\omega(\mathcal{F}^+(X)) \to \mathcal{F}^+(\mathcal{D}_\omega(X))$which makes this a monad; this way to combine the two forms of nondeterminism is used in a domain-theoretic context in [4]—modulo actions—and mentioned in [20] where an interesting idea of restricting to affine sets of evaluations is advocated.

In so far as we are successful with such examples, the question of how to treat other monads and their operations is the more pressing; exceptions, state and continuations all come immediately to mind. Possibly relevant here is the translational approach to defining operations in [5], but adapted to λ_c rather than the metalanguage; the idea would be to recover operational semantics via the translations. Ultimately, we would hope to incorporate the treatment of operational semantics into a modular approach to computational effects, e.g., along the lines of [26,28,29].

An obvious question is to consider language variations, such as an extension with recursive types or call-by-name; for the latter it would be preferable to use a framework incorporating both parameter-calling mechanisms, such as Levy's CBPV [18]. More intriguingly, one would wish to reconcile this work with the co-algebraic treatment of operational semantics in [33] with its use of behaviour functors and co-monads contrasting with our use of monads.

References

1. S. Abramsky and A. Jung, Domain Theory, in *Handbook of Logic in Computer Science* (eds. S. Abramsky, D.M. Gabbay and T. S. E. Maibaum), Vol. 3, Semantic Structures, Oxford: Clarendon Press, 1994.
2. S. O. Anderson and A. J. Power, A Representable Approach to Finite Nondeterminism, *Theoret. Comput. Sci.*, Vol. 177, No. 1, pp. 3–25, 1997.
3. E. Astesiano and G. Costa, Nondeterminism and Fully Abstract Models, in Informatique Théorique et Applications, Vol. 14, No. 4, pp. 323–347, 1980.
4. C. Baier and M. Kwiatkowska, *Domain Equations for Probabilistic Processes*, MSCS, Vol. 10, No. 6, pp. 665–717, 2000.
5. P. Cenciarelli and E. Moggi, A Syntactic Approach to Modularity in Denotational Semantics, in *Proc. 5th. Biennial Meeting on Category Theory and Computer Science*, LNCS, Berlin: Springer-Verlag, 1993.
6. V. Danos and R. Harmer, Probabilistic Game Semantics, in *Proc. 15th LICS*, pp. 204–213, Washington: IEEE Press, 2000.

7. M. Felleisen, and D. P. Friedman, Control Operators, the SECD-machine, and the Lambda-Calculus, in *Formal Description of Programming Concepts III* (ed. M. Wirsing), pp. 193–217, Amsterdam: Elsevier, 1986.
8. I. Guessarian, *Algebraic Semantics*, LNCS, Vol. 99, Berlin: Springer-Verlag, 1981.
9. R. Harmer and G. McCusker, A Fully Abstract Game Semantics for Finite Nondeterminism, in *Proc. 14th LICS*, pp. 422–430, Washington: IEEE Press, 1999.
10. M. C. B. Hennessy, The Semantics of Call-By-Value and Call-By-Name in a Nondeterministic Environment, in *SIAM J. Comput.*, Vol. 9, No. 1, pp. 67–84, 1980.
11. M. C. B. Hennessy and E. A. Ashcroft, A Mathematical Semantics for a Nondeterministic Typed Lambda-Calculus, in TCS Vol. 11, pp. 227-245, 1980.
12. M. C. B. Hennessy and G. Plotkin, Full Abstraction for a Simple Parallel Programming Language, in *Proc. 8th MFCS* (ed. J. Beĉvár̂), Olomouc, Czechoslovakia, LNCS, Vol. 74, pp. 108–120, Berlin: Springer-Verlag, 1979.
13. C. Jones, *Probabilistic Non-Determinism*, Ph.D. Thesis, University of Edinburgh, Report ECS-LFCS-90-105, 1990.
14. C. Jones and G. D. Plotkin, *A Probabilistic Powerdomain of Evaluations*, in *Proc. 4th LICS*, Asilomar, pp. 186–195, Washington: IEEE Press, 1989.
15. A. Jung and R. Tix, The Troublesome Probabilistic Powerdomain, in *Proc. Third COMPROX Workshop*, ENTCS, Vol. 13, Amsterdam: Elsevier, 1998.
16. G. M. Kelly, *Basic Concepts of Enriched Category Theory*, Cambridge: CUP, 1982.
17. Y. Kinoshita and A. J. Power, *Data Refinement for Call by Value Languages*, submitted, 2000.
18. P. B. Levy, Call-by-Push-Value: A Subsuming Paradigm, in *Proc. TLCA '99* (ed. J.-Y. Girard), LNCS, Vol. 1581, pp. 228-242, Berlin: Springer-Verlag, 1999.
19. J. Mezei and J. B. Wright, *Algebraic Automata and Context Free Sets*, in Information and Control, Vol. 11, pp. 3–29, 1967.
20. M. W. Mislove, Nondeterminism and Probabilistic Choice: Obeying the Laws, in *Proc. CONCUR 2000* (ed. C. Palamidessi), LNCS, Vol. 1877, pp. 350–364, Berlin: Springer-Verlag, 2000.
21. E. Moggi, Computational Lambda-Calculus and Monads, in *Proc. LICS '89*, pp. 14–23, Washington: IEEE Press, 1989.
22. E. Moggi, *An Abstract View of Programming Languages*, University of Edinburgh, Report ECS-LFCS-90-113, 1989.
23. E. Moggi, *Notions of Computation and Monads*, Inf. and Comp., Vol. 93, No. 1, pp. 55–92, 1991.
24. G. D. Plotkin, *A Powerdomain Construction*, SIAM J. Comput. Vol. 5, No. 3, pp. 452–487, 1976.
25. G. D. Plotkin, *Domains*, (http://www.dcs.ed.ac.uk/home/gdp/), 1983.
26. A. J. Power, Modularity in Denotational Semantics, in *Proc. MFPS XIII* (eds. S. Brookes and M. Mislove), ENTCS, Vol. 6, Amsterdam: Elsevier, 1997.
27. A. J. Power, Enriched Lawvere Theories, in *Lambek Festschrift* (eds. M. Barr, P. Scott and R. Seely), TAC, Vol. 7, pp. 83–93, 2000.
28. A. J. Power and E. P. Robinson, Modularity and Dyads, in *Proc. MFPS XV* (eds. S. Brookes, A. Jung, M. Mislove and A. Scedrov), ENTCS Vol. 20, Amsterdam: Elsevier, 1999.
29. A. J. Power and G. Rosolini, A Modular Approach to Denotational Semantics, in *Proc. ICALP '98* (eds. K. G. Larsen, S. Skyum and G. Winskel), LNCS, Vol. 1443, pp. 351–362 Berlin: Springer-Verlag, 1998.
30. A. J. Power and H. Thielecke, Closed *Freyd-* and κ-categories, in *Proc. 26th. ICALP* (eds. J. Wiedermann and P. van Emde Boas and M. Nielsen), LNCS, Vol. 1644, pp. 625–634, Berlin: Springer-Verlag, 1999.

31. K. Sieber, Call-by-Value and Nondeterminism, in *Proc. TLCA '93* (eds. M. Bezem and J. F. Groote), LNCS, Vol. 664, pp. 376–390, Berlin: Springer-Verlag, 1993.
32. A. Simpson, Computational Adequacy in an Elementary Topos, in *Proc. CSL '98*, LNCS, Vol. 1584, pp. 323–342, Berlin: Springer-Verlag, 1999.
33. D. Turi and G. D. Plotkin, Towards a Mathematical Operational Semantics, in *Proc. LICS 97*, pp. 268–279, Washington: IEEE Press, 1997.
34. G. Winskel, *The Formal Semantics of Programming Languages*, Cambridge: MIT Press, 1993.

Secrecy Types for Asymmetric Communication

Martín Abadi[1] and Bruno Blanchet[2]

[1] Bell Labs Research, Lucent Technologies,
abadi@research.bell-labs.com
[2] INRIA Rocquencourt***,
Bruno.Blanchet@inria.fr

Abstract. We develop a typed process calculus for security protocols in which types convey secrecy properties. We focus on asymmetric communication primitives, especially on public-key encryption. These present special difficulties, partly because they rely on related capabilities (e.g., "public" and "private" keys) with different levels of secrecy and scopes.

1 Introduction

A secret is something you tell to one person at a time, according to a popular definition. Research on security has led to several other concepts of secrecy (e.g., [12,16,2]). This paper studies secrecy in the context of security protocols with the methods of process calculi and type systems. Here, a secret is a datum (such as a cryptographic key) that never appears on a channel on which an adversary can listen, even if the adversary actively tries to obtain the datum. We develop a process calculus with a type system in which types convey secrecy properties and such that well-typed programs keep their secrets.

Many security protocols use asymmetric communication primitives, namely communication channels with only one fixed end-point (the receiver) and particularly public-key encryption. These primitives present special difficulties, partly because they rely on pairs of related capabilities (e.g., "public" and "private" keys) with different levels of secrecy and scopes. Therefore, we concentrate on these primitives, and their treatment constitutes the main novelty of this work. (An extended version of this paper gives corresponding type rules for symmetric primitives, such as shared-key encryption; they are more straightforward.)

Basically, our type system is concerned with the question of when it is permissible to tell a secret, and particularly to one person at a time. Telling it to one person at a time means that any communication channel used to transmit the secret has a unique receiver, or, if the secret is sent encrypted, that only one person has the corresponding decryption key. Of course, the person in question should be allowed to know the secret—it may still be a secret with respect to the adversary. Moreover, the person should realize that this is a secret, and treat it accordingly. The type system helps enforce these conditions.

*** This work was partly done while the author was at Bell Labs Research.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 25–41, 2001.

The rest of this introduction describes the process calculus that serves as setting for this work, the type system, and some of their properties. It also describes difficulties connected with public-key encryption and other asymmetric communication primitives. Along the way, it mentions relevant related work.

The process calculus. The process calculus is a relative of the pi calculus [21] and the spi calculus [3]. It includes channels with fixed receivers, as in the local pi calculus [19], the join calculus [11], and many object-oriented languages (e.g, [6]). Such a channel can be used for transmitting secrets if the adversary cannot listen on the channel. On the other hand, the capability for sending on the channel may be published. The channel may therefore convey not only secrets but also public data from the adversary. The receiver needs static and dynamic checks for distinguishing these two cases; our type system accounts for some of these mechanisms.

In addition, the process calculus includes cryptographic operations, specifically public-key encryption. In a public-key encryption scheme, the capabilities of encryption and decryption are separate (e.g., [18]), and can be handled differently. Hence public-key encryption is often called asymmetric encryption. Typically, the capability for decryption (the "private" key) remains with one principal, while the capability for encryption (the "public" key) may be published. Therefore, both secrets and public data from the adversary may be encrypted under the public key. Thus, pleasingly, public-key encryption resembles communication on channels with fixed receivers. Our process calculus and type system treat them analogously.

The phrase "public key" has at least two distinct meanings in this context. It may refer to one of the two keys in a public-key cryptosystem, used for encrypting data or for verifying digital signatures (which we do not treat here). Although such a key may be widely known, and it often is in examples, it can also be kept secret. Alternatively, a "public key" may be a key which happens to be public, that is, not secret. We try to use "public key" only with the latter meaning, and prefer "encryption key" for the former.

The type system. The type system is based on old ideas on secrecy levels [10] and on the newer trend of representing these levels in types (e.g., [28,1,13,14, 23,22,15,26,9]). For example, Public and Secret are the types of public data and secret data, respectively. In addition, the type system gives information on the intended usage and structure of data, like standard type systems. For example, $\mathrm{K}^{\mathrm{Secret}}[T_1, T_2]$ is the type of a secret encryption key that is used to encrypt pairs with components of respective types T_1 and T_2. Similarly, $\mathrm{K}^{\mathrm{Public}}[T_1, T_2]$ is the type of a public encryption key. Analogous types apply to channels.

Substantial difficulties arise because, in the study of security protocols, we cannot assume that the adversary politely obeys our typing discipline [1,27, 9]. Although honest protocol participants may use public channels and public keys according to declared types, the adversary may be an untyped process and disregard those types. Nevertheless, the declared types remain useful when combined with the static and dynamic checks mentioned above.

In this respect, asymmetric communication primitives (channels with fixed receivers, public-key encryption) are more delicate and interesting than their symmetric counterparts (shared channels as in the pi calculus, shared-key encryption). A shared channel or shared key that the adversary knows should never be employed for transmitting secrets, only public data. Therefore, types like $\mathrm{K}^{\mathrm{Public}}[\mathrm{Secret}]$ are useless in this setting; a public shared key may simply be given the type Public. On the other hand, a channel with a fixed receiver may be employed for transmitting secrets, even if the adversary knows it (as long as the adversary is not the receiver). Similarly, an encryption key in a public-key cryptosystem may be employed for encrypting secrets, even if the adversary knows it (as long as the adversary does not know the corresponding decryption key). Therefore, types like $\mathrm{K}^{\mathrm{Public}}[\mathrm{Secret}]$ give useful information. For instance, although the adversary may encrypt public data under a key of type $\mathrm{K}^{\mathrm{Public}}[\mathrm{Secret}]$, this type can tell others that encrypting secrets under the key is acceptable too, that these secrets will not escape. The typing of asymmetric communication primitives in the context of an untyped adversary is the main novelty of this work.

Secrecy results. We prove a subject-reduction property, namely, that typing is preserved by computation. Relying on this property, we also prove a secrecy theorem that shows that well-typed processes do not reveal their secrets. As an example, let us consider a well-typed process P with just two free names, a of type Public and k of type $\mathrm{K}^{\mathrm{Secret}}[T_1, T_2]$. As an adversary, we allow any process Q with the free name a (and possibly other free names, except for k). The secrecy theorem implies that the parallel composition of P with an adversary Q never outputs k on a.

This kind of secrecy guarantee is common and useful in the analysis of security protocols. It is particularly adequate and effective for dealing with the secrecy of fresh values that can be viewed as atomic, such as keys and nonces. In contrast, secrecy guarantees based on the concept of noninterference are better at excluding flows of partial information about compound values and implicit information flows. (See [2, section 6] for some further discussion and references.) Most type systems for secrecy concern noninterference; a recent exception is that of Cardelli, Ghelli, and Gordon [9].

Application to protocols. Our type system can be applied to some small but subtle security protocols. For example, in the Needham-Schroeder public-key protocol [24] (a standard test case), one might expect a certain nonce to be secret; however, the protocol fails to typecheck under the assumption that this nonce is secret. This failure is not a shortcoming of the type system: it manifests Lowe's attack on the protocol [17]. On the other hand, a corrected version of the protocol does typecheck under the assumption. Our secrecy theorem yields the expected secrecy property in this case.

A variety of other techniques have been applied to this sort of protocol analysis. They include theorem proving, model checking, and control-flow analysis

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$\{M_1, \ldots, M_n\}_M$	encryption
$P, Q ::=$	processes
$\overline{M}\langle M_1, \ldots, M_n\rangle$	output
$a(x_1, \ldots, x_n).P$	input
0	nil
$P \mid Q$	parallel composition
$!P$	replication
$(\nu a)P$	restriction
case M *of* $\{x_1, \ldots, x_n\}_k : P$ *else* Q	decryption
if $M = N$ *then* P *else* Q	conditional

Fig. 1. Syntax of the process calculus

methods (e.g., [25,20,17,8,7]). Type-based analyses, such as ours, are in part appealing because of their simplicity and because of their connections to classical information-flow methods. They do not require exhaustive state-space exploration; they take advantage of the structure of protocols. On the other hand, they are not always applicable: some reasonable examples fail to typecheck for trivial reasons (as in most typed programming languages and logics). Typing is a discipline, and it works best for processes that use channels and keys in disciplined ways, but disciplined design can lead to more robust protocols [4,5, 1].

Outline. The next section defines the process calculus. Section 3 defines a concept of secrecy. Section 4 presents the type system. Section 5 establishes the secrecy theorem and other results. Section 6 develops an example. Section 7 concludes. Because of space constraints, we leave for an extended version of this paper: some auxiliary rules, the study of the Needham-Schroeder public-key protocol, (negative) observations on "best" typings, the treatment of symmetric communication primitives, and details of proofs.

2 The Process Calculus (Untyped)

This section introduces our process calculus, by giving its syntax and informal semantics and then formalizing its operational semantics.

The syntax of our calculus is summarized in Figure 1. It assumes an infinite set of names and an infinite set of variables; a, b, c, k, s, and similar identifiers range over names, and x, y, and z range over variables. For simplicity, we do not formally separate the names for channels and those for keys. The syntax distinguishes a category of terms (data) and processes (programs). The terms are variables, names, and terms of the form $\{M_1, \ldots M_n\}_M$, which represent

encryptions. The processes include constructs for communication, concurrency, and dynamic name creation, roughly those of the pi calculus, a construct for decryption (*case* M *of* $\{x_1, \ldots, x_n\}_k : P$ *else* Q), and a conditional (*if* $M = N$ *then* P *else* Q). As usual, we may omit an "*else*" clause when it consists of the nil process 0.

The calculus is polyadic, in the sense that messages are tuples of terms, and asynchronous, in the sense that the output construct does not have a built-in acknowledgment. It is also local, as explained below. Therefore, the calculus could be called the local, asynchronous, polyadic spi calculus (but we refrain from such a jargon overload).

The calculus is based on asymmetric communication: channels with only one fixed end-point (the receiver) and public-key encryption. We adopt an elegant, economical approach to asymmetric communication from the local pi calculus [19]. In the local pi calculus, input is possible only on channels that are syntactically represented by names (and not variables). Output is possible on channels represented by names or variables. Thus, the input capability for a channel a remains within the scope of the restriction $(\nu a)P$ where a is created, while the output capability can be transmitted outside. Further, we extend this approach to public-key encryption, as follows. Decryption is possible only with keys that are syntactically represented by names (and not variables). Encryption is possible with keys represented by names or variables. Thus, we model that the encryption capability may be public while the decryption capability remains private, in the scope where it is generated. (We do not explicitly model the distribution of decryption keys across scopes; it is relatively unimportant in public-key cryptosystems.)

Thus, when a name a refers to a channel, it represents both end-points of the channel, that is, the capabilities for output and input on the channel. A variable can confer only the former capability, even if its value is a at run-time. Similarly, a name k will not represent a single encryption or decryption key, but rather the pair of an encryption key and a corresponding decryption key. A variable can confer only the capability of encrypting, even if its value is k at run-time.

Specifically, the constructs for asymmetric communication are output, input, encryption, and decryption:

- The process $\overline{M}\langle M_1, \ldots, M_n\rangle$ outputs the tuple $M_1, \ldots, M_n$ on M. Here an arbitrary term M is used to refer to a channel: M can be a variable, a name, or even an encryption. This last case is however unimportant for the present purposes; when M is an encryption, no process can receive the message $\overline{M}\langle M_1, \ldots, M_n\rangle$.
- The process $a(x_1, \ldots, x_n).P$ inputs a message with n components on channel a, then executes P with the variables $x_1, \ldots, x_n$ bound to the contents of the message. Note that a must be a name.
- The term $\{M_1, \ldots, M_n\}_M$ represents an encryption of the tuple $M_1, \ldots, M_n$ under M. Here an arbitrary term M is used to refer to an encryption key: M can be a variable, a name, or even an encryption. This last case

is however unimportant, again; when M is an encryption, no process can decrypt $\{M_1, \ldots, M_n\}_M$.
- In the process *case* M *of* $\{x_1, \ldots, x_n\}_k : P$ *else* Q, the term M is decrypted with the key k. If M is indeed a ciphertext encrypted under k, and the underlying plaintext has n components, then the process P is executed with the variables $x_1, \ldots, x_n$ bound to those components. Otherwise, the process Q is executed. Note that k must be a name.

The remaining constructs are standard. The nil process 0 does nothing. The process $P \mid Q$ is the parallel composition of P and Q. The replication $!P$ represents any number of copies of the process P in parallel. The restriction $(\nu a)P$ creates a new name a, and then executes P. The conditional *if* $M = N$ *then* P *else* Q executes P if M and N evaluate to the same closed term; otherwise, it executes Q. This construct is not always present in relatives of the pi calculus, but it is helpful in modeling security protocols.

For example, the following expression is a process:

$$(\nu k)(\overline{a}\langle k\rangle \mid !b(x).\mathit{case}\ x\ \mathit{of}\ \{y\}_k : \overline{c}\langle y\rangle)$$

This process relies on three public channels, a, b, and c. It generates a fresh key pair k; outputs the corresponding encryption key on a; and receives messages on b, filtering for one encrypted under k, of which it outputs the plaintext on c.

The name a is bound in $(\nu a)P$. The variables $x_1, \ldots, x_n$ are bound in P in the processes $a(x_1, \ldots, x_n).P$ and *case* M *of* $\{x_1, \ldots, x_n\}_k : P$ *else* Q. We write $fn(P)$ and $fv(P)$ for the sets of names and variables free in P, respectively. A process is closed if it has no free variables; it may have free names. We identify processes up to renaming of bound names and variables. We write $\{M_1/x_1, \ldots, M_n/x_n\}$ for the substitution that replaces $x_1, \ldots, x_n$ with $M_1, \ldots, M_n$, respectively.

The rules of Figure 2 axiomatize the reduction relation $\rightarrow$ for processes; they are a formal definition for the operational semantics of our calculus. Auxiliary rules axiomatize the structural congruence relation $\equiv$; this relation is useful for transforming processes so that the reduction rules can be applied. Both $\equiv$ and $\rightarrow$ are defined only on closed processes. (In particular, we do not include rules for structural congruence under an input, a decryption, or a conditional; such rules are not necessary to apply reduction rules.) All rules are fairly standard.

Using this operational semantics, we can give a precise definition of a simple concept of output, which we use below in a definition of secrecy. Here, $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$.

Definition 1. *The process P outputs M immediately on c if and only if $P \equiv \overline{c}\langle M\rangle \mid R$ for some process R. The process P outputs M on c if and only if $P \rightarrow^* Q$ and Q outputs M immediately on c for some process Q.*

3 A Definition of Secrecy

We think of an attacker as any process Q of the calculus, under some restrictions that characterize the attacker's initial capabilities. We formulate the restrictions

Structural congruence:

$$P \mid 0 \equiv P \qquad P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$$
$$P \mid Q \equiv Q \mid P \qquad P \equiv Q \Rightarrow\ !P \equiv\ !Q$$
$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \equiv Q \Rightarrow (\nu a)P \equiv (\nu a)Q$$
$$!P \equiv P \mid\ !P$$
$$P \equiv P$$
$$(\nu a_1)(\nu a_2)P \equiv (\nu a_2)(\nu a_1)P \text{ if } a_1 \neq a_2 \qquad Q \equiv P \Rightarrow P \equiv Q$$
$$(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q \text{ if } a \notin fn(P) \qquad P \equiv Q,\ Q \equiv R \Rightarrow P \equiv R$$

Reduction:

$$\overline{a}\langle M_1, \ldots, M_n\rangle \mid a(x_1, \ldots, x_n).P \rightarrow P\{M_1/x_1, \ldots, M_n/x_n\} \quad \text{(Red I/O)}$$

$$\textit{case } \{M_1, \ldots, M_n\}_k \textit{ of } \{x_1, \ldots, x_n\}_k : P \textit{ else } Q \rightarrow P\{M_1/x_1, \ldots, M_n/x_n\} \quad \text{(Red Decrypt 1)}$$

$$\textit{case } M \textit{ of } \{x_1, \ldots, x_n\}_k : P \textit{ else } Q \rightarrow Q \text{ if } M \text{ is not of the form } \{M_1, \ldots, M_n\}_k \quad \text{(Red Decrypt 2)}$$

$$\textit{if } M = M \textit{ then } P \textit{ else } Q \rightarrow P \quad \text{(Red Cond 1)}$$

$$\textit{if } M = N \textit{ then } P \textit{ else } Q \rightarrow Q \text{ if } M \neq N \quad \text{(Red Cond 2)}$$

$$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R \quad \text{(Red Par)}$$

$$P \rightarrow Q \Rightarrow (\nu a)P \rightarrow (\nu a)Q \quad \text{(Red Res)}$$

$$P' \equiv P,\ P \rightarrow Q,\ Q \equiv Q' \Rightarrow P' \rightarrow Q' \quad \text{(Red } \equiv\text{)}$$

Fig. 2. Operational semantics

by using a set of names RW ("read-write") and a set of closed terms W ("write"). Initially, the attacker is able to output, input, encrypt, and decrypt using the names of RW. He can output and encrypt using names in W. He has the terms in RW and W, and can compute on them and include them in messages. In the course of computation, the attacker may acquire capabilities not represented in RW and W, by creating fresh names and receiving terms in messages.

In order to express the limited use of the terms in W, we also introduce a set of variables $\{x_1, \ldots, x_l\}$ of the same cardinality l as W. When $W = \{M_1, \ldots, M_l\}$, the attacker is a process Q of the form $Q'\{M_1/x_1, \ldots, M_l/x_l\}$. Since Q' should be a well-formed process before the application of the substitution $\{M_1/x_1, \ldots, M_l/x_l\}$, it cannot input or decrypt using the variables $\{x_1, \ldots, x_l\}$. Further, in order to express that the attacker cannot initially use any other names or terms, we impose that $fn(Q') \subseteq RW$ and $fv(Q') \subseteq \{x_1, \ldots, x_l\}$.

Definition 2. *Let RW be a finite set of names and let $W = \{M_1, \ldots, M_l\}$ be a finite set of closed terms. The process Q is a (RW, W)-adversary if and only if it is of the form $Q'\{M_1/x_1, \ldots, M_l/x_l\}$ for some process Q' such that $fn(Q') \subseteq RW$ and $fv(Q') \subseteq \{x_1, \ldots, x_l\}$.*

We say that a process preserves the secrecy of a piece of data M if the process never publishes M, or anything that would permit the computation

of M, even in interaction with an attacker. This concept of secrecy is common in the literature on security protocols. A precise definition of it appears in [2], for the spi calculus; Cardelli, Ghelli, and Gordon use that definition in their work on secrecy and groups in the pi calculus [9]. Here we introduce and use a different definition that captures the same concept. This definition takes into account asymmetric communication; it is also more syntactic, and a little easier to treat in our proofs.

Definition 3. *Let RW be a finite set of names and let W be a finite set of closed terms. The process P preserves the secrecy of M from (RW, W) if and only if $P \mid Q$ does not output M on c for any (RW, W)-adversary Q and $c \in RW$.*

Clearly, if P preserves the secrecy of M from (RW, W), it cannot output M on some $c \in RW$, that is, on one of the channels on which an (RW, W)-adversary can read. This guarantee corresponds to the informal requirement that P never publishes M on its own. Moreover, P cannot publish data that would enable an adversary to compute M, because the adversary could go on to output M on some $c \in RW$.

For instance, $(\nu k)\overline{a}\langle\{s\}_k, k\rangle$ preserves the secrecy of s from $(\{a\}, \emptyset)$. This process publishes an encryption of s and the corresponding encryption key on the channel a, but keeps the decryption key, so s does not escape. Similarly, $\overline{a}\langle\{s\}_k, k\rangle$ preserves the secrecy of s from $(\{a\}, \{k\})$. However, $\overline{a}\langle\{s\}_k, k\rangle$ does not preserve the secrecy of s from $(\{a, k\}, \emptyset)$: the adversary $a(x, y).\mathit{case}\ x\ \mathit{of}\ \{z\}_k : \overline{a}\langle z\rangle$ can receive $\{s\}_k$ on a, decrypt s, and resend it on a. As a more substantial, untyped example we consider the following process P:

$$P \triangleq (\nu k)(\overline{a}\langle\{k_1, k_2\}_k\rangle \mid !b(x).\mathit{case}\ x\ \mathit{of}\ \{y_1, y_2\}_k : \overline{c}\langle y_1\rangle)$$

This process relies on three public channels, a, b, and c. It generates a fresh key pair k; outputs the encryption $\{k_1, k_2\}_k$ on a; and receives messages on b, filtering for one encrypted under k, of which it outputs the first component of the plaintext on c. Although P does not output k_1 in clear on its own, it does not preserve the secrecy of k_1 from $(\{a, c\}, \{b\})$: the adversary $a(x).\overline{b}\langle x\rangle$ can receive $\{k_1, k_2\}_k$ on a and resend it on b, causing k_1 to appear on c. On the other hand, P does preserve the secrecy of k_2 from $(\{a, c\}, \{b\})$.

4 The Type System

The types of our type system are defined by the grammar:

$T ::=$	types
Public	
Secret	
$\mathrm{C}^{\mathrm{Public}}[T_1, \ldots, T_n]$	
$\mathrm{C}^{\mathrm{Secret}}[T_1, \ldots, T_n]$	
$\mathrm{K}^{\mathrm{Secret}}[T_1, \ldots, T_n]$	
$\mathrm{K}^{\mathrm{Public}}[T_1, \ldots, T_n]$	

We let L range over $\{\text{Public}, \text{Secret}\}$, and may for example write $\mathrm{C}^L[T_1, \ldots, T_n]$ and $\mathrm{K}^L[T_1, \ldots, T_n]$. The subtyping relation is the least reflexive relation such that $\mathrm{C}^L[T_1, \ldots, T_n] \leq L$ and $\mathrm{K}^L[T_1, \ldots, T_n] \leq L$. (We do not have Secret $\leq$ Public or vice versa.)

The types have the following informal meanings:

- Public is the type of public data, and is a supertype of all types $\mathrm{C}^{\text{Public}}[T_1, \ldots, T_n]$ and $\mathrm{K}^{\text{Public}}[T_1, \ldots, T_n]$.
- Similarly, Secret is the type of secret data, and is a supertype of all types $\mathrm{C}^{\text{Secret}}[T_1, \ldots, T_n]$ and $\mathrm{K}^{\text{Secret}}[T_1, \ldots, T_n]$.
- $\mathrm{C}^{\text{Secret}}[T_1, \ldots, T_n]$ is the type of a channel on which the adversary cannot send messages, and which conveys n-tuples with components of respective types $T_1, \ldots, T_n$.
- Similarly, $\mathrm{K}^{\text{Secret}}[T_1, \ldots, T_n]$ is the type of an encryption key that the adversary does not have, and which is used to encrypt n-tuples with components of respective types $T_1, \ldots, T_n$.
- $\mathrm{C}^{\text{Public}}[T_1, \ldots, T_n]$ is the type of a channel on which the adversary may send messages; the channel may be intended to convey n-tuples with components of respective types $T_1, \ldots, T_n$, but the adversary may send any data it has (that is, any public data) on the channel.
- Similarly, $\mathrm{K}^{\text{Public}}[T_1, \ldots, T_n]$ is the type of an encryption key that the adversary may have; this key may be intended for encrypting n-tuples with components of respective types $T_1, \ldots, T_n$, but the adversary may encrypt any data it has (that is, any public data) under this key.

Figure 3 gives the main rules of the type system. In the rules, the metavariable u ranges over names and variables. The rules concern four judgments:

(1) $E \vdash \diamond$ means that E is a well-formed environment. The environment E is well-formed if and only if E is a list of pairs $u : T$ where each u is a name or a variable and distinct from all others in E.
(2) $E \vdash M : T$ means that M is a term of type T in the environment E. Basically, names and variables have the types declared in E, and any supertypes, while encryptions all have the type Public.
(3) $E \vdash_{\diamond} M : S$ means that S is the set of possible "true" types of M in the environment E (the declared type when M is a name, the declared type and any subtype when M is a variable, and Public when M is an encryption).
(4) $E \vdash P$ says that the process P is well-typed in the environment E.

Figure 3 omits the rules for proving the first and the third.

The type rules for output say that any public data can be sent on a public channel (Output Public), and tuples with the expected types $T_1, \ldots, T_n$ can be sent on a channel of type $\mathrm{C}^L[T_1, \ldots, T_n]$ (Output C^L). Therefore, by subtyping, any public data can be sent on a channel of type $\mathrm{C}^{\text{Public}}[T_1, \ldots, T_n]$. This use of the channel may not seem to conform to its declared type. However, it is unavoidable, since we expect that an attacker can use the channel; moreover, it does not cause harm from the point of view of secrecy. Similarly, the type rule

$$\frac{E \vdash \diamond \qquad (u : T) \in E}{E \vdash u : T} \quad \text{(Atom)}$$

$$\frac{E \vdash M : \text{Public} \qquad \forall i \in \{1, \ldots, n\}, E \vdash M_i : \text{Public}}{E \vdash \{M_1, \ldots, M_n\}_M : \text{Public}} \quad \text{(Encrypt Public)}$$

$$\frac{E \vdash M : \mathrm{K}^L[T_1, \ldots, T_n] \qquad \forall i \in \{1, \ldots, n\}, E \vdash M_i : T_i}{E \vdash \{M_1, \ldots, M_n\}_M : \text{Public}} \quad \text{(Encrypt } \mathrm{K}^L\text{)}$$

$$\frac{E \vdash M : T \qquad T \leq T'}{E \vdash M : T'} \quad \text{(Subsumption)}$$

$$\frac{E \vdash M : \text{Public} \qquad \forall i \in \{1, \ldots, n\}, E \vdash M_i : \text{Public}}{E \vdash \overline{M}\langle M_1, \ldots, M_n\rangle} \quad \text{(Output Public)}$$

$$\frac{E \vdash M : \mathrm{C}^L[T_1, \ldots, T_n] \qquad \forall i \in \{1, \ldots, n\}, E \vdash M_i : T_i}{E \vdash \overline{M}\langle M_1, \ldots, M_n\rangle} \quad \text{(Output } \mathrm{C}^L\text{)}$$

$$\frac{(a : \text{Public}) \in E \qquad E, x_1 : \text{Public}, \ldots, x_n : \text{Public} \vdash P}{E \vdash a(x_1, \ldots, x_n).P} \quad \text{(Input Public)}$$

$$\frac{(a : \mathrm{C}^{\text{Public}}[T_1, \ldots, T_m]) \in E \qquad E, x_1 : \text{Public}, \ldots, x_n : \text{Public} \vdash P \qquad E, x_1 : T_1, \ldots, x_m : T_m \vdash P \ \text{ if } m = n}{E \vdash a(x_1, \ldots, x_n).P} \quad \text{(Input } \mathrm{C}^{\text{Public}}\text{)}$$

$$\frac{(a : \mathrm{C}^{\text{Secret}}[T_1, \ldots, T_n]) \in E \qquad E, x_1 : T_1, \ldots, x_n : T_n \vdash P}{E \vdash a(x_1, \ldots, x_n).P} \quad \text{(Input } \mathrm{C}^{\text{Secret}}\text{)}$$

$$\frac{E \vdash \diamond}{E \vdash 0} \quad \text{(Nil)}$$

$$\frac{E \vdash P \qquad E \vdash Q}{E \vdash P \mid Q} \quad \text{(Parallel)}$$

$$\frac{E \vdash P}{E \vdash !P} \quad \text{(Replication)}$$

$$\frac{E, a : T \vdash P}{E \vdash (\nu a)P} \quad \text{(Restriction)}$$

$$\frac{E \vdash M : \text{Public} \qquad (k : \text{Public}) \in E \qquad E, x_1 : \text{Public}, \ldots, x_n : \text{Public} \vdash P \qquad E \vdash Q}{E \vdash \mathit{case}\ M\ \mathit{of}\ \{x_1, \ldots, x_n\}_k : P\ \mathit{else}\ Q} \quad \text{(Decrypt Public)}$$

$$\frac{E \vdash M : \text{Public} \qquad (k : \mathrm{K}^{\text{Public}}[T_1, \ldots, T_m]) \in E \qquad E, x_1 : \text{Public}, \ldots, x_n : \text{Public} \vdash P \qquad E \vdash Q \qquad E, x_1 : T_1, \ldots, x_m : T_m \vdash P \ \text{ if } m = n}{E \vdash \mathit{case}\ M\ \mathit{of}\ \{x_1, \ldots, x_n\}_k : P\ \mathit{else}\ Q} \quad \text{(Decrypt } \mathrm{K}^{\text{Public}}\text{)}$$

$$\frac{E \vdash M : \text{Public} \qquad (k : \mathrm{K}^{\text{Secret}}[T_1, \ldots, T_n]) \in E \qquad E, x_1 : T_1, \ldots, x_n : T_n \vdash P \qquad E \vdash Q}{E \vdash \mathit{case}\ M\ \mathit{of}\ \{x_1, \ldots, x_n\}_k : P\ \mathit{else}\ Q} \quad \text{(Decrypt } \mathrm{K}^{\text{Secret}}\text{)}$$

$$\frac{E \vdash_\diamond M : S_1 \qquad E \vdash_\diamond N : S_2 \qquad \text{if } S_1 \cap S_2 \neq \emptyset \text{ then } E \vdash P \qquad E \vdash Q}{E \vdash \mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q} \quad \text{(Cond)}$$

Fig. 3. Type rules: terms and processes

(Output Public) also permits processes that use an encryption as a channel, such as $\overline{\{M\}_k}\langle N\rangle$. A standard type system might attempt to exclude such processes. Ours does not, essentially because an attacker might run $\overline{\{M\}_k}\langle N\rangle$, but this does not cause harm from the point of view of secrecy. On the other hand, the attacker cannot have channels of type $\mathrm{C}^{\mathrm{Secret}}[T_1, \ldots, T_n]$. Therefore, we can guarantee that such channels are represented by names at run-time and that only tuples with types $T_1, \ldots, T_n$ can be sent on such channels.

As for input, we distinguish three cases, considering the type of the channel a on which an input happens:

- If a is of type Public, then the corresponding output must have been typed using (Output Public), so the input values are public. Rule (Input Public) treats this case. This rule may seem superfluously liberal, but it is helpful in our proofs. It enables us to type an arbitrary adversary, which can input public values on the public channels on which it listens (see section 5).
- When a is of type $\mathrm{C}^{\mathrm{Public}}[T_1, \ldots, T_n]$, two cases arise. In the first case, the corresponding output has been typed using (Output Public) and subtyping. Then the input values are of type Public. In the second case, the corresponding output has been typed using (Output C^L). In this case, the input values have the expected types $T_1, \ldots, T_n$. Rule (Input $\mathrm{C}^{\mathrm{Public}}$) takes into account both cases, by checking that the process P executed after the input is well-typed in both.
- When a is of type $\mathrm{C}^{\mathrm{Secret}}[T_1, \ldots, T_n]$, it cannot be known by the attacker, and the corresponding output must have been typed using (Output C^L). The input values are therefore of the expected types $T_1, \ldots, T_n$.

The type rules for encryption are similar to those for output. Any public data can be encrypted under a public encryption key (Encrypt Public), and data of types $T_1, \ldots, T_n$ can be encrypted under a key of type $\mathrm{K}^L[T_1, \ldots, T_n]$ (Encrypt K^L). Ciphertexts are always of type Public; this typing simplifies the rules and is reasonable for most protocols (particularly for most public-key protocols).

The type rules for decryption resemble those for input, in the same way as those for encryption resemble those for output.

The type rules for nil, parallel composition, replication, and restriction are standard. It is worth noting that we use a Curry-style typing for restriction, so we do not mention a type of a explicitly in the construct (νa). (That is, we do not write $(\nu a : T)$ for some T.) This style of typing gives rise to an interesting form of polymorphism: the type of a can change according to the environment. For instance, in $c(x).(\nu a)\overline{x}\langle a\rangle$, with c of type $\mathrm{C}^{\mathrm{Public}}[\mathrm{C}^{\mathrm{Secret}}[\mathrm{Secret}]]$, a can be of type Secret when x is of type $\mathrm{C}^{\mathrm{Secret}}[\mathrm{Secret}]$, and of type Public when x is of type Public, so that the output $\overline{x}\langle a\rangle$ is well-typed in both cases.

Rule (Cond) exploits the idea that if two terms M and N cannot have the same type, then they are certainly different. In this case, *if* $M = N$ *then* P *else* Q may be well-typed without P being well-typed. To determine whether M and N may have the same type, we determine the set of possible types of M and N. If M is a variable x, and $(x : T) \in E$, then x may of course have type T. Because of subtyping, x may also be substituted at run-time by a name whose type is

a subtype of T. Hence the possible types of x are $\{T' \mid T' \leq T\}$. When M is a name a, its only possible type is the type assigned to it in the environment. When M is a ciphertext, its only possible type is Public, by definition of the judgment $E \vdash M : T$.

The following example illustrates the use of rule (Cond), informally. Suppose that a participant A in a security protocol invents a fresh quantity a of type Secret (as a nonce challenge), sends it on a channel of type $\mathrm{C}^{\mathrm{Secret}}[\mathrm{Secret}]$, so a should remain secret. Some time later, A gets a ciphertext encrypted under a key k of type $\mathrm{K}^{\mathrm{Public}}[\mathrm{Secret}, T]$; decryption yields a pair x_1, x_2. The type system covers two possibilities: (1) x_1 has type Secret and x_2 has type T, (2) both are public (for example, because the attacker constructed the ciphertext). That is, the type rule for decryption (Decrypt $\mathrm{K}^{\mathrm{Public}}$) has hypotheses that correspond to each of these possibilities. Suppose further that A checks, dynamically, that $x_1 = a$ (and halts if $x_1 \neq a$). This check guarantees that x_1 has type Secret, and hence is not public. At the same time, it guarantees that x_2 has type T. After the check, A can assume that x_2 has type T, and act accordingly. Dynamic checks of this sort are common and important in security protocols.

This type system reflects a binary view of secrecy, according to which the world is divided into system and attacker, and a secret is something that the attacker does not have. When we wish to express that a piece of data is a secret for a given set of principals, we define the system to include only the processes that represent those principals.

In this respect and in others, our type system is most similar to that of Abadi [1] for the spi calculus and that of Cardelli, Ghelli, and Gordon [9, section 4] for the pi calculus. Both treat only symmetric communication primitives. The latter, however, is mostly introduced as an auxiliary type system for a proof. The proof concerns another type system, which elegantly exploits a powerful construct for group creation. Group creation directly supports a rich view of secrecy that does not simply divide the world into two parts. We believe that the type system with group creation can be extended with symmetric cryptographic primitives and further extended to deal with asymmetric communication. Unfortunately (as far as we can tell) this last extension does not retain the elegance of the original.

5 The Secrecy Theorem and Other Results

This section studies the type system of section 4. First it establishes a subject-reduction theorem and a typability lemma. Then it derives the secrecy theorem sketched in the introduction.

The subject-reduction theorem says that typing is preserved by computation. Its proof is mostly a fairly routine induction on computations with a case analysis on typing proofs.

Theorem 1 (Subject congruence and subject reduction). *If $E \vdash P$ and if $P \equiv Q$ or $P \to Q$ then $E \vdash Q$.*

The following typability lemma says that every process is well-typed, at least in a fairly trivial way that makes its free names public. This lemma is important because it means that any process that represents an adversary is well-typed. It is a formal counterpart to the informal idea that the type system cannot constrain the adversary. Its proof is an easy induction on the structure of P.

Lemma 1 (Typability). *Let P be an untyped process. If $fn(P) \subseteq \{a_1, \ldots, a_n\}$, $fv(P) \subseteq \{x_1, \ldots, x_m\}$, and $T_i \leq Public$ for all $i \in \{1, \ldots, m\}$, then*

$$a_1 : Public, \ldots, a_n : Public, x_1 : T_1, \ldots, x_m : T_m \vdash P$$

The secrecy theorem says that if a closed process P is well-typed in an environment E, and a name s has type Secret in E, then P preserves the secrecy of s from (RW, W), where RW is the set of names declared Public in E, and W is the set of names declared $\mathrm{C}^{\mathrm{Public}}[\ldots]$ or $\mathrm{K}^{\mathrm{Public}}[\ldots]$. The name s may be declared Secret in E, but it may also be declared $\mathrm{C}^{\mathrm{Secret}}[\ldots]$ or $\mathrm{K}^{\mathrm{Secret}}[\ldots]$. In other words, P preserves the secrecy of Secret names against adversaries that can output, input, encrypt, and decrypt on names declared Public, and output and encrypt on names declared $\mathrm{C}^{\mathrm{Public}}[\ldots]$ and $\mathrm{K}^{\mathrm{Public}}[\ldots]$.

Theorem 2 (Secrecy). *Let P be a closed process. Suppose that $E \vdash P$ and $E \vdash s : Secret$. Let*

$$RW = \{a \mid (a : Public) \in E\}$$
$$W = \{a' \mid (a' : C^{Public}[\ldots]) \in E \text{ or } (a' : K^{Public}[\ldots]) \in E\}$$

Then P preserves the secrecy of s from (RW, W).

Proof. The secrecy theorem is a fairly easy consequence of the subject-reduction theorem and the typability lemma. (In truth, the type system and the definition of secrecy were refined with this proof of the secrecy theorem in mind.) Suppose that $RW = \{a_1, ..., a_n\}$ and $W = \{a'_1, ..., a'_l\}$, and that T'_i is the type of a'_i in E, so $(a'_i{:}T'_i) \in E$ for all $i \in \{1, \ldots, l\}$. In order to derive a contradiction, we assume that P does not preserve the secrecy of s from (RW, W). Then there exists a process $Q = Q'\{a'_1/x_1, \ldots, a'_l/x_l\}$ with $fn(Q') \subseteq RW$ and $fv(Q') \subseteq \{x_1, \ldots, x_l\}$, such that $P \mid Q \rightarrow^* R$ and $R \equiv \overline{c}\langle s\rangle \mid R'$, where $c \in RW$. By Lemma 1, $a_1 : \mathrm{Public}, \ldots, a_n : \mathrm{Public}, x_1 : T'_1, \ldots, x_l : T'_l \vdash Q'$. By a standard substitution lemma, $E \vdash Q'\{a'_1/x_1, \ldots, a'_l/x_l\}$, that is, $E \vdash Q$. Therefore, $E \vdash P \mid Q$. By Theorem 1, $E \vdash R$ and $E \vdash \overline{c}\langle s\rangle \mid R'$. Since $c \in RW$, we have $(c : \mathrm{Public}) \in E$, so $E \vdash \overline{c}\langle s\rangle$ could have been derived only by (Output Public). Such a derivation would require that $E \vdash s : \mathrm{Public}$. However, this is impossible, since we have $E \vdash s : \mathrm{Secret}$ and the two typings are incompatible. We have a contradiction, so P preserves the secrecy of s from (RW, W). □

We restate a special case of the theorem, as it may be particularly clear.

Corollary 1. *Suppose that $a : Public, s : T \vdash P$ with $T \leq Secret$. Then P preserves the secrecy of s from $(\{a\}, \emptyset)$. That is, for all closed processes Q such that $fn(Q) \subseteq \{a\}$, $P \mid Q$ does not output s on a.*

For instance, we can obtain $a : \text{Public}, s : \text{Secret} \vdash (\nu k)\overline{a}\langle\{s\}_k, k\rangle$ by letting $k : \text{K}^{\text{Public}}[\text{Secret}]$. So this corollary implies that $(\nu k)\overline{a}\langle\{s\}_k, k\rangle$ preserves the secrecy of s from $(\{a\}, \emptyset)$, as claimed in section 3. In other words, if Q is a closed process and $fn(Q) \subseteq \{a\}$, then $((\nu k)\overline{a}\langle\{s\}_k, k\rangle) \mid Q$ does not output s on a. Thus, assuming that Q does not have s in advance, Q cannot guess s or compute it from the message on a.

6 Further Examples

This section applies the type system to security protocols. Although these protocols are often fairly small, informal reasoning about them is error-prone and difficult. The type system provides a simple yet rigorous approach for proving secrecy properties of these protocols.

Because of space constraints, we develop only one example. This example does not explicitly rely on cryptography but it brings up some issues common in cryptographic protocols. An extended version of this paper also covers the Needham-Schroeder public-key protocol and a variant (in 3–4 pages).

Our example concerns a protocol in which a principal A sends a secret s to a principal B. The following message sequence describes the protocol informally:

$$\begin{array}{l}\text{Message 1. } A \to B : k, a \text{ on } b\\ \text{Message 2. } B \to A : k, b' \text{ on } a\\ \text{Message 3. } A \to B : s \text{ on } b'\end{array}$$

Here, a and b are channels with A and B as only receivers, respectively; k is a secret nonce, created by A; and b' is a new channel, created by B, with B as only receiver and A as only sender. Instead of sending s directly on b, A creates the nonce k and sends it along with the return channel a on b; B's reply contains k, as proof of origin; the reply also includes the fresh channel b' on which A sends s. This channel is analogous to a session key in a cryptographic protocol.

We may represent the principals of this protocol by the processes:

$$\begin{array}{l}A \triangleq (\nu k)(\overline{b}\langle k, a\rangle \mid a(x, y).\mathit{if}\ x = k\ \mathit{then}\ \overline{y}\langle s\rangle)\\ B \triangleq b(x, y).(\nu b')(\overline{y}\langle x, b'\rangle \mid b'(z).0)\end{array}$$

We can then use our type system to prove that s remains secret, as expected. For this proof, we let:

$$\begin{array}{l}E \triangleq a : \text{C}^{\text{Public}}[\text{Secret}, \text{C}^{\text{Secret}}[\text{Secret}]],\\ \qquad b : \text{C}^{\text{Public}}[\text{Secret}, \text{C}^{\text{Public}}[\text{Secret}, \text{C}^{\text{Secret}}[\text{Secret}]]],\\ \qquad s : \text{Secret}\end{array}$$

and obtain $E, c : \text{Public} \vdash A \mid B$ for any c, as follows. In the typing of A, we choose $k : \text{Secret}$. The output $\overline{b}\langle k, a\rangle$ is then typed by (Output C^L). The input $a(x, y)$ is typed by (Input C^{Public}), and two cases arise:

(1) $x : \text{Public}, y : \text{Public}$: This case is vacuous by rule (Cond): in the test $x = k$, the two terms cannot have the same type.
(2) $x : \text{Secret}, y : \text{C}^{\text{Secret}}[\text{Secret}]$: In this case, $\overline{y}\langle s \rangle$ is typed by (Output C^{L}).

The input $b(x, y)$ is also typed by (Input C^{Public}), and again two cases arise:

(1) $x : \text{Public}, y : \text{Public}$: In this case, we let $b' : \text{Public}$.
(2) $x : \text{Secret}, y : \text{C}^{\text{Public}}[\text{Secret}, \text{C}^{\text{Secret}}[\text{Secret}]]$: In this case, instead, we let $b' : \text{C}^{\text{Secret}}[\text{Secret}]$.

In both cases, the rest of process B is easy to typecheck. Having derived $E, c : \text{Public} \vdash A \mid B$, we apply Theorem 2, and conclude that $A \mid B$ preserves the secrecy of s from $(\{c\}, \{a, b\})$.

We can also treat a more general system in which A and B communicate with other principals: A may initiate sessions with others than B, and B is willing to respond to several principals at once. Still, s should remain within $A \mid B$, and not escape to third parties. The following process represents the system:

$$P \triangleq (A \mid A') \mid !B$$

Here A' is an arbitrary process, notionally grouped with A, which may receive messages on a, under the assumption that $E, E' \vdash A'$ for some E'. In particular, this assumption implies that A' respects the secrecy of s and uses a in conformance with $a : \text{C}^{\text{Public}}[\text{Secret}, \text{C}^{\text{Secret}}[\text{Secret}]]$. It follows that $E, E' \vdash P$, so Theorem 2 still applies. For example, A' may be:

$$A' \triangleq (\nu k)(\overline{c}\langle k, a \rangle \mid a(x, y).\mathit{if}\ x = k\ \mathit{then}\ \overline{y}\langle s' \rangle)$$

that is, a variant of A that initiates a session with a third party on the channel c and sends s'. With $E' = c : \text{Public}, s' : \text{Public}$, Theorem 2 yields that P preserves the secrecy of s from $(\{c, s'\}, \{a, b\})$. Remarkably, the replication of B causes no complication whatsoever. In contrast, replication tends to be problematic for proof methods based on state-space exploration.

7 Conclusion

This paper presents a type system that can serve in establishing secrecy properties of processes. The type system is superficially straightforward: it consists of fairly elementary rules in a standard format. The proof of its soundness (the subject-reduction theorem) is also fairly standard; the subject-reduction theorem then yields the main secrecy theorem. This simplicity is not entirely accidental: we explored more complex type systems (with dependent types) and notions of secrecy before arriving at the current ones.

On the other hand, the type system is powerful enough to apply to some delicate security protocols, yielding concise proofs for subtle results. It is also fairly tricky, when examined more closely. For instance, as indicated above, the type rules allow certain forms of polymorphism.

A challenging subject for further work is to develop type systems with richer forms of polymorphism, with stronger theories, perhaps even with algorithms for inferring secrecy types. In another direction, it would be worthwhile to give type rules for more cryptographic primitives, for example for digital signatures. Finally, it would be useful to integrate proofs by typing with other proof methods.

References

[1] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.

[2] Martín Abadi. Security protocols and their properties. In F.L. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*, NATO Science Series, pages 39–60. IOS Press, 2000. Volume for the 20th International Summer School on Foundations of Secure Computation, held in Marktoberdorf, Germany (1999).

[3] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999. An extended version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998.

[4] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[5] Ross Anderson and Roger Needham. Robustness principles for public key protocols. In *Proceedings of Crypto '95*, pages 236–247, 1995.

[6] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software Practice and Experience*, S4(25):87–130, December 1995.

[7] Chiara Bodei. *Security Issues in Process Calculi.* PhD thesis, Università di Pisa, January 2000.

[8] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the π-calculus. In *CONCUR'98: Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 84–98. Springer Verlag, September 1998.

[9] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In Catuscia Palamidessi, editor, *CONCUR 2000: Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 365–379. Springer-Verlag, August 2000.

[10] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading, Mass., 1982.

[11] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, January 1996.

[12] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, April 1984.

[13] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[14] Matthew Hennessy and James Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 415–427. Springer-Verlag, 2000.

[15] Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In Gert Smolka, editor, *Programming Languages and Systems: Proceedings of the 9th European Symposium on Programming (ESOP 2000), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 2000.

[16] K. Rustan M. Leino and Rajeev Joshi. A semantic approach to secure information flow. In *Mathematics of Program Construction, 4th International Conference*, volume 1422 of *Lecture Notes in Computer Science*, pages 254–271. Springer Verlag, 1998.

[17] Gavin 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 *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996.

[18] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[19] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 856–867. Springer-Verlag, 1998.

[20] Jon Millen and Harald Ruess. Protocol-independent secrecy. In *Proceedings 2000 IEEE Symposium on Security and Privacy*, pages 110–119, May 2000.

[21] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, September 1992.

[22] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 228–241, January 1999.

[23] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 129–142, 1997.

[24] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[25] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[26] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, September 2000.

[27] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 93–104, January 1999.

[28] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187, 1996.

Axiomatizing Tropical Semirings

Luca Aceto[1], Zoltán Ésik[2⋆], and Anna Ingólfsdóttir[1]

[1] BRICS⋆⋆⋆, Department of Computer Science, Aalborg University,
Fredrik Bajers Vej 7-E, DK-9220 Aalborg Ø, Denmark.
[2] Department of Computer Science,
A. József University, Árpád tér 2, 6720 Szeged, Hungary.

Abstract. This paper studies the equational theory of various exotic semirings presented in the literature. Exotic semirings are semirings whose underlying carrier set is some subset of the set of real numbers equipped with binary operations of minimum or maximum as sum, and addition as product. Two prime examples of such structures are the $(\max, +)$ *semiring* and the *tropical semiring*. It is shown that none of the exotic semirings commonly considered in the literature has a finite basis for its equations, and that similar results hold for the commutative idempotent weak semirings that underlie them. For each of these commutative idempotent weak semirings, the paper offers characterizations of the equations that hold in them, explicit descriptions of the free algebras in the varieties they generate, and relative axiomatization results.

Keywords and Phrases: Equational logic, varieties, complete axiomatizations, relative axiomatizations, commutative idempotent weak semirings, tropical semirings, convexity, exponential time complexity.

1 Introduction

Exotic semirings, i.e., semirings whose underlying carrier set is some subset of the set of real numbers $\mathbb{R}$ equipped with binary operations of minimum or maximum as sum, and addition as product, have been invented and reinvented many times since the late fifties in various fields of research. This family of structures consists of semirings whose sum operation is idempotent—two prime examples are the $(\max, +)$ *semiring* $(\mathbb{R} \cup \{-\infty\}, \max, +, -\infty, 0)$ (see [5, Chapter 3] for a general reference), and the *tropical semiring* $(\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$ introduced in [20]. (Henceforth, we shall write $\vee$ and $\wedge$ for the binary maximum and minimum operations, respectively.) Interest in idempotent semirings arose in the 1950s through the observation that some problems in discrete optimization could be linearized over such structures (see, e.g., [22] for a survey). Since then, the study of idempotent semirings has forged productive connections with such diverse fields as, e.g., performance evaluation of manufacturing systems,

⋆ Partially supported by grant no. T30511 from the National Foundation of Hungary for Scientific Research.
⋆⋆⋆ Basic Research in Computer Science.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 42–56, 2001.

discrete event system theory, graph theory (path algebra), Markov decision processes, Hamilton-Jacobi theory, and automata and language theory (automata with multiplicities). The interested reader is referred to [10] for a survey of these more recent developments. Here we limit ourselves to mentioning some of the deep applications of variations on the tropical semiring in automata theory and the study of formal power series.

The tropical semiring $(\mathbb{N} \cup \{\infty\}, \wedge, +, \infty, 0)$ was originally introduced by Simon in his solution (see [20]) to Brzozowski's celebrated finite power property problem—i.e., whether it is decidable if a regular language L has the property that, for some $m \geq 0$,

$$L^* = 1 + L + \cdots + L^m \ .$$

The basic idea in Simon's argument was to use automata with multiplicities in the tropical semiring to reformulate the finite power property as a Burnside problem. (The original Burnside problem asks if a finitely generated group must necessarily be finite if each element has finite order [7].) The tropical semiring was also used by Hashiguchi in his independent solution to the aforementioned problem [12], and in his study of the star height of regular languages (see, e.g., [13,14]). (For a tutorial introduction on how the tropical semiring is used to solve the finite power property problem, we refer the reader to [18].) The tropical semiring also plays a key role in Simon's study of the nondeterministic complexity of a standard finite automaton [21].

The study of automata and regular expressions with multiplicities in the tropical semiring is by now classic, and has yielded many beautiful and deep results—whose proofs have relied on the study of further exotic semirings. For example, Krob has shown that the equality problem for regular expressions with multiplicities in the tropical semiring is undecidable [15] by introducing the *equatorial semiring* $(\mathbb{Z} \cup \{\infty\}, \wedge, +, \infty, 0)$, showing that the equality problem for it is undecidable, and finally proving that the two decidability problems are equivalent. Partial decidability results for certain kinds of equality problems over the tropical and equatorial semirings are studied in [16].

Another classic question for the language of regular expressions, with or without multiplicities, is the study of complete axiom systems for them (see, e.g., [9]). Along this line of research, Bonnier-Rigny and Krob have offered a complete system of identities for one-letter regular expressions with multiplicities in the tropical semiring [6]. However, to the best of our knowledge, there has not been a systematic investigation of the equational theory of the different exotic semirings studied in the literature. This is the aim of this paper.

Our starting points are the results we obtained in [1,2]. In [1] we studied the equational theory of the max-plus algebra of the natural numbers $\mathbf{N}_\vee = (\mathbb{N}, \vee, +, 0)$, and proved that not only its equational theory is not finitely based, but, for every n, the equations in at most n variables that hold in it do not form an equational basis. Another view of the non-existence of a finite basis for the variety generated by this algebra is offered in [2], where we showed that the collection of equations in two variables that hold in it has no finite equational axiomatization.

The algebra $\mathbf{N}_\vee$ is an example of a structure that we call in this paper *commutative idempotent weak semiring* (abbreviated to ciw-semiring). Since ciw-semirings underlie many of the exotic semirings studied in the literature, we begin our investigations in this paper by systematically generalizing the results from [1,2] to the structures $\mathbf{Z}_\vee = (\mathbb{Z}, \vee, +, 0)$ and $\mathbf{N}_\wedge = (\mathbb{N}, \wedge, +, 0)$. Our initial step in the study of the equational theory of these ciw-semirings is the characterization of the (in)equations that hold in them (Propositions 3 and 4). These characterizations pave the way to explicit descriptions of the free algebras in the varieties $\mathcal{V}(\mathbf{Z}_\vee)$ and $\mathcal{V}(\mathbf{N}_\wedge)$ generated by $\mathbf{Z}_\vee$ and $\mathbf{N}_\wedge$, respectively, (Theorem 1) and yield finite axiomatizations of the varieties $\mathcal{V}(\mathbf{N}_\vee)$ and $\mathcal{V}(\mathbf{N}_\wedge)$ relative to $\mathcal{V}(\mathbf{Z}_\vee)$ (Theorem 2). We then show that, like $\mathcal{V}(\mathbf{N}_\vee)$, the varieties $\mathcal{V}(\mathbf{Z}_\vee)$ and $\mathcal{V}(\mathbf{N}_\wedge)$ are not finitely based. The non-finite axiomatizability of the variety $\mathcal{V}(\mathbf{Z}_\vee)$ (Theorem 3) is a consequence of the similar result for $\mathcal{V}(\mathbf{N}_\vee)$ and of its finite axiomatizability relative to $\mathcal{V}(\mathbf{Z}_\vee)$. The proof of the non-existence of a finite basis for the variety $\mathcal{V}(\mathbf{N}_\wedge)$ (Theorem 4) is more challenging, and relies on a model-theoretic construction. This construction yields that, as for $\mathcal{V}(\mathbf{N}_\vee)$, for every natural number n, the set of equations in at most n variables that hold in $\mathcal{V}(\mathbf{N}_\wedge)$ does not form an equational basis for this variety. A similar strengthening of the non-finite axiomatizability result holds for the variety $\mathcal{V}(\mathbf{Z}_\vee)$, for which we also show that the collection of equations in two variables that hold in it does not have a finite axiomatization.

We then move on to study the equational theory of the exotic semirings presented in the literature that are obtained by adding bottom elements to the above ciw-semirings. More specifically, we examine the following semirings:

$$\begin{aligned}
\mathbf{Z}_{\vee,-\infty} &= (\mathbb{Z} \cup \{-\infty\}, \vee, +, -\infty, 0) \ , \\
\mathbf{N}_{\vee,-\infty} &= (\mathbb{N} \cup \{-\infty\}, \vee, +, -\infty, 0) \text{ and} \\
\mathbf{N}^-_{\vee,-\infty} &= (\mathbb{N}^- \cup \{-\infty\}, \vee, +, -\infty, 0) \ ,
\end{aligned}$$

where $\mathbb{N}^-$ stands for the set of nonpositive integers. Since $\mathbf{Z}_{\vee,-\infty}$ and $\mathbf{N}^-_{\vee,-\infty}$ are easily seen to be isomorphic to the semirings $\mathbf{Z}_{\wedge,\infty} = (\mathbb{Z} \cup \{\infty\}, \wedge, +, \infty, 0)$ and $\mathbf{N}_{\wedge,\infty} = (\mathbb{N} \cup \{\infty\}, \wedge, +, \infty, 0)$, respectively, the results that we obtain apply equally well to these algebras. In fact, rather than studying the algebra $\mathbf{N}^-_{\vee,-\infty}$, we work with the semiring $\mathbf{N}_{\wedge,\infty}$. (The semirings $\mathbf{Z}_{\wedge,\infty}$ and $\mathbf{N}_{\wedge,\infty}$ are usually referred to as the *equatorial semiring* [15] and the *tropical semiring* [20], respectively. The semiring $\mathbf{N}_{\vee,-\infty}$ is called the *polar semiring* in [17].) We offer non-finite axiomatizability results for the semirings $\mathbf{Z}_{\vee,-\infty}$, $\mathbf{N}_{\vee,-\infty}$ and $\mathbf{N}_{\wedge,\infty}$.

Throughout the paper, we shall use standard notions and notations from universal algebra that can be found, e.g., in [8]. All the proofs of our technical results are omitted for lack of space.

A full version of this study, containing full proofs and many more results, including those presented in [3], can be found in [4].

2 Background Definitions

We begin by introducing some notions that will be used in the technical developments to follow.

A *commutative idempotent weak semiring* (henceforth abbreviated to ciw-semiring) is an algebra $\mathbf{A} = (A, \vee, +, 0)$ such that $(A, \vee)$ is an idempotent commutative semigroup, i.e., a semilattice, $(A, +, 0)$ is a commutative monoid, and such that addition distributes over the $\vee$ operation. Thus, the following equations hold in $\mathbf{A}$:

$$\begin{aligned} x \vee (y \vee z) &= (x \vee y) \vee z \\ x \vee y &= y \vee x \\ x \vee x &= x \\ x + (y + z) &= (x + y) + z \\ x + y &= y + x \\ x + 0 &= x \\ x + (y \vee z) &= (x + y) \vee (x + z) \ . \end{aligned}$$

A homomorphism of ciw-semirings is a function which preserves the $\vee$ and $+$ operations and the constant 0.

A *commutative idempotent semiring* is an algebra $(A, \vee, +, \bot, 0)$ such that $(A, \vee, +, 0)$ is a ciw-semiring which satisfies the equations

$$\begin{aligned} x \vee \bot &= x \\ x + \bot &= \bot \ . \end{aligned}$$

A homomorphism of commutative idempotent semirings also preserves $\bot$.

Suppose that $\mathbf{A} = (A, \vee, +, 0)$ is a ciw-semiring. Assume that $\bot \notin A$ and let $A_\bot = A \cup \{\bot\}$. Extend the operations $\vee$ and $+$ given on A to $A_\bot$ by defining

$$\begin{aligned} a \vee \bot &= \bot \vee a = a \\ a + \bot &= \bot + a = \bot \ , \end{aligned}$$

for all $a \in A_\bot$. We shall write $\mathbf{A}_\bot$ for the resulting algebra.

Lemma 1. *For each ciw-semiring* $\mathbf{A}$*, the algebra* $\mathbf{A}_\bot$ *is a commutative idempotent semiring.*

Remark 1. In fact, $\mathbf{A}_\bot$ is the free commutative idempotent semiring generated by $\mathbf{A}$.

Let E_{ciw} denote the set of defining axioms of ciw-semirings, and E_{ci} the set of axioms of commutative idempotent semirings. Moreover, let $\mathcal{V}_{ciw}$ denote the variety axiomatized by E_{ciw}.

In the remainder of this paper, we shall use nx to denote the n-fold sum of x with itself, and we take advantage of the associativity and commutativity of

the operations. By convention, nx stands for 0 when $n = 0$. In the same way, the empty sum is defined to be 0.

For each integer $n \geq 0$, we use $[n]$ to stand for the set $\{1, \ldots, n\}$, so that $[0]$ is another name for the empty set.

Lemma 2. *With respect to the axiom system E_{ciw} (respectively, E_{ci}), every term t in the language of ciw-semirings (resp., commutative idempotent semirings), in the variables $x_1, \ldots, x_n$, may be rewritten in the form*

$$t = \bigvee_{i \in [k]} t_i$$

where $k > 0$ (resp., $k \geq 0$), each t_i is a "linear combination"

$$t_i = \sum_{j \in [n]} c_{ij} x_j \ ,$$

and each c_{ij} is in $\mathbb{N}$.

Terms of the form $\bigvee_{i \in [k]} t_i$, where each t_i ($i \in [k]$, $k \geq 0$) is a linear combination of variables, will be referred to as *simple terms*. When $k = 0$, the term $\bigvee_{i \in [k]} t_i$ is just $\bot$. (Note that $k = 0$ is only allowed for commutative idempotent semirings.) For any commutative idempotent (weak) semiring $\mathbf{A}$ and $a, b \in A$, we write $a \leq b$ to mean $a \vee b = b$. In any such structure, the relation $\leq$ so defined is a partial order, and the $+$ and $\vee$ operations are monotonic with respect to it. Similarly, we say that an inequation $t \leq t'$ between terms t and t' holds in $\mathbf{A}$ if the equation $t \vee t' = t'$ holds. We shall write $\mathbf{A} \models t = t'$ (respectively, $\mathbf{A} \models t \leq t'$) if the equation $t = t'$ (resp., the inequation $t \leq t'$) holds in $\mathbf{A}$. (In that case, we say that $\mathbf{A}$ is a model of $t = t'$ or $t \leq t'$, respectively.)

Definition 1. *A* simple inequation *in the variables $x_1, \ldots, x_n$ is of the form*

$$t \leq \bigvee_{i \in [k]} t_i,$$

where $k > 0$, and t and the t_i ($i \in [k]$) are linear combinations of the variables $x_1, \ldots, x_n$. We say that the left-hand side of the above simple inequation contains the variable x_j, or that x_j appears on the left-hand side of the simple inequation, if the coefficient of x_j in t is nonzero. Similarly, we say that the right-hand side of the above inequation contains the variable x_j if for some i, the coefficient of x_j in t_i is nonzero.

Note that, for every linear combination t over variables $x_1, \ldots, x_n$, the inequation $t \leq \bot$ is *not* a simple inequation.

Corollary 1. *With respect to the axiom system E_{ciw}, any equation in the language of ciw-semirings is equivalent to a finite set of simple inequations. Similarly, with respect to E_{ci}, any equation in the language of commutative idempotent semirings is equivalent to a finite set of simple inequations or to an inequation of the form $x \leq \bot$ (in which case the equation has only trivial models).*

Henceforth in this study, we shall often abbreviate a simple inequation

$$\sum_{j\in[n]} d_j x_j \leq \bigvee_{i\in[k]} \sum_{j\in[n]} c_{ij} x_j$$

as $\overline{d} \leq \{\overline{c}_1, \ldots, \overline{c}_k\}$, where $\overline{d} = (d_1, \ldots, d_n)$ and $\overline{c}_i = (c_{i1}, \ldots, c_{in})$, for $i \in [k]$. We shall sometimes refer to these inequations as *simple* $\vee$*-inequations*.

In the main body of the paper, we shall also study some ciw-semirings that, like the structure $\mathbf{N}_\wedge = (\mathbb{N}, \wedge, +, 0)$, have the minimum operation in lieu of maximum. The preliminary results that we have developed in this section apply equally well to these structures. Note that, for ciw-semirings of the form $(A, \wedge, +, 0)$, the partial order $\geq$ is defined by $b \geq a$ iff $a \wedge b = a$. In Defn. 1, we introduced the notion of simple $\vee$-inequation. Dually, we say that a *simple* $\wedge$*-inequation* in the variables $x_1, \ldots, x_n$ is an inequation of the form

$$\overline{d} \cdot \overline{x} \geq \bigwedge_{i\in[k]} \overline{c}_i \cdot \overline{x} \ ,$$

where $\overline{d}$ and the $\overline{c}_i$ $(i \in [k])$ are vectors in $\mathbb{N}^n$. We shall often write

$$\overline{d} \geq \{\overline{c}_1, \ldots, \overline{c}_k\}$$

as a shorthand for this inequation.

3 Min-Max-Plus Weak Semirings

Our aim in this section will be to study the equational theory of the ciw-semirings that underlie most of the tropical semirings studied in the literature. More specifically, we shall study the following ciw-semirings:

$$\mathbf{Z}_\vee = (\mathbb{Z}, \vee, +, 0),\ \mathbf{N}_\vee = (\mathbb{N}, \vee, +, 0) \text{ and } \mathbf{N}_\wedge = (\mathbb{N}, \wedge, +, 0)$$

equipped with the usual sum operation $+$, constant 0 and one of the operations $\vee$ for the maximum of two numbers and $\wedge$ for the minimum of two numbers. We shall sometimes use the fact that $\mathbf{Z}_\vee$ and $\mathbf{N}_\wedge$ are isomorphic to the ciw-semirings $\mathbf{Z}_\wedge = (\mathbb{Z}, \wedge, +, 0)$ and $\mathbf{N}_\vee^- = (\mathbb{N}^-, \vee, +, 0)$, respectively, where $\mathbb{N}^-$ stands for the set of nonpositive integers.

Our study of the equational theory of these algebras will be based on the following uniform pattern. First, we offer a characterization of the simple inequations that hold in these ciw-semirings. These characterizations pave the way to concrete descriptions of the free algebras in the varieties generated by the algebras we study, and yield relative axiomatization results (Sect. 3.2). Finally we show that none of the ciw-semirings we study is finitely based (Sect. 3.3). All of these technical results rely on a study of properties of convex sets, filters and ideals in $\mathbb{Z}^n$ and $\mathbb{N}^n$ presented in the following section.

3.1 Convex Sets, Filters, and Ideals

Suppose that $\overline{v}_1, \ldots, \overline{v}_k$ are vectors in $\mathbb{Z}^n$, or more generally, in $\mathbb{R}^n$. A *convex linear combination* of the $\overline{v}_i$ ($i \in [k]$) is any vector $\overline{v} \in \mathbb{R}^n$ which can be written as

$$\overline{v} = \lambda_1 \overline{v}_1 + \cdots + \lambda_k \overline{v}_k \ ,$$

where $\lambda_i \geq 0$, $i \in [k]$, are real numbers with $\sum_{i=1}^{k} \lambda_i = 1$.

Definition 2. *Suppose that U is any subset of $\mathbb{Z}^n$. We call U a* convex set *if for all convex linear combinations $\overline{v} = \lambda_1 \overline{v}_1 + \cdots + \lambda_k \overline{v}_k$ with $k > 0$ and $\overline{v}_i \in U$, $i \in [k]$, if $\overline{v} \in \mathbb{Z}^n$, then $\overline{v} \in U$.*

Suppose that $U \subseteq \mathbb{N}^n$. We call U an (order) ideal *if for all $\overline{u}, \overline{v}$ in $\mathbb{N}^n$, if $\overline{u} \leq \overline{v}$ and $\overline{v} \in U$ then $\overline{u} \in U$. Moreover, we call U a* filter*, if for all $\overline{u}$ and $\overline{v}$ as above, if $\overline{u} \in U$ then $\overline{v} \in U$. A convex ideal (respectively, convex filter) in $\mathbb{N}^n$ is any ideal (resp., filter) which is a convex set.*

Note that order ideals and filters are sometimes referred to as lower and upper sets, respectively. The following fact is clear:

Proposition 1. *The intersection of any number of convex sets in $\mathbb{Z}^n$ is convex. Moreover, the intersection of any number of convex ideals (convex filters) in $\mathbb{N}^n$ is a convex ideal (convex filter, respectively).*

Thus each set $U \subseteq \mathbb{Z}^n$ is contained in a smallest convex set $[U]$ which is the intersection of all convex subsets of $\mathbb{Z}^n$ containing U. We call $[U]$ the convex set generated by U, or the *convex hull* of U. When $\overline{u} \in \mathbb{Z}^n$, below we shall sometimes write $[\overline{u}]$ for $[\{\overline{u}\}] = \{\overline{u}\}$.

Suppose now that $U \subseteq \mathbb{N}^n$. By Proposition 1, there is a smallest convex ideal $\mathrm{ci}(U)$ and a smallest convex filter $\mathrm{cf}(U)$ in $\mathbb{N}^n$ containing U. We call $\mathrm{ci}(U)$ and $\mathrm{cf}(U)$ the convex ideal and the convex filter generated by U, respectively.

For each set $U \subseteq \mathbb{N}^n$, define the ideal $(U]$ generated by U thus:

$$(U] = \{\overline{d} \in \mathbb{N}^n : \exists \overline{c} \in U.\ \overline{d} \leq \overline{c}\} \ .$$

Similarly, the filter $[U)$ generated by U is defined as:

$$[U) = \{\overline{d} \in \mathbb{N}^n : \exists \overline{c} \in U.\ \overline{d} \geq \overline{c}\} \ .$$

Throughout this study, we shall use $\overline{u}_i$ ($i \in [n]$) to denote the ith unit vector in $\mathbb{R}^n$, i.e., the vector whose only nonzero component is a 1 in the ith position; $\overline{0}$ will denote the vector in $\mathbb{R}^n$ whose entries are all zero. When $\overline{u} \in \mathbb{N}^n$, below we shall sometimes write $(\overline{u}]$ and $[\overline{u})$ for $(\{\overline{u}\}]$ and $[\{\overline{u}\})$, respectively.

The following result, whose proof is nontrivial, plays an important role in the technical developments to follow.

Proposition 2. *Suppose that $U \subseteq \mathbb{N}^n$. Then:*

1. $\mathrm{ci}(U) = [(U]]$, *and*
2. $\mathrm{cf}(U) = [[U)]$.

We can also show that $\mathrm{ci}(U) = ([U]_R]$ and $\mathrm{cf}(U) = [[U]_R)$, where $[U]_R$ denotes the convex hull in $\mathbb{R}^n$ of the set U. However, the equalities $\mathrm{ci}(U) = ([U]]$ and $\mathrm{cf}(U) = [[U])$ fail.

Henceforth, we shall use $[(U]]$ and $[[U)]$ to denote the convex ideal and the convex filter generated by U, respectively. This notation is justified by the above proposition.

Recall that a simple $\vee$-inequation in the variables $\overline{x} = (x_1, \ldots, x_n)$ is an inequation of the form

$$\overline{d}\overline{x} \leq \overline{c}_1\overline{x} \vee \cdots \vee \overline{c}_k\overline{x} \ , \tag{1}$$

where $k > 0$, and $\overline{d}, \overline{c}_1, \ldots, \overline{c}_k \in \mathbb{N}^n$. Similarly, a simple $\wedge$-inequation is of the form

$$\overline{d}\overline{x} \geq \overline{c}_1\overline{x} \wedge \cdots \wedge \overline{c}_k\overline{x} \ , \tag{2}$$

where k, $\overline{d}$, and $\overline{c}_i$ ($i \in [k]$) are as above. We recall that (1) holds in a ciw-semiring $\mathbf{A}_\vee = (A, \vee, +, 0)$ if the equation

$$\overline{d}\overline{x} \vee \overline{c}_1\overline{x} \vee \cdots \vee \overline{c}_k\overline{x} = \overline{c}_1\overline{x} \vee \cdots \vee \overline{c}_k\overline{x}$$

does, i.e., when for all $\overline{v} \in A^n$,

$$\overline{d}\overline{v} \leq \overline{c}_1\overline{v} \vee \cdots \vee \overline{c}_k\overline{v} \ .$$

Similarly, we say that (2) holds in a ciw-semiring $\mathbf{A}_\wedge = (A, \wedge, +, 0)$ if the equation

$$\overline{d}\overline{x} \wedge \overline{c}_1\overline{x} \wedge \cdots \wedge \overline{c}_k\overline{x} = \overline{c}_1\overline{x} \wedge \cdots \wedge \overline{c}_k\overline{x}$$

does. Let U denote the set $\{\overline{c}_1, \ldots, \overline{c}_k\}$. We recall that we shall sometimes abbreviate (1) as $\overline{d} \leq U$ and (2) as $\overline{d} \geq U$.

We now proceed to characterize the collection of simple inequations that hold in the algebra $\mathbf{Z}_\vee$ (and, thus, in its isomorphic version $\mathbf{Z}_\wedge$).

Proposition 3. *Suppose that $\overline{d} \in \mathbb{N}^n$ and $U = \{\overline{c}_1, \ldots, \overline{c}_k\}$ is a finite nonempty set of vectors in $\mathbb{N}^n$. A simple inequation $\overline{d} \leq U$ holds in $\mathbf{Z}_\vee$ iff $\overline{d}$ belongs to the set $[U]$.*

For $\mathbf{N}_\vee$ and $\mathbf{N}_\wedge$ we have:

Proposition 4. *Suppose that $\overline{d} \in \mathbb{N}^n$ and U is a finite nonempty set of vectors in $\mathbb{N}^n$.*

1. *The simple inequation $\overline{d} \leq U$ holds in $\mathbf{N}_\vee$ iff $\overline{d}$ belongs to the set $[(U]]$.*
2. *The simple inequation $\overline{d} \geq U$ holds in $\mathbf{N}_\wedge$ iff $\overline{d}$ belongs to the set $[[U)]$.*

The problem of deciding whether an equation holds in any of $\mathbf{Z}_\vee$, $\mathbf{N}_\vee$ and $\mathbf{N}_\wedge$ can be reduced to deciding whether a finite set of simple inequations holds in it (Cor. 1). The obvious reduction may result in a number of simple inequations that is exponential in the number of variables. However, using Propositions 3 and 4, the validity of a simple inequation in any of these structures can be tested in polynomial time by using linear programming (see, e.g., [19]). We therefore have that:

Corollary 2. *There exists an exponential time algorithm to decide whether an equation holds in the structures* $\mathbf{Z}_\vee, \mathbf{N}_\vee$ *and* $\mathbf{N}_\wedge$*. Moreover, it is decidable in polynomial time whether a simple inequation holds in these structures.*

3.2 Free Algebras and Relative Axiomatizations

Let $C(\mathbb{N}^n)$, $CI(\mathbb{N}^n)$ and $CF(\mathbb{N}^n)$ denote the sets of all finite nonempty convex sets, finite nonempty convex ideals, and nonempty convex filters in $\mathbb{N}^n$, respectively. We turn each of these sets into a ciw-semiring. Suppose that $U, V \in C(\mathbb{N}^n)$. First of all, recall that the complex sum of U and V, notation $U \oplus V$, is defined thus:

$$U \oplus V = \{\overline{u} + \overline{v} \ : \ \overline{u} \in U, \ \overline{v} \in V\} \ .$$

We define

$$\begin{aligned} U \vee V &= [U \cup V] \\ U + V &= [U \oplus V] \\ 0 &= [\overline{0}] = \{\overline{0}\} \ . \end{aligned}$$

We define the operations in $CI(\mathbb{N}^n)$ and $CF(\mathbb{N}^n)$ in a similar fashion. Suppose that $U, V \in CI(\mathbb{N}^n)$ and $U', V' \in CF(\mathbb{N}^n)$. We set

$$\begin{aligned} U \vee V &= [(U \cup V]] \\ U + V &= [(U \oplus V]] \\ U' \wedge V' &= [[U' \cup V')] \\ U' + V' &= [[U' \oplus V')] \ . \end{aligned}$$

Moreover, we define $0 = (\overline{0}] = \{\overline{0}\}$ in $CI(\mathbb{N}^n)$, and $0 = [\overline{0}) = \mathbb{N}^n$ in $CF(\mathbb{N}^n)$.

Proposition 5. *Each of the structures* $C(\mathbb{N}^n) = (C(\mathbb{N}^n), \vee, +, 0)$*,* $CI(\mathbb{N}^n) = (CI(\mathbb{N}^n), \vee, +, 0)$ *and* $CF(\mathbb{N}^n) = (CF(\mathbb{N}^n), \wedge, +, 0)$ *is a ciw-semiring. In addition,* $CI(\mathbb{N}^n)$ *satisfies the equation*

$$x \vee 0 = x \ , \tag{3}$$

and $CF(\mathbb{N}^n)$ *the equation*

$$x \wedge 0 = 0 \ . \tag{4}$$

Note that (3) can be rephrased, with respect to E_{ciw}, as the inequation $0 \leq x$, and (4) as $x \geq 0$. Also, writing $\vee$ for $\wedge$, (4) takes the form $x \vee 0 = 0$ that one should have if $\vee$ is considered to be the signature symbol instead of $\wedge$.

For any structure $\mathbf{A}$, we use $\mathcal{V}(\mathbf{A})$ to denote the variety generated by $\mathbf{A}$, i.e., the class of algebras that satisfy the equations that hold in $\mathbf{A}$.

Theorem 1. *For each $n \geq 0$,*

1. *$C(\mathbb{N}^n)$ is freely generated in $\mathcal{V}(\mathbf{Z}_\vee)$ by the sets $[\overline{u}_i]$, $i \in [n]$,*
2. *$CI(\mathbb{N}^n)$ is freely generated in $\mathcal{V}(\mathbf{N}_\vee)$ by the sets $(\overline{u}_i]$, $i \in [n]$, and*
3. *$CF(\mathbb{N}^n)$ is freely generated in $\mathcal{V}(\mathbf{N}_\wedge)$ by the sets $[\overline{u}_i)$, $i \in [n]$.*

Remark 2. In any variety, any infinitely generated free algebra is the direct limit of finitely generated free algebras. More specifically, for any cardinal number κ, the free algebra on κ generators in $\mathcal{V}(\mathbf{Z}_\vee)$ can be described as an algebra of finite nonempty convex sets in $\mathbb{N}^\kappa$ consisting of vectors whose components, with a finite number of exceptions, are all zero. The free algebras in $\mathcal{V}(\mathbf{N}_\vee)$ and $\mathcal{V}(\mathbf{N}_\wedge)$ have similar descriptions using finitely generated convex ideals and filters, respectively.

Since $\mathbf{N}_\vee$ is a subalgebra of $\mathbf{Z}_\vee$, we have that $\mathcal{V}(\mathbf{N}_\vee) \subseteq \mathcal{V}(\mathbf{Z}_\vee)$. Also, since $\mathbf{N}_\wedge$ is isomorphic to the subalgebra $\mathbf{N}_\vee^-$ of $\mathbf{Z}_\vee$, it holds that $\mathcal{V}(\mathbf{N}_\wedge) \subseteq \mathcal{V}(\mathbf{Z}_\vee)$.

Definition 3. *Let $\mathcal{V}$ and $\mathcal{V}'$ be two varieties of algebras over the same signature Σ. Let E be a collection of equations over Σ. We say that $\mathcal{V}$ is* axiomatized over $\mathcal{V}'$ by E *if the collection of equations that hold in $\mathcal{V}'$ together with E form a basis for the identities of $\mathcal{V}$. We say that $\mathcal{V}$ has a* finite axiomatization relative to $\mathcal{V}'$ *if $\mathcal{V}$ is axiomatized over $\mathcal{V}'$ by some finite set of equations E.*

In the next result we show that both $\mathcal{V}(\mathbf{N}_\vee)$ and $\mathcal{V}(\mathbf{N}_\wedge)$ possess a finite axiomatization relative to $\mathcal{V}(\mathbf{Z}_\vee)$. Of course, $\mathcal{V}(\mathbf{Z}_\vee)$ is just $\mathcal{V}(\mathbf{Z}_\wedge)$, since $\mathbf{Z}_\vee$ and $\mathbf{Z}_\wedge$ are also isomorphic.

Theorem 2.

1. *$\mathcal{V}(\mathbf{N}_\vee)$ is axiomatized over $\mathcal{V}(\mathbf{Z}_\vee)$ by the equation (3).*
2. *$\mathcal{V}(\mathbf{N}_\wedge)$ is axiomatized over $\mathcal{V}(\mathbf{Z}_\wedge)$ by the equation (4).*

3.3 Non-finite Axiomatizability Results

Our order of business in this section is to show that the varieties generated by the ciw-semirings $\mathbf{Z}_\vee$ and $\mathbf{N}_\wedge$ are not finitely based. Our starting points are the results in [1,2] to the effect that the variety $\mathcal{V}(\mathbf{N}_\vee)$ is not finitely based. By using these results, and the first part of Theorem 2, we can derive:

Theorem 3. *The variety $\mathcal{V}(\mathbf{Z}_\vee)$ is not finitely based. Moreover, there exists no natural number n such that the collection of all equations in at most n variables that hold in $\mathcal{V}(\mathbf{Z}_\vee)$ forms an equational basis for $\mathcal{V}(\mathbf{Z}_\vee)$. Finally, the collection of equations in two variables that hold in the variety $\mathcal{V}(\mathbf{Z}_\vee)$ has no finite equational axiomatization.*

Our aim in the remainder of this section is to outline a proof of the following result to the effect that the variety $\mathcal{V}(\mathbf{N}_\wedge)$ has no finite equational basis.

Theorem 4. *The variety $\mathcal{V}(\mathbf{N}_\wedge)$ has no finite (equational) axiomatization.*

In fact, the above theorem is a consequence of the following stronger result.

Theorem 5. *There exists no natural number n such that the collection of all equations in at most n variables that hold in $\mathcal{V}(\mathbf{N}_\wedge)$ forms an equational basis for $\mathcal{V}(\mathbf{N}_\wedge)$.*

To prove Theorem 5, we begin by noting that the following equations e_n hold in $\mathbf{N}_\wedge$, for each $n \geq 2$:

$$e_n : \quad r_n \wedge s_n = s_n \ , \tag{5}$$

where

$$\begin{aligned} r_n &= x_1 + \cdots + x_n \\ s_n &= (2x_1 + x_3 + x_4 + \cdots + x_{n-1} + x_n) \\ &\quad \wedge (x_1 + 2x_2 + x_4 + \cdots + x_{n-1} + x_n) \\ &\quad \vdots \\ &\quad \wedge (x_1 + x_2 + x_3 + \cdots + x_{n-2} + 2x_{n-1}) \\ &\quad \wedge (x_2 + x_3 + x_4 + \cdots + x_{n-1} + 2x_n) \ . \end{aligned}$$

We then define a sequence of countably infinite ciw-semirings $\mathbf{B}_n$ $(n \geq 3)$ which are a model of all the equations in at most $n-1$ variables that hold in $\mathbf{N}_\wedge$, but in which e_n fails. The overall proof strategy is similar to the one we used to prove the main result in [1], but the actual technical details of the proof are significantly different.

We introduce the following notation for some vectors in $\mathbb{N}^n$ related to the equation e_n:

$$\begin{aligned} \overline{\delta} &= (1, \ldots, 1) \\ \overline{\gamma}_1 &= (2, 0, 1, 1, \ldots, 1, 1) \\ \overline{\gamma}_2 &= (1, 2, 0, 1, \ldots, 1, 1) \\ &\vdots \\ \overline{\gamma}_{n-1} &= (1, 1, 1, 1, \ldots, 2, 0) \\ \overline{\gamma}_n &= (0, 1, 1, 1, \ldots, 1, 2) \ , \end{aligned}$$

so that in $\overline{\gamma}_i$ $(i \in [n])$, the 2 is on the ith position and is followed by a 0. (Of course, we assume that the first position follows the nth.) All other components are 1's. Note that

$$\overline{\delta} = \frac{1}{n}\overline{\gamma}_1 + \cdots + \frac{1}{n}\overline{\gamma}_n \ . \tag{6}$$

Thus, $\overline{\delta}$ belongs to the convex filter generated by the vectors $\overline{\gamma}_i$ $(i \in [n])$. Moreover, the system consisting of any n of the vectors $\overline{\delta}, \overline{\gamma}_1, \ldots, \overline{\gamma}_n$ is linearly independent.

We define:

$$\begin{aligned} \Gamma &= [[\{\overline{\gamma}_1, \ldots, \overline{\gamma}_n\})] \\ \Delta &= \Gamma - \{\overline{\delta}\} \ , \end{aligned}$$

so that Γ is the convex filter generated by the $\overline{\gamma}_i$ ($i \in [n]$). By (6), the set Δ is not a convex filter.

We now proceed to define the algebras $\mathbf{B}_n = (B_n, \wedge, +, 0)$, $n > 0$. Let B_n consist of the nonempty convex filters in $\mathbb{N}^n$ and the set Δ.

Proposition 6. *If the intersection of a family of sets in B_n is not empty, then the intersection is in B_n.*

For each nonempty set $U \subseteq \mathbb{N}^n$, let $\mathbf{cl}(U)$ denote the least set in B_n containing U. For each $U, V \in B_n$, we define

$$U + V = \mathbf{cl}(U \oplus V)$$
$$U \wedge V = \mathbf{cl}(U \cup V) \ .$$

Moreover, we define the constant 0 to be the set $[[0)] = [0) = \mathbb{N}^n$. This completes the definition of the algebra $\mathbf{B}_n = (B_n, \wedge, +, 0)$.

Proposition 7. *If $n \geq 3$, then $\mathbf{B}_n$ is a ciw-semiring satisfying $x \wedge 0 = 0$ but not satisfying e_n.*

Indeed, for each $i \in [n]$, let $\overline{u}_i$ denote the ith n-dimensional unit vector whose components are all 0 except for a 1 in the ith position. We have

$$r_n([[\overline{u}_1)], \ldots, [[\overline{u}_n)]) = [[\overline{\delta})] \text{ and}$$
$$s_n([[\overline{u}_1)], \ldots, [[\overline{u}_n)]) = \Delta$$

in $\mathbf{B}_n$.

Proposition 8. *For each $n \geq 3$, the algebra $\mathbf{B}_n$ satisfies any equation in at most $n-1$ variables which holds in $\mathbf{N}_\wedge$.*

Theorem 5 is an immediate consequence of the above facts.

4 Tropical Semirings

We now proceed to investigate the equational theory of the tropical semirings studied in the literature that are obtained by adding bottom elements to the ciw-semirings presented in the previous section. More specifically, we shall study the following semirings:

$$\mathbf{Z}_{\vee,-\infty} = (\mathbb{Z} \cup \{-\infty\}, \vee, +, -\infty, 0) \ ,$$
$$\mathbf{N}_{\vee,-\infty} = (\mathbb{N} \cup \{-\infty\}, \vee, +, -\infty, 0) \text{ and}$$
$$\mathbf{N}^-_{\vee,-\infty} = (\mathbb{N}^- \cup \{-\infty\}, \vee, +, -\infty, 0) \ .$$

Since $\mathbf{Z}_{\vee,-\infty}$ and $\mathbf{N}^-_{\vee,-\infty}$ are isomorphic to $\mathbf{Z}_{\wedge,\infty} = (\mathbb{Z} \cup \{\infty\}, \wedge, +, \infty, 0)$ and $\mathbf{N}_{\wedge,\infty} = (\mathbb{N} \cup \{\infty\}, \wedge, +, \infty, 0)$, respectively, the results that we shall obtain apply equally well to these algebras. In fact, rather than studying the algebra

$\mathbf{N}^{-}_{\vee,-\infty}$, we shall work with the semiring $\mathbf{N}_{\wedge,\infty}$. (The semirings $\mathbf{Z}_{\wedge,\infty}$ and $\mathbf{N}_{\wedge,\infty}$ are usually referred to as the *equatorial semiring* [15] and the *tropical semiring* [20], respectively. The semiring $\mathbf{N}_{\vee,-\infty}$ is called the *polar semiring* in [17].)

In Sect. 2, we saw how to generate a commutative idempotent semiring $\mathbf{A}_\perp = (A_\perp, \vee, +, \perp, 0)$ from any ciw-semiring $\mathbf{A} = (A, \vee, +, 0)$ by freely adding a bottom element $\perp$ to it. We now proceed to study some general relationships between the equational theories of these two structures. The results that we shall obtain are the keys to obtain non-finite axiomatizability results for the above tropical semirings associated with the ciw-semirings.

Definition 4. *An inequation is* strictly regular *if its two sides contain the same variables.*

Let $\mathbf{A} \in \mathcal{V}_{ciw}$. *We say that* $\mathbf{A}$ *is* strictly regularly based *if every simple inequation that holds in it is provable, modulo* E_{ciw}, *from a strictly regular one that holds in* $\mathbf{A}$.

The following general result will allow us to lift non-finite axiomatizability results from some ciw-semirings to the commutative idempotent semirings they underlie.

Proposition 9. *Suppose that* $\mathbf{A} \in \mathcal{V}_{ciw}$ *is strictly regularly based. If* $\mathcal{V}(\mathbf{A})$ *has no finite axiomatization, then* $\mathcal{V}(\mathbf{A}_\perp)$ *has no finite axiomatization either.*

Theorem 6.

1. *The ciw-semirings* $\mathbf{Z}_\vee$ *and* $\mathbf{N}_\vee$ *are strictly regularly based.*
2. *The varieties* $\mathcal{V}(\mathbf{Z}_{\vee,-\infty})$ *and* $\mathcal{V}(\mathbf{N}_{\vee,-\infty})$ *are not finitely based.*

We now relate the equational theory of the original tropical semiring $\mathbf{N}_{\wedge,\infty} = (\mathbb{N} \cup \{\infty\}, \wedge, +, \infty, 0)$ from [20] to the equational theory of the algebra $\mathbf{N}_\wedge = (\mathbb{N}, \wedge, +, 0)$. For the sake of clarity, we recall that $+$ is standard addition (extended to $\mathbb{N} \cup \{\infty\}$ by stipulating that $\infty + x = x + \infty = \infty$).

The structures $\mathbf{N}_\wedge$ and $\mathbf{N}_{\wedge,\infty}$ satisfy the same equations in the language of $\mathbf{N}_\wedge$. This is the import of the following lemma.

Lemma 3. *Let* $t \geq t'$ *be a simple* $\wedge$*-inequation in the language of* $\mathbf{N}_\wedge$. *Then* $t \geq t'$ *holds in* $\mathbf{N}_\wedge$ *iff it holds in* $\mathbf{N}_{\wedge,\infty}$.

As a corollary of this result, we have that:

Theorem 7. *The varieties* $\mathcal{V}(\mathbf{N}_{\wedge,\infty})$ *and* $\mathcal{V}(\mathbf{N}^{-}_{\vee,-\infty})$ *have no finite axiomatization.*

5 Further Results

In this section, we outline some further results. For details, the reader is referred to the extended abstract [3] and the technical report [4].

We have shown that none of the tropical semirings $\mathbf{Z}_{\vee,-\infty}$, $\mathbf{N}_{\vee,-\infty}$ and $\mathbf{N}^{-}_{\vee,-\infty}$, or $\mathbf{N}_{\wedge,\infty}$, has a finite basis for its equations. But what is a complete

description of the equations, or rather, simple inequations that hold in these semirings? As regards $\mathbf{N}_{\wedge,\infty}$, we have already obtained a satisfactory answer. As for $\mathbf{Z}_{\vee,-\infty}$, it can be seen easily that a simple inequation holds in $\mathbf{Z}_{\vee,-\infty}$ if and only if it holds in $\mathbf{Z}_{\vee}$. The case of $\mathbf{N}_{\vee,-\infty}$ is more challenging. Suppose that U is a finite nonempty subset of $\mathbb{N}^n$ and $\vec{d} \in U$. Then we have that a simple inequation $\vec{d} \leq U$ holds in $\mathbf{N}_{\vee,-\infty}$ iff $\vec{d} \in \mathrm{ci}^+(U)$, where $\mathrm{ci}^+(U)$ denotes the *positive convex ideal* [3,4] generated by U. This characterization of the valid inequations has three corollaries. First, it follows that an algebra of the positive convex ideals in $\mathbb{N}^n$ can be characterized as the n-generated free algebra in $\mathcal{V}(\mathbf{N}_{\vee,-\infty})$. Second, it follows that $\mathcal{V}(\mathbf{N}_{\vee,-\infty})$ can be axiomatized over $\mathcal{V}(\mathbf{Z}_{\vee,-\infty})$ by the simple inequation $x \leq x + x$. Third, it follows that the equational theory of $\mathcal{V}(\mathbf{N}_{\vee,-\infty})$ is decidable in exponential time.

There are some further variations. These include structures based on the rational or real numbers, or just the nonnegative on nonpositive rational or real numbers. Semirings involving rationals or reals always have the same equational theory as the subalgebra of the integers included in the semiring. Thus, e.g., the semiring $\mathbf{Q}_{\vee,-\infty} = (\mathbb{Q} \cup \{-\infty\}, \vee, +, -\infty, 0)$ has the same equational theory as the semiring $\mathbf{Z}_{\vee,-\infty}$, and the equational theory of $\mathbf{Q}^+_{\vee,-\infty} = (\mathbb{Q}^+ \cup \{-\infty\}, \vee, +, -\infty, 0)$ agrees with that of $\mathbf{N}_{\vee,-\infty}$.

Acknowledgement. We would like to thank the anonymous referees whose suggestions helped us improve the paper.

References

1. L. Aceto, Z. Ésik, and A. Ingólfsdóttir, *The max-plus algebra of the natural numbers has no finite equational basis*, BRICS Report RS-99-33, October 1999. To appear in *Theoretical Computer Science*.
2. ———, *On the two-variable fragment of the equational theory of the max-sum algebra of the natural numbers*, in Proceedings of the 17th STACS, H. Reichel and S. Tison, eds., vol. 1770 of Lecture Notes in Computer Science, Springer-Verlag, Feb. 2000, pp. 267–278.
3. ———, *Nonfinitely based tropical semirings*, in Proceedings of the Int. Workshop on Max-Plus Algebras, Praha, August, 2001, to appear.
4. ———, *Axiomatizing tropical semirings*, to appear as BRICS research report.
5. F. Baccelli, G. Cohen, G.-J. Olsder, and J.-P. Quadrat, *Synchronization and Linearity*, Wiley Series in Probability and Mathematical Statistics, John Wiley, 1992.
6. A. Bonnier-Rigny and D. Krob, *A complete system of identities for one-letter rational expressions with multiplicities in the tropical semiring*, Theoretical Computer Science, 134 (1994), pp. 27–50.
7. W. Burnside, *On an unsettled question in the theory of discontinuous groups*, Q. J. Pure and Appl. Mathematics, 33 (1902), pp. 230–238.
8. S. Burris and H. P. Sankappanavar, *A Course in Universal Algebra*, Springer-Verlag, New York, 1981.
9. J. H. Conway, *Regular Algebra and Finite Machines*, Mathematics Series (R. Brown and J. De Wet eds.), Chapman and Hall, London, United Kingdom, 1971.

10. J. Gunawardena (ed.), *Idempotency*, Publications of the Newton Institute 11, Cambridge University Press, 1998.
11. ———, *An introduction to idempotency*, in [10], pp. 1–49.
12. K. Hashiguchi, *Limitedness theorem on finite automata with distance functions*, J. Comput. System Sci., 24 (1982), no. 2, pp. 233–244.
13. ———, *Regular languages of star height one*, Information and Control, 53 (1982), pp. 199–210.
14. ———, *Algorithms for determining relative star height and star height*, Information and Computation, 78 (1988), pp. 124–169.
15. D. Krob, *The equality problem for rational series with multiplicities in the tropical semiring is undecidable*, International J. Algebra Computation, 4 (1994), pp. 405–425.
16. ———, *Some consequences of a Fatou property of the tropical semiring*, Journal of Pure and Applied Algebra, 93 (1994), pp. 231–249.
17. ———, *Some automata-theoretic aspects of min-max-plus semirings*, in [10], pp. 70–79.
18. J.-E. Pin, *Tropical semirings*, in [10], pp. 50–69.
19. A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley and Sons, 1986.
20. I. Simon, *Limited subsets of a free monoid*, in Proceedings of the 19th Annual Symposium on Foundations of Computer Science (Ann Arbor, Mich., 1978), IEEE Computer Society, 1978, pp. 143–150.
21. ———, *The nondeterministic complexity of finite automata*, in Mots, M. Lothaire ed., Hermès, 1990, pp. 384–400.
22. U. Zimmermann, *Linear and Combinatorial Optimization in Ordered Algebraic Structures*, Annals of Discrete Mathematics 10, North-Holland, 1981.

Type Isomorphisms and Proof Reuse in Dependent Type Theory

Gilles Barthe[1] and Olivier Pons[2]

[1] INRIA Sophia-Antipolis, France Gilles.Barthe@inria.fr
[2] Departamento de Informática, Universidade do Minho, Portugal
olivier@lmf.di.uminho.pt

Abstract. We propose a theoretical foundation for proof reuse, based on the novel idea of a computational interpretation of type isomorphisms.

1 Introduction

Background. Proof development systems based on dependent type theory, such as Coq [4], Lego [29] and PVS [38], are increasingly used for large-scale formalisations of mathematics and programming languages semantics. These formalisations are usually available on-line and, in some instances, form an integral part of the system's distribution. In principle, these formalisations should allow users to benefit from a large library of basic results upon which to build up their own developments. In practice, users often face difficulties in adapting existing formalisations to their problem, leading to duplicate work and un-integrated contributions. Thus it is important to provide techniques and tools that facilitate proof reuse. However, existing techniques, which we detail below, fall short of providing a satisfactory answer to the problem.

Type isomorphisms provide a type-theoretical notion of equivalence between (generally simple) types: in a nutshell, a type isomorphism between A and B is a pair of expressions $f : A \to B$ and $g : B \to A$ that are mutually inverse to each other (usually w.r.t. $\beta\eta$-convertibility). Type isomorphisms have been used for a number of purposes, including program reuse [21,22,34,35], but the potential of type isomorphisms in proof development systems remains largely unexplored.

This article. The purpose of this paper is two-fold:

1. to highlight a number of situations where (a computational interpretation of) type isomorphisms can be used to good effect for proof reuse. More precisely, we suggest how, in the context of dependent type theory, type isomorphisms (1) enhance the usability of generic frameworks (2) allow to switch between equivalent representations of mathematical theories or data structures. En passant we also show how type isomorphisms are useful in the context of programming with dependent types;

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 57–71, 2001.

2. to propose an extension of dependent type theory where type isomorphisms are given a prominent rôle through computational rules. More precisely, we define an extension of dependent type theory in which type isomorphisms are captured by rewriting rules at the levels of types and elements. We show that the extension is well-behaved, i.e. enjoys all standard properties required for proof-checking, including subject reduction, consistency and decidable type-checking (modulo strong normalisation, which is not proved here).

Our broad conclusion is that type isomorphisms are useful in a number of situations (mostly connected to proof reuse) where traditional dependent type theory and its extensions are too weak.

Before proceeding any further, let us dispose of a red herring: in general, type isomorphisms are studied in extensional type theories and their existence is undecidable. Here we do not need extensionality because we enforce isomorphisms through new rewrite rules. As to undecidability, it simply reflects itself in the fact that our rewrite rules are not "complete ", and that it is possible to introduce new rewrite rules that preserve the good properties of the calculus.

Related work. Our work is original in that it suggests, for the first time to our best knowledge, to give a computational interpretation to (a class of) type isomorphisms. However, our work fits in a series of efforts to lay theoretical foundations for proof reuse in dependent type theory.

Subtyping. Subtyping [16,17] is a basic mechanism to enhance proof reuse in type systems. Specific forms of subtyping connected to reusability include record subtyping [11] and constructor subtyping [8] but these forms of subtyping are too specialised to provide a general solution to the problem of reusability.

Implicit coercions. Implicit coercions [3,5,28,36] provide a very powerful mechanism that subsumes a variety of previous and seemingly unrelated proposals in the area of subtyping. In a nutshell, a coercion from A to B is a function $f : A \to B$, declared in a *coercion context*, and which allows to view a as an element of B and a shorthand for $f\ a$ whenever $a : A$. While the initial idea is relatively simple, its realization involves a number of difficult concepts:

- coherence of coercions: a set of coercions is coherent if the graph of coercions it generates, notably by composition and instantiation, is such that, for every types A and B, any two edges (coercions) from A to B are convertible. Coherence is undecidable in general, and we currently lack of criteria to decide whether a set of coercions is indeed coherent. Thus current implementations of coercive subtyping, and in particular [4,13], do not check coherence to the detriment of conceptual clarity;
- computational interpretation of coercions: in most existing works, coercions are not given a computational meaning through rewrite rules. Instead, the meaning of coercive subtyping is given in a logical framework with equality judgements, as in [28], or by a translation of the extended language with

coercions into the original language without coercions, as in [3,36]. We prefer to follow the usual approach where expressions are given a computational interpretation independently of typing or of another system;
– back-and-forth coercions: the examples of this paper involve back-and-forth coercions, i.e. pairs of coercions $f : A \to B$ and $g : B \to A$ that coexist in a single coercion context. However, most works on implicit coercions do not consider such back-and-forth coercions. Moreover, in case back-and-forth coercions are considered the coercions f and g are required to be mutually inverse, which is not the case for our examples, as we work in a type theory without extensionality (η-conversion).

Thus our work may be viewed as contributing to the understanding of implicit coercions by suggesting (1) the usefulness of back-and-forth coercions (2) a novel computational interpretation of coercions.

Proof transformation. While the approaches presented above involve modifying the type system, several authors have developed methodologies to modify and adapt proofs to another context. Existing techniques include proof by analogy, see e.g. [20], proof by generalisation, see e.g. [23,25,27,33], or proof by transformation, see e.g. [30]. These solutions are appealing in that they do not involve modifying the type system and can address a wealth of problems that range beyond the issues considered in this paper. However, these approaches are not always implemented, can be heavy to put in practice and/or can yield large proof terms.

Type-theoretical preliminaries. The type theory considered in this paper is essentially the Calculus of Inductive Constructions, see e.g. [39]. We use **Prop**, **Set** and **Class** to denote the universes of sets, propositions and classes respectively—usually these are called **Prop**, $\mathbf{Type}_0$ and $\mathbf{Type}_1$ respectively. We let $\triangle$ range over universes.

In addition, we use $\mathbb{N}$ to denote the inductive type of natural numbers (with constructors 0 and S). Besides, we use $\{\mathsf{l} : A,\ \mathsf{l}' : A'\}$ for denoting record types, $\{\mathsf{l} = a,\ \mathsf{l}' = a'\}$ for denoting records, the standard dot notation to denote field selection and $.\langle . := . \rangle$ to denote substitution. Finally, we let $\doteq$ denote Leibniz equality.

We conclude with a warning on naming: most of the type isomorphisms introduced in this paper involve the generation of new constructor names for inductive types or label names for records. In this paper, we gloss over the way such labels can be generated and fixed uniquely.

Contents. The remaining of the paper is organised as follows: in Section 2, we illustrate a number of situations where type isomorphisms can be used for proof reuse. For each example, we suggest an extension of the type theory. Properties of the extensions are discussed in Section 3. Finally, we conclude in Section 4.

2 Motivating Examples

In this section, we introduce a number of examples where a computational interpretation of type isomorphisms allows users to switch between different representations of the same object/structure. Before delving into the examples, let us emphasise that the first three examples arise from our previous and ongoing work on the formalisation of mathematics and programming language semantics, and that in all cases, we were overwhelmed by the burden of switching between existing representations manually.

2.1 Universal Algebra

Universal algebra, see e.g. [18], is a generic framework for the study of algebraic structures. As such, it provides an appealing structuring mechanism for developing a large body of algebra in type theory. Unfortunately, instantiating universal algebra to a concrete algebraic theory is problematic. Definitions are only isomorphic, but not convertible to the ones we would expect. Below we show how type isomorphisms solve the problem.

Setting. In this paragraph, we briefly review how to formalise (some representative notions of) universal algebra in type theory. Recall that the framework is parametrised over the notion of signature. Informally, a signature consists of a set F of function symbols, and of a map $\mathsf{ar} : F \to \mathbb{N}$ which assigns its arity to each function symbol.

Definition 1. *The type* SIGNATURE *of* signatures *is defined by*

$$\textsc{Signature} : \mathbf{Class} = \{\mathsf{fun} : \mathbf{Set},\ \mathsf{ar} : \mathsf{fun} \to \mathbb{N}\}$$

Our representation is the standard one but there are alternatives formalisations that only focus on finite signatures, see e.g. [14].

Each signature has an associated notion of algebra. Recall that an algebra for a signature $S = (F, \mathsf{ar})$ consists of a set A, called the carrier of the algebra, and of a function $f : A^{\mathsf{ar}(f)} \to A$ for every $f \in F$. To formalise algebras, we therefore need to formalise sets and n-ary functions. For the sake of simplicity, we formalise sets as types and not as setoids [7]. The type of n-ary functions over a type A is denoted by FUN n A and is defined by the recursive equation

$$\begin{aligned}\textsc{Fun}[n : \mathbb{N}][A : \mathbf{Set}]\ &:\ \mathbf{Set}\\ &= \mathsf{case}\ n\ \mathsf{of}\quad 0\ \ \Rightarrow A\\ &\qquad\qquad \mid\ S\ m \Rightarrow A \to (\textsc{Fun}\ m\ A)\end{aligned}$$

Definition 2. *The type* ALGEBRA *of* algebras over a signature *is defined by*

$$\begin{aligned}\textsc{Algebra}[S : \textsc{Signature}]\ &:\ \mathbf{Class}\\ &= \{\mathsf{el} : \mathbf{Set},\ \mathsf{int} : \Pi f : (S \cdot \mathsf{fun}).\ \textsc{Fun}\ (S \cdot \mathsf{ar}\ f)\ \mathsf{el}\}\end{aligned}$$

The formalisation may be developed further, for example by introducing the notion of substructure, quotient... Finally some techniques such as reflection, see e.g. [9], require to define the type of terms of a signature.

Definition 3. *The type* Term *of* terms *is defined as*

$$\begin{aligned}&\textbf{Inductive}\ \textsc{Term}[S : \textsc{Signature}] : \textbf{Set}\\ &\quad = \mathit{vterm} : \mathbb{N} \to (\textsc{Term}\ S)\\ &\quad \mid \mathit{fterm} : \Pi f : (S \cdot \mathsf{fun}).\ (\textsc{Fun}\ (S \cdot \mathsf{ar}\ f)\ (\textsc{Term}\ S))\end{aligned}$$

Note that the type of *fterm* does not fit the official definition of constructor type but it is safe to extend the definition of constructor type for the above definition to be considered as legal (see [15] for an account of a similar mechanism). An alternative would be to use the type $(\textsc{Vec}\ (S \cdot \mathsf{ar}\ f)\ (\textsc{Term}\ S)) \to (\textsc{Term}\ S)$ instead of $\textsc{Fun}\ (S \cdot \mathsf{ar}\ f)\ (\textsc{Term}\ S)$ but it would introduce unnecessary technicalities in our presentation.

Problem. Now assume that we want to instantiate the framework to a specific signature, say the signature of groups defined below.

Definition 4. *The signature* GroupSig *is defined by*

$$\textsc{GroupSig} = \{\mathsf{fun} = \textsc{GroupSym},\ \mathsf{ar} = \textsc{GroupAr}\}$$

where

$$\begin{aligned}&\textbf{Inductive}\ \textsc{GroupSym} :\ \textbf{Set}\\ &\qquad = \textsc{o} : \textsc{GroupSym} \mid \textsc{e} : \textsc{GroupSym} \mid \textsc{i} : \textsc{GroupSym}\end{aligned}$$

and

$$\begin{aligned}&\textsc{GroupAr}[f : \textsc{GroupSym}] :\ \mathbb{N}\\ &\qquad = \mathsf{case}\ f\ \ \mathsf{of}\ \textsc{o} \Rightarrow 2\\ &\qquad\qquad \mid\ \textsc{e} \Rightarrow 0\\ &\qquad\qquad \mid\ \textsc{i} \Rightarrow 1\end{aligned}$$

The type of group algebras, i.e. Algebra GroupSig, may be evaluated to

$$\{\mathsf{el} : \textbf{Set},\ \mathsf{int} : \Pi f : \textsc{GroupSig}.\ \textsc{Fun}\ (\textsc{GroupAr}\ f)\ \mathsf{el}\}$$

and there is no further way to proceed. This is bad news, as one would prefer to use the more standard and palatable representation

$$\{\mathsf{el} : \textbf{Set},\ \mathsf{oint} : \mathsf{el} \to \mathsf{el} \to \mathsf{el},\ \mathsf{eint} : \mathsf{el}, \mathsf{iint} : \mathsf{el} \to \mathsf{el}\} \qquad (*)$$

Similarly, the type of group terms, i.e. Term GroupSig, may be evaluated to

$$\begin{aligned}&\textbf{Inductive}\ \textsc{TermGroupSig} : \textbf{Set}\\ &\quad = \mathit{vtermGroupSig} : \mathbb{N} \to \textsc{TermGroupSig}\\ &\quad \mid \mathit{ftermGroupSig} : \Pi f : \textsc{GroupSym}.\ \textsc{Fun}\ (\textsc{GroupAr}\ f)\ \textsc{TermGroupSig}\end{aligned}$$

(observe the renaming of constructors, over which we gloss here) whereas one would prefer to use the more standard and palatable representation

$$\begin{array}{l}\textbf{Inductive}\ \textsc{TermGroup} : \textbf{Set}\\ \quad = \mathit{vtermgroup} : \mathbb{N} \to \textsc{TermGroup}\\ \quad \mid \mathit{ftermgroupo} : \textsc{TermGroup} \to \textsc{TermGroup} \to \textsc{TermGroup}\\ \quad \mid \mathit{ftermgroupe} : \textsc{TermGroup}\\ \quad \mid \mathit{ftermgroupi} : \textsc{TermGroup} \to \textsc{TermGroup}\end{array}$$

Solution. One may achieve a simple solution to the problem by forcing suitable type isomorphisms. Indeed, consider the type of group algebras. The type

$$\Pi f : \textsc{GroupSig}.\ \textsc{Fun}\ (\textsc{GroupAr}\ f)\ A$$

is inhabited by functions over the 3-elements type GroupSig. Now, such functions can alternatively be described by triples assigning a value to each element of GroupSig, i.e. by inhabitants of the type
$\{\mathsf{oint} : \textsc{Fun}\ (\textsc{GroupAr}\ \textsc{o})\ A,\ \mathsf{eint} : \textsc{Fun}\ (\textsc{GroupAr}\ \textsc{e})\ A, \mathsf{iint} :$
$\textsc{Fun}\ (\textsc{GroupAr}\ \textsc{i})\ A\}$
which reduces to

$$\{\mathsf{oint} : A \to A \to A,\ \mathsf{eint} : A, \mathsf{iint} : A \to A\}$$

The approach advocated in this paper is to enforce an equational rule that relates dependent function spaces over an enumeration type to record types that collect the value of the function for each element of the enumeration type. More precisely, if X_n is an enumeration type with inhabitants $\bullet_1, \ldots, \bullet_n$, then we introduce the equational rule

$$\Pi f : \mathrm{X}_n.\ T \quad =_\Sigma \quad \{\mathsf{lab}_1 : T\langle f := \bullet_1\rangle,\ \ldots\ ,\ \mathsf{lab}_n : T\langle f := \bullet_n\rangle\}$$

for some previously agreed upon set of labels $\mathsf{lab}_1,\ \ldots\ ,\ \mathsf{lab}_n$. This equational rule is integrated into the conversion rule so as to allow to switch between the two representations. As a result, we introduce non-canonical inhabitants at function and record types. For example, Σ-conversion makes it possible to apply a record to an argument, or to select a field of a function! For example, taking G : Algebra GroupSig one can type $(G \cdot \mathsf{int})$ o, whereas $(G \cdot \mathsf{int})$ is of a record type. We therefore need to introduce new computational rules for such cases

$$\begin{array}{l}\{\mathsf{lab}_1 = f_1,\ \ldots\ , \mathsf{lab}_n = f_n\}\ \bullet_i \to_\sigma f_i\\ \qquad\qquad (\lambda f : \mathrm{X}_n.e) \cdot \mathsf{lab}_i \to_\sigma e\langle f := \bullet_i\rangle\end{array}$$

Coming back to the definition of group algebras, the new reduction relation enforces the type Algebra GroupSig to reduce to

$$\{\mathsf{el} : \textbf{Set},\ \mathsf{int} : \{\mathsf{oint} : \mathsf{el} \to \mathsf{el} \to \mathsf{el},\ \mathsf{eint} : \mathsf{el}, \mathsf{iint} : \mathsf{el} \to \mathsf{el}\}\}$$

which closely resembles $(*)$: in fact, one obtains $(*)$ simply by removing the nesting of records. In the next example, we show how new computational rules

allow to flatten records, thus forcing the two above types to be convertible with the usual representation of group algebras.

Let us now go back to the type of group terms. By applying the Σ-rule suggested above, one can rewrite TermGroupSig to

$$\begin{aligned}&\textbf{Inductive } \textsc{TermGroupSig} : \textbf{Set}\\ &\quad = \mathit{vtermGroupSig} : \mathbb{N} \to \textsc{TermGroupSig}\\ &\quad \mid \mathit{ftermGroupSig} :\\ &\qquad \{\, \mathsf{oint} : \textsc{TermGroupSig} \to \textsc{Term GroupSig} \to \textsc{TermGroupSig},\\ &\qquad\ \ \mathsf{eint} : \textsc{TermGroupSig},\\ &\qquad\ \ \mathsf{iint} : \textsc{TermGroupSig} \to \textsc{TermGroupSig}\}\end{aligned}$$

Clearly, the above expression is ill-formed as the type of *ftermGroupSig* does not fit the definition of constructor type. However, it is easy to recover a well-formed definition, and in fact the expected one, by splitting the second constructor into three. Formally, we adopt the compatibility rule:

$$\frac{T =_\Sigma \{\mathsf{lab}_1 : T_1,\ \ldots\ , \mathsf{lab}_n : T_n\}}{\begin{array}{c}\textbf{Inductive } I : A = \ \ldots \mid c : T \mid \ \ldots\\ =_\Sigma\\ \textbf{Inductive } I : A = \ \ldots \mid c_1 : T_1 \mid \ \ldots \mid c_n : T_n \mid \ \ldots\end{array}}$$

instead of the rule of compatible closure of Σ-equality for inductive types.

2.2 Switching between Mathematical Theories

In a recent article [32], R. Pollack surveys the different existing mechanisms to represent mathematical structures in type theory. Record types are one well-known such mechanism. However, as pointed out in *loc. cit.*, record types come in several flavours, including left-associating and right-associating records. All of these approaches are extensively used in the formalisation of mathematics.

Setting. We briefly compare left-associating and right-associating records in the context of Partial Equivalence Relations (PERs). PERs may be defined as flat, right-associative or left-associative records.

Definition 5.

1. *The* flat construction of Per *can be defined by*

$$\begin{aligned}&\textsc{Per} \ : \ \textbf{Class}\\ &\quad = \{\mathsf{S} : \textbf{Set},\ \mathsf{R} : \mathsf{S} \to \mathsf{S} \to \textbf{Prop},\ \mathsf{Sym} : (\textsc{sym}\ \mathsf{S}\ \mathsf{R}),\ \mathsf{Trans} : (\textsc{trans}\ \mathsf{S}\ \mathsf{R})\}\end{aligned}$$

2. *The* right-associative construction of Per *is defined by*

$$\begin{aligned}\textsc{Inner}[S : \textbf{Set}][R : S \to S \to \textbf{Prop}] : \textbf{Prop}\\ = \{\mathsf{Sym} : (\textsc{sym}\ S\ R),\ \mathsf{Trans} : (\textsc{trans}\ S\ R)\}\\ \textsc{Middle}[S : \textbf{Set}] : \textbf{Set} = \{\mathsf{R} : S \to S \to \textbf{Prop},\ \mathsf{i} : (\textsc{Inner}\ S\ \mathsf{R})\}\\ \textsc{Per}_r : \textbf{Class} = \{\mathsf{S} : \textbf{Set},\ \mathsf{r} : \textsc{Middle}\ \mathsf{S}\}\end{aligned}$$

3. The left-associative construction of PER *is defined by*

$$\begin{aligned}\textsc{Relation} &: \mathbf{Class} = \{\mathsf{S} : \mathbf{Set},\ \mathsf{R} : \mathsf{S} \to \mathsf{S} \to \mathbf{Prop}\}\\ \textsc{SymRel} &: \mathbf{Class} = \{\mathsf{P} : \textsc{Relation},\ \mathsf{Sym} : \textsc{sym}\ (\mathsf{P} \cdot \mathsf{S})\ (\mathsf{P} \cdot \mathsf{R})\}\\ \textsc{Per}_l &: \mathbf{Class} = \{\mathsf{SR} : \textsc{SymRel},\ \mathsf{Trans} : \textsc{trans}\ (\mathsf{SR}\cdot\mathsf{P}\cdot\mathsf{S})\ (\mathsf{SR}\cdot\mathsf{P}\cdot\mathsf{R})\}\end{aligned}$$

Problem. As noted by R. Pollack, all approaches have their own advantages and drawbacks. Right-associative constructions are easy to specialise but hard to extend. Conversely, left-associative constructions are easy to extend but hard to specialise. Both specialisation and extensibility are important and thus all approaches are extensively used in practice. It is therefore important to be able to switch between the three representations.

Solution. One may solve the problem by enforcing suitable equational rules that flatten record types. In order to avoid mind-boggling renaming problems with labels, we formalise these rules using Coq's view of record types as inductive types with a single constructor. The rule then becomes a

$$\begin{aligned}&\mathbf{Inductive}\ I : \triangle = c : \Pi \boldsymbol{x}\!:\!\boldsymbol{A}.\ \Pi y\!:\!(\mathbf{Inductive}\ J : \triangle' = d : \Pi \boldsymbol{z}\!:\!\boldsymbol{B}.\ J).\\ &\quad \Pi \boldsymbol{x'}\!:\!\boldsymbol{A'}.\ I\\ =_\Sigma\ &\mathbf{Inductive}\ I' : \triangle = c' : \Pi \boldsymbol{x}\!:\!\boldsymbol{A}.\ \Pi \boldsymbol{z}\!:\!\boldsymbol{B}.\ \Pi \boldsymbol{x'}\!:\!\boldsymbol{A'}\langle \mathrm{y} := d\ \boldsymbol{z}\rangle.\ I'\end{aligned}$$

As before the equational rule at the type level needs to be accompanied by rewrite rules at the object level. The rules allow to reduce case-expressions whose argument is not in the right shape.

$$\begin{aligned}&\mathsf{case}\ (c\ \boldsymbol{M}\ (d\ \boldsymbol{N})\ \boldsymbol{M'})\ \mathsf{of}\ \{c' \Rightarrow e\} \to_\sigma \mathsf{case}\ (c'\ \boldsymbol{M}\ \boldsymbol{N}\ \boldsymbol{M'})\ \mathsf{of}\ \{c' \Rightarrow e\}\\ &\quad \mathsf{case}\ (c'\ \boldsymbol{M}\ \boldsymbol{N}\ \boldsymbol{M'})\ \mathsf{of}\ \{c \Rightarrow e\} \to_\sigma \mathsf{case}\ (c\ \boldsymbol{M}\ (d\ \boldsymbol{N})\ \boldsymbol{M'})\ \mathsf{of}\ \{c \Rightarrow e\}\end{aligned}$$

Let us return to the example of PERs and observe the behaviour of the rewrite rules for this example. We focus on left-associating representations, since they are more intricate to handle. Formalising records as inductive types, the left-associating definition of PERs becomes

$$\begin{aligned}&\mathbf{Inductive}\ \textsc{Relation} : \mathbf{Class}\\ &= c_{Rel} : \Pi S : \mathbf{Set}.\ (S \to S \to \mathbf{Prop}) \to \textsc{Relation}\\ &\mathbf{Inductive}\ \textsc{SymRel}\ : \mathbf{Class}\\ &= c_{SymRel} : \Pi P : \textsc{Relation}.\ (\textsc{sym}\ (\textsc{Car}\ P)\ (\textsc{Rel}\ P)) \to \textsc{SymRel}\\ &\mathbf{Inductive}\ \textsc{Per}_l\ : \mathbf{Class}\\ &= c_{Perl} : \Pi SR : \textsc{SymRel}.\ (\textsc{trans}\ (\textsc{Car}\ (\mathrm{RSR}\ SR))\ (\textsc{Rel}\ (\mathrm{RSR}\ SR))) \to \textsc{Per}_l\end{aligned}$$

where

$$\begin{aligned}\textsc{Car} &: \textsc{Relation} \to \mathbf{Set}\\ \textsc{Rel} &: \Pi P : \textsc{Relation}.\ (\textsc{Car}\ P) \to (\textsc{Car}\ P) \to \mathbf{Prop}\\ \mathrm{RSR} &: \textsc{SymRel} \to \textsc{Relation}\end{aligned}$$

are defined in the obvious way. By using Σ-conversion as well as $\beta\iota\beta^+$-reduction, we can derive the flat representation of PERs

$$\begin{aligned}&\mathbf{Inductive}\ \textsc{Per'}_l : \mathbf{Class}\\ &\quad = c_{Perl'} : \Pi S : \mathbf{Set}.\ \Pi R : S \to S \to \mathbf{Prop}.\\ &\qquad\qquad (\textsc{Sym}\ S\ R) \to (\textsc{Trans}\ S\ R) \to \textsc{Per}'_l\end{aligned}$$

2.3 Switching between Equivalent Data Structures

Standard data structures, such as natural numbers or lists, often have several representations. In many cases, these representations serve different purposes: typically, some of them will be well-suited for logical reasoning whereas some others will be well-suited to serve as a basis for efficient algorithms. Below we illustrate some of the issues involved by considering natural numbers, and their representation in unary and binary schemes. Both representations are used for example in J. C. B. Almeida's formalisation of the RSA algorithm [1].

Setting. Natural numbers are traditionally formalised in type systems with the Peano unary representation scheme.

Definition 6. *The type* $\mathbb{N}$ *of unary natural numbers is defined by*

$$\begin{aligned}\textbf{Inductive}\ \mathbb{N}\ : \textbf{Set} = {} & 0 : \mathbb{N} \\ \mid\ & S : \mathbb{N} \to \mathbb{N}\end{aligned}$$

One can also use an alternative binary representation scheme in which a natural number is encoded as 0, $2n+1$ or $2n+2$ (this scheme slightly differs from the Coq encoding, which distinguishes between 0 and positive numbers).

Definition 7. *The type* $\mathbb{B}$ *of binary natural numbers is defined by*

$$\begin{aligned}\textbf{Inductive}\ \mathbb{B} : \textbf{Set} = {} & B_H : \mathbb{B} \\ \mid\ & B_O : \mathbb{B} \to \mathbb{B} \\ \mid\ & B_I\ : \mathbb{B} \to \mathbb{B}\end{aligned}$$

The two representations may be related by translations in both directions. These translations are based on encodings of a datatype's constructors as functions on the other datatype.

Definition 8.

1. *The successor function* $\mathrm{S}^{\mathbb{B}}$ *is defined by the recursive equation*

$$\begin{aligned}\mathrm{S}^{\mathbb{B}}[b : \mathbb{B}] : \mathbb{B} = {} & \mathsf{case}\ b\ \mathsf{of} \quad B_H \Rightarrow B_O\ B_H \\ \mid\ & B_O\ c \Rightarrow B_I\ c \\ \mid\ & B_I\ c \Rightarrow B_O\ (\mathrm{S}^{\mathbb{B}}\ c)\end{aligned}$$

2. *The conversion function* N2B *is defined by the recursive equation*

$$\begin{aligned}\textsc{n2b}[n : \mathbb{N}] : \mathbb{B} = {} & \mathsf{case}\ n\ \mathsf{of} \quad 0 \Rightarrow B_H \\ \mid\ & S\ m \Rightarrow \mathrm{S}^{\mathbb{B}}\ (\textsc{n2b}\ m)\end{aligned}$$

3. *The function* $\mathrm{B}_O^{\mathbb{N}}$ *is defined as*

$$\mathrm{B}_O^{\mathbb{N}}[n : \mathbb{N}] : \mathbb{N} = S\ (\textsc{Twice}\ n)$$

where TWICE *is the multiplication by 2 over* $\mathbb{N}$.

4. *The function* $\mathrm{B}_I^{\mathbb{N}}$ *is defined as*

$$\mathrm{B}_I^{\mathbb{N}}[n : \mathbb{N}] : \mathbb{N} = \textsc{Twice}\ (S\ n)$$

5. *The conversion function* B2N *is defined by the recursive equation*

$$\begin{aligned}\textsc{b2n}[b : \mathbb{B}] : \mathbb{N} := \ \mathsf{case}\ b\ \mathsf{of}\quad & B_H \Rightarrow 0\\ \mid\ & B_O\ c \Rightarrow B_O^{\mathbb{N}}\ (\textsc{b2n}\ c)\\ \mid\ & B_I\ c \Rightarrow B_I^{\mathbb{N}}\ (\textsc{b2n}\ c)\end{aligned}$$

Problem. We would like to use interchangeably both representations of natural numbers. In particular, we would like to derive the correctness of RSA on binary numbers from the correctness of RSA on unary numbers.

Solution. One may solve the problem by forcing the types $\mathbb{N}$ and $\mathbb{B}$ to be convertible by adding a new Σ-conversion rule

$$\mathbb{N} \quad =_\Sigma \quad \mathbb{B}$$

The new conversion rule introduces non-canonical inhabitants at each type. For example, it makes it possible to do a $\mathbb{N}$-case analysis on an inhabitant of $\mathbb{B}$ and vice-versa! We therefore need to cater for such cases, by introducing new computational rules which avoid "stuck redexes", in this case a case-expression applied to a term whose head symbol is a constructor of the "wrong" datatype

$$\begin{aligned}&\mathsf{case}\ F\ \mathsf{of}\ \{E_N\} \rightarrow_\sigma \mathsf{case}\ (\textsc{b2n}\ F)\ \mathsf{of}\ \{E_N\}\\ &\mathsf{case}\ G\ \mathsf{of}\ \{E_B\} \rightarrow_\sigma \mathsf{case}\ (\textsc{n2b}\ G)\ \mathsf{of}\ \{E_B\}\end{aligned}$$

where F is either B_H, $B_O\ M$ or $B_I\ M$ and G is either 0 or $S\ M$ and

$$\begin{aligned}E_N &= 0 \Rightarrow f_0 \mid S\ x \Rightarrow f_s\\ E_B &= B_H \Rightarrow f_H \mid B_O\ x \Rightarrow f_O \mid B_I\ x \Rightarrow f_I\end{aligned}$$

Let us now turn to closed terms in normal form. For every natural number n, its unary and binary encodings $\lceil n \rceil^{\mathbb{N}}$ and $\lceil n \rceil^{\mathbb{B}}$ are normal closed inhabitants of $\mathbb{N}$ and $\mathbb{B}$. Using the equality $(\mathsf{case}\ x\ \mathsf{of}\ 0 \Rightarrow 0 \mid S\ x \Rightarrow S\ x) \doteq x$, one can prove that $\lceil n \rceil^{\mathbb{N}} \doteq \lceil n \rceil^{\mathbb{B}}$ for every natural number n. Thus, our calculus does not have the so-called equality reflection property. This property may be recovered by introducing further equational rules $\lceil n \rceil^{\mathbb{N}} \leftrightarrow_\varsigma \lceil n \rceil^{\mathbb{B}}$. Embedding these equational rules in the conversion rule of the type system does not affect the decidability of type-checking.

2.4 Expressiveness of Programming Languages with Dependent Types

Our last example, which is not connected to proof reuse, shows how type isomorphisms can also be used in situations where the standard convertibility relation is too weak. Incidentally the following example, which deals with dependent lists (a.k.a. vectors), shows how a dependently typed programming language such as Cayenne [2], a dialect of Haskell that extends the Calculus of Constructions with full recursion, could be strengthened by using features from DML [40], a dependently typed extension of ML with a restricted form of dependent types.

Setting. Recall that vectors are formalised via the family $\textsc{vec} : \mathbb{N} \to \mathbf{Set} \to \mathbf{Set}$ which associates to every natural number n and type A the type of vectors of elements of A with length n. One can define vector concatenation and vector inversion with the types

$$\textsc{app} : \Pi m, n : \mathbb{N}.\ \Pi A : \mathbf{Set}.\ (\textsc{vec}\ m\ A) \to (\textsc{vec}\ n\ A) \to (\textsc{vec}\ (\textsc{plus}\ m\ n)\ A)$$
$$\textsc{rev} : \Pi m : \mathbb{N}.\ \Pi A : \mathbf{Set}.\ (\textsc{vec}\ m\ A) \to (\textsc{vec}\ m\ A)$$

The problem. In the context

$$m : \mathbb{N},\ l : \textsc{vec}\ m\ A,\ n : \mathbb{N},\ l' : \textsc{vec}\ n\ A$$

one would expect to be able to prove

$$\textsc{rev}\ (\textsc{plus}\ m\ n)\ A\ (\textsc{app}\ m\ n\ A\ l\ l') \doteq \textsc{app}\ n\ m\ A\ (\textsc{rev}\ n\ A\ l')\ (\textsc{rev}\ m\ A\ l)$$

Unfortunately, the above is not well-typed since:

- the left-hand side of the equality has type $\textsc{vec}\ (\textsc{plus}\ m\ n)\ A$;
- the right-hand side of the equality has type $\textsc{vec}\ (\textsc{plus}\ n\ m)\ A$;
- the two types are not convertible in general (since $\textsc{plus}\ m\ n$ and $\textsc{plus}\ n\ m$ are Leibniz equal but not convertible in general).

Solution. We impose a type isomorphism through the new equational rule

$$\textsc{vec}\ (\textsc{plus}\ m\ n)\ A \quad =_\Sigma \quad \textsc{vec}\ (\textsc{plus}\ n\ m)\ A$$

Such a use of type isomorphisms does not require the introduction of new computational rules at the term level, intuitively because $\textsc{plus}\ m\ n \doteq \textsc{plus}\ n\ m$ for every $m, n : \mathbb{N}$. We do not develop this example further, since a neat treatment of convertibility-checking and type-checking algorithms for this example requires to treat PLUS as a constant function symbol specified by pattern-matching, see e.g. [12], and thus to extend the type theory even further.

3 The Type Theory and Its Properties

3.1 The Extended Type System

The extended type system is very closely related to the Calculus of Inductive Constructions: it shares the same set of terms, the same notion of substitution, the same notion of constructor type, inductive definition, guarded recursive definitions... The sole difference with Calculus of Inductive Constructions is the computational behaviour of expressions. In addition to β-reduction, β^+-reduction and ι-reduction, which respectively account for the computational interpretation of functions, guarded recursive definitions and case-expressions, we consider the rules of σ-reduction displayed in Figure 1 and the equational rules of Σ-equality displayed in Figure 2; for the latter, special provisions in the compatibility rules are enforced so as to avoid the problems suggested in Subsection 2.1. These are embedded in the conversion rule, which becomes

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \quad A =_{\beta\beta^+\iota\sigma\Sigma} B$$

$$
\begin{array}{|r|}
\hline
\{\mathsf{lab}_1 = f_1,\ \ldots\ , \mathsf{lab}_n = f_n\}\ \bullet_i \to_\sigma f_i \\
(\lambda f : \mathrm{X}_n.e) \cdot \mathsf{lab}_i \to_\sigma e\langle f := \bullet_i \rangle \\
\hline
\mathsf{case}\ (c\ \boldsymbol{M}\ (d\ \boldsymbol{N})\ \boldsymbol{M'})\ \mathsf{of}\ \{c' \Rightarrow e\} \to_\sigma \mathsf{case}\ (c'\ \boldsymbol{M}\ \boldsymbol{N}\ \boldsymbol{M'})\ \mathsf{of}\ \{c' \Rightarrow e\} \\
\mathsf{case}\ (c'\ \boldsymbol{M}\ \boldsymbol{N}\ \boldsymbol{M'})\ \mathsf{of}\ \{c \Rightarrow e\} \to_\sigma \mathsf{case}\ (c\ \boldsymbol{M}\ (d\ \boldsymbol{N})\ \boldsymbol{M'})\ \mathsf{of}\ \{c \Rightarrow e\} \\
\hline
\mathsf{case}\ F\ \mathsf{of}\ \{E_N\} \to_\sigma \mathsf{case}\ (\textsc{b2n}\ F)\ \mathsf{of}\ \{E_N\} \\
\mathsf{case}\ G\ \mathsf{of}\ \{E_B\} \to_\sigma \mathsf{case}\ (\textsc{n2b}\ G)\ \mathsf{of}\ \{E_B\} \\
\hline
\end{array}
$$

where $F = B_H \mid B_O\ M \mid B_I\ M$ and $G = 0 \mid S\ M$ and

$$
\begin{array}{l}
E_N = 0 \Rightarrow f_0 \mid S\ x \Rightarrow f_s \\
E_B = B_H \Rightarrow f_H \mid B_O\ x \Rightarrow f_O \mid B_I\ x \Rightarrow f_I
\end{array}
$$

Fig. 1. Rules for σ-reduction

$$
\begin{array}{|c|}
\hline
\Pi f : \mathrm{X}_n.\ T =_\Sigma \{\mathsf{lab}_1 : T\langle f := \bullet_1 \rangle,\ \ldots\ ,\ \mathsf{lab}_n : T\langle f := \bullet_n \rangle\} \\
\mathbb{N} =_\Sigma \mathbb{B} \\
\hline
\mathbf{Inductive}\ I : \triangle = c : \Pi \boldsymbol{x}\!:\!\boldsymbol{A}.\ \Pi y\!:\!(\mathbf{Inductive}\ J : \triangle' = d : \Pi \boldsymbol{z}\!:\!\boldsymbol{B}.\ J).\ \Pi \boldsymbol{x'}\!:\!\boldsymbol{A'}.\ I \\
=_\Sigma\ \ \mathbf{Inductive}\ I' : \triangle = c' : \Pi \boldsymbol{x}\!:\!\boldsymbol{A}.\ \Pi \boldsymbol{z}\!:\!\boldsymbol{B}.\ \Pi \boldsymbol{x'}\!:\!\boldsymbol{A'}\langle \mathrm{y} := d\ \boldsymbol{z} \rangle.\ I' \\
\hline
\end{array}
$$

Fig. 2. Rules for Σ-equalities

3.2 Properties of the Extended Type System

First, the extended reduction relation is confluent. Indeed, the new σ-reduction rules are formulated in such a way that there are no critical pairs so the reduction relation is orthogonal and hence confluent [26].

Proposition 1. *$\beta\beta^+\iota\sigma$-reduction is confluent.*

Second, the calculus enjoys subject reduction.

Proposition 2. *If $\Gamma \vdash e : A$ and $e \to_{\beta\beta^+\iota\sigma} e'$ then $\Gamma \vdash e' : A$.*

Third, the calculus enjoys decidable type-checking, provided every legal term is strongly normalising. A key argument in the proof is that convertibility between legal types is decidable, see e.g. [6,31,37] for a recent survey of type-checking algorithms for dependent type theory.

Proposition 3. *If every legal term is normalising, then it is decidable whether $\Gamma \vdash e : A$ is derivable.*

Finally, the calculus is consistent. This can be established by a standard model construction, e.g. the proof-irrelevance model of [19].

Proposition 4. *There is no M such that $A : \mathbf{Prop} \vdash M : A$.*

We conjecture, but do not prove, that every legal term is $\beta\beta^+\iota\sigma$-strongly normalising. The most direct way to prove the conjecture seems to adapt the model constructions of e.g. [24,39]. An alternative would be to define a reduction-preserving translation from the extended type theory to the original system; however it is unclear to the authors on how to achieve such a result (whose interest goes beyond strong normalisation).

4 Conclusion

We have proposed a computational interpretation of type isomorphisms and detailed a number of situations in which such a mechanism allows proof reuse. For these examples, we have shown, up to the conjecture of strong normalisation, that this mechanism extends safely the Calculus of Inductive Constructions.

Our work suggests a new approach to (some form of) proof reuse and contributes to the understanding of implicit coercions. Yet much work remains to be done:

- from a practical perspective, it seems important to integrate the techniques proposed in this paper into a proof development system such as Coq, and to understand the interactions between our approach and the proof transformation techniques of [30,33];
- from a theoretical perspective, the outstanding question left unaddressed in this paper is the normalisation of the extended reduction relation. Besides, a number of technical developments seem much desirable. First, we are interested in understanding how the approach developed in this paper might be generalised (1) to a larger class of isomorphisms of types (2) beyond isomorphisms of types. W.r.t. the latter, one may drop the symmetry of isomorphisms of types and consider examples with unidirectional coercions. This would lead us to a calculus where implicit coercions determine a subtyping relation between datatypes and/or record types.
  In a different direction, it is also of interest to spell out the connections between our use of type isomorphisms in the example of vectors and H. Xi and F. Pfenning's use of constraints in DML [40]. This requires to understand under which circumstances the example of vectors, which does not require new computational rules at the term level, can be generalised.

Acknowledgements. The authors are grateful to Yves Bertot, Venanzio Capretta, Nicolas Magaud, Femke van Raamsdonk and the anonymous referees for their constructive comments on the paper. Olivier Pons is supported by the Portuguese Science Foundation (Fundação para a Ciencia e a Technologia) under the Fellowship PRAXIS-XXI/BPD/22108/99. The authors also acknowledge support from the cooperation program ICCTI/INRIA.

References

1. J. C. B. Almeida. A formalization of RSA in Coq, July 1999. Available from http://logica.di.uminho.pt/CryptoCoq/index.html.
2. L. Augustsson. Cayenne: A language with dependent types. In *Proceedings of ICFP'98*, pages 239–250. ACM Press, 1998.
3. A. Bailey. *The Machine-Checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, University of Manchester, 1998.
4. B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, P. Loiseleur, C. Muñoz, C. Murthy, C. Parent-Vigouroux, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant User's Guide. Version 6.3.1*, December 1999.

5. G. Barthe. Implicit coercions in type systems. In Berardi and Coppo [10], pages 16–35.
6. G. Barthe. Theoretical pearl: type-checking injective pure type systems. *Journal of Functional Programming*, 9:675–698, 1999.
7. G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. Submitted, 2000.
8. G. Barthe and M.J. Frade. Constructor subtyping. In D. Swiestra, editor, *Proceedings of ESOP'99*, volume 1576 of *Lecture Notes in Computer Science*, pages 109–127. Springer-Verlag, 1999.
9. G. Barthe, M. Ruys, and H. Barendregt. A two-level approach towards lean proof-checking. In Berardi and Coppo [10], pages 16–35.
10. S. Berardi and M. Coppo, editors. *Proceedings of TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
11. G. Betarte. *Dependent Record Types and Algebraic Structures in Type Theory*. PhD thesis, Department of Computer Science, Chalmers Tekniska Högskola, 1998.
12. F. Blanqui, J.-P. Jouannaud, and M. Okada. The algebraic calculus of constructions. In P. Narendran and M. Rusinowitch, editors, *Proceedings of RTA'99*, volume 1631 of *Lecture Notes in Computer Science*, pages 301–316. Springer-Verlag, 1999.
13. P. Callaghan and Z. Luo. Plastic: An implementation of typed LF with coercive subtyping and universes. *Journal of Automated Reasoning*, 200x. To appear.
14. V. Capretta. Universal algebra in type theory. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proceedings of TPHOL'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 131–148. Springer-Verlag, 1999.
15. V. Capretta. Recursive families of inductive types. In M. Aagard and J. Harrison, editors, *Proceedings of TPHOLs'00*, volume 1869 of *Lecture Notes in Computer Science*, pages 73–89. Springer-Verlag, 2000.
16. L. Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, March 1996.
17. L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
18. P. Cohn. *Universal algebra*, volume 6 of *Mathematics and its Applications*. D. Reidel, 1981.
19. T. Coquand. Metamathematical investigations of a calculus of constructions. In P. Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic Press, 1990.
20. R. Curien. *Outils pour la preuve par analogie*. PhD thesis, Université de Nancy, 1995.
21. D. Delahaye, R. di Cosmo, and B. Werner. Recherche dans une bibliothèque de preuves Coq en utilisant le type et modulo isomorphismes. In *Proceedings of the PRC/GDR de programmation, Pôle Preuves et Spécifications Algébriques*, November 1997.
22. R. Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhauser, 1995.
23. A. Felty and D. Howe. Generalization and reuse of tactic proofs. In F. Pfenning, editor, *Proceedings of LPAR'94*, volume 822 of *Lecture Notes in Artificial Intelligence*, pages 1–15. Springer-Verlag, 1994.
24. H. Geuvers. A short and flexible proof of strong normalisation for the Calculus of Constructions. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of TYPES'94*, volume 996 of *Lecture Notes in Computer Science*, pages 14–38. Springer-Verlag, 1995.

25. R. W. Hasker and U. S. Reddy. Generalization at higher types. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271. University of Pennsylvania, July 1992. Available as Technical Report MS-CIS-92-86.
26. 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.
27. T. Kolbe and C. Walther. Proof analysis, generalization, and reuse. In W. Bibel and P. H. Schmidt, editors, *Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques*. Kluwer Academic Publishers, 1998.
28. Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9:105–130, February 1999.
29. Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, University of Edinburgh, May 1992.
30. N. Magaud and Y. Bertot. Changements de représentation des structures de données dans Coq: le cas des entiers naturels. In P. Casteran, editor, *Proceedings of JFLA'01*, 2001.
31. R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.
32. R. Pollack. Dependently typed records for representing mathematical structures. In M. Aagard and J. Harrison, editors, *Proceedings of TPHOLs'00*, volume 1869 of *Lecture Notes in Computer Science*, pages 462–479. Springer-Verlag, 2000.
33. O. Pons. *Conception et réalisation d'outils d'aide au développement de grosses théories dans les systèmes de preuves interactifs*. PhD thesis, Conservatoire National des Arts et Métiers, 1999.
34. M. Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.
35. E. J. Rollins and J. M. Wing. Specifications as search keys for software libraries. In K. Furukawa, editor, *Proceedings of ICLP'91*, pages 173–187. MIT Press, June 1991.
36. A. Saibi. Typing algorithm in type theory with inheritance. In *Proceedings of POPL'97*, pages 292–301. ACM Press, 1997.
37. P. Severi. Type inference for pure type systems. *Information and Computation*, 143(1):1–23, May 1998.
38. N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.
39. B. Werner. *Méta-théorie du Calcul des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.
40. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of POPL'99*, pages 214–227. ACM Press, 1999.

On the Duality between Observability and Reachability *

Michel Bidoit[1], Rolf Hennicker[2], and Alexander Kurz[3]

[1] Laboratoire Spécification et Vérification (LSV), CNRS & ENS de Cachan, France
[2] Institut für Informatik, Ludwig-Maximilians-Universität München, Germany
[3] Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands

Abstract. Observability and reachability are important concepts in formal software development. While observability concepts allow to specify the required observable behavior of a program or system, reachability concepts are used to describe the underlying data in terms of data type constructors. In this paper we show that there is a duality between observability and reachability, both from a methodological and from a formal point of view. In particular, we establish a correspondence between observer operations and data type constructors, observational algebras and constructor-based algebras, and observational and inductive properties of specifications. Our study is based on the observational logic institution [7] and on a novel treatment of reachability which introduces the institution of constructor-based logic. The duality between both concepts is formalised in a category-theoretic setting.

1 Introduction

An important role in software specification and program development is played by observability and reachability concepts which deal with different aspects of software systems. While observational approaches focus on the observable properties of a system, reachability notions describe the underlying data manipulated by the system. Both concepts are treated in a formal way in various algebraic specification frameworks.

Considering observability, one can distinguish two main approaches: The first one is based on an observational equivalence relation between algebras which is used to abstract from the (standard) model class of a specification; cf. e.g. [17]. The second approach relaxes the (standard) satisfaction relation so that all algebras are accepted as observational models of a specification which satisfy a given set of axioms up to observational equality of the elements of the algebra. (This idea was originally introduced by Reichel; cf. e.g. [16].) Thereby two elements are considered to be observationally equal if they cannot be distinguished by a set of observable experiments.

Concerning reachability, the standard approach is to introduce a set of data type constructors and to consider those algebras which are reachable w.r.t. the

* This work is partially supported by the ESPRIT Working Group 29432 CoFI and by the Bayer. Forschungsstiftung.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 72–87, 2001.

given constructors. Most algebraic specification languages incorporate features to express reachability like, for instance, the CASL language [15]. Since observability and reachability are used for different purposes both concepts look quite unrelated. It is the aim of this study to show that there is a methodological and even formal duality between the two concepts which we believe contributes to a clarification of specification methodologies and their semantic foundations.[1] The correspondence will be based on the following working hypothesis (in the spirit of Hoare [9]):

The model class of a specification SP describes
the class of all correct realizations of SP.

The underlying paradigm of the algebraic approach is to model programs by (many-sorted) algebras and to describe the properties of these algebras by logical axioms provided by some specification SP. Then a program is a correct realization if it is a model of SP. Based on these assumptions we will study algebraic frameworks for observability and for reachability and we will compare both concepts.

First, in Section 2, we give an overview of the observational logic institution [7] which we will use as the basis for formalising observability. Then, in Section 3, we discuss reachability and we introduce a new institution, called constructor-based logic, to express reachability issues in accordance with the above working hypothesis. For this purpose we introduce, in particular, the notions of a constructor-based algebra and the constructor-based satisfaction relation. Section 4 exhibits the syntactic and semantic correspondences between all notions used in observational logic and in constructor-based logic. In Section 5, we focus on the properties that are valid consequences of an observational or constructor-based specification. In each case these properties can be characterized by the standard first-order theory of a class of algebras which represent the "idealized" models (also called "black box view") of a specification. By comparing the black box views of observational and constructor-based specifications it turns out that fully abstract models correspond to reachable models.

The results obtained so far show a syntactic, semantic and methodological analogy between observational and constructor-based specifications. In Section 6, we show that this correspondence can even be characterized by a formal duality in a category-theoretic setting. Thereby the syntactic aspects of the observational and the constructor-based notions are expressed by appropriate (pairs of) functors and the semantic aspects are expressed by using algebra-coalgebra pairs; cf. also [14]. Finally, some concluding remarks are given in Section 7.

We assume that the reader is familiar with the basic notions of algebraic specifications (see e.g., [13]), like the notions of (many-sorted) *signature* $\Sigma = (S, OP)$ (where S is a set of *sorts* and OP is a set of *operation symbols* $op : s_1, \ldots, s_n \to s$), *(total)* Σ*-algebra* $A = ((A_s)_{s\in S}, (op^A)_{op\in OP})$, class $\mathrm{Alg}(\Sigma)$ of all Σ-algebras, Σ*-term algebra* $T_\Sigma(X)$ over a family of variables X and *interpretation* $I_\alpha : T_\Sigma(X) \to A$ w.r.t. a *valuation* $\alpha : X \to A$.

[1] In the context of automata theory a similar duality was already investigated by Arbib and Manes in [2].

2 The Observational Logic Institution

Observability concepts provide a means to specify the observable behaviour of software systems in an abstract, implementation independent way. They take into account our working hypothesis (of the Introduction) in the sense that any program which satisfies the observable behaviour described by a specification SP is considered as a correct realization of SP. Observability concepts are particularly suited to specify the observable properties of state-based systems since they allow us to abstract from concrete state representations and to consider any two states which cannot be distinguished by observable experiments as "observationally equal". A flexible approach to formalise observable experiments is suggested (in a similar way) e.g. in [7] and [6] where the operations of an algebraic signature are split into a set of "observer operations" for building observable experiments and the "other" operations which can be used to manipulate states. In this study we will use the observational logic institution to formalise observability. An overview of observational logic is given in the remainder of this section (for more details see [7]).

Definition 1 (Observational signature). *Let $\Sigma = (S, OP)$ be a signature and $S_{\mathrm{Obs}} \subseteq S$ be a set of* observable sorts. *An* observer *is a pair (op, i) where $op : s_1, \ldots, s_n \to s \in OP$, $1 \leq i \leq n$, and $s_i \not\in S_{\mathrm{Obs}}$.*[2] *$(op, i)$ is a* direct observer *of s_i if $s \in S_{\mathrm{Obs}}$, otherwise it is an* indirect observer. *If $op : s_1 \to s$ is a unary observer we will simply write op instead of $(op, 1)$. An observational signature $\Sigma_{\mathrm{Obs}} = (\Sigma, S_{\mathrm{Obs}}, OP_{\mathrm{Obs}})$ consists of a signature $\Sigma = (S, OP)$, a set $S_{\mathrm{Obs}} \subseteq S$ of observable sorts and a set OP_{Obs} of observers (op, i) with $op \in OP$.*

Any observational signature determines a set of observable contexts which represent the observable experiments. In the following definition observable contexts are defined in a co-inductive style

Definition 2 (Observable context). *Let Σ_{Obs} be an observational signature, let $X = (X_s)_{s \in S}$ be a family of countable infinite sets X_s of variables of sort s and let $Z = (\{z_s\})_{s \in S \setminus S_{\mathrm{Obs}}}$ be a disjoint family of singleton sets (one for each non observable sort). For all $s \in S \setminus S_{\mathrm{Obs}}$ and $s' \in S_{\mathrm{Obs}}$ the set $\mathcal{C}(\Sigma_{\mathrm{Obs}})_{s \to s'}$ of* observable Σ_{Obs}-contexts *with "application sort" s and "observable result sort" s' is inductively defined as follows:*

1. *For each direct observer (op, i) with $op : s_1, \ldots, s_i, \ldots, s_n \to s'$ and pairwise disjoint variables $x_1 : s_1, \ldots, x_n : s_n$,*
   $op(x_1, \ldots, x_{i-1}, z_{s_i}, x_{i+1}, \ldots, x_n) \in \mathcal{C}(\Sigma_{\mathrm{Obs}})_{s_i \to s'}$,
2. *For each observable context $c \in \mathcal{C}(\Sigma_{\mathrm{Obs}})_{s \to s'}$, for each indirect observer (op, i) with $op : s_1, \ldots, s_i, \ldots, s_n \to s$, and for each pairwise disjoint variables $x_1 : s_1, \ldots, x_n : s_n$ not occurring in c,*
   $c[op(x_1, \ldots, x_{i-1}, z_{s_i}, x_{i+1}, \ldots, x_n)/z_s] \in \mathcal{C}(\Sigma_{\mathrm{Obs}})_{s_i \to s'}$
   where $c[op(x_1, \ldots, x_{i-1}, z_{s_i}, x_{i+1}, \ldots, x_n)/z_s]$ denotes the term obtained from c by substituting the term $op(x_1, \ldots, x_{i-1}, z_{s_i}, x_{i+1}, \ldots, x_n)$ for z_s.

[2] Non-observable sorts are also called "state-sorts".

The syntactic notion of an observable context induces, for any Σ-algebra A a semantic relation, called observational equality, which expresses indistinguishability of states w.r.t. the given observable contexts.

Definition 3 (Σ_{Obs}-equality). *Let Σ_{Obs} be an observational signature. For any Σ-algebra $A \in \mathrm{Alg}(\Sigma)$, the* observational Σ_{Obs}-equality *on A is denoted by $\approx_{\Sigma_{\mathrm{Obs}},A}$ and defined by:*
For all $s \in S$, two elements $a, b \in A_s$ are observationally equal w.r.t. Σ_{Obs}, i.e., $a \approx_{\Sigma_{\mathrm{Obs}},A} b$, if and only if

1. *$a = b$, if $s \in S_{\mathrm{Obs}}$,*
2. *for all observable sorts $s' \in S_{\mathrm{Obs}}$, for all observable contexts $c \in \mathcal{C}(\Sigma_{\mathrm{Obs}})_{s \to s'}$, and for all valuations $\alpha, \beta : X \cup \{z_s\} \to A$ with $\alpha(x) = \beta(x)$ if $x \in X$, $\alpha(z_s) = a$ and $\beta(z_s) = b$, we have $I_\alpha(c) = I_\beta(c)$, if $s \in S \setminus S_{\mathrm{Obs}}$.*

Note that only the observer operations are used to build observable contexts and hence to define the observational equality. As a consequence we require that the non-observer operations should not contribute to distinguish states. This requirement is guaranteed by observational algebras defined as follows.

Definition 4 (Observational algebra). *Let Σ_{Obs} be an observational signature. An* observational Σ_{Obs}-algebra *is a Σ-algebra A such that $\approx_{\Sigma_{\mathrm{Obs}},A}$ is a Σ-congruence on A. The class of all observational Σ_{Obs}-algebras is denoted by* $\mathrm{Alg}_{\mathrm{Obs}}(\Sigma_{\mathrm{Obs}})$. [3]

In the next step we define an observational satisfaction relation for observational algebras and first-order Σ-formulas. The underlying idea of the observational satisfaction relation is to interpret the equality symbol $=$ occurring in a first-order formula φ not by the set-theoretic equality but by the observational equality of elements.

Definition 5 (Observational satisfaction relation). *The* observational satisfaction relation *between observational Σ_{Obs}-algebras and first-order Σ-formulas is denoted by $\models_{\Sigma_{\mathrm{Obs}}}$ and defined as follows:*
Let $A \in \mathrm{Alg}_{\mathrm{Obs}}(\Sigma_{\mathrm{Obs}})$.

1. *For any two terms $t, r \in T_\Sigma(X)_s$ of the same sort s and for any valuation $\alpha : X \to A$, $A, \alpha \models_{\Sigma_{\mathrm{Obs}}} t = r$ holds if $I_\alpha(t) \approx_{\Sigma_{\mathrm{Obs}},A} I_\alpha(r)$.*
2. *For any arbitrary Σ-formula φ and for any valuation $\alpha : X \to A$, $A, \alpha \models_{\Sigma_{\mathrm{Obs}}} \varphi$ is defined by induction over the structure of the formula in the usual way.*
3. *For any arbitrary Σ-formula φ, $A \models_{\Sigma_{\mathrm{Obs}}} \varphi$ holds if for all valuations $\alpha : X \to A$, $A, \alpha \models_{\Sigma_{\mathrm{Obs}}} \varphi$ holds.*

Definition 6 (Basic observational specification). *A* basic observational specification $\mathrm{SP}_{\mathrm{Obs}} = \langle \Sigma_{\mathrm{Obs}}, \mathrm{Ax} \rangle$ *consists of an observational signature $\Sigma_{\mathrm{Obs}} = (\Sigma, S_{\mathrm{Obs}}, OP_{\mathrm{Obs}})$ and a set* Ax *of Σ-sentences, called the axioms of* $\mathrm{SP}_{\mathrm{Obs}}$. *The semantics of* $\mathrm{SP}_{\mathrm{Obs}}$ *is given by its signature* $\mathrm{Sig}_{\mathrm{Obs}}(\mathrm{SP}_{\mathrm{Obs}})$ *and by its class of models* $\mathrm{Mod}_{\mathrm{Obs}}(\mathrm{SP}_{\mathrm{Obs}})$ *which are defined by:*

$$\mathrm{Sig}_{\mathrm{Obs}}(\mathrm{SP}_{\mathrm{Obs}}) \stackrel{\mathrm{def}}{=} \Sigma_{\mathrm{Obs}}, \mathrm{Mod}_{\mathrm{Obs}}(\mathrm{SP}_{\mathrm{Obs}}) \stackrel{\mathrm{def}}{=} \{A \in \mathrm{Alg}_{\mathrm{Obs}}(\Sigma_{\mathrm{Obs}}) \mid A \models_{\Sigma_{\mathrm{Obs}}} \mathrm{Ax}\}$$

[3] Observational morphisms are defined as relations; see [7].

The definitions stated above provide the basic ingredients for defining the *observational logic institution* which is detailed in [7]. Thereby it is particularly important to use an appropriate morphism notion for observational signatures which guarantees encapsulation of observable properties (formally expressed by the satisfaction condition of institutions; cf. [5]). The basic idea for achieving this is to require that no "new"observations are introduced for "old" sorts when composing systems via signature morphisms. Thus the observational logic institution provides a suitable framework for instantiating the institution-independent specification-building operators introduced in [18], hence for defining structured observational specifications.

3 The Constructor-Based Logic Institution

Reachability concepts are used to describe the underlying data manipulated by a program. For this purpose, a standard approach is to declare a distinguished subset OP_{Cons} of the operation symbols OP (of a signature $\Sigma = (S, OP)$) as constructor symbols and to restrict the admissible models of a specification to those algebras which are reachable w.r.t. the given constructors. Syntactically we will follow this approach which leads to the notion of a constructor-based signature (see Definition 7 below). However, from the semantic point of view we do not adopt the above interpretation which we believe is too restrictive w.r.t. our working hypothesis (of the Introduction). Let us illustrate our viewpoint by a simple example.

Let NAT be a standard specification of the natural numbers with signature $\Sigma_{NAT} = (\{nat\},\ \{zero : \ \to nat,\ succ : nat \to nat,\ add : nat \times nat \to nat\})$ and with standard axioms. We declare $zero$ and $succ$ as constructor symbols. Then a Σ_{NAT}-algebra A is reachable w.r.t. the given constructors if any element of A is denotable by a term $succ^i(zero)$ with $i \geq 0$. Obviously the set $\mathbb{N}$ of the natural numbers (equipped with the usual operations) is a reachable algebra. But note that the set $\mathbb{Z}$ of the integers (equipped with the usual interpretations of $zero$, $succ$ and add) is not reachable w.r.t. the given constructors and therefore is not an admissible (standard) model of NAT. Nevertheless the integers can obviously be used as an implementation of the natural numbers which just happens to contain the negative integers as junk elements. Hence, in order to satisfy our working hypothesis, the integers should be admitted as a model of NAT. As a consequence we are interested in a more flexible framework where the constructor symbols are still essential, in the sense that they determine the data of interest, but nevertheless non-reachable algebras can be accepted as models if they satisfy certain conditions which are formalised by our notion of constructor-based algebra (see Definition 10 below).

In this way we obtain a novel treatment of reachability in algebraic specifications which finally leads to the institution of constructor-based logic. All steps performed in this section are quite analogous to the development of the observational logic institution. The correspondence will be analysed in Section 4 and formalised in Section 6.

Definition 7 (Constructor-based signature). *A constructor-based signature* $\Sigma_{\mathrm{Cons}} = (\Sigma, S_{\mathrm{Cons}}, OP_{\mathrm{Cons}})$ *consists of a signature* $\Sigma = (S, OP)$*, a set* $S_{\mathrm{Cons}} \subseteq S$ *of* constrained sorts *and a set* $OP_{\mathrm{Cons}} \subseteq OP$ *of* constructor symbols *such that, for any* $op \in OP_{\mathrm{Cons}}$ *with arity* $op : s_1, \ldots, s_n \to s$, $s \in S_{\mathrm{Cons}}$*. We assume also that for each constrained sort* $s \in S_{\mathrm{Cons}}$*, there exists at least one constructor in* OP_{Cons} *with range* s*.*

Any constructor-based signature determines a set of constructor terms.

Definition 8 (Constructor term). *Let* Σ_{Cons} *be a constructor-based signature, and let* $X = (X_s)_{s \in S}$ *be a family of countable infinite sets* X_s *of variables of sort* s*. A* constructor term *is a term* $t \in T_{\Sigma'}(X')$*, where* $\Sigma' = (S, OP_{\mathrm{Cons}})$*, and* $X' = (X'_s)_{s \in S}$ *with* $X'_s = X_s$ *if* $s \in S \setminus S_{\mathrm{Cons}}$ *and* $X'_s = \emptyset$ *if* $s \in S_{\mathrm{Cons}}$*. The set of constructor terms is denoted by* $\mathcal{T}(\Sigma_{\mathrm{Cons}})$*.*

The syntactic notion of a constructor term induces, for any Σ-algebra A, the definition of a family of subsets of the carrier sets of A, called reachable part, which consists of those data which are relevant according to the given constructors.

Definition 9 (Reachable part). *Let* Σ_{Cons} *be a constructor-based signature. For any* Σ*-algebra* $A \in \mathrm{Alg}(\Sigma)$*, the reachable part* $\langle A \rangle_{\mathrm{Cons}} = (\langle A \rangle_{\mathrm{Cons},s})_{s \in S}$ *of* A *is defined by:*
For each $s \in S$, $\langle A \rangle_{\mathrm{Cons},s} = \{a \in A_s \mid$ there exists a term $t \in \mathcal{T}(\Sigma_{\mathrm{Cons}})_s$ and a valuation $\alpha : X' \to A$ such that $I_\alpha(t) = a\}$. [4]

Note that only the constructor symbols are used to build constructor terms and hence to define the reachable part. Since the reachable part represents the data of interest we require that no further elements should be constructible by the non-constructor operations:

Definition 10 (Constructor-based algebra). *Let* Σ_{Cons} *be a constructor-based signature. A* constructor-based Σ_{Cons}-algebra *is a* Σ*-algebra* A *such that* $\langle A \rangle_{\mathrm{Cons}}$*, equipped with the canonical restrictions of the operations* op^A *of* A *to the carrier sets of* $\langle A \rangle_{\mathrm{Cons}}$*, is a* Σ*-subalgebra of* A*. The class of all constructor-based* Σ_{Cons}*-algebras is denoted by* $\mathrm{Alg}_{\mathrm{Cons}}(\Sigma_{\mathrm{Cons}})$*.*

Definition 11 (Constructor-based morphism). *Let* $A, B \in \mathrm{Alg}_{\mathrm{Cons}}(\Sigma_{\mathrm{Cons}})$ *be two constructor-based* Σ_{Cons}*-algebras.* A constructor-based Σ_{Cons}-morphism $h : A \to B$ *is a* Σ*-morphism between* $\langle A \rangle_{\mathrm{Cons}}$ *and* $\langle B \rangle_{\mathrm{Cons}}$*.*

For any constructor-based signature Σ_{Cons}, the class $\mathrm{Alg}_{\mathrm{Cons}}(\Sigma_{\mathrm{Cons}})$ together with the constructor-based Σ_{Cons}-morphisms is a category.

[4] Note that for any non-constrained sort s, $\langle A \rangle_{\mathrm{Cons},s} = A_s$.

The underlying idea of the constructor-based satisfaction relation is to restrict the valuations of variables to the generated values (i.e. to the elements of the reachable part) only.[5] Hence the following definition is quite similar to the definition of the standard satisfaction relation. The only difference concerns valuations: "$\alpha : X \to A$" is replaced by "$\alpha : X \to \langle A \rangle_{\mathrm{Cons}}$".

Definition 12 (Constructor-based satisfaction relation). *The* constructor-based satisfaction relation *between constructor-based Σ_{Cons}-algebras and first-order Σ-formulas is denoted by $\models_{\Sigma_{\mathrm{Cons}}}$ and defined as follows:*
Let $A \in \mathrm{Alg}_{\mathrm{Cons}}(\Sigma_{\mathrm{Cons}})$.

1. *For any two terms $t, r \in T_\Sigma(X)_s$ of the same sort s and for any valuation $\alpha : X \to \langle A \rangle_{\mathrm{Cons}}$, $A, \alpha \models_{\Sigma_{\mathrm{Cons}}} t = r$ holds if $I_\alpha(t) = I_\alpha(r)$.*
2. *For any arbitrary Σ-formula φ and for any valuation $\alpha : X \to \langle A \rangle_{\mathrm{Cons}}$, $A, \alpha \models_{\Sigma_{\mathrm{Cons}}} \varphi$ is defined by induction over the structure of the formula in the usual way.*
3. *For any arbitrary Σ-formula φ, $A \models_{\Sigma_{\mathrm{Cons}}} \varphi$ holds if for all valuations $\alpha : X \to \langle A \rangle_{\mathrm{Cons}}$, $A, \alpha \models_{\Sigma_{\mathrm{Cons}}} \varphi$ holds.*

As an example consider again the specification NAT and the integers which satisfy w.r.t. the constructor-based satisfaction relation the following Peano axiom

$$\mathbb{Z} \models_{\Sigma_{\mathrm{Cons}}} \forall x : nat.\ succ(x) \neq zero$$

Indeed this is true since the reachable part of $\mathbb{Z}$ w.r.t. the constructors *zero* and *succ* is just $\mathbb{N}$ and hence the universally quantified variable x is only interpreted in $\mathbb{N}$. Thus the integers will be an admissible model of NAT considered as a constructor-based specification w.r.t. the constructors *zero* and *succ*.

Definition 13 (Basic constructor-based specification). *A* basic constructor-based specification $\mathrm{SP}_{\mathrm{Cons}} = \langle \Sigma_{\mathrm{Cons}}, \mathrm{Ax} \rangle$ *consists of a constructor-based signature $\Sigma_{\mathrm{Cons}} = (\Sigma, S_{\mathrm{Cons}}, OP_{\mathrm{Cons}})$ and a set* Ax *of Σ-sentences, called the axioms of* $\mathrm{SP}_{\mathrm{Cons}}$. *The semantics of* $\mathrm{SP}_{\mathrm{Cons}}$ *is given by its signature* $\mathrm{Sig}_{\mathrm{Cons}}(\mathrm{SP}_{\mathrm{Cons}})$ *and by its class of models* $\mathrm{Mod}_{\mathrm{Cons}}(\mathrm{SP}_{\mathrm{Cons}})$ *which are defined by:*

$$\mathrm{Sig}_{\mathrm{Cons}}(\mathrm{SP}_{\mathrm{Cons}}) \stackrel{\mathrm{def}}{=} \Sigma_{\mathrm{Cons}}$$

$$\mathrm{Mod}_{\mathrm{Cons}}(\mathrm{SP}_{\mathrm{Cons}}) \stackrel{\mathrm{def}}{=} \{A \in \mathrm{Alg}_{\mathrm{Cons}}(\Sigma_{\mathrm{Cons}}) \mid A \models_{\Sigma_{\mathrm{Cons}}} \mathrm{Ax}\}$$

To obtain the constructor-based logic institution we still need an appropriate morphism notion for constructor-based signatures which guarantees encapsulation of properties w.r.t. the constructor-based satisfaction relation. The basic idea to achieve this is to require that no "new" constructors are introduced for "old" sorts when composing systems via signature morphisms which is formally captured by the following definition.

[5] This idea is related to the ultra-loose approach of [19] where the same effect is achieved by using formulas with relativized quantification.

Definition 14 (Constructor-based signature morphism). *Let* $\Sigma_{\mathrm{Cons}} = (\Sigma, S_{\mathrm{Cons}}, OP_{\mathrm{Cons}})$ *and* $\Sigma'_{\mathrm{Cons}} = (\Sigma', S'_{\mathrm{Cons}}, OP'_{\mathrm{Cons}})$ *be two constructor-based signatures with* $\Sigma = (S, OP)$ *and* $\Sigma' = (S', OP')$. *A* constructor-based signature morphism $\sigma_{\mathrm{Cons}} : \Sigma_{\mathrm{Cons}} \to \Sigma'_{\mathrm{Cons}}$ *is a signature morphism* $\sigma : \Sigma \to \Sigma'$ *such that:*

1. *For all* $s \in S$, $s \in S_{\mathrm{Cons}}$ *if and only if* $\sigma(s) \in S'_{\mathrm{Cons}}$.
2. *If* $op \in OP_{\mathrm{Cons}}$, *then* $\sigma(op) \in OP'_{\mathrm{Cons}}$.
3. *If* $op' \in OP'_{\mathrm{Cons}}$ *with* $op' : s'_1, \ldots, s'_n \to s'$ *and* $s' \in \sigma(S)$ *then there exists* $op \in OP$ *such that* $op \in OP_{\mathrm{Cons}}$ *and* $op' = \sigma(op)$.

Constructor-based signatures together with constructor-based signature morphisms form a category which has pushouts. Moreover, for any constructor-based signature morphism $\sigma_{\mathrm{Cons}} : \Sigma_{\mathrm{Cons}} \to \Sigma'_{\mathrm{Cons}}$, one can associate a constructor-based reduct functor $\_|_{\sigma_{\mathrm{Cons}}} : \mathrm{Alg}_{\mathrm{Cons}}(\Sigma'_{\mathrm{Cons}}) \to \mathrm{Alg}_{\mathrm{Cons}}(\Sigma_{\mathrm{Cons}})$ in a straightforward way. One can also show that the constructor-based satisfaction condition holds, i.e., for any constructor-based signature morphism $\sigma_{\mathrm{Cons}} : \Sigma_{\mathrm{Cons}} \to \Sigma'_{\mathrm{Cons}}$, constructor-based Σ'_{Cons}-algebra $A' \in \mathrm{Alg}_{\mathrm{Cons}}(\Sigma'_{\mathrm{Cons}})$ and Σ-sentence φ:
$A' \models_{\Sigma'_{\mathrm{Cons}}} \sigma(\varphi)$ if and only if $A'|_{\sigma_{\mathrm{Cons}}} \models_{\Sigma_{\mathrm{Cons}}} \varphi$.

This means that the definitions stated above provide the necessary ingredients for defining an institution (cf. [5]) which is called the *constructor-based logic institution*. As in the observational case this institution provides a suitable framework for instantiating the institution-independent specification-building operators introduced in [18], hence for defining structured constructor-based specifications.

4 A First Comparison

The observational logic institution and the constructor-based logic institution were developed step by step in a totally analogous way. Indeed there is a close correspondence between all notions of the observability and reachability concepts which is summarized in Table 1.

First, there is an obvious syntactic correspondence between an observational signature and a constructor-based signature which, on the one hand, leads to the notion of an observable context and, on the other hand, leads to the definition of a constructor term.

In any case the syntactic notions induce a semantic relation on any Σ-algebra A. In the observational case we obtain a binary relation $\approx_{\Sigma_{\mathrm{Obs}},A}$, called observational equality, and in the constructor case we obtain a unary relation $\langle A \rangle_{\mathrm{Cons}}$, called reachable part. Then we require that the operations of an algebra are compatible with the given relations. This means, in the observational case, that the observational equality is a Σ-congruence thus leading to the notion of an observational algebra. In the constructor case this means that the reachable part is a Σ-subalgebra thus leading to the notion of a constructor-based algebra.

In order to satisfy our working hypothesis we have relaxed the standard satisfaction relation such that, in the observational case, equality is considered

Table 1. Comparing Observability and Reachability

Observability	**Reachability**
observational signature $\Sigma_{\mathrm{Obs}} = (\Sigma, S_{\mathrm{Obs}}, OP_{\mathrm{Obs}})$	*constructor-based signature* $\Sigma_{\mathrm{Cons}} = (\Sigma, S_{\mathrm{Cons}}, OP_{\mathrm{Cons}})$
observable context	*constructor term*
observational equality $\approx_{\Sigma_{\mathrm{Obs}},A} \subseteq A \times A$	*reachable part* $\langle A \rangle_{\mathrm{Cons}} \subseteq A$
observational algebra $\approx_{\Sigma_{\mathrm{Obs}},A}$ is a Σ-congruence	*constructor-based algebra* $\langle A \rangle_{\mathrm{Cons}}$ is a Σ-subalgebra of A
observational satisfaction $A \models_{\Sigma_{\mathrm{Obs}}} \phi$ interpret "$=$" by "$\approx_{\Sigma_{\mathrm{Obs}},A}$"	*constructor-based satisfaction* $A \models_{\Sigma_{\mathrm{Cons}}} \phi$ use valuations $\alpha : X \to \langle A \rangle_{\mathrm{Cons}}$
observational specification $\mathrm{SP}_{\mathrm{Obs}} = \langle \Sigma_{\mathrm{Obs}}, \mathrm{Ax} \rangle$ $\mathrm{Mod}_{\mathrm{Obs}}(\mathrm{SP}_{\mathrm{Obs}}) \stackrel{\mathrm{def}}{=} \{A \in \mathrm{Alg}_{\mathrm{Obs}}(\Sigma_{\mathrm{Obs}}) \mid A \models_{\Sigma_{\mathrm{Obs}}} \mathrm{Ax}\}$	*constructor-based specification* $\mathrm{SP}_{\mathrm{Cons}} = \langle \Sigma_{\mathrm{Cons}}, \mathrm{Ax} \rangle$ $\mathrm{Mod}_{\mathrm{Cons}}(\mathrm{SP}_{\mathrm{Cons}}) \stackrel{\mathrm{def}}{=} \{A \in \mathrm{Alg}_{\mathrm{Cons}}(\Sigma_{\mathrm{Cons}}) \mid A \models_{\Sigma_{\mathrm{Cons}}} \mathrm{Ax}\}$
observational logic institution	*constructor-based logic institution*

as observational equality and, in the constructor case, variables are interpreted only by values of the reachable part. Then it is straightforward to introduce the notions of observational and constructor-based specifications whose semantics are defined according to the generalized satisfaction relations. Finally we have pointed out that both frameworks lead to an institution by using appropriate notions of signature morphisms.

It is still important to stress that there are also corresponding specification methods when writing observational and constructor-based specifications. In the observational case the idea is to specify the effect of each non-observer operation (in a co-inductive style) by a (complete) case distinction w.r.t. the given observers. A general schema for observer complete definitions is studied in [3]. As a standard example consider an observational specification of an alternating merge function `merge: stream x stream` $\to$ `stream` on streams with observers `head: stream` $\to$ `elem` and `tail: stream` $\to$ `stream`. Then the `merge` function is specified by the following complete case distinction w.r.t. the observers `head` and `tail`:

```
head(merge(s1, s2)) = head(s1)
tail(merge(s1, s2)) = merge(s2, tail(s1))
```

Analogously it is well-known that in the constructor case it is a standard technique to specify the non-constructor operations in an inductive-style by a (complete) case distinction w.r.t. the given constructors. In the categorical framework of algebras and co-algebras this analogy is described in [11].

5 Logical Consequences of Specifications: The Black Box View

So far we have emphasized the fact that the model class semantics of a specification should reflect all its correct realizations. According to our working hypothesis, a program P is a correct realization of SP_X if it determines a $\mathrm{Sig}_X(\mathrm{SP}_X)$-algebra which belongs to $\mathrm{Mod}_X(\mathrm{SP}_X)$.[6] In the following we will refer to $\mathrm{Mod}_X(\mathrm{SP}_X)$ as the *glass box semantics* of a specification since it reveals its correct realizations. Glass box semantics is appropriate from an implementor's point of view.

Of equal importance are the logical consequences of a given specification. In this section we focus on the properties φ that can be inferred from a given specification SP_X. This means that we are interested in statements $\mathrm{SP}_X \models_X \varphi$ which express that $\mathrm{Mod}_X(\mathrm{SP}_X) \models_X \varphi$ holds.

For this purpose it is convenient to *abstract* the models of a specification into "idealized" models, such that the consequences of the actual models of the specification of interest, in the chosen logic, are exactly the consequences of the idealized models, in *standard* first-order logic. Hence to any specification SP_X we will associate the class of its "idealized" models (which lie in the standard algebraic institution), and this class will be called the *black box semantics* of the specification. Black box semantics is appropriate from a client's point of view.

5.1 Black Box Semantics of Observational Specifications

Let Σ_{Obs} be an observational signature. Since for any Σ_{Obs}-algebra A, the observational equality $\approx_{\Sigma_{\mathrm{Obs}},A}$ is a Σ-congruence, we can construct its quotient $A/\approx_{\Sigma_{\mathrm{Obs}},A}$ which identifies all elements of A which are indistinguishable "from the outside". $A/\approx_{\Sigma_{\mathrm{Obs}},A}$ can be considered as the "black box view" of A and represents the "observable behaviour" of A w.r.t. Σ_{Obs}. $A/\approx_{\Sigma_{\mathrm{Obs}},A}$ is *fully abstract* in the sense that the observational equality (w.r.t. Σ_{Obs}) on $A/\approx_{\Sigma_{\mathrm{Obs}},A}$ coincides with the set-theoretic equality. By considering $A/\approx_{\Sigma_{\mathrm{Obs}},A}$ just as a Σ-algebra we obtain (for any observational signature Σ_{Obs}) a functor from the category $\mathrm{Alg}_{\mathrm{Obs}}(\Sigma_{\mathrm{Obs}})$ of observational algebras into the category $\mathrm{Alg}(\Sigma)$ of (standard) Σ-algebras which establishes a one to one correspondence between observational morphisms $h : A \to B$ and standard morphisms $k : A/\approx_{\Sigma_{\mathrm{Obs}},A} \to B/\approx_{\Sigma_{\mathrm{Obs}},B}$, i.e., this functor is full and faithful.

[6] We use the subscript $_X$ to denote the fact that we work either in the observational logic institution or in the constructor-based logic institution.

Theorem 1 (Behavior functor). *For any observational signature* Σ_{Obs} *with underlying standard signature* Σ*, the following defines a full and faithful functor* $\mathcal{FA}_{\Sigma_{\mathrm{Obs}}} : \mathrm{Alg}_{\mathrm{Obs}}(\Sigma_{\mathrm{Obs}}) \to \mathrm{Alg}(\Sigma)$*:*

1. *For each* $A \in \mathrm{Alg}_{\mathrm{Obs}}(\Sigma_{\mathrm{Obs}})$, $\mathcal{FA}_{\Sigma_{\mathrm{Obs}}}(A) \stackrel{\mathrm{def}}{=} A/{\approx_{\Sigma_{\mathrm{Obs}},A}}$ *and is called the* observational behavior *of* A*.*
2. *For each observational morphism* $h : A \to B$, $\mathcal{FA}_{\Sigma_{\mathrm{Obs}}}(h) : A/{\approx_{\Sigma_{\mathrm{Obs}},A}} \to B/{\approx_{\Sigma_{\mathrm{Obs}},B}}$ *is defined by* $\mathcal{FA}_{\Sigma_{\mathrm{Obs}}}(h)([a]) = [b]$ *if* $a\ h\ b$*.*

Definition 15 (Black box semantics). *Let* $\mathrm{SP}_{\mathrm{Obs}}$ *be an observational specification with signature* $\mathrm{Sig}_{\mathrm{Obs}}(\mathrm{SP}_{\mathrm{Obs}}) = \Sigma_{\mathrm{Obs}}$*. Its black box semantics is defined by* $[\![\mathrm{SP}_{\mathrm{Obs}}]\!] \stackrel{\mathrm{def}}{=} \mathcal{FA}_{\Sigma_{\mathrm{Obs}}}(\mathrm{Mod}_{\mathrm{Obs}}(\mathrm{SP}_{\mathrm{Obs}}))$*.*

It may be interesting to note that the black box semantics of an observational specification is exactly the class of its fully abstract models (viewed as ordinary algebras).

Fact 2 (Black box semantics relies on fully abstract models)
Let $\mathrm{SP}_{\mathrm{Obs}} = \langle \Sigma_{\mathrm{Obs}}, \mathrm{Ax} \rangle$ *be a basic observational specification. Then* $[\![\mathrm{SP}_{\mathrm{Obs}}]\!] = \{\Sigma\text{−algebra } A \mid A \models \mathrm{Ax} \text{ and } A \text{ is fully abstract w.r.t. } \approx_{\Sigma_{\mathrm{Obs}},A}\}$.

Theorem 3 (Observational consequences). *Let* Σ_{Obs} *be an observational signature with underlying standard signature* Σ*, let* φ *be a* Σ*-formula, let* A *be a* Σ_{Obs}*-algebra, and let* $\mathrm{SP}_{\mathrm{Obs}}$ *be an observational specification with signature* Σ_{Obs}*.*

1. $A \models_{\Sigma_{\mathrm{Obs}}} \varphi$ *if and only if* $\mathcal{FA}_{\Sigma_{\mathrm{Obs}}}(A) \models \varphi$*.*
2. $\mathrm{SP}_{\mathrm{Obs}} \models_{\Sigma_{\mathrm{Obs}}} \varphi$ *if and only if* $[\![\mathrm{SP}_{\mathrm{Obs}}]\!] \models \varphi$*.*

This theorem shows the adequacy of the black box semantics in the observational case. The theorem is a variant of Theorem 3.11 in [4] and it is related to similar results in [10].

5.2 Black Box Semantics of Constructor-Based Specifications

Let Σ_{Cons} be a constructor-based signature. Since for any Σ_{Cons}-algebra A, the reachable part $\langle A \rangle_{\mathrm{Cons}}$ of A is a Σ-algebra, which by definition contains only those elements that are generated by the given constructors (for the constrained sorts), we can consider the reachable part $\langle A \rangle_{\mathrm{Cons}}$ of A as its black box view (abstracting away from all junk values that may lie in A). This restriction to the reachable sub-algebra provides (for any constructor-based signature Σ_{Cons}) a functor from the category $\mathrm{Alg}_{\mathrm{Cons}}(\Sigma_{\mathrm{Cons}})$ of constructor-based algebras into the category $\mathrm{Alg}(\Sigma)$ of (standard) Σ-algebras which is full and faithful.

Theorem 4 (Restrict functor). *For any constructor-based signature* Σ_{Cons} *with underlying standard signature* Σ*, the following defines a full and faithful functor* $\mathcal{R}_{\Sigma_{\mathrm{Cons}}} : \mathrm{Alg}_{\mathrm{Cons}}(\Sigma_{\mathrm{Cons}}) \to \mathrm{Alg}(\Sigma)$*:*

1. *For each* $A \in \mathrm{Alg}_{\mathrm{Cons}}(\Sigma_{\mathrm{Cons}})$, $\mathcal{R}_{\Sigma_{\mathrm{Cons}}}(A) \stackrel{\mathrm{def}}{=} \langle A \rangle_{\mathrm{Cons}}$ *and is called the* reachable part *of* A.
2. *For each constructor-based morphism* $h : A \to B$, $\mathcal{R}_{\Sigma_{\mathrm{Cons}}}(h) : \langle A \rangle_{\mathrm{Cons}} \to \langle B \rangle_{\mathrm{Cons}}$ *is defined by* $\mathcal{R}_{\Sigma_{\mathrm{Cons}}}(h)(a) = h(a)$.

Definition 16 (Black box semantics). *Let* $\mathrm{SP}_{\mathrm{Cons}}$ *be a constructor-based specification with signature* $\mathrm{Sig}_{\mathrm{Cons}}(\mathrm{SP}_{\mathrm{Cons}}) = \Sigma_{\mathrm{Cons}}$. *Its black box semantics is defined by* $[\![\mathrm{SP}_{\mathrm{Cons}}]\!] \stackrel{\mathrm{def}}{=} \mathcal{R}_{\Sigma_{\mathrm{Cons}}}(\mathrm{Mod}_{\mathrm{Cons}}(\mathrm{SP}_{\mathrm{Cons}}))$.

Again, it may be interesting to note that the black box semantics of a constructor-based specification is exactly the class of its reachable models. Thereby an algebra is called reachable w.r.t. a set OP_{Cons} of constructor symbols if all elements of A are denotable by a constructor term.

Fact 5 (Black box semantics relies on reachable models)
Let $\mathrm{SP}_{\mathrm{Cons}} = \langle \Sigma_{\mathrm{Cons}}, \mathrm{Ax} \rangle$ *be a basic constructor-based specification. Then* $[\![\mathrm{SP}_{\mathrm{Cons}}]\!] = \{\Sigma\text{−algebra } A \mid A \models \mathrm{Ax} \text{ and } A \text{ is reachable w.r.t. } OP_{\mathrm{Cons}}\}$.

Theorem 6 (Inductive consequences). *Let* Σ_{Cons} *be a constructor-based signature with underlying standard signature* Σ, *let* φ *be a* Σ*-formula, let* A *be a* Σ_{Cons}*-algebra, and let* $\mathrm{SP}_{\mathrm{Cons}}$ *be a constructor-based specification with signature* Σ_{Cons}.

1. $A \models_{\Sigma_{\mathrm{Cons}}} \varphi$ *if and only if* $\mathcal{R}_{\Sigma_{\mathrm{Cons}}}(A) \models \varphi$.
2. $\mathrm{SP}_{\mathrm{Cons}} \models_{\Sigma_{\mathrm{Cons}}} \varphi$ *if and only if* $[\![\mathrm{SP}_{\mathrm{Cons}}]\!] \models \varphi$.

This theorem shows the adequacy of the black box semantics in the constructor-based case.

6 Formalising the Duality

In this section we establish a formal duality of the observability and reachability concepts considered in the previous sections. For this purpose we first need a precise notion of duality. This is provided by using category theory.

6.1 Categorical Duality

We briefly review categorical duality, for more details see e.g. [1]. A category $\mathcal{C}$ consists of a class of objects, also denoted by $\mathcal{C}$, and for all $A, B \in \mathcal{C}$ of a set of arrows (or morphisms) $\mathcal{C}(A, B)$. The *dual* (or opposite) category $\mathcal{C}^{\mathrm{op}}$ has the same objects and arrows $\mathcal{C}^{\mathrm{op}}(A, B) = \mathcal{C}(B, A)$. We write A^{op} and f^{op} for $A \in \mathcal{C}$ and $f \in \mathcal{C}(B, A)$ to indicate when we think of A as an object in $\mathcal{C}^{\mathrm{op}}$ and of f as an arrow in $\mathcal{C}^{\mathrm{op}}(A, B)$. Duality can now be formalised as follows: Let P be a property of objects or arrows in $\mathcal{C}$. We then say that

an object A (arrow f, respectively) in $\mathcal{C}$ has property co-P
iff A^{op} (f^{op}, respectively) has property P.

For example, an object A is co-initial in $\mathcal{C}$ (usually called terminal or final) iff A is initial in $\mathcal{C}^{\mathrm{op}}$; $C = A + B$ is a co-product iff C^{op} is the product $A^{\mathrm{op}} \times B^{\mathrm{op}}$.

The duality principle can also be extended to functors. The dual of a functor $F : \mathcal{C} \to \mathcal{D}$ is the functor $F^{\mathrm{op}} : \mathcal{C}^{\mathrm{op}} \to \mathcal{D}^{\mathrm{op}}$ which acts on objects and morphisms as F does. For instance, for an endofunctor F, the category of F-coalgebras is (isomorphic to) the dual of the category of F^{op}-algebras (cf. [11] for a study of algebras and co-algebras).

6.2 The Duality Principle for Observability and Reachability

We first give a categorical account of the signatures and models in the observational and in the constructor-based approach. The formal duality principle will then be an immediate consequence.

Motivated by the approach in [8], we represent a signature (over a base category $\mathcal{X}$) by two functors $\Omega, \Xi : \mathcal{X} \to \mathcal{X}$. A model is an algebra-coalgebra pair $\Omega X \to X \to \Xi X$ (cf. also [14]). We call Ω the *algebraic signature* and Ξ the *coalgebraic signature*.

Next, we introduce signatures for constructors and observers, given by functors $\mathcal{R}, \mathcal{O} : \mathcal{X} \to \mathcal{X}$, respectively. That is, for formalizing the observational signatures, we consider signatures $(\Omega, \Xi) = (\Omega, \Xi' \times \mathcal{O})$ and models

$$\Omega X \xrightarrow{\ \omega\ } X \xrightarrow{\ \langle \xi', o \rangle\ } (\Xi' \times \mathcal{O})X$$

where ω, ξ' are operations and o are observers.

On the other hand, for formalizing constructor-based signatures, we consider signatures $(\Omega, \Xi) = (\Omega' + \mathcal{R}, \Xi)$ and models

$$(\Omega' + \mathcal{R})X \xrightarrow{\ [\omega', \rho]\ } X \xrightarrow{\ \xi\ } \Xi X$$

where ω', ξ are operations and ρ are constructors.

Definition 17 (Observational models). *Let $\mathcal{X}$ be a category with binary products. An **observational signature** $(\Omega; \Xi', \mathcal{O})$ over $\mathcal{X}$ consists of functors $\Omega, \Xi', \mathcal{O} : \mathcal{X} \to \mathcal{X}$ such that a final $\mathcal{O}$-coalgebra $Z \xrightarrow{\ \zeta\ } \mathcal{O}Z$ exists. $\mathcal{O}$ is called the **observability constraint**. An algebra-coalgebra pair $(\omega, \langle \xi', o \rangle)$ for $(\Omega, \Xi' \times \mathcal{O})$ satisfies the observability constraint $\mathcal{O}$ and is called a **model** for $(\Omega; \Xi', \mathcal{O})$ iff there are dotted arrows such that the following diagram commutes*

$$\begin{array}{ccccc}
\Omega X & \xrightarrow{\ \omega\ } & X & \xrightarrow{\ \xi'\ } & \Xi' X \\
{\scriptstyle \Omega!}\downarrow & & \downarrow{\scriptstyle !} & & \downarrow{\scriptstyle \Xi'!} \\
\Omega Z & \dashrightarrow & Z & \dashrightarrow & \Xi' Z
\end{array}$$

*where $! : X \to Z$ is the unique coalgebra morphism from the observers $X \xrightarrow{\ o\ } \mathcal{O}X$ to the final $\mathcal{O}$-coalgebra. A **morphism** between models is an arrow which is simultaneously a morphism for the algebra and the coalgebra part. The resulting **category of models** is denoted by $\mathsf{Mod}(\Omega; \Xi', \mathcal{O})$. A model is **fully abstract** iff $!$ is an embedding (injective).*

Remark 1. The diagram expresses in an abstract way that the model satisfies the condition for observational algebras of Definition 4. Indeed, for any observational signature Σ_{Obs} (in the sense of Section 2) whose observers have only one non-observable argument sort, appropriate polynomial functors $\Omega, \Xi', \mathcal{O}$ over the base category Set^n can be found such that the observational equivalence induced by $\mathcal{O}$ is a congruence w.r.t. Ω and Ξ' (and anyway w.r.t. $\mathcal{O}$). This is due to the fact that ! identifies precisely all observationally equivalent points.

Definition 18 (Constructor-based models). *Let $\mathcal{X}$ be a category with binary coproducts. A* ***constructor-based signature*** *$(\Omega', \mathcal{R}; \Xi)$ over $\mathcal{X}$ consists of functors $\Omega', \mathcal{R}, \Xi : \mathcal{X} \to \mathcal{X}$ such that an initial $\mathcal{R}$-algebra $\mathcal{R}I \xrightarrow{\iota} I$ exists. $\mathcal{R}$ is called the* ***reachability constraint****. An algebra-coalgebra pair $([\omega', \rho], \xi)$ satisfies the reachability constraint $\mathcal{R}$ and is called a* ***model*** *for $(\Omega', \mathcal{R}; \Xi)$ iff there are dotted arrows such that the following diagram commutes*

$$\begin{array}{ccccc} \Omega' X & \xrightarrow{\omega'} & X & \xrightarrow{\xi} & \Xi X \\ {\scriptstyle \Omega'?}\uparrow & & {\scriptstyle ?}\uparrow & & \uparrow{\scriptstyle \Xi?} \\ \Omega' I & \dashrightarrow & I & \dashrightarrow & \Xi I \end{array}$$

where $? : I \to X$ is the unique algebra morphism from the initial $\mathcal{R}$-algebra to the constructor-algebra $\mathcal{R}X \xrightarrow{\rho} X$. A ***morphism*** *between models is an arrow which is simultaneously a morphism for the algebra and the coalgebra part. The resulting* ***category of models*** *is denoted by $\mathsf{Mod}(\Omega', \mathcal{R}; \Xi)$. A model is* ***reachable*** *iff ? is a quotient (surjective).*

Remark 2. The diagram expresses in an abstract way that the model satisfies the condition for constructor-based algebras of Definition 10: Indeed, for any constructor-based signature Σ_{Cons} (in the sense of Section 3) appropriate polynomial functors $\Omega', \mathcal{R}, \Xi$ over the base category Set^n can be found such that the image of I under ? is the reachable part and whenever one of the operations ω', ξ takes all its arguments from the image of I under ? the results are in the image again (and this is anyway true for ρ).

Definitions 17 and 18 give rise to a **duality principle** for constructor-based and observational models which is stated formally by the following isomorphisms of categories:

$$\mathsf{Mod}(\Omega; \Xi', \mathcal{O})^{\mathrm{op}} \simeq \mathsf{Mod}(\Xi'^{\mathrm{op}}, \mathcal{O}^{\mathrm{op}}; \Omega^{\mathrm{op}}),$$

$$\mathsf{Mod}(\Omega', \mathcal{R}; \Xi)^{\mathrm{op}} \simeq \mathsf{Mod}(\Xi^{\mathrm{op}}; \Omega'^{\mathrm{op}}, \mathcal{R}^{\mathrm{op}}).$$

The isomorphisms map models $(\alpha, \beta)^{\mathrm{op}} = (\alpha, \beta)$ to $(\beta^{\mathrm{op}}, \alpha^{\mathrm{op}})$. In the following, we identify $(\alpha, \beta)^{\mathrm{op}}$ with $(\beta^{\mathrm{op}}, \alpha^{\mathrm{op}})$.

As a consequence of the duality principle we obtain, for example:

Theorem 7.

1. *An algebra-coalgebra pair M for $(\Omega, \Xi' \times \mathcal{O})$ satisfies the observability constraint $\mathcal{O}$ iff M^{op} satisfies the reachability constraint $\mathcal{O}^{op}$.*
2. *An algebra-coalgebra pair M for $(\Omega' + \mathcal{R}, \Xi)$ satisfies the reachability constraint $\mathcal{R}$ iff M^{op} satisfies the observability constraint $\mathcal{R}^{op}$.*
3. *A model M is reachable iff M^{op} is fully abstract.*

The first theorem similar to (3) is due to Kalman [12] and was proved for linear systems in control theory. Later, Arbib and Manes [2] brought to light the general principles underlying this duality by considering—essentially—systems (automata) as Ω-algebras for arbitrary functors Ω. Compared to [2] the main point of our formalization consists in the use of coalgebras to formalise our notion of observation and in the consideration of observability and reachability constraints.

7 Conclusion

In this paper we have studied and formalised the duality between observability and reachability concepts used in algebraic approaches to software development taking into account observability and reachability constraints. We hope that the exhibition of this duality contributes to a clarification of specification methodologies and of their semantic foundations.

As a particular outcome we have presented the novel institution of constructor-based logic. The formal dualisation of the categorical representation of observational logic in [8] gave us the intuition for the adequate notions of constructor logic which provide sufficient flexibility to describe the semantically correct realizations of a specification from the reachability point of view (in the same way as observational logic does from the observational point of view). In this paper we have focused on a comparison of the two concepts and *not* on their integration. The combination of the two concepts offers a promising perspective of future reserach. We believe that such an integration will be strongly related to (a generalization of) the notion of partial observational equivalence considered e.g. in [4] and [10].

References

1. J. Adámek, H. Herrlich, and G. Strecker. *Abstract and Concrete Categories*. John Wiley & Sons, 1990.
2. M.A. Arbib and E.G. Manes. Adjoint machines, state-behaviour machines, and duality. *Journ. of Pure and Applied Algebra*, 6:313–344, 1975.
3. M. Bidoit and R. Hennicker. Observer complete definitions are behaviourally coherent. In *OBJ/CafeOBJ/Maude at Formal Methods '99*, pages 83–94. THETA, 1999.
4. M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 25:149–186, 1995.

5. J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39 (1):95–146, 1992.
6. J. Goguen and G. Roşu. Hiding more of hidden algebra. In J.M. Wing, J. Woodcock, and J. Davies, editors, *Formal Methods (FM'99)*, volume 1709 of *LNCS*, pages 1704–1719. Springer, 1999.
7. R. Hennicker and M. Bidoit. Observational logic. In Armando Haeberer, editor, *Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *LNCS*. Springer, 1999.
8. R. Hennicker and A. Kurz. (Ω, Ξ)-logic: On the algebraic extension of coalgebraic specifications. In B. Jacobs and J. Rutten, editors, *Coalgebraic Methods in Computer Science (CMCS'99)*, volume 19 of *Electronic Notes in Theoretical Computer Science*, pages 195–211, 1999.
9. C.A.R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
10. M. Hofmann and D.T. Sannella. On behavioural abstraction and behavioural satisfaction in higher-order logic. In *TAPSOFT '95*, volume 915 of *LNCS*, pages 247–261. Springer, 1995.
11. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62, 1997.
12. R. E. Kalman, P. L. Falb, and M. A. Arbib. *Topics in Mathematical System Theory.* McGraw-Hill, 1969.
13. J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types.* Wiley and Teubner, 1996.
14. Grant Malcolm. Behavioural equivalence, bisimulation, and minimal realisation. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specification*, volume 1130 of *LNCS*, pages 359–378. Springer, 1996.
15. CoFI Task Group on Language Design. Casl - the cofi algebraic specification language - summary. `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.
16. Horst Reichel. *Initial computability, algebraic specifications, and partial algebras.* Oxford, Clarendon Press, 1987.
17. D.T. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *Journal Comput. System Sci.*, 34:150–178, 1987.
18. D.T. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.
19. M. Wirsing and M. Broy. A modular framework for specification and information. In F. Orejas J. Diaz, editor, *TAPSOFT '89*, volume 351 of *LNCS*, pages 42–73. Springer, 1989.

The Finite Graph Problem for Two-Way Alternating Automata

Mikołaj Bojańczyk[1]

Uniwersytet Warszawski, Wydzia MIM, Banacha 2, Warszawa, Polska
bojan@mimuw.edu.pl

Abstract. Two-way alternating automata on trees were introduced by Vardi [Va98]. Here we consider alternating two-way automata on graphs and show the decidability of the following problem: ,,does a given automaton with the Büchi condition accept any *finite* graph?" Using this result we demonstrate the decidability of the finite model problem for a certain fragment of the modal μ-calculus with backward modalities.

1 Introduction

In this paper we consider the propositional μ-calculus with *backwards modalities*. It is an extension of the propositional μ-calculus introduced by Kozen [Ko83], which in itself is a very strong logic, subsuming such formalisms as Program Decision Logic, various temporal logics such as CTL and process logics such as YAPL. The various propositional μ-calculi are subject to much research, because, while being expressive, they still have reasonable computational complexity.

The propositional μ-calculus extends propositional logic with least and greatest fix-point operators μ, ν and modal quantification $\exists$, $\forall$ (sometimes written as $\Diamond$, $\Box$). The calculus with *backwards modalities*, aside from the above, allows for quantification over backward modalities, denoted by $\exists^-$ and $\forall^-$. Thus, for example, a formula of the form $\exists^-\phi$ (respectively $\exists\phi$) states that ϕ occurs in some predecessor (respectively succsessor) of the current state. This calculus is notably stronger than the propositional modal μ-calculus without backward modalities, in particular it no longer has the *finite model property*, i. e. it admits sentences which are satisfiable only in infinite structures. This gives rise to the following natural decision problem: ,,Is a given sentence of the propositional modal μ-calculus with backward modalities satisfiable in some *finite* structure?"

The more or less standard way to tackle this problem would be to use automata on infinite trees. Extensions of modal logic often have the *tree model* property, which states that structures can be *unraveled* into indistinguishable tree-like structures. This allows one to reap from the rich resources of automata on infinite trees. And thus, for instance, the decidability of the satisfiability problem for the propositional μ-calculus can be proven by a reduction to the emptiness problem for *alternating* automata on infinite trees with the *parity condition*, while the satisfiability problem for the μ-calculus with backward modalities can be reduced to the emptiness problem for alternating *two-way* automata on infinite trees with the parity condition [Va98].

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 88–103, 2001.

As much as the tree model property is helpful in researching the satisfiability problem of the μ-calculus with backward modalities, things get more complicated where the finite model problem is concerned. The reason is that, unfortunately, finite models rarely turn out to be trees. On the contrary, the class of sentences satisfiable in finite tree models is a small subset of the class of sentences with arbitrary finite models, a class for which the finite model problem is quite easily decidable. For this reason, while investigating the finite model problem we will consider automata on arbitrary graphs, not on trees. Using a close correspondence between such automata and the μ-calculus with backward modalities, we will reduce the finite model problem to the question of whether a given automaton accepts some finite graph, the *finite graph* problem.

In this paper we consider alternating two-way automata with the Büchi acceptance condition. The Büchi acceptance condition means the automaton has a special, *accepting*, subset of the set of all states and in order for a run of the automaton to be accepting, one of the accepting states must occur infinitely often on every computation path. Even though it is weaker than the full parity condition of normal two-way automata, the Büchi condition is sufficient to recognize a large class of graph languages. The main result of this paper is a proof of the decidability of the finite graph problem for two-way alternating automata with the Büchi acceptance condition. Having done this, we decide the finite model property for a certain subset of the μ-calculus with backward modalities by a reduction to the finite graph problem for alternating two-way Büchi automaton.

For the decidability proof of the finite graph problem, given an alternating two-way automaton A, we construct a nondeterministic automaton A' on trees that accepts unravelings of finite graphs accepted by A. In order to find a way of distinguishing unravelings of finite and infinite graphs, we introduce the concept of a graph *signature*. A two-way alternating automaton's signature in a particular vertex of a graph says what is the longest sequence of non-accepting states that can appear in a run of the automaton beginning with that vertex. It can be proven that finite graphs have finite signatures, moreover, unravelings of finite graphs also have finite signatures. It also turns out that accepting a graph, perhaps infinite, of finite signature is in a way sufficient for accepting a finite graph; we discover one can ,,loop" a finite signature tree into an acceptable finite graph. The essential technical ,,small signature" theorem allows us to find a tractable bound on the signature of finite graphs. The proof of this theorem uses a new approach of successive tree approximations of the final bound signature tree. The automaton A' accepts trees where the signature of the automaton A is bound by the constant in the small signature theorem.

In the last sections of the paper we introduce the μ-calculus with backward modalities and, closely following a paper by Vardi [Va98], show its correspondence with alternating automata with the parity condition. We then show what fragment of the calculus corresponds to Büchi condition automata and prove the decidability of the finite model problem for this fragment. Finally, we comment on the possibility of further work regarding the so-called Guarded Fragment.

2 Games with the Parity Condition

Games with the parity condition are an important concept in the theory of infinite tree automata. In particular, the semantics of two-way alternating automata used in this paper are defined by a certain game with the parity condition.

Definition 2.1 (Parity condition game) A *game with the parity condition* is a tuple $\mathbb{G} = \langle V_0, V_1, E, v_0, \Omega \rangle$, where V_0 and V_1 are disjoint sets of *positions*, the function $\Omega : V = V_0 \cup V_1 \to \{0, \ldots, N\}$ is called the *coloring* function, $E \subseteq V \times V$ is the set of *edges*, and $v_0 \in V$ is some fixed *starting position*. We additionally assume that for every position $v \in V$, the set of outgoing edges $(v, w) \in E$ is finite.

The game is played as follows. The play starts in the vertex v_0. Assuming the game has reached in turn j a vertex $v_j \in V_i$, $i \in \{0, 1\}$, the player i chooses some vertex v_{j+1} such that $(v_j, v_{j+1}) \in E$. If at some point one of the players players cannot make a move, she loses. Otherwise, assume $v_0, v_1, \ldots$ is the infinite sequence of vertices visited in the game. This infinite play is winning for player 0 if the sequence $\Omega(v_0), \Omega(v_1), \ldots$ satisfies the parity condition, otherwise it is winning for player 1.

Definition 2.2 (Parity condition) A sequence $\{a_i\}$ of numbers belonging to some finite set of natural numbers is said to satisfy the *parity condition* if the smallest number occurring infinitely often in $\{a_i\}_{i \in \mathbb{N}}$ is even.

The notions of strategy and winning strategy are introduced in the usual manner. We say a strategy is *memoryless* if the choice of the next vertex depends solely upon the current vertex. A very important theorem [EJ91,Mo91], which will enable us to consider only memoryless strategies, says:

Theorem 2.1 (Memoryless determinacy theorem). *Every game with the parity condition is determined, i. e. one of the players has a winning strategy. Moreover, the winner also has a memoryless winning strategy.*

3 Two-Way Alternating Automata on Graphs

Two-way alternating automata were introduced by Vardi in [Va98] as a tool for deciding the satisfiability problem of the modal μ-calculus with backward modalities. As opposed to ,,normal" alternating automata, two-way automata can travel backwards across vertices. Here we consider automata not on trees, as in the original paper, but on arbitrary graphs.

Given a set of states Q, we consider formulas built using the logical connectives $\vee$ and $\wedge$ from atoms of the form $\forall Q$, i. e. from the set $\{\forall q, \forall^- q : q \in Q\}$ and $\exists Q$ i. e. $\{\exists q, \exists^- q : q \in Q\}$. We will denote the set of such formulas by $Form(A)$. Moreover, we partition the set $Form(A)$ into *conjunctive* formulas $Con(A)$, i. e. either atoms from $\forall Q$ or formulas of the form $\phi_1 \wedge \phi_2$ and *disjunctive* formulas $Dis(A)$, i. e. atoms from $\exists Q$ and formulas of the form $\phi_1 \vee \phi_2$.

Definition 3.1 (Two-way alternating automaton) A *two-way alternating automaton* on Σ-labeled graphs is the tuple:

$$\langle Q, q_0, \Sigma, \delta, \Omega \rangle$$

Q is a finite set of *states*, $q_0 \in Q$ is called the *starting state* and Ω is a function assigning to each state $q \in Q$ a natural number $\Omega(q)$ called the *color* or *priority* of q. The *transition function* δ is of the form $\delta : Q \times \Sigma \to Form(A)$.

In this paper, when speaking of graphs, we will use *Σ-labeled graphs with a starting position*, where Σ is some finite set of labels. Such a graph is a tuple $G = \langle V, E, e, v_0 \rangle$, where V is the set of *vertices*, $E \subseteq V \times V$ is the set of *edges*, the *labeling* is a function $e : V \to \Sigma$ and $v_0 \in V$ is the *starting position*.

To define the semantics of two-way alternating automata, we shall use games with the parity condition. Given a Σ-labeled graph $G = \langle V, E, e, v_0 \rangle$ and a two-way alternating automaton $A = \langle Q, q_0, \Sigma, \delta, \Omega \rangle$, we define the game $\mathbb{G}(A, G) = \langle V_0, V_1, E', v_0', \Omega' \rangle$. The set of positions of $\mathbb{G}(A, G)$ is defined $V_0 = Dis(A) \times V$ and $V_1 = Con(A) \times V$.

For briefer notation, let $\overset{\exists}{\forall}$ stand for any one of the quantifiers $\exists^{\pm}$ and $\forall^{\pm}$. For any edge $(u, w) \in E$ we will write $(u, w)^{-1}$ to denote the reverse of the edge, that is (w, u). To further simplify notation, assume for $(u, w)^1$ the edge (u, w), similarly let $\exists^1 = \exists$, $\exists^{-1} = \exists^-$, $\forall^1 = \forall$ and $\forall^{-1} = \forall^-$.

The edges of the game are set as follows:

- For an atom $(\overset{\exists}{\forall} q, v) \in (\forall Q \times V) \cup (\exists Q \times V)$ there exists an edge to $(\delta(q, e(w)), w)$ if $(v, w)^i \in E$, where i is 1 if the quantifier is positive and -1 otherwise.
- For a non-atomic formula (ϕ, v) there exists an edge to (ϕ', v) for each subformula ϕ' of ϕ.

The coloring Ω' in the game $\mathbb{G}(A, G)$ is defined as follows: for $(\overset{\exists}{\forall} q, v) \in (\forall Q \times V) \cup (\exists Q \times V)$ we set $\Omega'(\overset{\exists}{\forall} q, v) = \Omega(q)$. For the remaining positions we set, say, $max(\Omega(Q)) + 1$, so that their color is irrelevant. The starting position in $\mathbb{G}(A, G)$ is $(\delta(q_0, e(v_0)), v_0)$.

Definition 3.2 (Acceptance by the automaton) We say the automaton A *accepts* a graph G under strategy s if s is a winning strategy for player 0 in the game $\mathbb{G}(A, G)$. Such a strategy s is called *accepting*. We say A accepts graph G if there exists a strategy s such that A accepts G under s.

A very important concept that will be used here is the tree unraveling of a graph. By a *two-way path* in a graph $G = \langle V, E, e, v_0 \rangle$ we mean any sequence of neighboring vertices, that is, any sequence $v_0, \ldots, v_i$ such that $(v_j, v_{j+1}) \in E$ or $(v_{j+1}, v_j) \in E$.

Definition 3.3 (Tree unraveling) Given a graph $G = \langle V, E, e, v_0 \rangle$, its *tree unraveling* is the graph $Un(G) = \langle V', E', e', v_0 \rangle$, where the set of vertices V' is the set of finite two-way paths in G starting in v_0, the set of edges is defined

$E' = \{((\pi, v), (\pi, v, w))^i : (\pi, v, w) \in V', (v, w)^i \in E\}$ and the labeling is set as $e'(\pi, v) = e(v)$.

Note that this is a *two-way* tree, that is, edges between a son and father can be either forward or backward. The *depth* of a vertex $\pi = (v_0, \ldots, v_n)$ is the number n. For two vertices π_1 and π_2 of such a tree, we say π_1 is a successor of π_2 if π_2 is an initial fragment of the path π_1. For a tree T we use $T|_\pi$ to signify the subtree of T with the root at π, $T|^i_\pi$ is the subtree with the root at π and of depth i and $T|^i$ is defined as $T|^i_\lambda$, where λ is the root of T.

Having a tree unraveling we define the canonical projection $\Pi : V' \to V$, so that $\Pi(\pi, v) = v$ and expand this projection onto the positions of $\mathbb{G}(A, Un(G))$ and $\mathbb{G}(A, G)$ so that $\Pi(x, (\pi, v)) = (x, v)$.

Definition 3.4 (Strategy unraveling) We say the strategy $Un(s)$ is the *unraveling of strategy* s if $\Pi \circ Un(s) = s \circ \Pi$.

We omit the trivial proof of:

Lemma 3.1. *The automaton A accepts a graph G under strategy s iff A accepts $Un(G)$ under $Un(s)$.*

It can be shown that one-way alternating automata on graphs have a certain finite graph property, that is, if a given one-way alternating automaton accepts any kind of graph, it also accepts a finite graph. This, however, is not the case when speaking of two-way alternating automata. We will conclude this section with an example of an automaton that accepts only infinite graphs.

Example 3.0.1

Consider the following two-way automaton $A = \langle Q, q_0, \Sigma, \delta, \Omega \rangle$, where $Q = \{q_0, q_1\}$, $\Omega(q_0) = 0, \Omega(q_1) = 1$, and $\Sigma = \{a\}$. The transition function δ is defined as follows:

$$\delta(q_0, a) = \exists q_0 \wedge \forall^- q_1$$

$$\delta(q_1, a) = \forall^- q_1$$

We will consider a play in the game $\mathbb{G}(A, G)$ where $G = \langle \mathbb{N}, \{(n, n+1) : n \in \mathbb{N}\}, e, 0\} \rangle$, such that $e(n) = a$ for all $n \in \mathbb{N}$. The play starts in formula $\exists q_0 \wedge \forall^- q_1$ at vertex 0. This is a position for player 1, let's assume he chooses the subformula $\exists q_0$ (the play stays at 0). Now player 0 has to choose a neighboring (in G) vertex along a forward edge. He has to choose 1; the position is now $\exists q_0 \wedge \forall^- q_1$ at vertex 0. This goes on until, say, we reach 10. Now let's assume player 1 chooses the subformula $\forall^- q_1$. Now it is his choice to choose a neighboring vertex in G, along a backward edge; he has to choose vertex 9. The play then goes on through positions $(\forall^- q_1, 9), \ldots, (\forall^- q_1, 0)$ in which last position player 1 loses for a lack of possible moves. Consider now a different play – player 1 always chooses the subformula $\exists q_0$. The play goes through positions $(\exists q_0 \wedge \forall^- q_1, 0), (\exists q_0, 0), \ldots, (\exists q_0 \wedge \forall^- q_1, k), (\exists q_0, k), \ldots$. The only color appearing infinitely often in this play is 0, thus player 0 wins.

Analyzing the game $\mathbb{G}(A, G)$, one will notice that in the graph G, player 0 has a winning strategy. It can also be proven, that the automaton A accepts only graphs with an infinite forward path where no infinite backward path is ever reachable. In particular A accepts only infinite graphs.

3.1 Automaton Paths

Let us fix a two-way alternating automaton A. Given a play r in the game $\mathbb{G}(A, G)$ we can define $\tilde{r}$ as the sequence of state-vertex pairs visited in the play r. For instance, in the example above, for the first play r, we have $\tilde{r} = (q_0, 0), (q_1, 1), \ldots, (q_0, 10), (q_1, 9), (q_1, 8), \ldots, (q_1, 0)$ The following is a key definition:

Definition 3.5 (Automaton path) Let r be a play consistent with the strategy s in $\mathbb{G}(A, G)$ and let $\tilde{r} = (q_1, v_1), \ldots$ be the projection of r onto $Q \times V$. Any contiguous subsequence of $\pi(r)$ is called an *automaton path* G consistent with the strategy s, or, more concisely, an automaton path in G, s.

We use $\omega(G, s)$ to denote the set of all automaton paths in G, s. Sometimes we shall omit the word automaton and simply say path, where confusion can arise we shall distinguish automaton paths from graph paths. The length of a path is denoted by $|\omega|$. Given a path $\omega = (q_1, v_1), \ldots, (q_n, v_n)$, we say the path $\omega' = (q_2, v_2), \ldots, (q_{n-1}, v_{n-1})$ *leads* from (q_1, v_1) to (q_n, v_n), which is written as $(q_1, v_1) \rightarrow^{\omega'} (q_n, v_n)$. By ω_i we denote the i-th element of the path ω, that is (q_i, v_i) Sometimes we shall simply write ω_i to denote simply either q_i or v_i, where the context clearly defines what type of result is needed. A path ω is a *sub-path* of ω', written as $\omega \sqsubseteq \omega'$, if ω is a contiguous subsequence of ω'. We define $||\omega||_Q$ as the set of states visited in ω and $||\omega||_V$ as the set of vertices visited in ω.

We say that a finite path ω *ends well* under the strategy s if it corresponds to a finite play f in $\mathbb{G}(A, G)$ winning for 0, that is one where player 1 cannot make a move. The following lemma gives a path characterization of the acceptance of automata:

Lemma 3.2. *The automaton A accepts a graph G under the strategy s iff every maximal (in terms of $\sqsubseteq$) finite path ends well under s and every infinite path $(q_0, v_0), (q_1, v_1), \ldots$ satisfies the parity condition for the sequence $\Omega(q_0), \Omega(q_1), \ldots$*

Corollary 3.1. *If the automaton A accepts the graph G under s there is no cycle ω in which the lowest priority in the set $\Omega(||\omega||_Q)$ is odd.*

4 The Finite Graph Problem

The example in Section 3.0.1 is a motivation for the following problem: ,,does a given alternating two-way automaton accept some finite graph?". Let us denote

this problem by *FIN-ALT* . We are not able to prove the decidablity of this full problem and we consider a simpler case. Fix a set of states Q and some subset $F \subseteq Q$. We say that the sequence of states $\{q_i\}_{i\in\mathbb{N}}$, $q_i \in Q$ satisfies the *Büchi acceptance condition*, if there exists a state $q \in F$ which appears infinitely often in the sequence $\{q_i\}_{i\in\mathbb{N}}$. This condition is obviously equivalent to the parity condition where we put $\Omega(q) = 0$ for $q \in F$ and $\Omega(q) = 1$ for $q \in Q/F$. Note that the example automaton in the previous section is is an automaton with the Büchi acceptance condition. It is thus meaningful to consider the *FIN-ALT* problem restricted to automata with the Büchi acceptance condition; we shall call this problem *FIN-ALT(B)*. Consider a graph $G = \langle V, E, e, v_0\rangle$ accepted by A. We shall now define the concept of an automaton *signature*, used in the key Theorem 4.1 of this paper.

Definition 4.1 (Signature) Let $v \in G$, $\omega \in \omega(G, s)$, and $k, i \in \mathbb{N}$.

- $Sig(\omega, i) \equiv_{def} min\{j : \Omega(\omega_{i+j}) = 0\}$
- $Sig^{G,s}(q, v) \equiv_{def} max\{Sig(\omega, i) : \omega \in \omega(G, s), \omega_i = (q, v)\}$.
- $Sig^{G,s}(v) \equiv_{def} (Sig^{G,s}(q, v))_{q\in Q}$

If the context is clear as to what graph and strategy are concerned, instead of $Sig^{G,s}(q, v)$ we shall write simply $Sig(q, v)$. Intuitively, $Sig(q, v)$ gives the longest possible length of an automaton path consisting of odd states starting in (q, v). We shall assume $Sig(q, v) = \infty$ if there is no such bound.

The following theorem is the main technical result of this paper.

Theorem 4.1 (Small signature theorem). *For any alternating two-way automaton with the Büchi acceptance condition $A = \langle Q, q_0, \Sigma, \delta, \Omega\rangle$ there exists a constant M doubly exponential on $|Q|$ such that the following three conditions are equivalent:*

1. *There exists a finite graph G such that A accepts G.*
2. *There exist a bound $N \in \mathbb{N}$, a tree T and an accepting strategy s such that for every vertex $v \in T$ and every state $q \in Q$, $Sig^{T,s}(q, v) < N$.*
3. *There exist a tree T and an accepting strategy s, such that for every vertex $v \in T$ and every state $q \in Q$, $Sig^{T,s}(q, v) < M$.*

Let us fix the automaton $A = \langle Q, q_0, \Sigma, \Omega\rangle$. The proof of this theorem is long and will be distributed across three subsections.

4.1 Proof of 1 ⇒ 2

First we shall state two lemmas, whose trivial proofs will be omitted.

Lemma 4.1. *If the automaton A accepts the finite graph $G = \langle V, E, e, v_0\rangle$ under s, then for every $q \in Q, v \in V$ we have $Sig^{G,s}(q, v) \leq |V||Q|$.*

Lemma 4.2. *The tree unwinding does not increase the signature, i. e.*

$$Sig^{Un(G),Un(s)}(q,(\pi v)) \leq Sig^{G,s}(q,v)$$

For the proof of $1 \Rightarrow 2$, assume A accepts the graph G. Then A accepts $Un(G)$ under the strategy $Un(s)$ (Lemma 3.1). Moreover, for every vertex πv of the tree $Un(G)$ and every state $q \in Q$ we have $Sig^{Un(G),Un(s)}(q,\pi v) \leq Sig^{G,s}(q,v) \leq |V||Q|$. The first inequality is due to Lemma 4.2, the second due to 4.1.

4.2 Proof of 3 ⇒ 1

It can be shown that for each graph G accepted by A, one can reduce the edge set of G, obtaining a graph G' also accepted by A, where the degree of each vertex is bounded by a certain constant $\hat{A}$, linear with respect to the size of the formulas in the transition function of A. We will now try to think of a way to encode strategies. Generally speaking, an accepting strategy can choose potentially any vertex for existential atoms, making a bounded representation difficult. Fortunately however, one can show that in graphs of degree bounded by $\hat{A}$ a strategy can be encoded as a number from 1 to a constant $\tilde{A}$, where $\tilde{A}$ is exponential over $\hat{A}$.

With every vertex v of the tree $T = \langle V, E, e, \lambda\rangle$ we shall associate two pieces of information constituting the *type* of v: the strategy s in the vertex v and $Sig(v)$. Because strategies are encoded by a number from 1 to $\tilde{A}$ and since by assumption we have $Sig(q,v) < M$ for every $q \in Q$, there exists a finite number of vertex types. We can thus find such a number i that all vertex types in the subtree $T|^{i+1}$ appear already in the subtree $T|^i$.

Let $f : T|^{i+1} \to T|^i$ be any function such that f restricted to $T|^i$ is the identity mapping and for every vertex $v \in T|^{i+1}$, the equalities $s(f(v)) = s(v)$ and $Sig(f(v)) = Sig(v)$ hold. In other words f assigns to vertices in $T|^{i+1}$ vertices of the same type from $T|^i$. Such a function exists by assumption on i. Consider now the following graph $T' = \langle T|^i, E', e \circ f, \lambda\rangle$ resulting from „looping" the tree T on the level i. We define the set of edges E' of the graph T' as follows: $E' = \{(f(v), f(v')) : (v,v') \in E\}$, where E is the set of edges of the original tree T.

It is an easy exercise to show that the function $Sig^{T,s}$ satisfies the conditions in the below lemma for the tree T', thus proving that A accepts the finite graph T'.

Lemma 4.3. *A accepts a finite graph G under the strategy s iff there exists a number $N \in \mathbb{N}$ and a function $\sigma : Q \times V \to \{0,..,N\}$ such that if $(q,v)(q',v') \in \omega(G,s)$ then $\sigma(q',v') \leq \sigma(q,v)$ and the inequality is proper if $\Omega(q) = 1$.*

Proof. For the left to right implication it is sufficient to notice that the signature Sig satisfies the above conditions. For the other direction, one has to show that the function σ majorizes Sig, so that each state with odd priority appears at most N times before an even priority state appears. □

4.3 Proof of 2 ⇒ 3

Let T, s be as in condition 2 of Theorem 4.1. By assumption we know there exists a certain, if perhaps difficult to estimate, bound N on the signature. We will now modify the tree T in such a way as to bound the signature by a tractable constant M.

First, we shall introduce the following definitions. We say a path $\omega \in \omega(G, s)$ is *bad* iff $\Omega(||\omega||_Q) = \{1\}$. We say (q, w) is 1-*accessible* from (q', v) in G, s, written as $(q', v) \rightarrow_1 (q, w)$ in G, s, if there exists a bad path $\omega \in \omega(G, s)$ such that $(q', v) \rightarrow^\omega (q, w)$. Moreover, we write $v \rightarrow_1 w$ if there exist states $q, q' \in Q$ such that $(q', v) \rightarrow_1 (q, w)$. We define the *bad neighborhood* $X_{D,s}(v)$ of a vertex v in the tree D under the strategy s as the set of v's 1−accessible (both ways) successors, i. e. $X_{D,s}(v) \equiv_{def} \{w \in D|_v : w \rightarrow_1 v \vee v \rightarrow_1 w \text{ in } D, s\}$

Let M' be a constant whose exact size depending on the size of the automaton A we will estimate later in this paper. For a tree D, strategy s and vertex v of the tree D, denote the following property as (*):

$$(*) X_{D,s}(v) \subseteq D|_v^{M'}$$

We will now construct a sequence of trees and accepting strategies $(D^0, s^0), (D^1, s^1), \ldots$ such that each two tree-strategy pairs (D^i, s^i) and (D^j, s^j) are identical up to depth $min(i, j) - 1$ and, moreover, if a vertex v of the tree D^i has a depth less than i, then the property (*) holds for v in D^i, s^i.

We will define the trees inductively with respect to i. Let D^0, s^0 be simply T^0, s^0. The above condition obviously holds for (D^0, s^0) since there are no vertices of depth less than zero. Assume now that we have constructed (D^i, s^i). We will define (D^{i+1}, s^{i+1}) by iterating the following Lemma 4.4 for successive vertices of depth i. Of course the conditions in the lemma are satisfied by D^0, s^0.

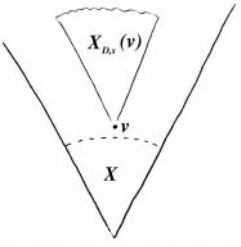


Fig. 1. The tree D

Lemma 4.4. *Let D, s be such that*

1. *A accepts D under s*
2. *$X_{D,s}(w)$ is finite for every vertex w of the tree D.*
3. *(*) holds for some contiguous subset of vertices X closed under the ancestor relation.*

Let v be the successor of any vertex in X. There exist a tree D' and strategy s' identical with D, s on the set $X \cup \{v\}$ such that the above three conditions hold for D', s' and (*) holds for v.

Proof. We prove 4.4 by iterating Lemma 4.5. □

Lemma 4.5. *Assume D, s and v satisfy the assumptions of Lemma 4.4. If (*) does not hold for v, there exists a tree D' and strategy s', identical with D, s on the set $X \cup \{v\}$ such that D', s' satisfies the assumptions of Lemma 4.4 and moreover the following inequality holds (the cardinality of both sets is finite by assumption 3):*

$$|X_{D',s'}(v)| < |X_{D,s}(v)|$$

Proof. We will consider a certain equivalence relation $\simeq$ defined on $X_{D,s}(v)$. Vertices equivalent under this relation are in a sense interchangeable (along with their subtrees). First, introduce for any two vertices v, w in the tree D the following symbol $O(v, w) \subseteq Q \times P(\{0, 1\}) \times Q$. Let $O(v, w) = \{(q, R, q') : \exists \omega \in \omega(D, s).(q, v) \rightarrow^{\omega} (q', w) \wedge \Omega(||\omega||_Q) = R\}$. Using this notation, we write $w \simeq w'$ if and only iff all the following conditions hold:

1. $s(w) = s(w')$
2. $O(w, w) = O(w', w')$
3. $O(v, w) = O(v, w')$
4. $O(w, v) = O(w', v)$

Let M' be the number of abstraction classes of the relation $\simeq$. It is easy to see that the bound M' is exponential with respect to $|Q|$. Assume now that the depth of $X_{D,s}(v)$ is greater than M'. In such a case we can find two vertices $w, w' \in X_{D,s}(v), w < w'$ such that $w \simeq w'$. Now take for D' (from Lemma 4.5) the tree resulting from substituting $D|_w$ for $D|_{w'}$, and let s' be the strategy s restricted to the new, smaller tree.

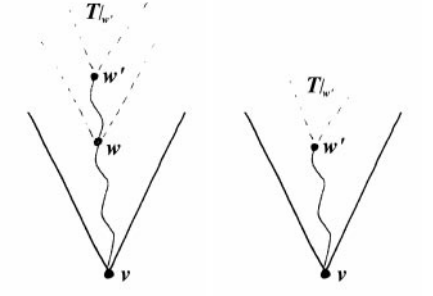


Fig. 2. Before and after the cut ($D|_v$ and $D'|_v$)

Definition 4.2 (Clean path) We say the path ω is *clean*, if $w' \not\in ||\omega||_V$. Clean paths can be either *upper*, that is contained in $D'|_{w'}$, or *lower* – the remainder.

Fact 4.1 For any state $q \in Q$, if (q, w') is accessible in D', s', then both (q, w) and (q, w') are accessible in D, s.

Proof. Let ω be such that $(q_0, v_0) \rightarrow^{\omega} (q, w')$ in D', s', where v_0 is the root and starting vertex of D'. The proof is by induction on the number of times w' appears in ω. For clean paths this is obvious, using condition 3 of $w \simeq w'$. Assume ω is not clean. Then $\omega = \omega^1(q', w')\omega^2$, where ω^2 is clean. Let us just consider the case where ω^2 is upper, the proof of the other is analogous. If ω^2 is upper then it is a correct path in D, s such that $(q', w') \rightarrow^{\omega^2} (q, w')$ in D, s. By condition 2 of $w \simeq w'$ we have $(q', w) \rightarrow^{\omega^3} (q, w)$ in D, s for some path ω^3. By induction hypothesis, both (q', w') and (q', w) are accessible in D, s, say by paths ω^A and ω^B. Linking the paths ω^A and ω^B with ω^2 and ω^3 respectively we obtain the desired assertion. □

Corollary 4.1. *For any $q \in Q, u \in D'$, if (q, u) is accessible in D', s' then it is also accessible in D, s.*

Now we are in a position to prove Lemma 4.5. That A accepts the tree D' under s is a rather tedious technical result whose proof we omit here. Secondly, we want to show that $X_{D',s'}(u)$ is finite for each vertex u in the tree D', s'. Take any bad path ω that goes through u and consider two cases:

- ω is clean. Using Corollary 4.1, we show $\omega \in \omega(D, s)$, so we can, according to the assumptions of Lemma 4.5, bound the depth of ω by a constant dependent only on the vertex u.
- ω is not clean. Again – divide ω into clean sub-paths:

$$\omega = \omega^{(1)}(w', q^{(1)})\omega^{(2)} \dots (w', q^{(n-1)})\omega^{(n)}$$

  By easy induction we can prove that if $\omega^{(i)}$ upper, then $w' \rightarrow_1 \omega^{(i)}$ w D, s. This proves that the depth of the path ω is no greater than the depth of $X_{D,s}(w')$, in particular it is bound.

Finally, the facts that the cut preserves the property (*) for vertices $u \in X$ and that $|X_{D',s'}(v)| < |X_{D,s}(u)|$ are implied by the following fact:

Fact 4.2 If $u \in X \cup \{v\}$, then $X_{D',s'}(u) \subseteq X_{D,s}(u) \cap D'$

To prove Fact 4.2, we need to prove the following lemma:

Lemma 4.6. *If $(q, v) \rightarrow_1 (q', w')$ [respectively $(q', w') \rightarrow_1 (q, v)$] in D', s', then we have both $(q, v) \rightarrow_1 (q', w)$ and $(q, v) \rightarrow_1 (q', w')$ [$(q', w') \rightarrow_1 (q, v)$ and $(q', w) \rightarrow_1 (q, v)$] in D, s.*

The proof is wholly analogous to the proof of Lemma 4.1 and will be omitted. Thus equipped we can finish the proof of Fact 4.2 and along with that, the whole Lemma 4.5.

Let $u \in X \cup \{v\}$ and $u' \in X_{D',s'}(u)$. We want to show that $u' \in X_{D,s}(u) \cap D'$. We will only consider the first case in the definition of $X_{D',s'}(u)$, the other one is similar. Assume then that $u \to_1 u'$. There exists states $q, q' \in Q$ and a bad path $\omega \in \omega(D', s')$ such that $(q, u) \to^\omega (q', u')$. If ω is clean, then obviously $(q, u) \to^\omega (q', u')$ holds also in D, s and so $u' \in X_{D,s}(u) \cap D'$. In the other case, let $\omega = \omega_1(q'', w')\omega_2$, where ω_2 is clean. Consider two cases:

- ω_2 is upper. According to Lemma 4.6, we have $(q, u) \to_1 (q'', w')$ in D, s. Since the path ω_2 is a correct path in D, s, we have $(q, u) \to_1 (q', u')$ in D', s' and, consequently, $u' \in X_{D,s}(u) \cap D'$.
- ω_2 is lower. Analogously to the above, except using $(q, u) \to_1 (q', w)$.

□

(end of proof of 4.5)

Having thus proven 4.5 we can conclude by using the trees $(D^0, s^0), (D^1, s^1), \ldots$ to prove the 2 $\Rightarrow$ 3 implication. Since the trees $(D^i, s^i), (D^j, s^j)$ are identical up to depth $min(i, j) - 1$, we can define the limit tree D and strategy s which are identical with each D^i, s^i up to depth $1 - i$. Now take a vertex $v \in D$ and let $n = |v| + M'$. Since $X_{D^n,s^n}(v)$ contains only vertices of depth less than n, we see that $X_{D,s}(v) \subseteq D|_v^{M'}$, or, in other words, (*) holds for all all vertices of D, s. It is now a trivial observation that the maximal length M of a bad path contained within $D|_v^{M'}$ is at most exponential with respect to M', otherwise we would have a cycle.

4.4 The *FIN-ALT(B)* Problem Is Decidable

Armed with Theorem 4.1 we are ready to show the decidability of the *FIN-ALT(B)* problem.

Theorem 4.2. *The FIN-ALT(B) problem is decidable in DEXPTIME.*

Proof. Given an automaton A we shall construct a nondeterministic automaton A' on full $\hat{A}$-ary trees over the alphabet $\Sigma \times \{1, \ldots, \tilde{A}\}$. The automaton A' will recognize an infinite tree T labeled by the strategy s iff for every vertex v of the tree T, the signature $Sig^{T,s}$ is smaller than the constant M from Theorem 4.1. The automaton A' guesses a function $\sigma : V \times Q \to \{1, \ldots, M\}$ defined on vertices of the tree T and then checks if the assumptions of Lemma 4.3 hold.

This kind of local property can be checked by a nondeterministic automaton whose number of states is doubly exponential with regards the size of Q of the automaton A (this is how big the constant M is). The fact that A' works on $\hat{A}$-ary trees is not limiting, as we have observed before. Thanks to Theorem 4.1, we need only look for trees whose signature is bounded by M, so the emptiness problem for A' (in terms of nondeterministic automata) is equivalent to the emptiness of A (in terms of two-way alternating automata).

Since the automaton A' checks only a local consistency, it has the following nice property: if A' has an infinite run, then it has an accepting run. It can be proved that for such automata the emptiness problem is decidable in polynomial time and thus we obtain the time in the theorem's conclusion. □

5 The Propositional μ-Calculus

Let $AP = \{p, q, \ldots\}$ be a set of atomic propositions, and let $VAR = \{X, Y, \ldots\}$ be a set of propositional variables.

Definition 5.1 (Formulas of the calculus) The set of formulas of the μ-calculus is the smallest set such that:

- Every atomic proposition $p \in AT$ and its negation $\neg p$ are formulas
- Every variable $X \in VAR$ is a formula
- If α and β are formulas and $X \in VAR$ then the following are formulas:

$$\alpha \vee \beta, \alpha \wedge \beta, \exists\alpha, \forall\alpha, \exists^{-}\alpha, \forall^{-}\alpha, \mu X.\alpha, \nu X.\alpha$$

We call μ and ν, respectively, the least and greatest fix-point operators. We will write ϑ to signify any one of the two operators. Formulas of the calculus are interpreted in so-called *Kripke structures*.

Definition 5.2 (Kripke structure) A *Kripke structure* $K = \langle V, E, S\rangle$ consists of a graph $\langle V, E\rangle$ along with a function $S : V \to P(AP)$ which assigns to each vertex the set of atomic propositions true in that vertex.

Let $K = \langle V, E, S\rangle$ be a Kripke structure, v a valuation, i. e. any function $v : VAR \to P(V)$. As usual, we define $v[W/X]$ as the valuation obtained from v by substituting the set $W \subseteq V$ for the variable X. The interpretation of a formula ϕ in a given Kripke structure under the valuation v, written as $\phi^K[v]$, is defined inductively as follows:

- For atomic propositions $p \in AP$, $p^K[v] = \{u \in V : p \in S(u)\}$
- For variables $X \in VAR$, $X^K[v] = v(X)$
- $(\phi_1 \wedge \phi_2)^K[v] = \phi_1^K[v] \cap \phi_2^K[v]$
- $(\phi_1 \vee \phi_2)^K[v] = \phi_1^K[v] \cup \phi_2^K[v]$
- $(\exists^k\phi)^K[v] = \{u \in V : \exists w \in V.(u, w)^k \in E \wedge w \in \phi^K[v]\}, k \in \{1, -1\}$
- $(\forall^k\phi)^K[v] = \{u \in V : \forall w \in V.(u, w)^k \in E \Rightarrow w \in \phi^K[v]\}, k \in \{1, -1\}$
- $(\mu X.\phi)^K[v] = \bigcap\{V' \subseteq V : \phi^K[v[V'/X]] \subseteq V'\}$
- $(\nu X.\phi)^K[v] = \bigcup\{V' \subseteq V : V' \subseteq \phi^K[v[V'/X]]\}$

5.1 Enhanced Automata

For a briefer notation we will add two new mechanisms to two-way alternating automata which do not expand their expressive power. Let $\circ Q$ be the set $\{\circ q : q \in Q\}$. Our new *enhanced automata* are identical to alternating two-way automata, save they have a more complicated transition function. In an enhanced automaton, δ assigns to each state-label pair $(q, a) \in Q \times \Sigma$ a formula $\delta(q, a)$ built from atoms of the form $\exists Q$, $\forall Q$ (as before) and, additionally, $\circ Q$ and *true* and $false$.

We interpret *true* and $false$ as follows: when player 0 reaches *true*, he wins, while when he reaches $false$, he looses; the reverse holds for player 1. On the other hand, $\circ q$ means the automaton stays in the same vertex, only changes its state to q. We omit the easy, yet technical, proof of:

Lemma 5.1. *Every enhanced automaton is equivalent to a two-way alternating automaton with only a polynomial blowup of size. The finite graph problem for enhanced automata with the Büchi condition is decidable in double exponential time with respect to the size of states.*

5.2 Automata on Models

In this section we sketch the correspondence between the μ-calculus and enhanced automata. Let $AP(\phi)$ be the set of atomic predicates $p \in AP$ occurring in ϕ. Let $\Sigma_\phi = P(AP(\phi))$.

Definition 5.3 (Encoding) The *encoding* of a Kripke structure $K = \langle V, E, S\rangle$ from vertex $v_0 \in V$ is the graph $G(K, v_0) = \langle V, E, e, v_0\rangle$ where $e : V \to \Sigma_\phi$ is the restriction of S to Σ_ϕ.

Let ϕ be a sentence of the μ-calculus. We will construct an enhanced automaton A_ϕ on graphs that will recognize the encodings of models for ϕ. By $cl(\phi)$ we mean the smallest set of formulas closed under subformulas such that $\phi \in cl(\phi)$ and if $\vartheta X.\varphi(X) \in cl(\phi)$ then $\varphi(\vartheta X.\varphi(X) \in cl(\phi)$. Let $A_\phi = \langle cl(\phi), \phi, \Sigma_\phi, \delta, \Omega\rangle$. The transition function δ is defined as follows:

- $\delta(p, \sigma) = true$ if $p \in \sigma$, $false$ otherwise.
- $\delta(\neg p, \sigma) = false$ if $p \in \sigma$, $true$ otherwise.
- $\delta(\phi_1 \vee \phi_2, \sigma) = \circ\phi_1 \vee \circ\phi_2$
- $\delta(\phi_1 \wedge \phi_2, \sigma) = \circ\phi_1 \wedge \circ\phi_2$
- $\delta(\vartheta X.\varphi(X), \sigma) = \circ\varphi(\vartheta X.\varphi(X))$
- $\delta(\exists^k\varphi, \sigma) = \exists^k\varphi$ for $k \in \{1, -1\}$
- $\delta(\forall^k\varphi, \sigma) = \forall^k\varphi$ for $k \in \{1, -1\}$

We fix the coloring Ω so as to satisfy the following conditions:

- If the variable Y occurs freely in $\vartheta X.\phi_1(X)$ then $\Omega(\vartheta X.\phi_1(X)) > \Omega(\vartheta Y.\phi_2(Y))$.
- $\Omega(\vartheta X.\varphi(X))$ is even if and only if $\vartheta = \nu$
- Formulas not beginning with ϑ have a color no smaller than any fix-point formula.

Lemma 5.2. *[Va98] For every sentence ϕ and every Kripke structure K, $v_0 \in \phi^K$ if and only if A_ϕ accepts the encoding $G(K, v_0)$.*

Corollary 5.1. *(of Lemma 5.2) A sentence ϕ of the μ-calculus has a finite model if and only if the automaton A_ϕ accepts some finite graph.*

We say a sentence of the μ-calculus is *simple*, if it has no sub-formulas $\phi_\nu = \nu X.\phi_1(X)$ and $\mu Y.\phi_2(Y)$ such that Y occurs freely in ϕ_ν. Now it is easy to see that in Lemma 5.2, simple sentences are translated into enhanced automata whose acceptance condition is equivalent to the Büchi condition (the coloring consists of first a number of small even priorities, then larger odd priorities).

Theorem 5.1. *The problem whether a simple sentence of the μ-calculus with backward modalities has a finite model is in DEXPTIME.*

Proof. Given a simple sentence ϕ we construct an equivalent automaton A'_ϕ and solve this instance of the enhanced *FIN-ALT(B)* problem. Both the alphabet and set of states of the automaton A'_ϕ are polynomial with respect to the length of the formula ϕ. □

6 Closing Remarks

The main result of this paper is a proof of the decidability of the finite graph problem for two-way alternating automata with the Büchi condition. This can be used to prove the decidability of the finite model problem for a certain sub-logic of the propositional μ-calculus with backward modalities.

The proof is based on Theorem 4.1, which uses the concept of signature. In this theorem, implications $1 \Rightarrow 2$ and $3 \Rightarrow 1$ can be easily generalized for automata with an arbitrary parity condition. However, it remains an open problem whether such a generalization is possible for the implication $2 \Rightarrow 3$.

It seems that a decidability proof for the whole problem is desirable for ends other than the finite model problem of the μ-calculus with backward modalities. Two-way alternating automata are used in the paper [GW99] to decide the satisfiability of formulas of the so-called Guarded Fragment with fixed points.

The Guarded Fragment [ABN98] is a subset of first order logic, originally introduced as an elaboration on the translation of modal logic into first order logic. Currently the Guarded Fragment is subject to much research. It can be supposed that the *FIN-ALT* problem can be applied to solving the open problem of whether the finite model property for formulas of the Guarded Fragment with fixed points is decidable.

References

[ABN98] H. Andreka, J. van Benthem and I. Nemeti, Modal Languages and Bounded Fragments of Predicate Logic, *Journal of Philosophical Logic* , 27 (1998), pp. 217-274.

[EJ91] E. A. Emerson and C. Jutla: Tree Automata, Mu-Calculus and Determinacy, in *Proc. 32th IEEE Symposium on Foundations of Computer Science* pages 368-377.

[ES89] R. S. Street and E. A. Emerson: An Automata theoretic procedure for the propositional mu-calculus. *Information and Computation* , 81:249-264.

[GW99] E. Grädel and I. Walukiewicz: Guarded Fixed Point Logic, *Proc. 14th IEEE Symp. on Logic in Computer Science* , pages 45-54.

[Ko83] D. Kozen: Results on the Propositional μ-calculus, in *Theoretical Computer Science, Vol. 27* pages 333-354.

[LPZ85] O. Lichtenstein, A. Pnueli and L. Zuck: The Glory of the Past, in *Logics of Programs, Vol. 193 LNCS* pages 196-218.

[Mo91] A. W. Mostowski: Games with Forbidden Positions, Technical Report 78, University of Gdañsk, 1991.

[MS87] D. E. Muller and P. E. Schupp: Alternating automata on infinite trees, *Theoretical Computer Science* , 54:267-276, 1987.

[Ni88] D. Niwiński: Fixed Points vs. Infinite Generation, in *Proc. 3rd IEEE LICS* pages 402-409.

[St82] R. S. Streett:Propositional dynamic logic of looping and converse, in *Information and Control* , Vol. 54, pages 121-141.

[Th97] Wolfgang Thomas: Languages, Automata, and Logic, in *Handbook of Formal Language Theory, III* , Springer, pages 389-455.

[Va97] M. Vardi: Why is modal logic so robustly decidable?, in *Descriptive Complexity and Finite Models* , AMS, pages 149-184.

[Va98] M. Vardi: Reasoning About the Past with Two-way Automata, in *Vol. 1443 LNCS* pages 628-641.

High-Level Petri Nets as Type Theories in the Join Calculus

Maria Grazia Buscemi and Vladimiro Sassone

DMI, Università di Catania, Italy
buscemi@dmi.unict.it, vs@dmi.unict.it

Abstract. We study the expressiveness of the join calculus by comparison with (generalised, coloured) Petri nets and using tools from type theory. More precisely, we consider four classes of nets of increasing expressiveness, Π_i, introduce a hierarchy of type systems of decreasing strictness, Δ_i, $i = 0, \ldots, 3$, and we prove that a join process is typeable according to Δ_i if and only if it is (strictly equivalent to) a net of class Π_i. In the details, Π_0 and Π_1 contain, resp., usual place/transition and coloured Petri nets, while Π_2 and Π_3 propose two natural notions of high-level net accounting for dynamic reconfiguration and process creation and called reconfigurable and dynamic Petri nets, respectively.

1 Introduction

The join calculus [5,7] is an algebra of mobile processes with asynchronous name-passing communication that simplifies the π-calculus by enforcing the hypothesis of *unique receptors*. This means that there is at most one process receiving messages on a name, and is a distinctive feature of the join calculus that makes it suitable for distributed implementations, as channels may be allocated at their receptor process. The present work focuses on a version of the join calculus, studying its expressiveness by establishing a tight link to Petri nets.

Petri nets [16,17] are a fundamental model in concurrency, representing basic distributed machines that, although rudimentary, may exhibit complex interaction behaviours when processes (transitions) compete for shared resources (tokens). Operational in nature, Petri nets have been studied extensively from the semantic viewpoint (see [18] for some references).

The analogy between join terms and nets is relatively simple, and was first noticed in [1] and, recently, in [14]. Names, messages, and elementary definitions of the join calculus (cf. §2) correspond respectively to places, tokens, and transitions of Petri nets (cf. §3). The correspondence, however, runs short soon because nets are *not* a value-passing formalism and can express *no* mobility. Regrettably, they have a static, immutable network topology. This suggests to look for suitable extensions of Petri nets – in particular nets with mobility – that might be profitably applied to the study of mobile networks.

Supported by MURST project *TOSCA*. The authors wish to thank **BRICS**, Basic Research in Computer Science and project MIMOSA, INRIA Sophia Antipolis.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 104–120, 2001.

In the present paper we consider three extensions of place/transition Petri nets (Π_0) obtained by adding one by one, in a hierarchical fashion, the features needed to achieve the full expressiveness of join calculus. Namely, *value-passing*, accounted for by (a version of) *coloured nets* (Π_1), mobility as *network reconfigurability*, achieved by introducing *reconfigurable nets* (Π_2), and dynamically growing, *open networks*, modelled by the notion of *dynamic nets* (Π_3).

High level nets exist in several shapes and versions. The distinctive features of the mobile nets presented here is that input places of transitions are private and presets are immutable. A reconfigurable net can deliver tokens in different places at each firing, according to the input it receives, while a dynamic net can spawn a new net to life. But they *cannot* change the input arcs of any transitions. As in [13], where an algebra with such properties has been considered, we believe that this is the key to a tractable compositional semantic framework, the key to control generality. Most noticeably, it corresponds to unique receptiveness in the join calculus. A further naturality criteria in our view is that, as it turns out, dynamic nets can be proved *equivalent* to the join calculus.

The technical bulk of this work is characterising classes of join terms corresponding to the classes of the high-level nets discussed above. More precisely, our approach consists of designing four type systems Δ_i, $i = 0, \ldots, 3$, that single out the terms corresponding to the nets in Π_i, respectively for $i = 0, \ldots, 3$. In particular, for a fixed, semantic-preserving, well-behaved translation $[\![\_]\!]$, a join term P is typeable in Δ_i if and only if $[\![P]\!]$ belongs to Π_i. System Δ_0 is aimed at constraining terms to place/transition nets and, therefore, forbids dynamic process creation and nontrivial messages, Δ_1 relaxes the latter limitation, but enforces a strict distinction between channels and values, supporting value-passing but forbidding any mobility. Such distinction is then relaxed in Δ_2. System Δ_3 relaxes the dynamicity constraint and, therefore, turns out to be trivial, i.e., each term is typeable in Δ_3. That is, dynamic nets *coincide* with join terms. Finally, we provide a system Δ_4 that sums the features of all the others.

It is worth remarking that these systems are very rudimentary from the type theoretic viewpoint: there are no complex types or rules, nor sophisticated issues such as polymorphism [8,15] or similar. However, we believe that our formalisation is quite interesting, suggestive, and worth pursuing, because the nature of join-terms makes it natural to express our conditions in systems of rules. Nevertheless, this paper remains a paper about comparing models. As a matter of fact, relating join calculus and Petri nets may be beneficial for both. On one of the edges of the connection, in fact, the join calculus may provide Petri nets with a compositional framework, together with behavioural semantics such as testing [10] and bisimulation [3]. Also, it may suggest interesting, semantically well-founded extensions of Petri nets, such as reconfigurable and dynamic nets here, mobile nets in [1], and functional nets in [14]. On the other edge, it opens the join calculus to an entire body of results on the semantics of noninterleaving concurrency, such as those supported by monoidal categories (see, e.g., [4]).

Here we actually consider a generalisation of the join calculus in that join patterns may not be linear, a choice that we discuss at length in §2.

Related Work. Although focus and approach here differ radically from [1], our models are clearly related to those of Asperti and Busi. While we aim at representing the join calculus precisely, the *dynamic nets* of [1] are more loosely inspired by it and are more general than ours. In particular, they do not enforce privateness of input places – that is unique receptors – as our nets do. The analogy between Petri nets and join calculus has also provided the inspiration for Odersky's *functional nets* [14]. Although *loc. cit.* is quite different in spirit from the present paper, the relationships between Odersky's framework and our ideas here are worth of further investigation. This, together with the fine comparison with [1], we leave to a later paper.

Structure of the Paper. The paper is organised as follows. In §2 we recall the basic definitions of the join calculus, while §3 introduces Petri nets, Δ_0 and relates typeability to PT nets. The following sections repeat the same pattern for coloured, reconfigurable, dynamic nets and the respective type systems. Finally, §7 introduces Δ_4. Due to space limitations all proofs are omitted.

2 The Join Calculus

We shall focus on a monadic version of the join calculus [5,7], writing its operational semantics in terms of a reduction system as in [15]. For a thorough introduction the reader is referred to the literature. Let $\mathcal{N}m$ be an infinite set of names and let $x, y \ldots, u, v, \ldots$ range over $\mathcal{N}m$. Join terms, definitions, and patters are given by the following grammar.

$$\begin{array}{lll} \textsc{Terms} & P, Q & ::= 0 \mid x\langle y\rangle \mid P \mid Q \mid \texttt{def}\ D\ \texttt{in}\ P \\ \textsc{Definitions} & D, E & ::= J \triangleright P \mid D \wedge E \\ \textsc{Join Patterns} & J, K & ::= x\langle y\rangle \mid J \mid K \end{array}$$

Thus, a join term P is either the 'null' term, an emission $x\langle y\rangle$ of message y on channel x, a parallel composition of terms, or a local definition. A definition D is a set of elementary definitions $J \triangleright P$ matching join patterns J to terms P.

The sets of defined names dn, received names rn, and free names fn are defined below, and then extended to sets of terms in the obvious way.

$$\begin{array}{llll} rn(x\langle y\rangle) & = \{y\} & rn(J \mid K) & = rn(J) \cup rn(K) \\ dn(x\langle y\rangle) & = \{x\} & dn(J \mid K) & = dn(J) \cup dn(K) \\ dn(D \wedge E) & = dn(D) \cup dn(E) & dn(J \triangleright P) & = dn(J) \\ fn(x\langle y\rangle) & = \{x\} \cup \{y\} & fn(P \mid Q) & = fn(P) \cup fn(Q) \\ fn(D \wedge E) & = fn(D) \cup fn(E) & fn(J \triangleright P) & = dn(J) \cup (fn(P) \setminus rn(J)) \end{array}$$

$$fn(\texttt{def}\ D\ \texttt{in}\ P) = (fn(P) \cup fn(D)) \setminus dn(D)$$

A *renaming* σ is a map from names to names that is the identity except on a finite number of names. We indicate by $dom(\sigma)$ the set of names on which σ

is not the identity, and by $cod(\sigma)$ its image, using function application for the renaming of free names in terms.

Definition 1. The *structural congruence* $\equiv$ is the smallest substitutive (i.e., closed for contexts) equivalence relation on terms that satisfies the following rules, where θ and σ are *one-to-one* renamings such that $dom(\theta) \subseteq rn(D)$, $cod(\theta) \cap fn(D) = \varnothing$ and $dom(\sigma) \subseteq dn(D)$, $cod(\sigma) \cap fn(\theta D, P) = \varnothing$.

$$\texttt{def}\ D\ \texttt{in}\ P \equiv \texttt{def}\ \sigma\theta D\ \texttt{in}\ \sigma P \tag{1}$$

$$P \mid \texttt{def}\ D\ \texttt{in}\ Q \equiv \texttt{def}\ D\ \texttt{in}\ P \mid Q \qquad fn(P) \cap dn(D) = \varnothing \tag{2}$$

$$\texttt{def}\ D\ \texttt{in}\ \texttt{def}\ E\ \texttt{in}\ P \equiv \texttt{def}\ D \wedge E\ \texttt{in}\ P \qquad dn(E) \cap (dn(D) \cup fn(D)) = \varnothing \tag{3}$$

plus equations stating that 0 is the unit for $\mid$ and that the operators $\mid$ and $\wedge$ are associative and commutative.

Taking $dom(\sigma) = \varnothing$, rule (1) expresses the α-equivalence of definitions up to a renaming of their received names, while considering $dom(\theta) = \varnothing$ we obtain α-equivalence of terms up to renaming of defined names. Rule (2) formalises the scope extrusion of names, and rule (3) states that, under conditions that avoid name clashes, definitions can equivalently be gathered together by the $\wedge$ connective. Interestingly, rule (3) is problematic in the design of polymorphic type systems for the join calculus [8,15], essentially because it may introduce polymorphic (mutual) recursion. This, however, is not an issue in the present setting. On the contrary, the law is instrumental in establishing our results.

The operational semantics adopted here corresponds to the original one based on the chemical abstract machine [2]. Terms within a reduction context play the role of 'molecules', definitions determine the 'reactions'. The reduction context determines which join pattern is matched in a definition and binds its received names. The reduction rule replaces in the context the matched pattern by the right-hand side of its definition.

Definition 2 (cf. [15]). The *reduction* $\longrightarrow$ is the smallest substitutive relation on $\equiv$-classes that satisfies the following rule:

$$\texttt{def}\ D \wedge J \triangleright P\ \texttt{in}\ \mathcal{C}[\sigma J] \longrightarrow \texttt{def}\ D \wedge J \triangleright P\ \texttt{in}\ \mathcal{C}[\sigma P] \qquad \text{if } dom(\sigma) \subseteq rn(J),$$

where D may be absent (if $J \triangleright P$ is the only definition), $\mathcal{C}$ is a *reduction context*, i.e.,

$$\mathcal{C} ::= [\,] \mid \texttt{def}\ D\ \texttt{in}\ \mathcal{C} \mid P \mid \mathcal{C},$$

and $\mathcal{C}[P]$ denotes the reduction context $\mathcal{C}$ with its hole filled by P.

On Linearity. Here we no longer require join patterns to be linear, i.e., we allow names to occur several times in patterns. There are two aspects to this. The first one consists of renouncing the linearity of defined names in join patterns. It presents no further difficulties for implementations [11], and we see no reason not to consider it. In terms of Petri nets it corresponds to adding multiplicities to

arcs. More problematic in distributed implementations is relaxing the linearity of received names, because this amounts to equating values on different channels. Nevertheless we decided to adopt it, as it corresponds to a form of pattern matching essential in coloured Petri nets. It is worth remarking that our work is largely independent of this choice: we could as well keep linearity of received names at the price of disallowing matching in our coloured nets, and our results would stay, *mutatis mutandis*.

3 Place Transition Nets

A *multiset* on a set P is a function $\mu: P \to \mathbb{N}$. We shall use $\mu(P)$ to denote the set of finite multisets, i.e., markings, on P. The *sum* of $\mu_0, \mu_1 \in \mu(P)$ is the multiset $\mu = \mu_0 \oplus \mu_1$ such that $\forall p \in P.\ \mu(p) = \mu_0(p) + \mu_1(p)$.

Petri nets consists of places, transitions, and tokens. In view of relating nets and join terms, we identify places with names. In addition to $\mathcal{N}m$ – that will play the role of *public* places (*free* names) – we shall consider an infinite set ω of distinguished *private* places (*bound* names). In the following, $\mathcal{N}m_\omega$ stands for the union $\mathcal{N}m + \omega$.

Definition 3 (PT Nets). A *place/transition net* is a tuple $N = (T, \partial_0, \partial_1, \mu_0)$, where T is a finite set of transitions, $\partial_0, \partial_1 : T \to \mu(\mathcal{N}m_\omega)$ are the pre- and post-set functions, and $\mu_0 \in \mu(\mathcal{N}m_\omega)$ is the initial marking. We shall use Π_0 to refer to PT Petri nets.

Thus, to simplify notations, we include in every net all the names of $\mathcal{N}m_\omega$ with the understanding that *only* marked places, i.e., those carrying tokens, and places connected to some transitions are effectively to be considered in the net. In particular, the *empty net*, that we denote by $\varnothing$, formally consists of all the places in $\mathcal{N}m_\omega$, but no tokens and no transitions. Analogously, the net consisting of a distribution of tokens on a *multiset* of places μ, has all places but only $\mu(x)$ tokens in each $x \in \mathcal{N}m_\omega$. With abuse of notations, we shall use μ to denote both the multisets and the corresponding net. Also, for $\mu_0, \mu_1 \in \mu(\mathcal{N}m_\omega)$, $\mathbf{tran}\,\mu_0\,\mu_1$ stands for the net with a unique transition t with $\partial_0(t) = \mu_0$ and $\partial_1(t) = \mu_1$.

Following our intuition, isomorphisms preserve free names. This is detailed in the definition below, where we use *id* for identities, $+$ for function coproducts, and take the liberty of identifying functions with their extensions to multisets.

Definition 4. Nets $(T, \partial_0, \partial_1, \mu_0)$ and $(T', \partial'_0, \partial'_1, \mu'_0)$ are *isomorphic* if there exist isomorphisms $f_t : T \to T'$ and $f_p : \omega \to \omega$ such that $\partial'_i \circ f_t = (id_{\mathcal{N}m} + f_p) \circ \partial_i$, for $i = 0, 1$, and $(id_{\mathcal{N}m} + f_p)(\mu_0) = \mu'_0$.

Observe that nets that only differ for the use of names in ω are isomorphic, yielding a form of α-conversion. In the following, isomorphic nets will be regarded as equal. In particular, as we shall see, net isomorphism corresponds to structural congruence of join terms.

Definition 5. Let $N = (T, \partial_0, \partial_1, \mu_0)$ and $N' = (T', \partial'_0, \partial'_1, \mu'_0)$ be nets. Denoted by $\otimes$, *parallel composition* juxtaposes nets, merging public places without confusing the private ones. Formally,

$$N \otimes N' = (T + T', \partial_0 + \partial'_0, \partial_1 + \partial'_1, \mu_0 \oplus \mu'_0), \qquad \text{if } \omega(N) \cap \omega(N') = \varnothing,$$

where $\omega(N)$ denotes the names in ω that are used in N, i.e., that are either marked or connected to some transition. Observe that the side condition on private names can always be achieved up to isomorphism. Clearly, $\otimes$ is commutative and associative.

Denoted by ν_x, *restriction* 'hides' the place $x \in \mathcal{N}m$ replacing it by a *fresh* name $i \in \omega$, i.e., a name not occurring in $\omega(N)$. Formally, for $N = (T, \partial_0, \partial_1, \mu_0)$

$$\nu_x N = N[x \leftrightarrow i], \qquad i \text{ is fresh in } N$$

where $N[x \leftrightarrow i]$ is the net $(T, \partial'_0, \partial'_1, \mu'_0)$ whose μ'_0 and ∂'_j coincide respectively with μ_0 and ∂_j $(j = 0, 1)$, but for the values they yield on x and i, that are exchanged, i.e., for $\delta \in \{\partial_0, \partial_1, \mu_0\}$, we have that $\delta'(k)$ is equal to $\delta(i)$ if $k = x$, to $\delta(x)$ if $k = i$, and to $\delta(k)$ otherwise.

For $X = \{x_1, \ldots, x_n\} \subseteq \mathcal{N}m$, we use $\nu_X N$ to mean $\nu_{x_1}(\cdots \nu_{x_n} N)$. This is a correct definition because $\nu_x \nu_y N = \nu_y \nu_x N$ for each x, y, up to isomorphism.

The evolution of nets is described in terms of the 'firing' of its transitions. As usual, the firing of t consumes and produces resources, as prescribed by the pre- and post-set functions ∂_i. In the present setting, using $\otimes$ and ν_X to form net contexts, this can be expressed as the least *substitutive* relation $[\cdot\rangle$ such that

$$\mathbf{tran}\, \mu_0\, \mu_1 \otimes \mu_0 \; [\cdot\rangle \; \mathbf{tran}\, \mu_0\, \mu_1 \otimes \mu_1.$$

This should be read as saying that, in any context, a marking matching the preset of a transition can be replaced by the associated post-set.

Observe that we stop at the single transition semantics of nets, rather than considering also the usual step semantics. We do so in order to match the standard reduction semantics of the join calculus, and it is not a hard limitation. We could easily extend the results to the classical step semantics, provided we equip join terms with multiple concurrent reductions.

The Type System Δ_0

The purpose of Δ_0 is to single out those join terms that are PT Petri nets. In order to achieve this we need to prevent any form of name passing and mobility, imposing a static network structure to terms. In particular, since definitions represent transitions, it is fundamental that their right-hand sides consist of messages only. Also, corresponding to the fact that there is only one atom (*token*) of information delivered in PT nets, we enforce that only empty messages may be exchanged. For notational convenience, we assume in the following the existence of a distinguished name $\bullet \in \mathcal{N}m$ that represents the empty tuple.

The types of Δ_0 are $\diamond$ and $\diamond\diamond$, with τ ranging over them. There are three kinds of type judgement:

$$\vdash P : \diamond\diamond \quad (P \text{ is ok and no } \texttt{def}\ \_\ \texttt{in}\ \_ \text{ occurs in it})$$
$$\vdash P : \diamond \quad (P \text{ is ok}) \qquad \vdash D : \diamond \quad (D \text{ is ok}).$$

The typing rules are the following.

$$\text{(P-Zero)}\ \vdash 0 : \diamond\diamond \qquad \text{(P-Mess)}\ \vdash x\langle\bullet\rangle : \diamond\diamond \qquad \text{(P-Par)}\ \frac{\vdash P : \tau \quad \vdash Q : \tau}{\vdash P \mid Q : \tau}$$

$$\text{(P-Def)}\ \frac{\vdash D : \diamond \quad \vdash P : \diamond}{\vdash \texttt{def}\ D\ \texttt{in}\ P : \diamond} \qquad \text{(P-Sub)}\ \frac{\vdash P : \diamond\diamond}{\vdash P : \diamond}$$

$$\text{(D-Pat)}\ \frac{\vdash P : \diamond\diamond}{\vdash J \triangleright P : \diamond}\ rn(J)=\{\bullet\} \qquad \text{(D-And)}\ \frac{\vdash D : \diamond \quad \vdash E : \diamond}{\vdash D \wedge E : \diamond}$$

The rules are elementary. The key is (D-Pat) that allows *only* simple processes as right-hand sides of definitions, while (P-Mess) ensures that only emissions of $\bullet$ are well typed and, together with (P-Par), that only parallel composition of messages can be used inside definitions. We use $\Delta_0 \vdash P$ as a shorthand for typeability, i.e., $\vdash P : \diamond$, in the system Δ_0 above.

Proposition 1 (Subject Reduction). *If $\Delta_0 \vdash P$ and $P \longrightarrow Q$, then $\Delta_0 \vdash Q$.*

Translating Δ_0 Terms to PT Nets

We intend to define a translation from Δ_0-typed join terms to PT nets that can be extended to other type systems and other classes of nets. Therefore, we first give a general map on all join terms, and then prove that it restricts to a well defined encoding of terms typeable in Δ_0 into PT nets. By induction on the structure of terms, $[\![\_]\!]$ is defined as follows.

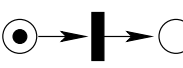
$$\begin{array}{lll} [\![0]\!] = \varnothing & & \\ [\![x\langle v\rangle]\!] = (x, v) & & [\![J \triangleright P]\!] = \mathbf{tran}[\![J]\!]\,[\![P]\!] \\ [\![P \mid Q]\!] = [\![P]\!] \otimes [\![Q]\!] & & [\![D \wedge E]\!] = [\![D]\!] \otimes [\![E]\!] \\ [\![\texttt{def}\ D\ \texttt{in}\ P]\!] = \nu_{dn(D)}([\![D]\!] \otimes [\![P]\!]) & & \end{array}$$

where $[\![J]\!]$ is J seen as a multiset, i.e., $[\![x\langle v\rangle]\!] = (x, v)$ and $[\![J \mid K]\!] = [\![J]\!] \oplus [\![K]\!]$.

Roughly speaking, we map names to places, messages to markings, and definitions to (groups of) transitions. Since we shall soon consider nets whose tokens carry information, we use (x, v) to denote a marking of place x with value v. Of course, in the current case v always equals $\bullet$, and $(x, \bullet)$ should be read as

an exotic way to mean x. In more details, $J \triangleright P$ is mapped into a transition whose pre-set is essentially J and whose post-set is the encoding of P, while $|$ and $\wedge$ are mapped homomorphically to $\otimes$. Similarly for $\texttt{def}\ D\ \texttt{in}\ P$, where the use of restriction ensures the correct treatment of local definitions. Here it is important to understand the different roles of $x\langle v\rangle$ because, depending on the context, $(x, \bullet)$ is treated as the net consisting of a single token in place x, or as an arc connecting place x to a transition. We shall get back to this in §4.

Example 1. Let P be the Δ_0-typeable term $\texttt{def}\ x\langle\bullet\rangle \triangleright y\langle\bullet\rangle\ \texttt{in}\ x\langle\bullet\rangle$. Then, using the standard representation of Petri nets enriched with labels that decorate *public* places with their name, the translation of P is

$$[\![P]\!] = \nu_x\big(\mathbf{tran}\{x\}\{y\} \otimes (x, \bullet)\big) =$$  $.$

Observe that, in force of the restriction ν_x, the name x does not appear.

As announced, the encoding is well defined on well-typed terms.

Proposition 2 (Correctness). *If $\Delta_0 \vdash P$, then $[\![P]\!]$ is a PT net.*

In addition to the previous proposition, we can prove a completeness theorem for Δ_0, expressed as follows.

Theorem 1 (Completeness). *The translation $[\![\_]\!]$ is an isomorphism between the set of terms typeable in Δ_0 and PT nets in Π_0.*

The rest of this section is devoted to illustrating that the $[\![\_]\!]$ is very tight a connection, preserving both structural congruence and dynamic behaviour.

Proposition 3. *Let P and Q be join terms typeable in Δ_0. Then,*

$$P \equiv Q \quad \textit{if and only if} \quad [\![P]\!] = [\![Q]\!].$$

Observe that there is a one-to-one correspondence between transitions in $[\![P]\!]$ and simple definitions $J \triangleright Q$ in P. Here and in the following we use $N\ [t\rangle\ N'$ to mean that N becomes N' by firing transitions t, and use $P \xrightarrow{-t\rightarrow} Q$ to signify that P reduces to Q by means of the simple definition from which the transition t of $[\![P]\!]$ was generated.

Proposition 4. *Let P be a join term typeable in Δ_0. Then,*

$$P \xrightarrow{-t\rightarrow} Q \quad \textit{if and only if} \quad [\![P]\!]\ [t\rangle\ [\![Q]\!]$$

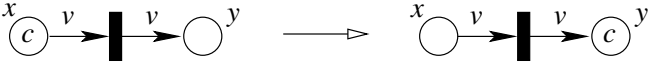
This implies that $P \longrightarrow^* Q$ if and only if there exist transitions $t_1, \ldots, t_n$ such that $[\![P]\!]\ [t_1\rangle \cdots [t_n\rangle\ [\![Q]\!]$. Also, P has a barb $x \in fn(P)$, i.e., $P \equiv \mathcal{C}[x\langle v\rangle]$ for $\mathcal{C}$ a context that does not bind x, if and only if x is marked in $[\![P]\!]$. Together with surjectivity and compositionality of $[\![\_]\!]$, this describes a very strong relationship, allowing to identify (the system represented by) P and (that represented by) $[\![P]\!]$. In other words, it allows us to identify typeability in Δ_0 with Π_0.

4 Coloured Nets

In coloured nets, tokens carry information: the 'colours'. Arcs between places and transitions are labelled by *arc expressions*, which evaluate to multisets of coloured tokens after binding free variables to colours. Expressions on input and output arcs describe respectively the resources needed for the firing and those generated by it. Although our treatment below attempts at providing an essential version of coloured Petri nets, we maintain one of their fundamental features. Namely, we permit a name to label several of the input arcs of a transition, with the intended meaning that firing is allowed only if the same colour can be fetched along all those arcs. This form of pattern matching is the reason why we relaxed the linearity requirement of join patterns. Technically, our model is obtained simply by allowing PT Nets to circulate tokens other than '$\bullet$'. The obvious choice in the present context is to identify colours with names.

Definition 6 (CNets). A *coloured net* is a tuple $(T, \partial_0, \partial_1, \mu_0)$, where T is a finite set of transitions, $\partial_0, \partial_1 : T \to \mu(\mathcal{N}m_\omega \times \mathcal{N}m)$, are the pre and post-set functions, and $\mu_0 \in \mu(\mathcal{N}m_\omega \times \mathcal{N}m)$ is the initial marking. We use Π_1 to denote the class of coloured Petri nets.

Example 2. In the above definition we interpret the pairs (p, c) of $\mathcal{N}m_\omega \times \mathcal{N}m$ as representing a place p that carries a colour (name) c. The coloured net $\mathbf{tran}\{(x, v)\}\{(y, v)\} \otimes (x, c)$ is represented graphically as below on the left.



As usual, the role of names is twofold: in markings they represent colours, e.g., the constant c in x; in transitions they represent bound variables. The net above fetches any token from x, binds it to v, and then delivers the actual value of v to y, as illustrated by the picture above and formalised by the definition below.

Definition 7. The set of *received colours* of a marking $\mu \in \mu(\mathcal{N}m_\omega \times \mathcal{N}m)$ is

$$\mathtt{rc}(\mu) = \{x \in \mathcal{N}m \mid \ x \neq \bullet,\ \exists p.\ (p, x) \in \mu\}.$$

A *binding* for μ is a function $\mathtt{b} : \mathcal{N}m \to \mathcal{N}m$ which is the identity on $\mathcal{N}m \setminus \mathtt{rc}(\mu)$. We shall let $\mu\langle \mathtt{b} \rangle$ denote the multiset $\{(p, \mathtt{b}(c)) \mid (p, c) \in \mu\}$.

The firing rule for coloured nets is the *substitutive* relation generated by the reduction rule below, where $\mathtt{b}$ is a binding for μ_0

$$\mathbf{tran}\, \mu_0\, \mu_1 \otimes \mu_0\langle \mathtt{b} \rangle\ [\cdot\rangle\ \mathbf{tran}\, \mu_0\, \mu_1 \otimes \mu_1\langle \mathtt{b} \rangle.$$

Isomorphisms of coloured nets are the obvious extension of those of PT nets given in Definition 4. They allow renaming of private places, but map public ones identically. It is important to remark that coloured nets are considered up to α-conversion, that is renaming of received colours. In particular, coloured nets are isomorphic if there is an isomorphism between them after possibly renaming received names.

The Type System Δ_1

The purpose of type system Δ_1 is to characterise coloured Petri nets among join terms. Having introduced names, we now face the issue of distinguishing among two kind of names: channels and parameters. In fact, coloured nets are a strict value-passing formalism. They are not allowed to use values received along channels as channels themselves, nor to send private names as messages. In order to enforce this, we consider typing environments holding assumptions on free names, *viz.*, if they are channels or messages. Type environments are therefore pairs of *disjoint* sets Γ (the channels) and ∇ (the messages). Type judgements are exactly as before, but for the presence of type environments.

(P-Zero) $\quad \Gamma;\nabla \vdash 0 : \diamond\diamond$

(P-Mess) $\quad \Gamma;\nabla \vdash x\langle y\rangle : \diamond\diamond \quad \binom{x \notin \nabla}{y \notin \Gamma}$

(P-Par)
$$\frac{\Gamma;\nabla \vdash P : \tau \quad \Gamma;\nabla \vdash Q : \tau}{\Gamma;\nabla \vdash P \mid Q : \tau}$$

(P-Def)
$$\frac{\Gamma, dn(D);\nabla \vdash D : \diamond \quad \Gamma, dn(D);\nabla \vdash P : \diamond}{\Gamma;\nabla \vdash \texttt{def}\ D\ \texttt{in}\ P : \diamond}$$

(P-Sub)
$$\frac{\Gamma;\nabla \vdash P : \diamond\diamond}{\Gamma;\nabla \vdash P : \diamond}$$

(D-Pat)
$$\frac{\Gamma;\nabla, rn(J) \vdash P : \diamond\diamond}{\Gamma;\nabla \vdash J \triangleright P : \diamond}$$

(D-And)
$$\frac{\Gamma;\nabla \vdash D : \diamond \quad \Gamma;\nabla \vdash E : \diamond}{\Gamma;\nabla \vdash D \wedge E : \diamond}$$

The structure of the rules matches exactly those of Δ_0, the only difference being the use of Γ in (P-Def) and of ∇ in (D-Pat), so to be able to control the use of names in (P-Mess).

Proposition 5. *$\Delta_0 \vdash P$ implies $\Delta_1 \vdash P$.*

Proposition 6 (Subject Reduction). *If $\Delta_1 \vdash P$ and $P \longrightarrow Q$, then $\Delta_1 \vdash Q$.*

A simple inspection of the rules shows that, as Δ_0, system Δ_1 does not allow processes of the form `def _ in _` to appear inside definitions. The other fundamental properties of Δ_1 are expressed in the proposition below.

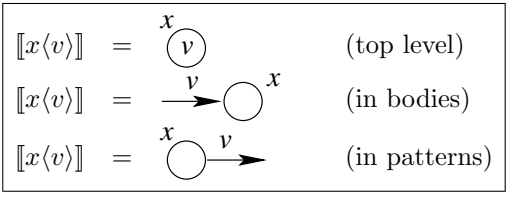
Proposition 7. *If $\Delta_1 \vdash P$, then P never emits messages on received names nor emits bound names out of the scope of their definitions.*

Translating Δ_1 Terms to CNets

It follows as a consequence of Proposition 7 that the translation $[\![\_]\!]$ extends to a well defined map from Δ_1 typeable terms to coloured Petri nets.

Proposition 8 (Correctness). *If $\Delta_1 \vdash P$, then $[\![P]\!]$ is a coloured net.*

Here more than before it is important to understand the three different roles played by $x\langle v\rangle$ in the translation. First, when $x\langle v\rangle$ appears outside all definitions, at top level, it is translated as the net consisting of a token v in the place x. Secondly, when it appears in a join pattern J, it is translated in $[\![J]\!]$ as (x, v), representing the input from place x on a bound variable v. Finally, when $x\langle v\rangle$ appears in the body of a definition, it is translated as (x, v) by the same clause above but, considered as a marking, it represents the output of a free or bound variables in the place x. This is summarised by the pictures below.



Example 3. Let P be the Δ_1 typeable term $\texttt{def}\ x\langle v\rangle \triangleright y\langle v\rangle\ \texttt{in}\ x\langle c\rangle$. Then $[\![P]\!]$ is the coloured net of Example 2.

Our results about Δ_1 match exactly those for Δ_0. They are listed below.

Theorem 2 (Δ_1 vs Π_1). *The translation $[\![\_]\!]$ is an isomorphism between the set of terms typeable in Δ_1 and coloured nets in Π_1. Moreover, if P and Q are typeable in Δ_1, then*

$$P \equiv Q \quad \text{if and only if} \quad [\![P]\!] = [\![Q]\!]$$
$$P \xrightarrow{-t\rightarrow} Q \quad \text{if and only if} \quad [\![P]\!]\ [t\rangle\ [\![Q]\!]$$

5 Reconfigurable Nets

Coloured Petri nets have a static structure, i.e., their firings only affects the marking. While this accounts faithfully for value-passing theories, it is completely inadequate to represent dynamically changing structures. Reconfigurable nets generalise coloured ones by adding precisely one ingredient: each firing of a transition may deliver to a different set of output places. This equips nets with a mechanism to model networks with reconfigurable topologies, that is networks in which the set of components is fixed, but the connectivity among them may change in time. Formally, this is achieved by a very smooth alteration of the definitions of coloured nets – one that allows markings and messages to contain private names – and of their firing rule.

Definition 8 (RNets). A *reconfigurable net* is a tuple $(T, \partial_0, \partial_1, \mu_0)$, where T is a finite set of transitions, $\partial_0 \colon T \to \mu(\mathcal{N}m_\omega \times \mathcal{N}m)$ and $\partial_1 \colon T \to \mu(\mathcal{N}m_\omega \times \mathcal{N}m_\omega)$, are, respectively, the pre and post-set functions, and $\mu_0 \in \mu(\mathcal{N}m_\omega \times \mathcal{N}m_\omega)$ is the initial marking. We use Π_2 to denote the set of reconfigurable Petri nets.

Restriction ν_x and parallel composition $\otimes$ extends straightforwardly to reconfigurable nets. Also, isomorphisms of reconfigurable Petri nets are obtained extending those of coloured nets in the obvious way.

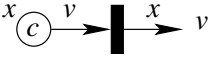
The firing rule is generalised by extending the scope of the binding of received names to include the output places in addition to the names on the output arcs.

Definition 9. Let $\mathtt{b}$ be a binding. For $\mu \in \mu(\mathcal{N}m_\omega \times \mathcal{N}m_\omega)$, let $X\langle\langle \mathtt{b} \rangle\rangle$ denote the multiset $\{(\mathtt{b}_\omega(p), \mathtt{b}_\omega(c)) \mid (p, c) \in \mu\}$, where $\mathtt{b}_\omega = \mathtt{b} + id_\omega$.

For $\mathtt{b}$ a binding for μ_0, the firing is generated closing by net contexts the rule

$$\mathbf{tran}\,\mu_0\,\mu_1 \otimes \mu_0\langle \mathtt{b} \rangle \; [\cdot\rangle \; \mathbf{tran}\,\mu_0\,\mu_1 \otimes \mu_1\langle\langle \mathtt{b} \rangle\rangle.$$

Example 4. The simple reconfigurable net represented below binds v to any name found in x – in this case c – and delivers the name x to the place to which v is bound.



The Type System Δ_2

In order to design a type system Δ_2 corresponding to Π_2 we only need to remove the difference between channels and messages upon which Δ_1 is based, which can be done easily by removing Γ and ∇. This yields a system identical to Δ_0, apart from (P-Mess) that allows any message to be transmitted on any name.

(P-Zero) $\vdash 0 : \diamond\diamond$ (P-Mess) $\vdash x\langle y \rangle : \diamond\diamond$ (P-Par) $\dfrac{\vdash P : \tau \quad \vdash Q : \tau}{\vdash P \mid Q : \tau}$

(P-Def) $\dfrac{\vdash D : \diamond \quad \vdash P : \diamond}{\vdash \mathtt{def}\ D\ \mathtt{in}\ P : \diamond}$ (P-Sub) $\dfrac{\vdash P : \diamond\diamond}{\vdash P : \diamond}$

(D-Pat) $\dfrac{\vdash P : \diamond\diamond}{\vdash J \triangleright P : \diamond}$ (D-And) $\dfrac{\vdash D : \diamond \quad \vdash E : \diamond}{\vdash D \wedge E : \diamond}$

Proposition 9 (Subject Reduction). *If $\Delta_2 \vdash P$ and $P \longrightarrow Q$, then $\Delta_2 \vdash Q$.*

Proposition 10. *$\Delta_1 \vdash P$ implies $\Delta_2 \vdash P$.*

Translating Δ_2 Terms to RNets

The property guaranteed by Δ_2 is that there is no process of the kind `def _ in _` inside definitions, i.e., the topology of the net may change by redirecting output edges, but the components of the net are fixed once and for all. In force of this, we can extend to Δ_2 the correspondence between well-typed terms and nets.

Theorem 3 (Δ_2 vs Π_2). *The translation $[\![\_]\!]$ is an isomorphism between the set of terms typeable in Δ_2 and reconfigurable nets in Π_2. Moreover, if P and Q are typeable in Δ_2, then*

$$P \equiv Q \quad \text{if and only if} \quad [\![P]\!] = [\![Q]\!]$$
$$P \xrightarrow{-t\rightarrow} Q \quad \text{if and only if} \quad [\![P]\!] \; [t\rangle \; [\![Q]\!]$$

6 Dynamic Nets

The obvious generalisation of reconfigurable nets is to allow the dynamic creation of components, that we achieve by means of the notions of *dynamic Petri nets*. The idea behind such structures is that the firing of a transition allocates a new net parametric in the actual values of the received names. As the net may consists simply of a marking and no transitions, this includes the standard definition of PT nets. Also coloured and reconfigurable nets are, of course, special kinds of dynamic nets. The characteristic feature of dynamic nets, that to our best knowledge distinguishes them by other approaches in the literature and draws a connection to [13], is that, as for reconfigurable nets, input arcs are never modified. While it is possible to modify dynamically the post-set of a transition, and also to spawn new subnets, it is not possible to add places to the presets of transitions. This allows us to formalise our intuition by simply generalising the post-set functions of nets, allowing $\partial_1(t)$ to be a dynamic net. Of course, this means that the definition of nets becomes recursive.

Definition 10 (DNets). Let *DNets* be the least set satisfying the equation

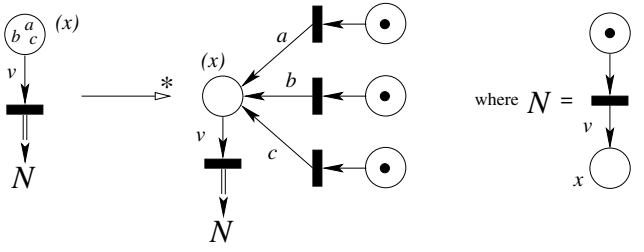
$$DNets \cong \big\{(T, \partial_0 : T \to \mu(\mathcal{N}m_\omega \times \mathcal{N}m), \partial_1 : T \to DNets, \mu_0)\big\},$$

where T is a finite set and μ_0, the initial marking, is in $\mu(\mathcal{N}m_\omega \times \mathcal{N}m_\omega)$. A *dynamic* net DN is an element of *DNets*. We use Π_3 to refer to the set of dynamic nets.

Once again, the operations of parallel composition and restriction, and the notions of α-conversion and isomorphism lift smoothly to this setting.

Definition 11. Dynamic nets $(T, \partial_0, \partial_1, \mu_0)$ and $(T', \partial'_0, \partial'_1, \mu'_0)$ are isomorphic if, up to α-conversion, there exists a pair $(f_t : T \to T', f_p : \omega \to \omega)$ of isomorphisms such that $(id_{\mathcal{N}m} + f_p \times id_{\mathcal{N}m} + f_p)(\mu_0) = \mu'_0$, $\partial'_0 \circ f_t = (id_{\mathcal{N}m} + f_p \times id_{\mathcal{N}m}) \circ \partial_0$, and, recursively, $\partial_1(t)$ and $\partial'_1(f_t(t))$ are isomorphic for each $t \in T$.

Example 5. For $P = \texttt{def}\ x\langle v\rangle \triangleright (\texttt{def}\ y\langle\bullet\rangle \triangleright x\langle v\rangle\ \texttt{in}\ y\langle\bullet\rangle)\ \texttt{in}\ x\langle a\rangle \mid x\langle b\rangle \mid x\langle c\rangle$, the structure $[\![P]\!]$ is the dynamic net illustrated below together with one of its possible firing sequences. The transition may fire three times, binding v in turn to a, b, and c, and spawning three new nets instantiating v appropriately in N. The free name x of N maintains its meaning across instantiations. This rules are formalised below, and amounts to a recursive extension of binding.



Definition 12. Let $\texttt{b}$ be a binding. For $X = (T, \partial_0, \partial_1, \mu_0) \in DNets$, let $X\langle\!\langle\!\langle \texttt{b} \rangle\!\rangle\!\rangle$ denote the dynamic net obtained from X by substituting free names according to $\texttt{b}$ in μ_0, $\partial_0(t)$ and, recursively, applying $\langle\!\langle\!\langle \texttt{b} \rangle\!\rangle\!\rangle$ to $\partial_1(t)$, for all $t \in T$.

The firing for dynamic nets is generated as follows, for $\texttt{b}$ a binding for μ_0.

$$\mathbf{tran}\,\mu_0\,X \otimes \mu_0\langle \texttt{b}\rangle\ [\cdot\rangle\ \mathbf{tran}\,\mu_0\,X \otimes X\langle\!\langle\!\langle \texttt{b} \rangle\!\rangle\!\rangle$$

The Type System Δ_3

In order to capture dynamic nets in terms of the join calculus, we simply have to relax the restriction that disallows definitions inside definitions. At the level of type system, this can be achieved by removing the distinction between types $\diamond$ and $\diamond\diamond$ from Δ_2. As an obvious consequence, any join term results typeable.

Proposition 11. *For each join term P, $\Delta_3 \vdash P$.*

Thus, extending our results for $[\![\_]\!]$ to Δ_3 amounts to proving that dynamic nets coincide with join terms, which is the content of the following.

Theorem 4 (Δ_3 vs Π_3). *The translation $[\![\_]\!]$ is an isomorphism between join terms (terms typeable in Δ_3) and dynamic nets in Π_3. Moreover, if P and Q are typeable in Δ_3, then*

$$P \equiv Q \quad \textit{if and only if} \quad [\![P]\!] = [\![Q]\!]$$
$$P \xrightarrow{t} Q \quad \textit{if and only if} \quad [\![P]\!]\ [t\rangle\ [\![Q]\!]$$

7 The Type System Δ_4

The information contained in Δ_i, $i = 0, \ldots, 3$ can be summarised in a single type system Δ_4 that tags terms with indices that characterise them as nets of Π_i.

Types range over $i^{\diamond\diamond} \in \{0^{\diamond\diamond}, 1^{\diamond\diamond}, 2^{\diamond\diamond}, 3^{\diamond\diamond}\}$ and $i^{\diamond} \in \{0^{\diamond}, 1^{\diamond}, 2^{\diamond}, 3^{\diamond}\}$. For $\tau = i^{?}$ a type, we write $\tau{\downarrow}$ for i and $\tau{\uparrow}$ for ?. By $\tau \leq \tau'$ we mean that $\tau{\downarrow} \leq \tau'{\downarrow}$ and $\tau{\uparrow} \leq \tau'{\uparrow}$, with the convention that $\diamond\diamond \leq \diamond$. Type environments are pairs $\Gamma; \nabla$ as for Δ_1, and type judgements are as follows:

$$\Gamma; \nabla \vdash P : i^{\diamond\diamond} \quad (P \text{ well-typed, in } \Pi_i, \text{ and containing no } \texttt{def}\ \_\ \texttt{in}\ \_)$$
$$\Gamma; \nabla \vdash P : i^{\diamond} \quad (P \text{ well-typed and in } \Pi_i)$$
$$\Gamma; \nabla \vdash D : i^{\diamond} \quad (D \text{ well-typed and containing terms in } \Pi_i)$$

The typing rules are the following.

(P-Mess$_0$) $\quad \Gamma; \nabla \vdash x\langle\bullet\rangle : 0^{\diamond\diamond}$ $(x \notin \nabla)$

(P-Mess$_1$) $\quad \Gamma; \nabla \vdash x\langle y\rangle : 1^{\diamond\diamond}$ $\binom{x \notin \nabla}{y \notin \Gamma}$

(P-Mess$_2$) $\quad \Gamma; \nabla \vdash x\langle y\rangle : 2^{\diamond\diamond}$

(P-Def) $\quad \dfrac{\Gamma, dn(D); \nabla \vdash D : i^{\diamond} \quad \Gamma, dn(D); \nabla \vdash P : i^{\diamond}}{\Gamma; \nabla \vdash \texttt{def}\ D\ \texttt{in}\ P : i^{\diamond}}$

(P-Zero) $\quad \Gamma; \nabla \vdash 0 : 0^{\diamond\diamond}$

(P-Par) $\quad \dfrac{\Gamma; \nabla \vdash P : \tau \quad \Gamma; \nabla \vdash Q : \tau}{\Gamma; \nabla \vdash P \mid Q : \tau}$

(P-Sub) $\quad \dfrac{\Gamma; \nabla \vdash P : \tau \quad \tau \leq \tau'}{\Gamma; \nabla \vdash P : \tau'}$

(D-Patt$_0$) $\quad \dfrac{\Gamma; \nabla \vdash P : 0^{\diamond\diamond}}{\Gamma; \nabla \vdash J \triangleright P : 0^{\diamond}}$ $(rn(J) = \{\bullet\})$

(D-Patt$_1$) $\quad \dfrac{\Gamma; \nabla, rn(J) \vdash P : i^{\diamond\diamond}}{\Gamma; \nabla \vdash J \triangleright P : i^{\diamond}}$

(D-Patt$_2$) $\quad \dfrac{\Gamma; \nabla, rn(J) \vdash P : i^{\diamond}}{\Gamma; \nabla \vdash J \triangleright P : 3^{\diamond}}$

(D-And) $\quad \dfrac{\Gamma; \nabla \vdash D : i^{\diamond} \quad \Gamma; \nabla \vdash E : i^{\diamond}}{\Gamma; \nabla \vdash D \wedge E : i^{\diamond}}$

(D-Sub) $\quad \dfrac{\Gamma; \nabla \vdash D : i^{\diamond} \quad i < j}{\Gamma; \nabla \vdash D : j^{\diamond}}$

Here (P-Mess) is split into three rules, each behaving as in the corresponding Δ_i; (P-Mess$_2$) ignores type environments, so achieving the effect of Δ_2. Definitions can be typed by i if the terms they contain are typeable by i; (D-Patt$_2$) constrains to 3 the type of definitions containing a $\texttt{def}\ \_\ \texttt{in}\ \_$ term, so forcing to 3 the type of enclosing processes by means of rule (P-Def). For Δ_4 we have the following results.

Proposition 12 (Subject Reduction). *If $\Delta_4 \vdash P$ and $P \longrightarrow Q$, then $\Delta_4 \vdash Q$.*

Theorem 5. *$\Delta_4 \vdash P : i^{\diamond}$ if and only if $P \in \Pi_i$, for $i = 0, \ldots, 3$.*

Proof. It follows by proving that $\Delta_4 \vdash P : i^{\diamond}$ if and only if $\Delta_i \vdash P$.

Conclusions and Future Work

We have provided a full correspondence between join calculus and a hierarchy of Petri net classes. It would be interesting to study relevance and adaptability to coloured nets of the existing polymorphic type systems [8,15] and extensional semantic equivalence [10,3,6] for join terms. On the other hand, the body of work on the semantics of nets suggests a noninterleaving semantics based on monoidal categories for the join-calculus.

Our version of coloured nets simplifies considerably those in the literature. It is worth investigating whether it can be interesting for the coloured Petri net community by putting it at work on suitable applications.

Also, we plan to compare our nets to Milner's named nets [12] – clearly closely related to the nets of §3 – to Asperti and Busi's dynamic nets [1], and to Odersky's functional nets [14].

Acknowledgements. We heartily thank Cedric Fournet, Alan Schmitt, and Emilio Tuosto for their useful comments. Special thanks to Roberto Bruni, who made substantial suggestions, and to an anonymous referee, who spotted some technical flaws.

References

1. A. Asperti and N. Busi (1996), *Mobile Petri Nets*, Technical Report UBLCS 96-10, Università di Bologna.
2. G. Berry and G. Boudol (1992), The Chemical Abstract Machine, *Theoretical Computer Science*, 96:217–248.
3. M. Boreale, C. Fournet, and C. Laneve (1998), Bisimulations for the Join-Calculus, In *Proc. PROCOMET'98*, D. Gries and W.P. de Roever (Eds.), 68–86, IFIP, Chapman & Halls.
4. R. Bruni, J. Meseguer, U. Montanari, and V. Sassone (2000), Functorial Models for Petri Nets, *Information and Computation*. To appear.
5. C. Fournet and G. Gonthier (1996), The Reflexive Chemical Abstract Machine and the Join-Calculus, In *Proc. POPL'96*, 372–385, ACM.
6. C. Fournet and G. Gonthier (1996), A Hierarchy of Equivalences for Asynchronous Calculi, In *Proc. ICALP'98*, Lecture Notes in Computer Science 1443, 844–855, Springer.
7. C. Fournet, G. Gonthier, J. Lévy, L. Maranget, and D. Rémy (1996), A Calculus of Mobile Agents, In *Proc. CONCUR'96*, Lecture Notes in Computer Science 1119, 406–421, Springer.
8. C. Fournet, C. Laneve, L. Maranget, and D. Rémy (1997), Implicit Typing à la ML for the Join-Calculus, In *Proc. CONCUR'97*, Lecture Notes in Computer Science 1243, 196–212, Springer.
9. K. Jensen (1992), *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Monographs on Theoretical Computer Science, Springer.
10. C. Laneve (1996), May and Must Testing in the Join-Calculus, Technical Report UBLCS 96-04, Università di Bologna.
11. F. Le Fessant and L. Maranget (1998), Compiling Join-Patterns, In *Proc. HLCL'98*, Electronic Notes in Computer Science 16(3), Elsevier.

12. R. Milner (1993), Action Calculi, or Syntactic Action Structures, In *Proc. MFCS'93*, Lecture Notes in Computer Science 711, 105-121, Springer.
13. M. Nielsen, L. Priese, and V. Sassone (1995), Characterizing Behavioural Congruences for Petri Nets, in *Proc. CONCUR 95*, Lecture Notes in Computer Science 962, 175–189, Springer.
14. M. Odersky (2000), Functional Nets, In *Proc. ESOP'2000*, Lecture Notes in Computer Science 1782, 1–25, Springer.
15. M. Odersky, C. Zenger, M. Zenger, and G. Chen (1999), A Functional View of Join, Technical Report ACRC-99-016, University of South Australia.
16. C.A. Petri (1962), *Kommunikation mit Automaten*. Ph.D. thesis, Institut für Instrumentelle Mathematik, Bonn.
17. W. Reisig (1985), *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science, Springer.
18. V. Sassone (2000), On the Algebraic Structure of Petri Nets, *Bulletin of EATCS* 72, 133–148.

Temporary Data in Shared Dataspace Coordination Languages*

Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura Anteo Zamboni 7, I-40127 Bologna, Italy.
busi,gorrieri,zavattar@cs.unibo.it

Abstract. The shared dataspace metaphor is historically the most prominent representative of the family of coordination models. According to this approach, concurrent processes interact via the production, consumption, and test for presence/absence of data in a common repository. Recently, the problem of the accumulation of outdated and unwanted information in the shared repository has been addressed and it has been solved by introducing non-permanent data, obtained by associating an expiration time to data. In this paper we investigate the impact of the adoption of different notions of non-permanent data on the expressiveness of a Linda based process calculus.

1 Introduction

The rapid evolution of computers and networks is calling for the development of middleware platforms responsible for the management of dynamically reconfigurable federations of devices, where processes cooperate and compete for the use of shared resources. In this scenario one of the most challenging topics is concerned with the coordination of the activities performed by the federated components.

The shared dataspace model has recently received a renewed interest; for example, Sun and IBM have respectively proposed JavaSpaces [8] and TSpaces [9] as middlewares for distributed Java programming. Both these products exploit generative communication: processes communicate through production, consumption and test for presence or absence of data in a shared dataspace; after its insertion in the dataspace, a datum has an independent existence, until it is explicitly withdrawn by a consumer.

The adoption of generative communication to manage the interaction of distributed components causes the following features to come into play: data are shared by a large, unpredictable number of processes, and the entity producing an information does not necessarily coincide with its user. The conjunction of these aspects leads to the unwanted effect of accumulations of outdated information. Such useless information can grow indefinitely, hence compromising

* Work partially supported by Italian Ministry of University - MURST 40% - Progetto TOSCA.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 121–136, 2001.

the performance of the whole system. A first solution could consists in leaving to each process the responsibility to take care of the data it produces, i.e., to explicitly free the resources when they are no longer needed. A first drawback of this approach is that a failure of the producer may cause the resources used by the data to be never freed. Moreover, this approach clashes with the basic principle of generative communication, i.e., the independence of a datum from any process, once it has been produced. In this setting, it is not unusual that the lifetime of the datum is longer than the lifetime of its producer, thus making a management of the datum by its producer impossible.

A commonly adopted solution to this problem (see, e.g., the leasing mechanism of JavaSpaces) is based on a notion of temporary data: rather than maintaining a datum until it has been explicitly consumed, the lifetime of the datum is decided by the producer. After this time has been expired, the existence of the datum is no longer granted.

The removal of outdated information can be carried out by a sort of expired-data collector, which is invoked when the workload of the system is low or when storage space needs to be freed for incoming data. We consider two possible implementations of the collector. The first one, called *unordered collection*, removes one of the expired data, whereas the second one, called *ordered collection*, removes one of the data which expired first.

The introduction of an upper bound to the guaranteed lifetime of datum, as well as different implementation policies, have an impact on the expressive power of the coordination primitives of Linda-like languages, which is aim of this paper to investigate.

We start by introducing a traditional Linda-like process calculus (a slight variation of the calculus in [1]) with permanent data, i.e., where data remain available in the dataspace until they are not explicitly consumed by some process.

Then, we move to temporary data by enriching each datum with a lifetime information. When an output operation is performed, the process specifies the time the datum is required to remain available in the dataspace. After the specified time has expired, the datum can be removed from the dataspace. Regarding the technique of removal of expired data, we consider both unordered and ordered collection.

We compare the three approaches described above from an expressiveness viewpoint. More precisely, we investigate decidability of properties, such as the existence of a terminating or a divergent computation, and the ability to provide an encoding of a Random Access Machine [7] that preserves some of these properties.

Regarding permanent data, we recall a deterministic encoding of RAMs, proposed in [1], which preserves the existence of terminating and divergent computations.

The removal policy influences the expressive power of the calculus with temporary data. The unordered collection policy gives rise to the weaker model. We show that it is possible to decide the existence of a divergent computation, by reduction to the same (decidable) property on place/transition nets with reset

arcs [3]. On the contrary, the existence of terminating computations remains undecidable, because it is still possible to provide a (nondeterministic) encoding of RAMs which preserves the existence of terminating computations.

The ordered collection policy increases the expressive power of temporary data. In fact, we provide a nondeterministic encoding of RAMs that preserves the existence of divergent computations. Moreover, the termination-preserving encoding presented for unordered collection works also in this case.

As both termination and divergence are undecidable for temporary data with ordered collection, one might wonder if the calculus with ordered collection is equivalent to the one with permanent data. We prove that the former is weaker than the latter by showing the impossibility of providing a deterministic encoding of any RAM which preserves divergence (and termination) in the calculus with ordered collection. This result is achieved by reducing the divergence problem for a superclass of deterministic processes (namely, processes whose computations are either all finite or all infinite) to the divergence problem for place/transition nets with reset arcs.

The paper is organized as follows. In Section 2 we define the calculus with permanent data and we show a deterministic encoding of RAMs, preserving both termination and divergence. Section 3 is devoted to the presentation of the calculus with temporary data and of the two aforementioned collection policies. In Sections 4 and 5 we investigate the expressiveness of temporary data with unordered and ordered collection, respectively. Section 5 reports on related work.

2 Permanent Data

In this section we introduce the calculus with permanent data, and we show that it is Turing equivalent.

Let *Name* be a denumerable set of data ranged over by a, b, ..., and *Const* be a set of program constants ranged over by K, K', ...

Let *Prog*, ranged over by P, Q, ..., be the set of the programs defined by the following grammar:

$$P ::= \mathbf{0} \mid in(a).P \mid out(a).P \mid inp(a)?P\_P \mid P|P \mid K$$

The program $\mathbf{0}$ is the empty process. The program $in(a).P$ requires the consumption of an instance of datum a from the dataspace; after the consumption the continuation P is activated. The program $out(a).P$ produces a new instance of datum a and then behaves like P. The program $inp(a)?P\_Q$ represents a non-blocking version of input; an instance of datum a is required to be consumed, if it is present the datum is removed and the first continuation P is activated, otherwise the second continuation Q is activated.

Programs can be composed in parallel by using the operator |. We adopt program constants in order to permit recursive program definition: we assume that each constant is equipped with a definition $K = P$ and, as usual, we admit guarded recursion only [5].

The state of the dataspace is modeled by a multiset of data. An instance of datum a is denoted by $\langle a \rangle$. Formally, we define *DataSpace*, ranged over by DS, DS', ..., as $DataSpace = \mathcal{M}(\{\langle a \rangle \mid a \in Name\})$, where $\mathcal{M}(S)$ denotes the set of all the multisets on S. In the following we use $\oplus$ to denote multiset union, and we usually omit the brackets in the denotation of singletons (e.g., we use $\langle a \rangle$ instead of $\{\langle a \rangle\}$).

Let *Conf*, ranged over by C, C', ..., be the set of the configurations; a configuration is a pair composed of the active processes and the dataspace, i.e., $Conf = \{[P, DS] \mid P \in Prog, DS \in DataSpace\}$.

Table 1. The operational semantics for permanent data (symmetric rule of (5) omitted).

(1)	$[in(a).P, DS \oplus \langle a \rangle] \longrightarrow [P, DS]$	
(2)	$[out(a).P, DS] \longrightarrow [P, DS \oplus \langle a \rangle]$	
(3)	$[inp(a)?P\_Q, DS \oplus \langle a \rangle] \longrightarrow [P, DS]$	
(4)	$[inp(a)?P\_Q, DS] \longrightarrow [Q, DS]$	if $\langle a \rangle \not\in DS$
(5)	$\dfrac{[P, DS] \longrightarrow [P', DS']}{[P \mid Q, DS] \longrightarrow [P' \mid Q, DS']}$	
(6)	$\dfrac{[P, DS] \longrightarrow [P', DS']}{[K, DS] \longrightarrow [P', DS']}$	if $K = P$

The semantics is described by a transition system $(Conf, \longrightarrow)$ defined as the minimal relation satisfying the axioms and rules in Table 1. Axiom (1) and (2) describe the execution of $in(a)$ and $out(a)$: in the first case one datum $\langle a \rangle$ is removed from the dataspace, in the second one it is added. The $inp(a)$ operation is described by the axioms (3) and (4); if the required datum is available it is consumed and the first continuation is activated (axiom (3)), otherwise the second continuation is chosen and the space is left unchanged (axiom (4)). Rules (5) and (6) describe the behaviour of local computation and of program constants, respectively (the symmetric rule of (5) is omitted).

A configuration C is *deterministic* if for each D such that $C \longrightarrow^* D$, the reached configuration D has at most one outgoing transition, i.e., for all D', D'', if $D \longrightarrow D'$ and $D \longrightarrow D''$ then $D' = D''$. A configuration C is *terminated* (denoted by $C \not\longrightarrow$) if it has no outgoing transition, i.e., if and only if there exists no C' such that $C \longrightarrow C'$. A configuration C has a terminating computation (denoted by $C\downarrow$) if C can block after a finite amount of computation steps, i.e., there exists C' such that $C \longrightarrow^* C$ and $C' \not\longrightarrow$. A configuration C has an infinite computation (denoted by $C\uparrow$) if there exists an infinite computation starting from C, i.e., there exist an infinite sequence $C_0, C_1, C_2, \ldots$ such that $C = C_0$ and, for each $i \geq 0$, $C_i \longrightarrow C_{i+1}$. Observe that, because of nondeterminism,

the two above conditions are not in general mutually exclusive, i.e., given a program C both $C\downarrow$ and $C\uparrow$ may hold. A configuration C is *uniform w.r.t. termination* (uniform for short) if it satisfies the following: if C has a terminating computation, then all its computations are finite. Formally, C is uniform if and only if $C\downarrow$ implies $C\not\uparrow$

We define a *structural congruence* for programs (denoted by $\equiv$) as the minimal congruence relation satisfying the usual laws for the parallel composition operator: $P \equiv P|\mathbf{0}, P|Q \equiv Q|P, P|(Q|R) \equiv (P|Q)|R$. Two structurally congruent programs are observationally indistinguishable (they act on the dataspace exactly in the same way); for this reason, in the remainder of the paper, we will make no distinction between $[P, DS]$ and $[P', DS]$ in the case $P \equiv P'$.

Random Access Machines

A Random Access Machine [7], simply RAM in the following, is a computational model composed of a finite set of registers $r_1 \ldots r_n$, that can hold arbitrary large natural numbers, and a program $I_1 \ldots I_k$, that is a sequence of simple numbered instructions.

The execution of the program begins with the first instruction and continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached.

In [6] it is shown that the following two instructions are sufficient to model every recursive function:

- $Succ(r_j)$: adds 1 to the content of register r_j;
- $DecJump(r_j, s)$: if the content of register r_j is not zero, then decreases it by 1 and go to the next instruction, otherwise jumps to instruction s.

We restrict to RAMs which start and finish their computation (in the case they terminate) with all the registers empty; it is well known result that this restriction does not decrease the expressiveness of the formalism.

The (computation) state is represented by $(i, c_1, c_2, \ldots, c_n)$, where i indicates the next instruction to execute and c_l is the content of the register r_l for each $l \in \{1, \ldots, n\}$. Let R be a program $I_1 \ldots I_k$, and $(i, c_1, c_2, \ldots, c_n)$ be the corresponding state; we use the notation $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (i', c'_1, c'_2, \ldots, c'_n)$ to state that after the execution of the instruction I_i with contents of the registers $c_1, \ldots, c_n$, the program counter points to the instruction $I_{i'}$, and the registers contain $c'_1, \ldots, c'_n$. Moreover, we use $(i, c_1, c_2, \ldots, c_n) \not\longrightarrow_R$ to indicate that $(i, c_1, c_2, \ldots, c_n)$ is a terminal state, i.e., $i > k$.

Observe that the computation proceeds deterministically; that is, given a state reached during the computation, the subsequent state, if it exists, is unique. Thus, given a program R and its initial state $(1, 0, 0, \ldots, 0)$ the computation either terminates (denoted by $R\downarrow$) or proceeds indefinitely (denoted by $R\uparrow$).

We compare the expressiveness of our calculi by studying the ability to provide encodings that satisfy some basic properties, outlined below. As RAMs

are deterministic, the best encoding one may provide consists of a deterministic configuration, which terminates if and only if the RAM terminates. There are however weaker forms of encodings that, though adding some nondeterministic computation to the original behaviour, still permit to observe relevant properties of the encoded RAM.

Let $[\![(i, c_1, c_2, \ldots, c_n)]\!]_R$ be the encoding of the RAM with program R and corresponding state $(i, c_1, c_2, \ldots, c_n)$. We say that the encoding *preserves termination* if, for any RAM program R, $R\downarrow$ iff $([\![(1, 0, 0, \ldots, 0)]\!]_R)\downarrow$. The encoding *preserves divergence* if, for any RAM program R, $R\uparrow$ iff $([\![(1, 0, 0, \ldots, 0)]\!]_R)\uparrow$.

A Deterministic Encoding of RAMs

In this section we recall an encoding of RAMs [1] in the deterministic fragment of the calculus with permanent data.

Consider the state $(i, c_1, c_2, \ldots, c_n)$ with corresponding RAM program R. We represent the content of each register r_l by putting c_l occurrences of $\langle r_l \rangle$ in the dataspace. Suppose that the program R is composed of the sequence of instructions $I_1 \ldots I_k$; we consider k programs $P_1 \ldots P_k$, one for each instruction. The program P_i behaves as follows: if I_i is a *Succ* instruction on register r_j, it simply emits an instance of datum $\langle r_j \rangle$ and then activates the program P_{i+1}; if it is an instruction $DecJump(r_j, s)$, it performs an $inp(r_j)$ operation, if this operation succeeds (i.e., an instance of $\langle r_j \rangle$ has been withdrawn) then the subsequent program is P_{i+1}; otherwise it is P_s. According to this approach we consider the following definitions for each $i \in \{1, \ldots, k\}$:

$$\begin{array}{ll} P_i = out(r_j).P_{i+1} & \text{if } I_i = Succ(r_j) \\ P_i = inp(r_j)?P_{i+1}\_P_s & \text{if } I_i = DecJump(r_j, s) \end{array}$$

We also consider a definition $P_i = \mathbf{0}$ for each $i \notin \{1, \ldots, k\}$ which appears in one of the previous definitions. This is necessary in order to model the termination of the computation occurring when the next instruction to execute has an index outside the range $1, \ldots, k$.

The encoding is then defined as follows:

$$[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \quad = \quad [P_i, \bigoplus_{1 \le l \le n} \{\underbrace{\langle r_l \rangle, \ldots, \langle r_l \rangle}_{c_l \text{ times}}\}]$$

The correctness of the encoding is stated by the following theorem.

Theorem 1. *Given a RAM program R and a state $(i, c_1, c_2, \ldots, c_n)$, we have $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (i', c'_1, c'_2, \ldots, c'_n)$ if and only if $[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \longrightarrow [\![(i', c'_1, c'_2, \ldots, c'_n)]\!]_R$.*

Hence, the encoding preserves both termination and divergence. A consequence of this theorem is that, for the language with permanent data, both termination and divergence are undecidable.

3 Temporary Data

In this section we study temporary data, i.e., data whose presence in the dataspace is granted at least for a minimal period specified by the producer. At the end of this period the data may be removed in order to free space needed, e.g., to store incoming data. We consider two possible policies for the collection of the data to be removed: the first, called *unordered collection*, removes one datum chosen among those with granted lifetime expired, while the second, called *ordered collection*, removes one of the data which expired first, i.e., one of those with oldest expiration time.

In order to model temporary data we need to represent the passing of time. To be as general as possible, we do not fix any specific model of time. We only assume what follows: *Time*, ranged over by t, t', ..., is a generic set of time instants; *Inter*, ranged over by Δt, $\Delta t'$, ..., is a set of time intervals; $\leq$ is a total order on *Time* such that $t \leq t'$ means that the time instant t' follows the instant t; $+ : \mathit{Time} \times \mathit{Inter} \to \mathit{Time}$ is an addition operation such that $t + \Delta t$ is the time instant in which a time interval Δt, starting at time instant t, will finish. We make the minimal reasonable assumption that, for any time instant t and time interval Δt, $t \leq t + \Delta t$; this means that the time instant in which a time interval finishes follows the instant in which it starts.

The syntax of programs under temporary data is defined as for the calculus with permanent data with the unique difference that the output operation requires a further parameter indicating the minimal lifetime of the produced datum; namely, $\mathit{out}(a, \Delta t)$ is substitute for the $\mathit{out}(a)$ prefix. When a datum is produced with minimal lifetime Δt, its expiration time is computed as $t + \Delta t$ where t is the current time.

In the following we use $\langle a \rangle_t$ to denote an instance of datum a with expiration time t; the index t is sometimes omitted when not relevant. Formally, we redefine $\mathit{DataSpace} = \mathcal{M}(\{\langle a \rangle_t \mid a \in \mathit{Name}, t \in \mathit{Time}\})$. The current time is added as a third parameter to configurations, thus we redefine $\mathit{Conf} = \{[P, DS, t] \mid P \in \mathit{Prog}, DS \in \mathit{DataSpace}, t \in \mathit{Time}\}$. In the following, we sometimes omit also the third component t from the denotation of configurations in the case the current time has no relevance.

The operational semantics for the new calculus is defined by the axioms and rules in Table 2: (1)...(6) are simple adaptations of the corresponding axioms and rules in Table 1. The most signicant difference is that whenever a a coordination operation is performed the current time is updated according to the relation $t \leq t'$ specifying the passing of time. Observe that axiom (2) computes the expiration time of the produced datum considering the current time at the moment the datum is effectively introduced in the dataspace, i.e., the time instant t' of the target configuration.

The two new axioms (7_u) and (7_o) define the unordered and ordered collection policies, respectively: (7_u) removes one of the expired data (simply by checking whether its expiration time preceeds the current time) while (7_o) removes one of the objects which expired first (requiring also that the expiration time is the minimum among those associated to the data currently in the dataspace).

Table 2. The operational semantics for temporary data (symmetric rule of (5) omitted).

(1)	$[in(a).P, \langle a\rangle \oplus DS, t] \longrightarrow [P, DS, t']$	if $t \leq t'$
(2)	$[out(a, \Delta t).P, DS, t] \longrightarrow [P, \langle a\rangle_{t'+\Delta t} \oplus DS, t']$	if $t \leq t'$
(3)	$[inp(a)?P\_Q, \langle a\rangle \oplus DS, t] \longrightarrow [P, DS, t']$	if $t \leq t'$
(4)	$[inp(a)?P\_Q, DS, t] \longrightarrow [Q, DS, t']$	if $t \leq t'$ and $\langle a\rangle_t \not\in DS$ for any t
(5)	$\dfrac{[P, DS, t] \longrightarrow [P', DS', t']}{[P \mid Q, DS, t] \longrightarrow [P' \mid Q, DS', t']}$	
(6)	$\dfrac{[P, DS, t] \longrightarrow [P', DS', t']}{[K, DS, t] \longrightarrow [P', DS', t']}$	$K = P$
(7_u)	$[P, \langle a\rangle_{t''} \oplus DS, t] \longrightarrow [P, DS, t']$	if $t \leq t'$ and $t'' \leq t'$
(7_o)	$[P, \langle a\rangle_{t''} \oplus DS, t] \longrightarrow [P, DS, t']$	if $t \leq t'$ and $t'' \leq t'$ and $t'' = min\{t \mid \langle a\rangle_t \in DS\}$

4 Unordered Collection

In this section we consider the new calculus with temporary data under the assumption that the collection policy is unordered. In particular, we prove an expressiveness gap with respect to the calculus with permanent data: first of all we show that the RAM encoding presented in Section 2 no longer works; then we show the existence of an alternative encoding, which preserves termination only; finally, we show that it is not possible to define any divergence preserving encoding because $C\!\uparrow$ is decidable.

A Termination Preserving Encoding of RAMs

The encoding of RAMs presented in Section 2 is not valid under temporary data, because the tuples representing the content of the registers may disappear; in this way, an undesired decrement of the registers may occurs. For example, the encoding of the terminating RAM program:

$$1 : inc(r_1)$$
$$2 : decJump(r_1, 1)$$

may never terminate in the case the datum $\langle r_1\rangle$ always expires and is removed before the second instruction is executed.

We now provide a nondeterministic encoding of RAMs which preserves termination. This encoding checks whether an undesired decrement of a register occurred during a completed finite computation; if this happens, we force the computation to diverge. In this way, if the computation terminates it is a correct computation.

In order to check the occurrence of undesired decrements of registers, we use the fact that the computation of the RAM we consider start and finish with all the registers empty; for this reason a completed RAM computation contains the same number of increments and decrements.

The encoding is modified by producing two particular programs *LOOP* and *KILL* every time an increment or a decrement is performed, respectively. *LOOP* has the ability to perform an infinite computation until it communicates with an instance of the program *KILL*. In this way, if the total number of increments is strictly greater than the total number of decrements then the computation cannot terminate, due to the existence of some *LOOP* programs unable to synchronize with a corresponding *KILL* program. For this reason, if the computation terminates we can conclude that no undesired decrements of register occurred thus it is a correct computation. On the other hand, if the computation does not terminate we cannot conclude anything about the corresponding RAM because it could be the case that the computation diverges due to the presence of some *LOOP* programs. *LOOP* and *KILL* are defined as follows:

$$\begin{aligned} LOOP &= inp(a)?\mathbf{0}\_LOOP \\ KILL &= out(a, \Delta t).\mathbf{0} \end{aligned}$$

where Δt can be any time interval. Observe that it could happen that a datum $\langle a \rangle_t$, produced by some *KILL* program, expires and is removed before being consumed by any corresponding *LOOP* program. If this happens, the corresponding *LOOP* program will loop forever, implying that the computation will never terminate. As we are interested in terminating computations only, this does not introduce any undesired behaviours.

The new encoding redefines the program constants P_i by adding the spawning of the *KILL* and *LOOP* programs:

$$\begin{aligned} P_i &= out(r_j, \Delta t).(LOOP|P_{i+1}) && \text{if } I_i = Succ(r_j) \\ P_i &= inp(r_j)?(KILL|P_{i+1})\_P_s && \text{if } I_i = DecJump(r_j, s) \end{aligned}$$

Finally, the encoding of the state $(i, c_1, c_2, \ldots, c_n)$ is defined as follows:

$$[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \;=\; [P_i|\prod_{c_1} LOOP|\ldots|\prod_{c_n} LOOP, \bigoplus_{1\le l\le n} \{\underbrace{\langle r_l\rangle, \ldots, \langle r_l\rangle}_{c_l \text{ times}}\}]$$

where by $\prod_j P$ we denote the parallel composition of j instances of the program P. We do not indicate the expiration time of the data in the dataspace because it is not relevant in order to evaluate the content of the registers of the considered state. Observe that each instance of $\langle r_j \rangle$ is equipped with a corresponding program *LOOP*.

The proof that the encoding preserves termination is based on two separated theorems. The first shows that each computation of the RAM may be simulated by a computation in the encoding.

Theorem 2. *Given a state* $(i, c_1, c_2, \ldots, c_n)$ *and a RAM program* R*, we have that* $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (i', c_1', c_2', \ldots, c_n')$ *implies* $[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \longrightarrow^* [\![(i', c_1', c_2', \ldots, c_n')]\!]_R$.

As we are assuming that terminal RAM states have all the registers empty, we have also that the corresponding encoding contains no *LOOP* programs, thus it is terminated. Due to this observation and the above theorem, we can conclude that a RAM computation leading to a terminal state has a corresponding finite computation of the encoding.

The second theorem states that any partial computation of an encoding can be either extended to reach a correct configuration or it can only admit infinite computations.

Theorem 3. *Given an initial state* $(1, 0, 0, \ldots, 0)$ *and a RAM program* R*, we have that if* $[\![(1, 0, 0, \ldots, 0)]\!]_R \longrightarrow^* C$ *then one of the following holds:*

1. *there exists* C' *such that* $C \longrightarrow^* C'$ *and* $C' = [\![(i, c_1, c_2, \ldots, c_n)]\!]$ *where* $(1, 0, 0, \ldots, 0) \longrightarrow_R^* (i, c_1, c_2, \ldots, c_n)$*;*
2. C *has only infinite computations.*

A consequence of this theorem is that any computation of an encoding leading to a terminated configuration corresponds to a correct RAM computation.

Finally, we can conclude that as the RAM encoding preserves the existence of terminating computations, $C\downarrow$ is undecidable also for the calculus with temporary data.

Divergence Is Decidable

In order to show the impossibility to define a divergence preserving encoding, we prove that $C\uparrow$ is decidable. In order to prove this result we resort to a semantics in terms of Place/Transition nets extended with *reset* arcs, a formalism for which the existence of an infinite firing sequence is decidable (see [3]). Here, we report a definition of this formalism adapted to our purposes.

Definition 1. *Given a set* S*, we denote by* $\mathcal{P}_{fin}(S)$ *and* $\mathcal{M}_{fin}(S)$ *the set of the finite sets and multisets on* S*, respectively. A P/T net with reset arcs is a triple* $N = (S, T, m_0)$ *where* S *is the set of* places*,* T *is the set of* transitions *(which are triples* $(c, p, r) \in \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S) \times \mathcal{P}_{fin}(S)$ *such that* r *has no intersection with both* c *and* p*), and* m_0 *is a finite multiset of places. Finite multisets over the set* S *of places are called* markings*;* m_0 *is called* initial marking*. Given a marking* m *and a place* s*,* $m(s)$ *denotes the number of occurrences of* s *inside* m *and we say that the place* s *contains* $m(s)$ tokens*. A P/T net with reset arcs is finite if both* S *and* T *are finite.*

A transition $t = (c, p, r)$ *is usually written in the form* $c \xrightarrow{r} p$ *and* r *is omitted when empty. The marking* c *is called the* preset *of* t *and represents the*

Table 3. Definition of the decomposition function *dec* and net transitions $\mathcal{T}$.

$$dec(\mathbf{0}) = \emptyset \qquad dec(in(a).P) = \{in(a).P\}$$
$$dec(out(a, \Delta t).P) = \{out(a).P\} \qquad dec(inp(a)?P\_Q) = \{inp(a)?P\_Q\}$$
$$dec(K) = dec(P) \quad \text{if } K = P \qquad dec(P|Q) = dec(P) \oplus dec(Q)$$
$$dec(\langle a \rangle_t \oplus DS) = \langle a \rangle \oplus dec(DS) \qquad dec(\emptyset) = \emptyset$$

`in(a,Q)`	$in(a).Q \oplus \langle a \rangle \longrightarrow dec(Q)$
`out(a,Q)`	$out(a).Q \longrightarrow \langle a \rangle \oplus dec(Q)$
`dis(a)`	$\langle a \rangle \longrightarrow \emptyset$
`inp+(a,Q,R)`	$inp(a)?Q\_R \oplus \langle a \rangle \longrightarrow dec(Q)$
`inp-(a,Q,R)`	$inp(a)?Q\_R \xrightarrow{\langle a \rangle} dec(R)$

tokens to be consumed. *The marking p is called the* postset *of t and represents the tokens to be* produced. *The set of places r denotes the* reset places. *The meaning of r is the following: when the transition fires all the tokens inside a place in r are removed.*

A transition $t = (c, p, r)$ is enabled *at m if $c \subseteq m$. The execution of the transition produces the new marking m' such that $m'(s) = m(s) - c(s) + p(s)$ if s is not in r, and $m'(s) = 0$ otherwise. This is written as $m \xrightarrow{t} m'$ or simply $m \longrightarrow m'$ when the transition t is not relevant. A marking m is* dead *if no transition is enabled at m. The net has a* deadlock *if it has a legal firing sequence leading to a dead marking.*

The basic idea underlying the definition of an operational net semantics for a process calculus is to decompose a term into a multiset of sequential components, which can be thought of as running in parallel. Each sequential component has a corresponding place in the net, and will be represented by a token in that place. Reductions are represented by transitions which consume and produce multisets of tokens.

In our particular case we deal with different kinds of *sequential components* represenfing the programs $in(a).P$, $out(a, \Delta t).P$, or $inp(a)?P\_Q$.

Any datum is represented by a token in a particular place $\langle a \rangle$. The way we represent input and output operations is standard; $in(a)$ removes a token from the place $\langle a \rangle$, while $out(a)$ produces a new token in the same place. In order to model ephemeral data, we connect to each place $\langle a \rangle$ a transition which simply removes a token from the place.

More interesting is the mechanism we adopt to model the execution of an $inp(a)$ operation. The idea is that whenever an $inp(a)?Q\_R$ process moves, the corresponding net may have two possible behaviours: either (i) a token is

consumed from place $\langle a \rangle$ and the continuation Q is activated, or (ii) the continuation R is activated and all the tokens in the place $\langle a \rangle$ are removed. This global consumption is achieved by using a reset arc.

The behaviour (i) corresponds to the successful execution of the $inp(a)$ operation, while (ii) corresponds to a sequence of moves: first each available datum $\langle a \rangle$ expires, then they are all removed, and finally the $inp(a)$ operation fails because no $\langle a \rangle$ is available any more.

The axioms in the first part of Table 3, describing the decomposition of programs and dataspaces in corresponding markings, state that the agent **0** generates no tokens; a sequential component produces one token in the corresponding place; a program constant is treated as its corresponding program definition; and the parallel composition is interpreted as multiset union, i.e, the decomposition of $P|Q$ is $dec(P) \oplus dec(Q)$. On the other hand, the decomposition of dataspaces is obtained simply by removing the time index from each single datum. Given a configuration $C = [P, DS, t]$ we define $dec(C) = dec(P) \oplus dec(DS)$ the marking containing the representation of both the active processes and the available data. Observe that the current time t of the configuration does not play any role in the definition of the corresponding marking.

The axioms in the second part of Table 3 define the possible transitions denoted by $\mathcal{T}$. Axioms `in(a,Q)` and `out(a,Q)` deal with the execution of the primitives $in(a)$ and $out(a)$, respectively: in the first case a token from place $\langle a \rangle$ is consumed, in the second one it is introduced. Axiom `dis(a)` removes one token from $\langle a \rangle$ in order to model one ephemeral datum $\langle a \rangle$ which disappears. Finally, axioms `inp+(a,Q,R)` and `inp-(a,Q,R)` describes the two possible behaviours for the $inp(a)$ operation; in the first case a token from place $\langle a \rangle$ is consumed and the first continuation is activated, in the second case the second continuation is activated and all the tokens in $\langle a \rangle$ are removed as effect of the presence of the reset arc.

Definition 2. *Let $C = [P, DS, t]$ be a configuration such that P has the following related program constant definitions: $K_1 = P_1, \ldots, K_n = P_n$. We define the triple $Net(C) = (S, T, m_0)$, where:*

$$
\begin{aligned}
S &= \{Q \mid Q \text{ is a sequential component of either } P, P_1, \ldots, P_n\} \cup \\
&\quad \{\langle a \rangle \mid a \text{ is a message name in either } P, P_1, \ldots, P_n, \text{ or } DS\} \\
T &= \{c \xrightarrow{r} p \in \mathcal{T} \mid \text{the components and the data in } c \text{ are also in } S\} \\
m_0 &= dec(C)
\end{aligned}
$$

Given a configuration C the corresponding $Net(C)$ is a finite P/T net with reset arcs.

In order to prove that $C\uparrow$ is decidable, we show that for any configuration C, $Net(C)$ has an infinite computation if and only if $C\uparrow$. This is a consequence of the following theorem based on two sentences. The first states that each computation step of the configuration C is matched by a transition in the corresponding net. The second states a similar symmetric result: each transition firable in the net can be mimicked by a sequence of computation steps of the corresponding configuration. The proof of the second sentence is by cases on the possible transitions; the unique non trivial case is `inp-(a,Q,R)` where we assume that all the

data $\langle a \rangle$ expire and are removed in the configuration before the corresponding program $inp(a)?Q\_R$ performs its $inp(a)$ operation.

Theorem 4. *Consider the Net(D) for some configuration D. Let m be a marking of the net such that $m = dec(C)$ for some configuration C.*

1. *If $C \longrightarrow C'$ then $m \longrightarrow dec(C')$ in Net(D).*
2. *If $m \longrightarrow m'$ then $C \longrightarrow^+ C'$ with $dec(C') = m'$.*

5 Ordered Collection

Here we prove the main results regarding the ordered collection policy: (i) $C\uparrow$ is no more decidable because a divergence preserving encoding of RAM exists, and (ii) there exists no encoding of RAM which preserves both termination and divergence.

A Divergence Preserving Encoding of RAMs

In the previous section we have discussed that it is not possible to define a divergence preserving encoding of RAMs when using temporary data under the unordered collection policy. The reason is that it is not possible, in the case of infinite computation, to check whether an undesired decrement of register occurs during the computation. Here we show that moving to ordered collection this check becomes possible.

The encoding presented in this section is based on the following idea. Each datum $\langle r_j \rangle$ is produced with an associated granted lifetime Δt. The granted lifetime is renewed before the execution of each instruction. The renewal is realized by removing and reproducing each datum. Between two subsequent renewals we check whether some of the data expire and are removed by expoiting the followign technique: before the first renewal we produce a special datum with a granted lifetime $\Delta t'$ shorter than Δt, then we check if this datum is still present at the end of the second one. In the case this special datum is still available, this means that the data emitted during the first renewal surely did not disappear before the second one (otherwise also the special datum should be removed as it should expire first) On the other hand, if the special datum expires and is removed, we cannot conclude any more that the computation is valid; in this case we block the computation. This forced termination is not a limitation of the encoding as we are interested in preserving the infinite computations only.

Concerning the renewal procedure, we have to divide it in two phases: first we rename each $\langle r_j \rangle$ in $\langle s_j \rangle$, and then each $\langle s_j \rangle$ in $\langle r_j \rangle$. In order to do this we need also two different special data $\langle a \rangle$ and $\langle b \rangle$.

The refreshing procedure is embedded in each program constant definition P_i. The new definitions are parametric in a program constant Q_i representing the effective part of the computation:

$$\begin{aligned} Q_i &= out(r_j, \Delta t).P_{i+1} && \text{if } I_i = Succ(r_j) \\ Q_i &= inp(r_j)?P_{i+1}\_P_s && \text{if } I_i = DecJump(r_j, s) \end{aligned}$$

We are now able to define the program constant P_i:

$$\begin{array}{lll}
P_i & = out(b, \Delta t').R_1 toS_1 & \\
R_k toS_k & = inp(r_k)?(out(s_k, \Delta t).R_k toS_k)_R_{k+1} toS_{k+1} & \text{for } k \in \{1, \ldots, n-1\} \\
R_n toS_n & = inp(r_n)?(out(s_n, \Delta t).R_n toS_n)_(in(a).out(a, \Delta t').S_1 toR_1) & \\
S_k toR_k & = inp(s_k)?(out(r_k, \Delta t).S_k toR_k)_S_{k+1} toR_{k+1} & \text{for } k \in \{1, \ldots, n-1\} \\
S_n toR_n & = inp(s_n)?(out(r_n, \Delta t).S_n toR_n)_(in(b).Q_i) &
\end{array}$$

where Δt and $\Delta t'$ are two generic time intervals such that for any time instant t we have $t + \Delta t' \leq t + \Delta t$.

We assume that before the execution of the program P_i the special datum $\langle a \rangle$ has been already emitted. In order to ensure this, we start the computation with the program $out(a, \Delta t').P_1$ instead of P_1 only.

The encoding of a RAM program R acting on the state $(i, c_1, c_2, \ldots, c_n)$ is then defined as follows:

$$[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \quad = \quad [P_i, \langle a \rangle \oplus \bigoplus_{1 \leq l \leq n} \{\underbrace{\langle r_l \rangle, \ldots, \langle r_l \rangle}_{c_l \text{ times}}\}]$$

The proof that the encoding preserves the existence of infinite computations is based on two separated theorems. Also in this case, the first shows that each computation step of the RAM may be simulated by a sequence of steps of the encoding.

Theorem 5. *Given a state $(i, c_1, c_2, \ldots, c_n)$ and a RAM program R, we have that $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (i', c'_1, c'_2, \ldots, c'_n)$ implies $[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \longrightarrow^+ [\![(i', c'_1, c'_2, \ldots, c'_n)]\!]_R$.*

The second theorem has to deal with all those intermediary configurations related to the renewal procedure. Nevertheless, we show that given a configuration reachable during a computation of the encoding, we have that either it is possible to extend the computation in order to reach the encoding of a correct RAM configuration, or the computation cannot be infinite.

Theorem 6. *Given an initial configuration $(1, 0, 0, \ldots, 0)$ and a RAM program R, we have that if $[\![(1, 0, 0, \ldots, 0)]\!]_R \longrightarrow^* C$ then one of the following holds:*

1. *there exists a conf. C' such that $C \longrightarrow^* C'$ and $C' = [\![(i, c_1, c_2, \ldots, c_n)]\!]_R$ where $(1, 0, 0, \ldots, 0) \longrightarrow^*_R (i, c_1, c_2, \ldots, c_n)$;*
2. *the configuration C has no infinite computations.*

Divergence Is Decidable (for Uniform Configurations)

In order to prove the impossibility to define a RAM encoding preserving both termination and divergence, we proceed by contraposition. We first assume the existence of such an encoding; then we show that for each RAM the corresponding encoding should be a configuration uniform with respect to termination;

finally, we prove that divergence is decidable for uniform configurations. This implies that the divergence of RAM is decidable (contradiction).

Suppose there exists a divergence and termination preserving encoding of RAMs: if the RAM terminates then its encoding has only finite computations, while if the RAM diverges then its encoding has only infinite computations. Hence, such a RAM encoding is uniform.

We resort to P/T nets to prove the decidability of divergence for uniform configurations. The P/T net semantics defined for the unordered collection is not satisfactory under the ordered policy. This because when a transition `inp-(a,Q,R)` fires, the connected reset arc removes all the tokens from the place $\langle a \rangle$ only. However, it could be the case that also other data expire before the some of the data $\langle a \rangle$; due to the ordered collection policy, also these data should be removed.

In order to overcome this limitation we extend the net with new reset arcs connecting each transition `inp-(a,Q,R)` with all the places representing data. In this way, when an `inp-(a,Q,R)` transition fires, all the data are removed. This behaviour corresponds to a sequence of computation steps in which all the data initially expire, then they are globally removed, and finally the considered $inp(a)$ fails.

Formally, given a configuration C and the previously defined $Net(C) = (S', T', m'_0)$, we define the new net semantics $Nnet = (S, T, m_0)$ as follows:

$$
\begin{aligned}
S &= S' \\
T &= T' \setminus \{t \mid t \text{ is one of the } \texttt{inp-(a,Q,R)} \text{ transitions}\} \cup \\
&\quad \{t = c \xrightarrow{\{\langle b \rangle \mid \langle b \rangle \in S\}} p \mid c \xrightarrow{\{\langle a \rangle\}} p \in T' \text{ for some } a\} \\
m_0 &= m'_0
\end{aligned}
$$

here $\setminus$ denotes set difference.

The new net semantics has the following properties.

Theorem 7. *Let C be a configuration. We have that:*

1. *if $Nnet(C)$ has an infinite firing sequence then $C\uparrow$;*
2. *if $Nnet(C)$ has a deadlock then $C\downarrow$.*

The first sentence is the consequence of the fact that each transition in the net can be mimicked by the corresponding configuration. The unique non-trivial case is the new kind of transitions related to the $inp(a)$ operation which removes all the tokens from the places representing data: as described above this transition is simulated by a sequence of computation steps of the configuration.

The second sentence states that a computation in the net which leads to a dead marking has a corresponding finite computation of the considered configuration. The intuition behind this result is that each dead marking in the net represents has a corresponding terminated configuration. Indeed, a dead marking in the net models a configuration composed by a set of programs which are either terminated or blocked, because they are trying to execute an *in* operation on one datum which is not actually available. This kind of configuration is trivially terminated.

We conclude by showing that given a uniform configuration C we have that the net $Nnet(C)$ has an infinite firing sequence *if and only if* $C\uparrow$. The *only if*

part, i.e., if $Nnet(C)$ has an infinite firing sequence then $C\uparrow$, has been proved as first sentence of the Theorem. The *if* part, i.e., if $C\uparrow$ then $Nnet(C)$ has an infinite firing sequence, is proved by contraposition: suppose that $C\uparrow$ and $Nnet(C)$ has no infinite firing sequence; this implies that the net has a deadlock, thus (by the second sentence of the Theorem) also $C\downarrow$; by uniformity of C we have $C \not\uparrow$ (contradiction).

6 Related Work

Recently, two proposals dealing with time in Linda-like languages appeared in the literature. Although both of them deal with timeout primitives, and not with temporary data, we briefly compare their models of time with our one. In [2] the passing of time is explicitely represented by means of transitions, and each action is assumed to take a unit of time to be performed. We claim that our results continue to hold also in this approach to time modeling. In [4] the two-phase functioning approach, tipical of synchronous languages, is adopted: in the first phase all actions are performed; when no further action can be performed, the second phase, consisting in a progress of time, takes place. If we follow this approach, our RAM encoding for permanent data also works in the case of temporary data.

References

1. N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, 156(1/2):90–121, 2000.
2. F.S. de Boer, M. Gabbrielli, and M.C. Meo. A Timed Linda Language. In *Proc. of Coordination2000*, volume 1906 of *Lecture Notes in Computer Science*, pages 299–304. Springer-Verlag, Berlin, 2000.
3. C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. of ICALP'98*, volume 1061 of *Lecture Notes in Computer Science*, pages 103–115. Springer-Verlag, Berlin, 1998.
4. J.M. Jacquet, K. De Bosschere, and A. Brogi. On Timed Coordination Languages. In *Proc. of Coordination2000*, volume 1906 of *Lecture Notes in Computer Science*, pages 81–98. Springer-Verlag, Berlin, 2000.
5. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
6. M.L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, 1967.
7. J.C. Shepherdson and J.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.
8. Sun Microsystem, Inc. *JavaSpaces Specifications*, 1998.
9. P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998.

On Garbage and Program Logic*

Cristiano Calcagno[1,2] and Peter W. O'Hearn[1]

[1] Queen Mary, University of London
[2] DISI, University of Genova

Abstract. Garbage collection relieves the programmer of the burden of managing dynamically allocated memory, by providing an automatic way to reclaim unneeded storage. This eliminates or lessens program errors that arise from attempts to access disposed memory, and generally leads to simpler programs. One might therefore expect that reasoning about programs in garbage collected languages would be much easier than in languages where the programmer has more explicit control over memory. But existing program logics are based on a low level view of storage that is sensitive to the presence or absence of unreachable cells, and Reynolds has pointed out that the Hoare triples derivable in these logics are even incompatible with garbage collection. We present a semantics of program logic assertions based on a view of the heap as finite, but extensible; this is for a logical language with primitives for dereferencing pointer expressions. The essential property of the semantics is that all propositions are invariant under operations of adding or removing garbage cells; in short, they are garbage insensitive. We use the assertion language to formulate notions of partial and total correctness for a small programming language, and provide logical characterizations of two natural notions of observational equivalence between programs.

1 Introduction

Garbage collection is an essential method used to reclaim heap-allocated objects whose lifetime cannot be easily predicted at compile time. It is most strongly associated with high-level languages such as Lisp, ML and Java, where heap allocation is the norm. But it can also be used in a lower level language like C, coexisting with explicit deallocation primitives [8]. In any case, garbage collection relieves the programmer of the burden of explicitly managing dynamically allocated memory. This generally leads to simpler programs, and removes or lessens errors that result from incorrect attempts to access disposed memory, errors that are often difficult to diagnose or even reproduce.

Given the intuitive rationale for garbage collection, one would expect that reasoning about garbage collected languages (especially memory safe languages) would be much easier than for lower level languages. But existing program logics take a view of storage that is sensitive to the presence of unreachable cells, a view that is not invariant under garbage collection.

* Work partially supported by the EPSRC

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 137–151, 2001.

1.1 A Logical Conundrum

The following problem was raised by Reynolds in [11].

Consider a statement form $x := \mathtt{cons}(E, E')$ that allocates and initializes a new cons cell and places a pointer to that cell in storage variable x. Then the following sequence of instructions creates a new cell, and then makes it garbage.

$$x := \mathtt{cons}(3, 4);\ x := z.$$

Now, ask the question: is there a pointer y to a cell in the heap where $y.1 = 3$, after these statements have been executed? From the point of view of execution the answer is that it depends, on whether a garbage collector has taken control or not. (We use $x.1$ and $x.2$ for the combination of dereferencing a pointer and accessing one of the two components of a cons cell.)

The conundrum is that program logic seems to take a particular stance, one that is incompatible with garbage collection. That is, previous logics for pointer programs would allow us to derive a Hoare triple

$$\{true\}\, x := \mathtt{cons}(3, 4);\ x := z\, \{\exists y.\, y.1 = 3\}.$$

The problem is that on termination there might not actually be a cell whose car is 3, if a garbage collector reclaims the detached cell.

It is worth looking at this example in more detail. After the statement $x := \mathtt{cons}(3, 4)$ has been executed, it has to be admitted that there is such a cell, because it cannot be garbage collected. So we would expect to have a true Hoare triple $\{true\}\, x := \mathtt{cons}(3, 4)\, \{\exists y.\, y.1 = 3\}$. But then either Hoare's of Floyd's assignment axiom gives us $\{\exists y.\, y.1 = 3\}\, x := z\, \{\exists y.\, y.1 = 3\}$ because x is free in neither the precondition nor the postcondition. And now the die is cast: we obtain the triple above by sequencing.

The conundrum related to the problem of *full abstraction*. That is, there are Hoare triple contexts that can distinguish programs that are observationally equivalent, if we take a standard notion of observation (such as termination). For example, one would expect the program fragment given above to be observationally equivalent to $x := z$ on its own, but if the standard assignment axiom is to be believed then the weakest precondition of $\exists y.\, y.1 = 3$ is just itself, so we only get $\{\exists y.\, y.1 = 3\}\, x := z\, \{\exists y.\, y.1 = 3\}$. Thus, if there are states where this precondition is false, then the logical context $\{true\} - \{\exists y.\, y.1 = 3\}$ will distinguish two "equivalent" commands, since it will not be satisfied by $x := z$. (In this discussion we have ignored the possibility of running out of memory. But a similar point can be made if we bound the number of possible pointers by strengthening the precondition to say that there is at least one cell available, and by using $x := \mathtt{cons}(7, 8);\ x := z$ in place of the single statement $x := z$.)

Of course, the foregoing hinges on several points: the substitution-oriented treatment of assignment and a typical reading of the logical operators. And there are several ways that one might react to the conundrum; we discuss several of the possibilities in the conclusion. The purpose of this paper is to provide one solution, in the form of a semantics of pointer assertions that is insensitive to garbage.

1.2 A Resolution

The semantics we give is based on a view of memory as finite but extensible. What we mean by this is that at any given time the heap consists of a finite collection of cons cells in storage, but the quantifiers are arranged so as to permit arbitrary (but finite) extensions to the heap. This has echoes of the idea of "finite but potentially infinite", but the semantics is also compatible with overflow-sensitive situations where a bound is placed.

We use a possible-world semantics, where the current heap (a finite collection of cells) is the world. The most important case is the interpretation of $\exists x.\, P$. As usual, x here ranges over values, including pointers and even pointers not in the current heap. If such a new pointer is chosen, then the world is extended as well, to provide a binding for (at least) the new pointer. The other connectives are interpreted pointwise, without changing worlds.

After presenting the model we prove three main results. The first is garbage insensitivity: the semantics of assertions is invariant under the operations of removing or adding garbage cells.

The second and third are both full abstraction results, which establish that two programs satisfy the same Hoare triples just when they are observationally equivalent. One of these results connects total correctness specifications with an equivalence obtained from observing termination, and the other connects partial correctness specifications with an equivalence obtained from observing the presence of runtime errors (such as dereferencing *nil*) as well as termination. The first equivalence, for total correctness, conflates divergence and runtime error. Both equivalences equate the program fragments discussed in the conundrum above.

We also establish a connection between the total states model and a partial states model based on the celebrated "dense topology" semantics of classical logic [3,9]. This connection provides some theoretical justification for the unusual semantics of quantification in the total states model.

We work with a pared down storage model and programming language for this study, where the heap consists of a collection of binary cons cells, and where there are data pointers but no code pointers or closures. Garbage insensitivity appears to extend to higher-order store, but the extension of full abstraction is not obvious. Even though it is simple, this kind of storage model has interesting structure that has only recently begun to be uncovered [11,7].

In this paper we concentrate on garbage and the relation between specifications and observational equivalence; we do not consider program logic axioms for the individual statements of our programming language. In a future paper we will show how the approach of [11,7] can be adapted, to provide a logic that is sound for an operational semantics with rules for garbage collection.

2 The Storage Model

The storage model we use supports pointer allocation, dereferencing and assignment, and divides the state into *stack* and *heap* components. The stack consists

of associations of values to variables, and is altered by the standard assignment statement $x := E$. As is common in Hoare logic, we do not distinguish between a variable name and the l-value it denotes; this is justified because we are not considering aliasing between stack variables. The heap consists of a collection of binary cons cells, which can only be accessed via pointers; the extension to records of other sizes is straightforward.

The basic domains of the model are as follows.

$$\begin{aligned}
\mathtt{Pointers} &\stackrel{\Delta}{=} \{p, q, ...\} \quad \mathtt{Bool} \stackrel{\Delta}{=} \{false, true\} \quad \mathtt{Tags} \stackrel{\Delta}{=} \{a, b, ...\} \\
\mathtt{Variables} &\stackrel{\Delta}{=} \{x, y, ...\} \quad \mathtt{Nat} \stackrel{\Delta}{=} \{0, 1, ...\} \\
\mathtt{Values} &\stackrel{\Delta}{=} \mathtt{Nat} + \mathtt{Bool} + \mathtt{Pointers} + \{nil\} \\
\mathtt{Stacks} &\stackrel{\Delta}{=} \mathtt{Variables} \rightharpoonup_{fin} \mathtt{Values} \\
\mathtt{Heaps} &\stackrel{\Delta}{=} \mathtt{Pointers} \rightharpoonup_{fin} \mathtt{Values} \times \mathtt{Values} \\
\mathtt{States} &\stackrel{\Delta}{=} \mathtt{Stacks} \times \mathtt{Heaps}
\end{aligned}$$

Although there is no ordering on variables, in the programming language to be presented later allocation and deallocation of variables will obey a stack discipline. Stacks and heaps are represented by finite partial functions: we write $dom(s)$ or $dom(h)$ for the domain of definition of a stack or heap.

Notice that a state $s, h \in \mathtt{Stacks} \times \mathtt{Heaps}$ might have dangling pointers, pointers that are reachable but not defined in h. Since we will consider a programming language without memory disposal, total states play a central role. We use the following definitions.

- Pointer q is reachable from p in h if $q = p$ or $h(p) = \langle v_1, v_2 \rangle$ and q is reachable from v_1 or v_2 in h;
- p is reachable in s, h if there is $x \in dom(s)$ such that p is reachable from $s(x)$ in h;
- (s, h) is total if every reachable pointer is in the domain of h.

We will also be concerned with the presence or absence of garbage.

- (s, h) is garbage free if every $p \in dom(h)$ is reachable in s, h.

Garbage collection is intimately connected to the relation of heap extension, which gives us a partial order on $\mathtt{Heaps}$.

- $g \sqsubseteq h$ indicates that the graph of heap g is a subset of the graph of heap h.

From the point of view of this model, given a state s, h the effect of a garbage collector is to shrink h by selecting $g \sqsubseteq h$ in a way that doesn't produce any (new) dangling pointers. In the best case, the collector will remove all garbage.

- $prune(s, h) = (s, g)$, where $g \sqsubseteq h$ is the subheap of h restricted to those pointers reachable in s, h.

Actually, a relocating collector can move heap cells around. For this the notion of isomorphism is important.

– $=_\alpha$ is equality of states modulo renaming of pointers.
(To be explicit, if $f : \texttt{Pointers} \to \texttt{Pointers}$ is a permutation, let f^* be the induced function from values to values which is the identity on non-pointer values. Then $s, h =_\alpha s', h'$, where $s' = s; f^*$ and $h' = f^{-1}; h; (f^* \times f^*)$.)

Lemma 1. *The following hold:*

– *prune*(s, h) *is garbage free.*
– *If* (s, h) *is total then so is prune*(s, h).
– *The relation* $=_\alpha$ *preserves totality and garbage freedom: If* $(s, h) =_\alpha (s', h')$ *and* s, h *is total (garbage free) then* (s', h') *is total (garbage free).*

3 Expressions

We restrict our attention to expressions that are free from side effects. A programming language for transforming states will be given later in Section 6, after we have presented the assertion language.

We include standard operations of arithmetic, along with generic equality testing and pointer dereference. The syntax of expressions is given by the following grammar:

$$\begin{aligned} E ::= {} & x \\ \mid {} & 0 \mid 1 \mid E + E \mid E \times E \mid E - E \\ \mid {} & \texttt{true} \mid \texttt{not}\, E \mid E \,\texttt{and}\, E \mid E == E \\ \mid {} & \texttt{nil} \\ \mid {} & E.1 \mid E.2 \end{aligned}$$

To keep things simple we have avoided all issues of typing. A type system could eliminate type errors in many cases but not, without great complication, for pointer dereferencing, where `nil` or some other value is typically used as a special "pointer" that should not be dereferenced.

So, the semantics of an expression E will determine an element

$$[\![E]\!]s, h \in \texttt{Values} + \{wrong\}$$

where s, h is a total state and the domain of s includes the free variables of E. A *wrong* result indicates a type error; this is treated essentially as in a partial function semantics, where *wrong* stands for undefined.

Selected semantics definitions are as follows.

$$
\begin{array}{rll}
[\![x]\!]s,h \stackrel{\Delta}{=} s(x) & & \\
[\![E_1 == E_2]\!]s,h \stackrel{\Delta}{=} & wrong & \text{if } [\![E_1]\!]s,h = wrong \text{ or } [\![E_2]\!]s,h = wrong \\
& true & \text{if } wrong \neq [\![E_1]\!]s,h = [\![E_2]\!]s,h \\
& false & \text{if } wrong \neq [\![E_1]\!]s,h \neq [\![E_2]\!]s,h \neq wrong \\
[\![\texttt{not}\ E]\!]s,h \stackrel{\Delta}{=} & wrong & \text{if } [\![E]\!]s,h = wrong \\
& false & \text{if } [\![E]\!]s,h = true \\
& true & \text{if } [\![E]\!]s,h = false \\
[\![E.i]\!]s,h \stackrel{\Delta}{=} & wrong & \text{if } [\![E]\!]s,h \notin dom(h) \\
& \pi_i(h(p)) & \text{if } [\![E]\!]s,h = p \in dom(h)
\end{array}
$$

It is not difficult to verify that the semantics of expressions is insensitive to garbage.

Lemma 2. $[\![E]\!]s,h \;=\; [\![E]\!]prune(s,h)$.

Such a result is much more difficult for assertions than expressions, because of quantifiers which may quantify over unreachable elements.

4 Propositions

The grammar for propositions P is as follows.

$$P ::= E = E \mid P \to P \mid \texttt{false} \mid \exists x.\, P$$

We can define various other connectives as usual: $\neg P \stackrel{\Delta}{=} P \to \texttt{false}$; $\texttt{true} \stackrel{\Delta}{=} \neg(\texttt{false})$; $P \vee Q \stackrel{\Delta}{=} (\neg P) \to Q$; $P \wedge Q \stackrel{\Delta}{=} \neg(\neg P \wedge \neg Q)$; $\forall x.\, P \stackrel{\Delta}{=} \neg\exists x.\, \neg P$.

The assertion language looks rather sparse, but using the expressions a number of properties can be defined. For example, $E = E + 0$ says that E is a number, and $E.1 = E.1$ says that E is a pointer. In practice, one would also want atomic predicates for describing reachability properties (e.g. [1]), or a definitional mechanism for defining them. Any such predicates could be included, as long as they satisfy the properties Growth, Shrinkage and Renaming described in the next section.

The semantics is described in terms of a judgement of the form

$$s,h \models P$$

which asserts that P holds of total state s,h. We require $free(P) \subseteq dom(s)$, where $free(P)$ indicates the set of variables occurring free in P.

The quantifier-free fragment is straightforward.

$$
\begin{array}{ll}
s,h \models E = E' \stackrel{\Delta}{\iff} & [\![E]\!]s,h = [\![E']\!]s,h = v \in \texttt{Values} \\
s,h \models \texttt{false} & \text{never} \\
s,h \models P \to Q \stackrel{\Delta}{\iff} & \text{if } s,h \models P \text{ then } s,h \models Q
\end{array}
$$

The only point to note is the interpretation of equality, which requires that neither side is wrong.

Normally, one would expect the following interpretation of $\exists$.

EXISTENTIAL: FIRST TRY

$$s, h \models \exists x.P \overset{\Delta}{\iff} \exists v \in \texttt{Values}.(s \mid x \mapsto v), h \models P$$

($(s \mid x \mapsto v)$ is the stack like s except that x maps to v.) The immediate problem is that this is not even well defined: there is no guarantee that $(s \mid x \mapsto v), h$ be a total state, so the right-hand side is not well formed. We might attempt to allow partial states and maintain this interpretation, but then we run into the conundrum mentioned in the Introduction.

Our solution is to accompany the selection of a value with a change of world. That is, if v is a pointer whose value is not determined in the current heap, then we look for a bigger heap where the value is determined. And in doing so, we must also ensure that no dangling pointers are introduced by following v further into the heap. Thus, we look for an extended heap, which maintains total state status.

EXISTENTIAL: OFFICIAL VERSION

$$\begin{aligned} s, h \models \exists x.P \overset{\Delta}{\iff} \exists v \in \texttt{Values}.\, \exists g \sqsupseteq h.& \\ (s \mid x \mapsto v), g \text{ is total and } (s \mid x \mapsto v), g \models P& \end{aligned}$$

Another idea that occurs is to quantify over reachable pointers only; we discuss in the conclusion.

It is worth spelling out the induced semantics of $\forall$, that obtains from the $\neg\exists\neg$ encoding:

$$\begin{aligned} s, h \models \forall x.P \overset{\Delta}{\iff} \forall v \in \texttt{Values}.\, \forall g \sqsupseteq h.\ &\text{if } (s \mid x \mapsto v), g \text{ is total} \\ &\text{then } (s \mid x \mapsto v), g \models P \end{aligned}$$

The semantics handles the example from Section 1.1 as follows. Given any heap, the semantics of $\exists$ allows a new pointer to be selected, along with a heap extension (change of possible world) which has 3 in the first component; $\exists y.\, y.1 = 3$ is always true. So, both of the Hoare triple contexts in Section 1.1 boil down to $\{true\}$–$\{true\}$. And, because of the full abstraction results in Sections 7 and 8, we know that the semantics is not *too* abstract: it makes *enough* distinctions to distinguish genuinely inequivalent commands.

One property of the model is worth noting, the truth of

$$\forall y.\, \exists x.\, x.1 = y$$

This says that for any value we consider, there is some cell we can find which has y in its first component. And the same goes for the second component. This illustrates the extensible heap view taken by the semantics, where there is an extensible stock of pointers, or locations, with arbitrary possible contents, to choose from. (This formula is true in all states only if `Pointers` is infinite. With a finite collection of pointers, $\forall y$ could saturate the set of available pointers leaving none for $\exists x.\, x.1 = y$.)

Many specifications one uses when reasoning about pointer programs are garbage insensitive. For example: x points to a non-circular linked list; or x points to a linked list representing the sequence L. With the aid of recursive definitions (which we have avoided for theoretical simplicity in this paper) these properties can be expressed in our language, as in $nclist(x) \Leftrightarrow x = \mathtt{nil} \vee (\exists z.\, x.2 = z \wedge nclist(z))$.

Examples of properties that are not garbage insensitive include some that are invariant under pointer renaming (relocation)and some that are not. An example of the former is "the heap has exactly n cells", while an example of the latter is "if we add 2 to pointer p, we get pointer q". Garbage sensitive properties are useful in lower level languages.

5 Garbage Insensitivity

There are two aspects to garbage insensitivity, which we label growth and shrinkage.

Growth. If $s, g \models P$, $g \sqsubseteq h$, and s, h is total then $s, h \models P$.

Shrinkage. If $s, h \models P$, $g \sqsubseteq h$, and s, g is total then $s, g \models P$.

Growth says just that if a proposition is true then it remains true if we add extra cells to the heap. *Shrinkage* says the opposite: truth is preserved by removing cells from the heap. In both cases there is the proviso that the cells added or removed do not result in dangling pointers, as our semantics is defined only for total states.

Growth is essentially Kripke's monotonicity property for intuitionistic logic, but for one difference: the side condition on totality is a property that refers to both the stack and heap components. Conventional Kripke monotonicity would consider all larger heaps (possible worlds), without any side conditions that refer to stacks/environments.

Shrinkage looks like a backwards form of Kripke monotonicity, but is not. Because of the restriction to total states, we will only be able to go back as far as the heap obtained by pruning, which is dependent on s. (In the dense semantics presented later, which allows partial states, forwards Kripke monotonicity does hold, while backwards does not.)

Theorem 1 (Garbage Insensitivity Theorem). *All propositions satisfy Growth and Shrinkage. As a consequence, each proposition is completely determined by its meaning at garbage-free states:*

$s, h \models P$ *iff* $prune(s, h) \models P$.

The proof of the theorem relies on the

Lemma 3 (Renaming Lemma). *All propositions are invariant under pointer renaming. If* $s, h \models P$ *and* $s, h =_\alpha s', h'$ *then* $s', h' \models P$.

If we combine this with Garbage Insensitivity then we obtain a high level correspondent to the idea that propositions are invariant under the operation of a relocating garbage collector. The Renaming Lemma is true because we have provided no facilities for pointer arithmetic among the expressions we consider.

The theorem holds for any collection of atomic predicates that are closed under Growth, Shrinkage and Renaming.

Proof: [of Garbage Insensitivity] By structural induction on P.

The case of `false` is trivial. For $P \to Q$ we consider *Shrinkage*. Suppose $s, h \models P \to Q$ and consider $g \sqsubseteq h$ where s, g is total. Either $s, h \not\models P$ or $s, h \models Q$ must hold. If the former, we use the *Growth* part of the induction hypothesis for P to conclude $s, g \not\models P$, and thus $s, g \models P \to Q$. If the latter we use the *Shrinkage* induction hypothesis to conclude $s, g \models Q$, and thus $s, g \models P \to Q$. The proof of *Growth* for $P \to Q$ is symmetric.

For the *Growth* case of $\exists x.\, P$, if $s, g \models \exists x.\, P$ then there is v and $k \sqsupseteq g$ with $(s \mid x \mapsto v), k \models P$. Consider a permutation that renames the pointers in the domain of k but not g to be different from all pointers in h, while leaving pointers in g fixed. Let k' be the heap obtained from k via this permutation. Then $(s \mid x \mapsto v'), k' \models P$ by the Renaming Lemma, where v has been perhaps renamed to v' (if v is in the domain of k but not g). Let j denote the lub of h and k'. j exists: it is just the union of h and k', which have g in common but which are otherwise defined on distinct pointers. It is not difficult to see that $(s \mid x \mapsto v'), k'$ is total. (Since the permutation fixes g, no renaming is required in s.) Since $j \sqsupseteq k'$, we can apply the *Growth* induction hypothesis, to obtain $(s \mid x \mapsto v'), j \models P$. Since $j \sqsupseteq h$ we satisfied the right-hand-side of the definition of $s, h \models \exists x.\, P$.

The *Shrinkage* case of $\exists x.\, P$ follows immediately from the clause for $\exists$ and the transitivity of $\sqsubseteq$.

The case of the equality predicates is immediate from Lemma 2. ■

Interestingly, the validity of the usual law of $\exists$ elimination

$$\frac{Q \wedge P \models R \quad Q \models \exists x.P}{Q \models R} \ (x \text{ not free in } Q \text{ or } R)$$

relies on Garbage Insensitivity. (Here, $\models$ refers to the semantic consequence relation between propositions, where $P \models Q$ if any state satisfying P also satisfies Q.) The proof of validity of this law requires a variable weakening lemma, which involves a change of world as well as an additional variable.

Lemma 4 (Variable Weakening Lemma). *If $s, h \models P$, $g \sqsupseteq h$, x is not free in P, and $(s \mid x \mapsto v), g$ is total, then $(s \mid x \mapsto v), g \models P$.*

The proof of this lemma uses only the *Growth* part of Garbage Insensitivity, but *Growth* itself relies on *Shrinkage* for propositions of the form $P \to Q$

The following version of $\exists$ introduction also holds:

$$\frac{Q \models P[E/x] \quad Q \models E = E}{Q \models \exists x.\, P}$$

Recall that our interpretation of equality is like in a partial function logic, where $E = F$ can hold only when neither side is *wrong*. Thus, the premise $E = E$ is not trivially true: it ensures that E denotes a genuine value.

6 A Programming Language

We define a small programming language for altering stacks and heaps.

$$\begin{array}{l} C ::= x := E \mid E.i := E \mid x := \mathtt{cons}(E_1, E_2) \\ \quad \mid \; C; C \mid \mathtt{while}\ E\ \mathtt{do}\ C\ \mathtt{od} \mid \mathtt{local}\ x\ \mathtt{in}\ C\ \mathtt{end} \end{array}$$

The binding form $\mathtt{local}\ x\ \mathtt{in}\ C\ \mathtt{end}$ declares a new stack variable, which is de-allocated on block exit. This is why we refer to the s component of the semantics as the stack.

The commands are interpreted using a relation $\rightsquigarrow$ on configurations. Configurations include terminal configurations s, h as well as triples C, s, h. Also, in order to treat `local` we introduce a new command $\mathtt{dealloc}(x)$; this command form is used only in the operational semantics, and is not part of the language. In other sections metavariable C will refer exclusively to the unextended language, while here it includes `dealloc`.

The $\rightsquigarrow$ relation is specified by the following rules.

$$\frac{[\![E]\!]s, h = v}{x := E, s, h \rightsquigarrow (s \mid x \mapsto v), h} \qquad \frac{[\![F]\!]s, h = v \quad [\![E]\!]s, h = p \in dom(h)}{E.i := F, s, h \rightsquigarrow s, (h \mid p.i \mapsto v)}$$

$$\frac{p \notin dom(h) \quad p \in \mathtt{Pointers} \quad [\![E_1]\!]s, h = v_1 \in \mathtt{Values} \quad [\![E_2]\!]s, h = v_2 \in \mathtt{Values}}{x := \mathtt{cons}(E_1, E_2), s, h \rightsquigarrow (s \mid x \mapsto p), (h \mid p \mapsto (v_1, v_2))}$$

$$\frac{C_1, s, h \rightsquigarrow C_1', s', h'}{(C_1; C_2), s, h \rightsquigarrow (C_1'; C_2), s', h'} \qquad \frac{C_1, s, h \rightsquigarrow s', h'}{(C_1; C_2), s, h \rightsquigarrow C_2, s', h'}$$

$$\frac{[\![E]\!]s, h = false}{\mathtt{while}\ E\ \mathtt{do}\ C\ \mathtt{od}, s, h \rightsquigarrow s, h}$$

$$\frac{[\![E]\!]s, h = true \quad (C; \mathtt{while}\ E\ \mathtt{do}\ C\ \mathtt{od}), s, h \rightsquigarrow K}{\mathtt{while}\ E\ \mathtt{do}\ C\ \mathtt{od}, s, h \rightsquigarrow K}$$

$$\frac{}{\mathtt{local}\ x\ \mathtt{in}\ C\ \mathtt{end}, s, h \rightsquigarrow C; \mathtt{dealloc}(x), (s \mid x \mapsto nil), h}$$

$$\frac{}{\mathtt{dealloc}(x), s, h \rightsquigarrow s - x, h}$$

In the last rule $s - x$ is the stack like s but undefined on x. $(h \mid p.i \mapsto v)$ is the heap like h except that the i'th component of the $h(p)$ cell is v.

We say that

- "C, s, h is stuck" in case there is no configuration K such that $C, s, h \rightsquigarrow K$;
- "C, s, h goes wrong" when there exists a configuration K such that $C, s, h \rightsquigarrow^* K$ and K is stuck;
- "C, s, h terminates normally" just if there is s, h such that $C, s, h \rightsquigarrow^* s', h'$.

It is possible to prove a number of properties about this operational semantics, such as that normal termination, going wrong and divergence are mutually exclusive, invariance under pointer renaming, and a kind of determinacy, where the prunes of any two output states gotten from the same initial state must be isomorphic.

One can also entertain an extension of the operational semantics with rules for garbage collection or even relocation; e.g. $s, h \rightsquigarrow s, g$ when $g \sqsubseteq h$ and s, g is total. Such extensions do not affect the results that follow.

7 Partial Correctness

If P and Q are propositions and C is a command then we have the specification form $\{P\}\, C\, \{Q\}$.

Partial Correctness
We say that $\{P\}\, C\, \{Q\}$ is true just if for all s, h, if $s, h \models P$ then
- C, s, h doesn't go wrong, and
- if $C, s, h \rightsquigarrow^* s', h'$ then $s', h' \models Q$.

We have arranged the definition of partial correctness so that well specified programs don't go wrong. That is, if $\{P\}C\{Q\}$ holds then executing C in a state satisfying P never leads to a runtime error.

Using partial correctness assertions it is possible to distinguish between programs that differ in when they go wrong, but that agree on all outputs states when they are reached. As an extreme example, consider a command *diverge* that always diverges without getting stuck (e.g. `while true do` $x := x$ `od`) and another command $x :=$ `nil`; $x.1 := 17$ that always gets stuck. The former satisfies the triple $\{true\}\,-\,\{true\}$ while the latter does not. This indicates that the notion of observational equivalence appropriate to partial correctness must distinguish divergence and stuckness.

These considerations lead to a notion $\cong_{pc}$ of observational equivalence based on observing both normal termination and stuckness. In formulating observational equivalence, we will only consider the execution of closed commands, ones where all variables have been bound by `local`.

We define $C \cong_{pc} C'$ to hold just if
for all closing contexts $G[\cdot]$,
- $G[C], (), ()$ goes wrong just if $G[C'], (), ()$ does, and
- $G[C], (), ()$ terminates normally just if $G[C'], (), ()$ does.

Theorem 2 (Full Abstraction, Partial Correctness). *$C \cong_{pc} C'$ iff C and C' satisfy the same partial correctness assertions.*

The proof goes by first relating $\cong_{pc}$ to a function $[\![C]\!]^{\sim}_{pc}$ determined by a command, which ignores garbage and renaming of locations, and then relating this relation to partial correctness. First, we define

$$[\![C]\!]_{pc} \subseteq \texttt{TStates} \times \texttt{TStates}_*$$

where `TStates` is the set of total states, and $\texttt{TStates}_* \stackrel{\Delta}{=} \texttt{TStates} \cup \{wrong\}$.

$$((s,h),(s',h')) \in [\![C]\!]_{pc} \overset{\Delta}{\Longleftrightarrow} C,s,h \rightsquigarrow^* s',h'$$
$$((s,h),wrong) \in [\![C]\!]_{pc} \overset{\Delta}{\Longleftrightarrow} C,s,h \text{ goes wrong}$$

The equivalence relation $\sim \subseteq \mathtt{TStates} \times \mathtt{TStates}$ is defined as

$$(s,h) \sim (s',h') \quad \overset{\Delta}{\Longleftrightarrow} \quad prune(s,h) =_\alpha prune(s',h')$$

where $=_\alpha$ is equality modulo renaming of locations. The theorem is then established by proving three lemmas.

Lemma 5. *$[\![C]\!]_{pc}$ induces a partial function between quotients*

$$[\![C]\!]^{\sim}_{pc} : (\mathtt{TStates}/\sim) \rightharpoonup (\mathtt{TStates}_*/\sim_*)$$

where $-/-$ is the quotient operation and $\sim_*$ extends $\sim$ with $(wrong, wrong)$.

Lemma 6. *$C \cong_{pc} C' \Leftrightarrow [\![C]\!]^{\sim}_{pc} = [\![C']\!]^{\sim}_{pc}$.*

Lemma 7. *$[\![C]\!]^{\sim}_{pc} = [\![C']\!]^{\sim}_{pc} \Leftrightarrow C$ and C' satisfy the same partial correctness assertions.*

The proofs of lemmas 6 and 7 use as main ingredient the fact that for each equivalence class c, there exists a command that creates an instance of c and an expression/proposition that characterizes c.

It is also possible to prove a version of the theorem which characterizes an observational approximation relation, rather than equivalence.

8 Total Correctness

If P and Q are assertions and C is a command then we have the specification form $[P]\,C\,[Q]$.

Total Correctness We say that $[P]\,C\,[Q]$ is true just if
- $\{P\}\,C\,\{Q\}$ is true, and
- for all s,h, if $s,h \models P$ then C,s,h terminates normally.

Using total correctness assertions we can no longer distinguish between divergence and going wrong in the way that we did with partial correctness. That is, neither divergence nor an always sticking command satisfies $[true]\,-\,[true]$, because neither terminates normally. This suggests to define a notion of observational equivalence that conflates divergence and going wrong. Since, in our language, there is no way to recover from, or handle, a runtime error, we can do this by observing termination only (ignoring wrongness).

We define $C \cong_{tc} C'$ to hold just if
for all closing contexts $G[\cdot]$,
- $G[C]()()$ terminates normally just if $G[C']()()$ does.

Theorem 3 (Full Abstraction, Total Correctness). *$C \cong_{tc} C'$ iff C and C' satisfy the same total correctness assertions.*

The proof idea is similar to the one for partial correctness, except that we use a function on quotients that conflates non-termination and getting stuck.

9 Partial States and the Dense Semantics

We now give an alternate presentation of the model using partial states, which leads to a connection with the dense semantics of classical logic [3,9]. We begin by defining a notion of support, which works for partial states. Intuitively, $s, h \Vdash P$ holds if h contains enough information to conclude P. Technically, this means that, for any total state we wish to extend h to, P will hold. Suppose s, h is a state, perhaps partial. Then

$$s, h \Vdash P \quad \stackrel{\Delta}{\iff} \quad \forall g \sqsupseteq h. \text{ if } s, g \text{ is total then } s, g \models P$$

Support has the following properties.

Totality Condition: $s, h \Vdash P \iff \forall g \sqsupseteq h.$ if s, g is total then $s, g \Vdash P$.

Shrinkage Condition: If $s, h \Vdash P$, $g \sqsubseteq h$, and the domain of g contains all those pointers reachable in s, h, then $s, g \Vdash P$.

Density Condition: $s, h \Vdash P \iff \forall h' \sqsupseteq h. \exists h'' \sqsupseteq h'. s, h'' \Vdash P$.

These conditions are not independent. In particular, Density follows from Monotonicity and Shrinkage. Totality and Shrinkage are of interest from the perspective of pointers, while Density is more of a logical property.

Theorem 4. *All propositions satisfy Totality, Shrinkage and Density. Furthermore, the standard clauses of dense semantics all hold.*

$$\begin{array}{lll} s, h \Vdash \texttt{false} & & \text{never} \\ s, h \Vdash P \to Q & \iff & \forall h' \sqsupseteq h \,.\, \text{if } s, h' \Vdash P \;\; \textit{then} \;\; s, h' \Vdash Q \\ s, h \Vdash \exists x.P & \iff & \forall h' \sqsupseteq h,\, \exists h'' \sqsupseteq h'. \exists v \in \texttt{Values}. (s \mid x \mapsto v), h'' \Vdash P \end{array}$$

As a result, the forcing relation $\Vdash$ satisfies all the laws of classical logic (with the appropriate side-condition on $\exists$-intro to account for definedness).

The conditions in this proposition can be taken as an alternate definition of $\Vdash$, one that does not appeal to $\models$. It is standard that the semantic clauses above validate all of classical logic, as long as all atomic propositions satisfy density [3,9]. The reader may enjoy verifying $((P \to \texttt{false}) \to \texttt{false}) \to P$.

Thus, because $\Vdash$ and $\models$ agree on total states, the total state semantics can be viewed as a specialization of a well known semantics. Beyond the justification it provides for the total semantics, the partial state perspective reveals further territory worth exploring. For example, there is a wider collection of properties that are garbage insensitive but neither necessarily dense nor total (such properties can be expressed using an $\exists$-free fragment of intuitionistic logic). We started with the total states model in this paper because the conceptual justification for the use of partial states in a garbage collected language is less than immediate. We just stress that totality is not theoretically necessary for garbage insensitivity (and we defer further discussion of this point to a future occasion).

10 Conclusion

As far as we are aware, no previous semantics of pointer assertions is garbage insensitive, including all of the pointer logic references in [4,2,7]. We take a few representative examples. In an early work Oppen and Cook described a complete proof system for deriving true Hoare triples about pointer programs [10], but their interpretation of quantifiers is the usual first-order interpretation; as a result their propositions are not garbage insensitive, and the Hoare triple context $\{true\} - \{\exists y.\, y.1 = 3\}$ behaves exactly as described in Section 1.1. In a series of papers, the most recent of which is [5], de Boer has advocated an approach where the quantifiers are restricted to range over currently active cells only. However, the currently active cells can contain garbage, and because of this de Boer's approach falls foul of the conundrum, and fails to characterize observational equivalence, using essentially the same examples we used in the Section 1.1. Finally, Honsell, Smith, Mason and Talcott [6] have given a characterization of equivalence (the "ciu theorem") in an expressive language with higher-order store. However, after giving this characterization a notion of "contextual assertion" is introduced, and an example is given showing a logical context that breaks observational equivalence. The sticking point in all of these approaches is the interpretation of quantifiers. If one considers a quantifier free language then garbage insensitivity is easy to achieve, at the cost of some expressivity. The decidable logic of [1] very nearly qualifies, except for the use of a garbage sensitive atomic predicate hs for describing sharing constraints.

We have given one resolution of the conundrum involving program logic and garbage collection, but our answer is not the only possible reaction. We conclude by discussing several of the others.

One reaction is to lay the blame not on the interpretation of propositions, but on the Floyd-Hoare approach to assignment. This reaction is difficult to uphold. For, although the soundness of the assignment axioms requires certain assumptions (such as no clashes between named variables), none of these are the essential problem here.

Another approach that is often suggested is to restrict quantifiers so that they range over reachable elements only. Although tantalizing, this notion has several obstacles to overcome. It invalidates Hoare's assignment axiom, as we would expect $\{\exists y.\, y.1 = 3\}\, x := z\, \{\exists y.\, y.1 = 3\}$ to fail if $\exists$ ranges over reachable elements only, since $x := z$ can detach a cell whose first component is 3. Just as significantly, it invalidates the rule of Weakening of contexts in first-order logic. This rule says that if P holds in a context with a collection X of variables, then we know that P holds for all bigger collections. These points do not erect a comprehensive roadblock to the "reachable elements" approach, but they do indicate that such a suggestion requires careful analysis and development.

There is a final reaction to the logical conundrum, and that is not to worry about a higher level of abstraction, on the basis we naturally do consider lower level aspects of program execution, even in garbage collected languages. Although this stance has some intuitive merit, it still has to deal with the logical

problem. That is, if we admit that a garbage collector *might* intervene, and we adopt a concrete semantics of $\exists$, then we will have to deny

$$\{true\}\, x := \mathtt{cons}(3,4);\ x := z\, \{\exists y.\, y.1 = 3\}.$$

The question then would be what alterations to program logic axioms (for assignment and memory allocation) could give a useful logic, while maintaining this denial.

References

1. M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *ESOP '99: European Symposium on Programming*, pages 2–19. Lecture Notes in Computer Science, Vol. 1576, S.D. Swierstra (ed.), Springer-Verlag, New York, NY, 1999.
2. C. Calcagno, S. Ishtiaq, and P.W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In *ACM-SIGPLAN 2nd International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*. ACM Press, September 2000.
3. P. Cohen. *Set Theory and the Continuum Hypothesis*. Benjamin, San Fancisco, 1966.
4. P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 843–993. Elsevier, Amsterdam, and The MIT Press, Cambridge, Mass., 1990.
5. F. de Boer. A WP calculus for OO. In *Proceedings of FOSSACS'99*, 1999.
6. F. Honsell, I. A. Mason, S. Smith, and C. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, may 1995.
7. S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. To appear in POPL'01, 2001.
8. R. Jones and R. Lins. *Garbage Collection*. Wiley, 1996.
9. S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic*. Springer-Verlag, 1992.
10. D. C. Oppen and S. A. Cook. Proving assertions about programs that manipulate data structures. In *Conference Record of Seventh Annual ACM Symposium on Theory of Computation*, pages 107–116, Albuquerque, New Mexico, 5–7 May 1975.
11. J.C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*, Palgrave, 2000.

The Complexity of Model Checking Mobile Ambients

Witold Charatonik[1,3], Silvano Dal Zilio[2], Andrew D. Gordon[2], Supratik Mukhopadhyay[1], and Jean-Marc Talbot[1,4]

[1] Max-Planck-Institut für Informatik, Germany.
[2] Microsoft Research, United Kingdom.
[3] University of Wroc law, Poland.
[4] Laboratoire d'Informatique Fondamentale de Lille, France.

Abstract. We settle the complexity bounds of the model checking problem for the replication-free ambient calculus with public names against the ambient logic without parallel adjunct. We show that the problem is PSPACE-complete. For the complexity upper-bound, we devise a new representation of processes that remains of polynomial size during process execution; this allows us to keep the model checking procedure in polynomial space. Moreover, we prove PSPACE-hardness of the problem for several quite simple fragments of the calculus and the logic; this suggests that there are no interesting fragments with polynomial-time model checking algorithms.

1 Introduction

The ambient calculus [1,2,3] is a formalism for describing the mobility of both software and hardware. An ambient is a named cluster of running processes and nested sub-ambients. Each computation state has a spatial structure, the tree induced by the nesting of ambients. Mobility is abstractly represented by re-arrangement of this tree: an ambient may move inside or outside other ambients.

The ambient logic [4] is a modal logic designed to specify properties of distributed and mobile computations programmed in the ambient calculus. As well as standard temporal modalities for describing the evolution of ambient processes, the logic includes novel spatial modalities for describing the tree structure of ambient processes. Serendipitously, these spatial modalities can also usefully describe the tree structure of semistructured databases [5]. Other work on the ambient logic includes a study of the process equivalence induced by the satisfaction relation [9] and a study of the logic extended with constructs for describing private names [6].

The *model checking* problem is to decide whether a given object (in our case, an ambient process) satisfies (that is, is a model of) a given formula. Cardelli and Gordon [4] show decidability of the model checking problem for a finite-state fragment of the ambient calculus against the fragment of the ambient logic without their parallel adjunct modality. This finite-state ambient calculus omits the constructs for unbounded replication and dynamic name generation of the full calculus. Cardelli and Gordon give no complexity analysis for their algorithm. Still, given the various possible applications of the logic, it

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 152–167, 2001.

is of interest to analyse the complexity of model checking mobile ambients. In fact, a naive analysis of the algorithm of Cardelli and Gordon gives only a doubly exponential bound on its use of time and space. A more sophisticated analysis based on results in this paper shows that their algorithm works in single-exponential time on single-exponential space.

In this paper we settle the complexity bounds of the model checking problem for the finite-state ambient calculus (that is, the full calculus apart from replication and name generation) against the logic without parallel adjunct. Our main result (embodied in Theorems 3.1 and 4.1) is that the problem is PSPACE-complete. Hence, this situates model checking the ambient logic in the same complexity class as model checking concurrent programs against CTL and CTL* [8].

As we discuss in Sect. 2, there are two reasons that the algorithm in [4] uses exponential space. One of them is that a process may grow exponentially during its execution; the other is that there may be exponentially many processes reachable from a given one. In Sect. 3, we present a new model checking algorithm that avoids these problems as follows. We avoid the first problem by devising a new representation of processes using a form of closure. The main feature of this representation is that substitutions that occur when communications take place within an ambient are not applied directly, but are kept explicit. These explicit substitutions prevent the representation blowing up exponentially in the size of the original process. The idea of using closures comes from DAG representations used in unification for avoiding exponential blow-up. A sequential substitution that we use here can be seen as a DAG representation of the substitution. To avoid the second problem, we first devise a non-deterministic algorithm for testing reachability that does not have to store all the reachable processes, but instead tests it on-the-fly, and then remove nondeterminism using Savitch's theorem [10]. Hence we prove Theorem 3.1, that the model checking problem is solvable in PSPACE.

We show this upper bound to be tight in Sect. 4; Theorem 4.1 asserts that the model checking problem is PSPACE-hard. Actually, we give PSPACE-hardness results for various fragments of the logic and of the calculus. For instance, by Theorem 4.2, even for a calculus of purely mobile ambients (that is, a calculus without communication or the capability to dissolve ambients) and the logic without quantifiers, the problem is PSPACE-hard. Moreover, by Theorem 4.3, for a calculus of purely communicative ambients (that is, a calculus without the capabilities to move or to dissolve ambients) and the logic without quantifiers, the problem is also PSPACE-hard. Often in the study of model checking fixing the model or the formula makes the problem easier. Here this is not the case. Even if we fix the process to be the constant $\mathbf{0}$, the model checking problem remains PSPACE-hard. Although we do not prove PSPACE-hardness for fixed arbitrary formulas, our result is not much weaker: Theorem 4.4 asserts that for any level of the polynomial-time hierarchy we can find a fixed formula such that the model checking problem is hard for that level.

A more complete presentation of the calculus and the logics, together with examples and omitted proofs, may be found in an extended version of this paper [7].

2 Review of the Ambient Calculus and Logic

We present a finite-state ambient calculus (that is, the full calculus [1] apart from replication and name generation) and the ambient logic without parallel adjunct. This is the same calculus and logic for which Cardelli and Gordon present a model checking algorithm [4].

2.1 The Ambient Calculus with Public Names

The following table describes the expressions and processes of our calculus.

Expressions and Processes:

$M, N ::=$	expressions	$P, Q, R ::=$	processes
n	name	$\mathbf{0}$	inactivity
$in\ M$	can enter M	$P \mid Q$	composition
$out\ M$	can exit M	$M[P]$	ambient
$open\ M$	can open M	$M.P$	action
ϵ	null	$(n).P$	input
$M.M'$	path	$\langle M \rangle$	output

A name n is said to be *bound* in a process P if it occurs within an input prefix (n). A name is said to be *free* in a process P if there is an occurrence of n outside the scope of any input (n). We write $bn(P)$ and $fn(P)$ for respectively the set of bound names and the set of free names in P. We say two processes are α-equivalent if they are identical apart from the choice of bound names. We write $M\{n \leftarrow N\}$ and $P\{n \leftarrow N\}$ for the outcomes of capture-avoiding substitutions of the expression N for the name n in the expression M and the process P, respectively.

The semantics of the calculus is given by the relations $P \equiv Q$ and $P \rightarrow Q$. The *reduction* relation, $P \rightarrow Q$, defines the evolution of processes over time. The *structural congruence* relation, $P \equiv Q$, is an auxiliary relation used in the definition of reduction. When we define the satisfaction relation of the modal logic in the next section, we use an auxiliary relation, the *sublocation* relation, $P \downarrow Q$, which holds when Q is the whole interior of a top-level ambient in P. We write $\rightarrow^*$ and $\downarrow^*$ for the reflexive and transitive closure of $\rightarrow$ and $\downarrow$, respectively.

Structural Congruence $P \equiv Q$:

P, Q are α-equivalent $\Rightarrow P \equiv Q$ (Str Refl)
$Q \equiv P \Rightarrow P \equiv Q$ (Str Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$ (Str Trans)

$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$ (Str Par)
$P \equiv Q \Rightarrow M[P] \equiv M[Q]$ (Str Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$ (Str Action)
$P \equiv Q \Rightarrow (n).P \equiv (n).Q$ (Str Input)

$P \mid Q \equiv Q \mid P$ (Str Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$ (Str Par Assoc)
$P \mid \mathbf{0} \equiv P$ (Str Zero Par)
$\epsilon.P \equiv P$ (Str ϵ)
$(M.M').P \equiv M.M'.P$ (Str .)

Reduction $P \to Q$ and Sublocation $P \downarrow Q$:

$n[in\ m.P \mid Q] \mid m[R] \to m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out\ m.P \mid Q] \mid R] \to n[P \mid Q] \mid m[R]$	(Red Out)
$open\ n.P \mid n[Q] \to P \mid Q$	(Red Open)
$\langle M \rangle \mid (n).P \to P\{n \leftarrow M\}$	(Red I/O)
$P \to Q \Rightarrow P \mid R \to Q \mid R$	(Red Par)
$P \to Q \Rightarrow n[P] \to n[Q]$	(Red Amb)
$P' \equiv P, P \to Q, Q \equiv Q' \Rightarrow P' \to Q'$	(Red $\equiv$)
$P \equiv n[Q] \mid P' \Rightarrow P \downarrow Q$	(Loc)

The following example shows that the size of reachable processes may be exponential, and that there may be a reduction path of exponential length. The algorithm given in [4] may use exponential space to check properties of this example.
Consider the family of processes $(P_k)_{k \geq 0}$, recursively defined by the equations $P_0 = (n).(p[n] \mid q[\mathbf{0}])$ and $P_{k+1} = (n_{k+1}).(\langle n_{k+1}.n_{k+1} \rangle \mid P_k)$. Intuitively, the process P_{k+1} inputs a capability, calls it n_{k+1}, doubles it, and outputs the result to the process P_k. We have the following, where $M^1 = M$ and $M^{k+1} = M.M^k$.

$$\begin{array}{ll} \langle in\ q.out\ q \rangle \mid P_0 \to^1 & p[in\ q.out\ q] \mid q[\mathbf{0}] \\ \langle in\ q.out\ q \rangle \mid P_1 \to^2 & p[(in\ q.out\ q)^2] \mid q[\mathbf{0}] \\ \langle in\ q.out\ q \rangle \mid P_k \to^{k+1} & p[(in\ q.out\ q)^{2^k}] \mid q[\mathbf{0}] \end{array}$$

Since $(in\ q.out\ q)^{2^k}$ is a sequence of 2^k copies of $in\ q.out\ q$, the process $\langle in\ q.out\ q \rangle \mid P_k$ reduces in $(k+1) + 2^{k+1}$ steps to $p[\mathbf{0}] \mid q[\mathbf{0}]$.
This example points out two facts. First, using a simple representation of processes (such as the one proposed in [4]), it may be that the size of a process considered during model checking grows exponentially bigger than the size of the initial process. Second, during the model checking procedure, there may be an exponential number of reachable processes to consider. Therefore, a direct implementation of the algorithm proposed in [4] may use space exponential in the size of the input process. In this paper, using a new representation of processes, we show that the model checking problem $P \models \mathcal{A}$ can be solved using only polynomial space in the sizes of P and $\mathcal{A}$, in spite of this example.

2.2 The Logic (for Public Names)

We describe the formulas and satisfaction relation of the logic.

Logical Formulas:

η	a name n or a variable x		
$\mathcal{A}, \mathcal{B} ::=$	formula		
$\mathbf{T}$	true	$\neg\mathcal{A}$	negation
$\mathcal{A} \vee \mathcal{B}$	disjunction	$\exists x.\mathcal{A}$	existential quantification
$\mathbf{0}$	void	$\mathcal{A} \mid \mathcal{B}$	composition match
$\eta[\mathcal{A}]$	ambient match	$\mathcal{A}@\eta$	location adjunct
$\Diamond\mathcal{A}$	somewhere modality	$\Diamond\mathcal{A}$	sometime modality

We assume that names and variables belong to two disjoint vocabularies. We write $\mathcal{A}\{x{\leftarrow}m\}$ for the outcome of substituting each free occurrence of the variable x in the formula $\mathcal{A}$ with the name m. We say a formula $\mathcal{A}$ is closed if and only if it has no free variables (though it may contain free names).

Intuitively, we interpret closed formulas as follows. The formulas $\mathbf{T}$, $\neg\mathcal{A}$, and $\mathcal{A} \vee \mathcal{B}$ embed propositional logic. The formulas $\mathbf{0}$, $\eta[\mathcal{A}]$, and $\mathcal{A} \mid \mathcal{B}$ are spatial modalities. A process satisfies $\mathbf{0}$ if it is structurally congruent to the empty process. It satisfies $n[\mathcal{A}]$ if it is structurally congruent to an ambient $n[P]$ where P satisfies $\mathcal{A}$. A process P satisfies $\mathcal{A} \mid \mathcal{B}$ if it can be decomposed into two subprocesses, $P \equiv Q \mid R$, where Q satisfies $\mathcal{A}$, and R satisfies $\mathcal{B}$. The formula $\exists x.\mathcal{A}$ is an existential quantification over names. The formulas $\Diamond\mathcal{A}$ (sometime) and $\diamond\!\!\!\!-\mathcal{A}$ (somewhere) quantify over time and space, respectively. A process satisfies $\Diamond\mathcal{A}$ if it has a temporal successor, that is, a process into which it evolves, that satisfies $\mathcal{A}$. A process satisfies $\diamond\!\!\!\!-\mathcal{A}$ if it has a spatial successor, that is, a sublocation, that satisfies $\mathcal{A}$. Finally, a process P satisfies the formula $\mathcal{A}@n$ if the ambient $n[P]$ satisfies $\mathcal{A}$.

The satisfaction relation $P \models \mathcal{A}$ provides the semantics of our logic.

Satisfaction $P \models \mathcal{A}$ (for $\mathcal{A}$ Closed):

$$P \models \mathbf{T} \qquad\qquad P \models \neg\mathcal{A} \triangleq \neg(P \models \mathcal{A})$$
$$P \models \mathcal{A} \vee \mathcal{B} \triangleq P \models \mathcal{A} \vee P \models \mathcal{B} \qquad\qquad P \models \mathbf{0} \triangleq P \equiv \mathbf{0}$$

$$P \models n[\mathcal{A}] \triangleq \exists P'.P \models n[P'] \wedge P' \models \mathcal{A}$$
$$P \models \mathcal{A} \mid \mathcal{B} \triangleq \exists P', P''.P \equiv P' \mid P'' \wedge P' \models \mathcal{A} \wedge P'' \models \mathcal{B}$$

$$P \models \exists x.\mathcal{A} \triangleq \exists m.P \models \mathcal{A}\{x{\leftarrow}m\} \qquad\qquad P \models \Diamond\mathcal{A} \triangleq \exists P'.P \rightarrow^* P' \wedge P' \models \mathcal{A}$$
$$P \models \diamond\!\!\!\!-\mathcal{A} \triangleq \exists P'.P \downarrow^* P' \wedge P' \models \mathcal{A} \qquad\qquad P \models \mathcal{A}@n \triangleq n[P] \models \mathcal{A}$$

We use $\Box\mathcal{A}$ (everytime modality), $\boxminus\mathcal{A}$ (everywhere modality) and $\forall x.\mathcal{A}$ (universal quantification) as abbreviations for $\neg(\Diamond\neg\mathcal{A})$, $\neg(\diamond\!\!\!\!-\neg\mathcal{A})$ and $\neg(\exists x.\neg\mathcal{A})$, respectively.

3 A Model Checking Algorithm

We show that the model checking problem can be decided in polynomial space by devising a new representation of processes (Sect. 3.1) that remains polynomial in the size of the initial process (Sect. 3.2). In Sect. 3.3 we present a new model checking algorithm based on this representation.

Since the reduction relation is defined up to α-equivalence, we may assume for the purposes of computing reachable processes that the free and bound names of every ambient process are distinct, and moreover that the bound names are pairwise distinct.

3.1 A Polynomial-Space Representation

We give in this section a new representation for ambient processes based on *normal closures*. (It is different from the *normal form* of processes introduced in [4].) We also present basic operations on closures and prove that closures indeed simulate the processes they represent.

Annotated Processes, Substitutions, Closures:

$\tilde{P} ::= \prod_{i\in I} \pi_i$	annotated process, multiset of primes, I finite indexing set		
$\pi ::=$	prime		
$M[\tilde{P}]$	ambient	$M(o).\tilde{P}$	action, with offset $o \geq 0$
$(n).\tilde{P}$	input	$\langle M \rangle$	output
$\sigma ::= \{n_1 \leftarrow M_1\} \cdots \{n_k \leftarrow M_k\}$	sequential substitution, $k \geq 0$		
$\langle \tilde{P}; \sigma \rangle$	closure		

In a sequential substitution $\{n_1 \leftarrow M_1\} \cdots \{n_k \leftarrow M_k\}$, the expression M_i bound to n_i lies in the scope of the bindings for the remaining names $n_{i+1}, \ldots, n_k$. We denote by ι the empty sequence of substitutions and treat it as the identity substitution. For an annotated process $\tilde{P}$, we define free and bound names in the same way as for ambient processes. Let $names(\sigma)$ be the set of all names occurring in σ.

We define a partial mapping $\mathcal{U}$ from closures to the set of ambient processes. Intuitively, it unfolds a closure to the process it represents by applying the substitution and cutting off the prefix defined by the offset. Roughly speaking, the expression $\mathcal{U}(\tilde{P}, \sigma)$ is defined if the offsets within the annotated process do not exceed the length of the expression they are associated with. The unfolding $\mathcal{U}(\tilde{P}, \sigma)$ is defined as follows.

The Unfolding $\mathcal{U}(\tilde{P}, \sigma)$ of a Closure $\langle \tilde{P}; \sigma \rangle$:

$$\mathcal{U}(\textstyle\prod_{i\in I} \pi_i, \sigma) = \begin{cases} \mathcal{U}(\pi_1, \sigma) \mid \ldots \mid \mathcal{U}(\pi_n, \sigma) & \text{if } I = \{1, \ldots, n\} \neq \varnothing \\ \mathbf{0} & \text{otherwise} \end{cases}$$

$$\mathcal{U}(M[\tilde{P}], \sigma) = M\sigma[\mathcal{U}(\tilde{P}, \sigma)]$$

$$\mathcal{U}(M(o).\tilde{P}, \sigma) = \begin{cases} N_{o+1}.\cdots.N_l.\mathcal{U}(\tilde{P}, \sigma) & \text{if } M\sigma = N_1.\cdots.N_l, o < l \text{ and } N_i \\ & \text{being either a name or of the form} \\ & cap\ N \text{ with } cap \in \{in, out, open\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{U}((n).\tilde{P}, \sigma) = (n).\mathcal{U}(\tilde{P}, \sigma) \qquad \mathcal{U}(\langle M \rangle, \sigma) = \langle M\sigma \rangle$$

We are only interested in a particular kind of closure, which we refer to as *normal*. Let a closure $\langle \tilde{P}; \sigma \rangle$ be normal if each of the following conditions hold: (1) $\mathcal{U}(\tilde{P}, \sigma)$ is defined; (2) the bound names of $\tilde{P}$ are pairwise distinct and such that $bn(\tilde{P}) \cap (fn(\tilde{P}) \cup names(\sigma)) = \varnothing$; (3) every offset occurring in the scope of an input in $\tilde{P}$ is zero.

The next proposition says that our representation of ambient processes with normal closures preserves their basic properties. We write $\{\}$ and $+\!+$ for the empty multiset and the multiset union operation, respectively.

Proposition 3.1 (Structural Congruences). *Let $\tilde{P} = \prod_{i\in I} \pi_i$ and $\langle \tilde{P}; \sigma \rangle$ be a normal closure. Then*

(1) $\mathcal{U}(\tilde{P}, \sigma) \equiv \mathbf{0}$ *iff* $I = \varnothing$.
(2) $\mathcal{U}(\tilde{P}, \sigma) \equiv M[Q]$ *iff* $\exists M', \tilde{Q} :$ I *is a singleton* $\{i\}$, $\pi_i = M'[\tilde{Q}]$, $M'\sigma = M$, *and* $\mathcal{U}(\tilde{Q}, \sigma) \equiv Q$.

(3) $\mathcal{U}(\tilde{P},\sigma) \equiv P' \mid P''$ *iff* $\exists J,K : J \cup K = I,\ J \cap K = \varnothing,\ P' \equiv \mathcal{U}(\prod_{j\in J}\pi_j,\sigma)$, *and* $P'' \equiv \mathcal{U}(\prod_{k\in K}\pi_k,\sigma)$.
(4) $\mathcal{U}(\tilde{P},\sigma) \equiv \langle M\rangle$ *iff* $\exists M' :$ I *is a singleton* $\{i\}$, $\pi_i = \langle M'\rangle$ *and* $M'\sigma = M$.
(5) $\mathcal{U}(\tilde{P},\sigma) \equiv (n).Q$ *iff* $\exists \tilde{Q} :$ I *is a singleton* $\{i\}$, $\pi_i = (n).\tilde{Q}$ *and* $\mathcal{U}(\tilde{Q},\sigma) \equiv Q$.

Next, we present how the reduction and sublocation transitions can be defined on closures. Due to this particular representation and the fact that some part of the ambient process is contained in the sequential substitution, some auxiliary subroutines are needed. One can see in the definition of $\mathcal{U}$ that only expressions M in the annotated process are affected by the sequential substitution. For the sublocation transition, it is important to extract the name represented by the expression M under the substitution σ. So, one of those subroutines, $nam(M,\sigma)$, consists in recovering from an expression M the name it effectively represents within the substitution σ and is inductively defined over sequential substitution as follows: for the empty substitution, $nam(n,\iota) = n$ and for an non-empty one,

$$nam(n,\{m{\leftarrow}M\}\sigma) = \begin{cases} nam(M,\sigma) \text{ if } n = m \\ nam(n,\sigma) \quad \text{otherwise} \end{cases}$$

The reduction transition for a closure $\langle \tilde{P};\sigma\rangle$ requires some other auxiliary subroutines. Intuitively, the outcome of applying the substitution σ to an expression M contained within $\tilde{P}$ is a finite sequence of either capabilities of the form $in\,M'$, $out\,M'$, $open\,M'$, or names not bound by the substitution. We need a subroutine to compute the length of the sequence, and to do so in polynomial space, even though the length may be exponential in the size of the closure. Therefore, the subroutine $len(M,\sigma)$ cannot explicitly apply σ to M. It is defined by the equations:

$$\begin{aligned} &len(\epsilon,\sigma) = 0 \\ &len(cap\ N,\sigma) = 1 \text{ if } cap \in \{in, out, open\} \\ &len(n,\iota) = 1 \\ &len(n,\{m{\leftarrow}N\}\sigma) = \begin{cases} len(N,\sigma) \text{ if } n = m \\ len(n,\sigma) \quad \text{otherwise} \end{cases} \\ &len(M.N,\sigma) = len(M,\sigma) + len(N,\sigma) \end{aligned}$$

Now, from the definition of the reduction on ambient processes, one can see that the reduction consumes one capability: once the reduction is done, the involved capability disappears from the resulting process. This is slightly different for the representation we have proposed: a sequence of capabilities can be partially contained in a sequential substitution σ. This substitution remains fixed during the execution of capabilities and the offset attached to this sequence plays the role of a program counter. Therefore, to perform a reduction step one has to extract the first capability to execute from a sequence of capabilities, M, a substitution, σ, and an offset, o. This is computed by $fst(M,o,\sigma)$ defined by:

$$\begin{aligned} &fst(M.N,o,\sigma) = \begin{cases} fst(M,o,\sigma) & \text{if } len(M,\sigma) > o \\ fst(N,o-len(M,\sigma),\sigma) & \text{otherwise} \end{cases} \\ &fst(cap\ N,0,\sigma) = cap\ (nam(N,\sigma)) \text{ for } cap \in \{in, out, open\} \\ &fst(n,o,\{m{\leftarrow}M\}\sigma) = \begin{cases} fst(M,o,\sigma) \text{ if } n = m \\ fst(n,o,\sigma) \quad \text{otherwise} \end{cases} \end{aligned}$$

The next subroutine introduced here, $split(M(o).\tilde{P}, \sigma)$, computes a pair from a prime, $M(o).\tilde{P}$, and a sequential substitution, σ. The first component of this result is the first capability of $\langle\{M(o).\tilde{P}\};\sigma\rangle$ (the one in head position). The second component is the remaining annotated process once this first capability has been executed.

$$split(M(o).\tilde{P}, \sigma) = \begin{cases} (fst(M,o,\sigma), \{M(o+1).\tilde{P}\}) & \text{if } len(M,\sigma) > o+1 \\ (fst(M,o,\sigma), \tilde{P}) & \text{otherwise} \end{cases}$$

Suppose $\langle\tilde{P};\sigma\rangle$ is a normal closure containing an action $M(o).\tilde{Q}$. From the definition of a normal closure, $len(M,\sigma) > o$, and if the action occurs under an input variable n, then the offset $o = 0$. If n occurs in M and gets bound to ϵ by an I/O step, it may be that $len(M, \{n{\leftarrow}\epsilon\}\sigma) = 0$. So, in the transition rule for I/O, we need to re-normalise the closure representing the outcome of the transition. We do so using the following subroutines $norm(\tilde{P}, \sigma)$ and $norm(\pi, \sigma)$ that return the annotated process obtained by removing from $\tilde{P}$ and π, respectively, any prefix $M(o)$ such that $len(M,\sigma) = 0$.

$$norm(\textstyle\prod_{i\in 1..k} \pi_i, \sigma) = \begin{cases} \{\} & \text{if } k = 0 \\ norm(\pi_1,\sigma) +\!\!+ \cdots +\!\!+ norm(\pi_k,\sigma) & \text{otherwise} \end{cases}$$

$$norm(M[\tilde{P}], \sigma) = \{M[norm(\tilde{P},\sigma)]\}$$

$$norm(M(o).\tilde{P}, \sigma) = \begin{cases} \{M(o).norm(\tilde{P},\sigma)\} & \text{if } len(M,\sigma) = 0 \\ norm(\tilde{P},\sigma) & \text{otherwise} \end{cases}$$

$$norm((n).\tilde{P}, \sigma) = \{(n).norm(\tilde{P},\sigma)\} \qquad norm(\langle M\rangle, \sigma) = \{\langle M\rangle\}$$

Notice that $nam(M,\sigma)$ is undefined if M is of the form ϵ, $N.N'$, $in\ N$, $out\ N$, or $open\ N$. Therefore, the expression $nam(M,\sigma)$ is either undefined or is evaluated to a name. Moreover, we can compute the name returned by $nam(M,\sigma)$, or whether it is undefined, in linear time. The number returned by $len(M,\sigma)$ can be computed in polynomial space. We can compute the capability returned by $fst(M,o,\sigma)$ and the pair returned by $split(M(o).\tilde{P},\sigma)$, or whether they are undefined, in polynomial space. We can compute the outcome of $norm(\tilde{P},\sigma)$ and $norm(\pi,\sigma)$ in polynomial space.
Next, we define transition and the sublocation relations on closures.

Transitions $\langle\tilde{P};\sigma\rangle \to \langle\tilde{P}';\sigma'\rangle$ and Sublocations $\langle\tilde{P};\sigma\rangle \downarrow \langle\tilde{P}';\sigma'\rangle$ of Closures:

(Trans In)

$$\frac{split(\pi,\sigma) = (in\ m, \tilde{P}) \quad nam(M,\sigma) = m \quad nam(N,\sigma) = n}{\langle\{N[\{\pi\} +\!\!+ \tilde{Q}], M[\tilde{R}]\};\sigma\rangle \to \langle\{M[\{N[\tilde{P} +\!\!+ \tilde{Q}]\} +\!\!+ \tilde{R}]\};\sigma\rangle}$$

(Trans Out)

$$\frac{split(\pi,\sigma) = (out\ m, \tilde{P}) \quad nam(M,\sigma) = m \quad nam(N,\sigma) = n}{\langle\{M[\{N[\{\pi\} +\!\!+ \tilde{Q}]\} +\!\!+ \tilde{R}]\};\sigma\rangle \to \langle\{N[\tilde{P} +\!\!+ \tilde{Q}], M[\tilde{R}]\};\sigma\rangle}$$

(Trans Open)

$$\frac{split(\pi,\sigma) = (open\ n, \tilde{P}) \quad nam(M,\sigma) = n}{\langle\pi, \{M[\tilde{Q}]\};\sigma\rangle \to \langle\tilde{P} +\!\!+ \tilde{Q};\sigma\rangle}$$

(Trans I/O)

$$\frac{\tilde{P}' = norm(\tilde{P}, \{n{\leftarrow}M\}\sigma)}{\langle\{(n).\tilde{P}, \langle M\rangle\};\sigma\rangle \to \langle\tilde{P}';\{n{\leftarrow}M\}\sigma\rangle}$$

(Trans Par)

$$\frac{\langle\tilde{P};\sigma\rangle \to \langle\tilde{P}';\sigma'\rangle}{\langle\tilde{P} +\!\!+ \tilde{Q};\sigma\rangle \to \langle\tilde{P}' +\!\!+ \tilde{Q};\sigma'\rangle}$$

$$\text{(Trans Amb)}\quad \frac{\langle \tilde{P};\sigma\rangle \to \langle \tilde{P}';\sigma'\rangle \quad nam(M,\sigma)=n}{\langle \{M[\tilde{P}]\};\sigma\rangle \to \langle \{M[\tilde{P}']\};\sigma'\rangle} \qquad \text{(Loc)}\quad \frac{nam(M,\sigma)=m}{\langle \tilde{Q} +\!\!+ \{M[\tilde{P}]\};\sigma\rangle \downarrow \langle \tilde{P};\sigma\rangle}$$

The condition for (Loc) ensures simply that the expression M together with σ is a name. For two normal closures $\langle P;\sigma\rangle$, $\langle P';\sigma'\rangle$, deciding whether $\langle P;\sigma\rangle \downarrow \langle P';\sigma'\rangle$ can be achieved in polynomial space. There is no rule corresponding to (Red $\equiv$) since we always keep closures in normal form. The two rules (Trans Par) and (Trans Amb) correspond to the congruence rules (Red Par) and (Red Amb) for reduction.
In the same way as for ambient processes, we define the relations $\to^*$ and $\downarrow^*$ (on closures) as the reflexive and transitive closures of $\to$ and $\downarrow$, respectively.

Proposition 3.2. *If $\langle \tilde{P};\sigma\rangle$ is normal and $\langle \tilde{P};\sigma\rangle \downarrow^* \langle \tilde{P}';\sigma\rangle$ then $\langle \tilde{P}';\sigma\rangle$ is normal. If $\langle \tilde{P};\sigma\rangle$ is normal and $\langle \tilde{P};\sigma\rangle \to^* \langle \tilde{P}';\sigma'\rangle$ then $\langle \tilde{P}';\sigma'\rangle$ is normal.*

The next propositions says that the representation of processes as closures preserves sublocations and reductions.

Proposition 3.3 (Sublocation Equivalences). *Assume $\langle \tilde{P};\sigma\rangle$ is a normal closure. If $\langle \tilde{P};\sigma\rangle \downarrow \langle \tilde{Q};\sigma\rangle$ then $\mathcal{U}(\tilde{P},\sigma) \downarrow \mathcal{U}(\tilde{Q},\sigma)$. If $\mathcal{U}(\tilde{P},\sigma) \downarrow Q$ then there exists $\tilde{Q}$ such that $\langle \tilde{P};\sigma\rangle \downarrow \langle \tilde{Q};\sigma\rangle$ and $\mathcal{U}(\tilde{Q},\sigma) \equiv Q$.*

Proposition 3.4 (Reduction Equivalences). *Assume $\langle \tilde{P};\sigma\rangle$ is a normal closure. If $\langle \tilde{P};\sigma\rangle \to \langle \tilde{P}';\sigma'\rangle$ then $\mathcal{U}(\tilde{P},\sigma) \to \mathcal{U}(\tilde{P}',\sigma')$. If $\mathcal{U}(\tilde{P},\sigma) \to P'$ then there exists $\langle \tilde{P}';\sigma'\rangle$ such that $\langle \tilde{P};\sigma\rangle \to \langle \tilde{P}';\sigma'\rangle$ and $\mathcal{U}(\tilde{P}',\sigma') \equiv P'$.*

Propositions 3.1–3.4 are enough to prove that normal closures indeed simulate the processes they represent.

3.2 Size of the Representation

We show that closures indeed give a polynomial representation of processes. To do this, we have to bound the size of offsets that occur in closures.
For a given object (a closure or a process) O, by $|O|$ we mean the length of its string representation and by $\|O\|$ the number of nodes in its tree representation. We assume that an offset is represented by a single node in the tree representation.

Lemma 3.1. *Suppose that $\langle \tilde{P};\sigma\rangle \to \langle \tilde{P}';\sigma'\rangle$. Then $\|\langle \tilde{P}';\sigma'\rangle\| \le \|\langle \tilde{P};\sigma\rangle\|$.*

Proof. A simple case analysis on the derivation of $\langle \tilde{P};\sigma\rangle \to \langle \tilde{P}';\sigma'\rangle$. □

Proposition 3.5. *Assume $\langle \tilde{P};\sigma\rangle$ is normal and $\langle \tilde{P};\sigma\rangle \to \langle \tilde{P}';\sigma'\rangle$. Then all offsets used in $\tilde{P}$ and $\tilde{P}'$ can be represented by the same number of bits, polynomial in $|\langle \tilde{P};\sigma\rangle|$ and, with such a representation, $|\langle \tilde{P}';\sigma'\rangle| \le |\langle \tilde{P};\sigma\rangle|$.*

Proof. A simple induction on the length of the substitution σ' proves that the offsets in $\tilde{P}'$ are bounded by the value $\|\langle \tilde{P}'; \sigma' \rangle\|^{\|\langle \tilde{P}'; \sigma' \rangle\|}$. By Lemma 3.1, they are also bounded by $\|\langle \tilde{P}; \sigma \rangle\|^{\|\langle \tilde{P}; \sigma \rangle\|}$ and then all offsets used in $\tilde{P}$ and $\tilde{P}'$ are bounded by this value, which can be represented on $\|\langle \tilde{P}; \sigma \rangle\| \cdot (\lfloor \log(\|\langle \tilde{P}; \sigma \rangle\|) \rfloor + 1)$ bits. With this representation of offsets, incrementing an offset does not increase the size of its string representation. Thus no transitions can increase the length of the string representations of closures. □

The following proposition is a key fact in the proof that our model checking algorithm and also the algorithm in [4] terminate in exponential time. It implies that the computation tree of a given process might be very deep and very narrow (as in our example in Sect. 2) or not so deep and wider; in any case the number of nodes in the tree remains exponentially bounded. A naive argument (without using closures) gives only a doubly exponential bound on the number of reachable processes: one can prove that the computation tree of a given process is at most exponentially deep (as our example in Sect. 2 shows, this bound is tight) and that the number of successors for every node is at most polynomial. For example, the closure $\langle \{n[in\ n(0).\tilde{P}_0], \ldots, n[in\ n(0).\tilde{P}_k]\}; \sigma \rangle$ has at most k^2 different successors. These two facts do not give, however, the exponential bound on the number of nodes in the tree, which is given by the following proposition.

Proposition 3.6. *Let $\langle \tilde{P}; \sigma \rangle$ be a normal closure. Then there exist at most exponentially many $\langle \tilde{P}'; \sigma' \rangle$ such that $\langle \tilde{P}; \sigma \rangle \rightarrow^* \langle \tilde{P}'; \sigma' \rangle$.*

Proof. This is a direct consequence of Proposition 3.5 and the observation that there are only exponentially many strings of polynomial length. □

Proposition 3.7. *The reachability problem for normal closures is decidable in PSPACE.*

Proof. Take any instance $\langle \tilde{P}; \sigma \rangle$, $\langle \tilde{P}'; \sigma' \rangle$ of the reachability problem. To decide whether $\langle \tilde{P}; \sigma \rangle \rightarrow^* \langle \tilde{P}'; \sigma' \rangle$, we first define a nondeterministic algorithm that starting from $\langle \tilde{P}; \sigma \rangle$ guesses an immediate successor of the current closure until it reaches $\langle \tilde{P}'; \sigma' \rangle$ or there are no further successors. By Proposition 3.5 the algorithm requires only polynomial space (we have to store only the current closure and its one immediate successor); Proposition 3.6 implies termination. Finally, using the general statement of Savitch's theorem [10] (NPSPACE$(S(n)) \subseteq$ PSPACE$(S(n)^2)$), this non-deterministic algorithm can be turned into a deterministic one. □

3.3 A New Algorithm

We propose a new algorithm, $Check(\tilde{P}, \sigma, \mathcal{A})$, to check whether the ambient process simulated by $\langle \tilde{P}; \sigma \rangle$ satisfies the closed formula $\mathcal{A}$. For each ambient process, P, we only consider the closure, $\mathcal{F}(P)$, obtained using the *folding* function defined as follows. We prove (Proposition 3.9), that $P \models \mathcal{A}$ if and only if $Check(\mathcal{F}(P), \iota, \mathcal{A})$ returns the Boolean value **T**.

The Folding $\mathcal{F}(P)$ of a Process P:

$$\mathcal{F}(\mathbf{0}) = \{\} \qquad \mathcal{F}(P \mid Q) = \mathcal{F}(P) +\!\!+ \mathcal{F}(Q)$$
$$\mathcal{F}(M[P]) = \{M[\mathcal{F}(P)]\} \qquad \mathcal{F}((n).P) = \{(n).\mathcal{F}(P)\}$$
$$\mathcal{F}(\langle M\rangle) = \{\langle M\rangle\}$$
$$\mathcal{F}(M.P) = \begin{cases} \mathcal{F}(P) & \text{if } len(M,\iota) = 0 \\ \{M(0).\mathcal{F}(P)\} & \text{otherwise} \end{cases}$$

For any process P, the closure $\langle \mathcal{F}(P); \iota\rangle$ is normal and $\mathcal{U}(\mathcal{F}(P), \iota)$ is structurally congruent to P. Furthermore, $\mathcal{F}(P)$ can be computed in linear time in the size of P.
For the model checking problem, $P \models \mathcal{A}$, we may assume without loss of generality that the free names of $\mathcal{A}$ are disjoint from the bound names of P. We denote by $fn(\tilde{P}, \sigma)$ the set $(fn(\tilde{P}) \cup names(\sigma)) \setminus dom(\sigma)$.

Computing whether a Process Satisfies a Closed Formula:

$$Check(\tilde{P}, \sigma, \mathbf{T}) = \mathbf{T}$$
$$Check(\tilde{P}, \sigma, \neg\mathcal{A}) = \neg Check(\tilde{P}, \sigma, \mathcal{A})$$
$$Check(\tilde{P}, \sigma, \mathcal{A} \vee \mathcal{B}) = Check(\tilde{P}, \sigma, \mathcal{A}) \vee Check(\tilde{P}, \sigma, \mathcal{B})$$
$$Check(\textstyle\prod_{i\in I} \pi_i, \sigma, \mathbf{0}) = \begin{cases} \mathbf{T} \text{ if } I = \varnothing \\ \mathbf{F} \text{ otherwise} \end{cases}$$
$$Check(\textstyle\prod_{i\in I} \pi_i, \sigma, n[\mathcal{A}]) = \begin{cases} Check(\tilde{Q}, \sigma, \mathcal{A}) \text{ if } I = \{i\},\ \pi_i = M[\tilde{Q}], \\ \qquad\qquad\qquad\qquad nam(M, \sigma) = n \\ \mathbf{F} \qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$$
$$Check(\textstyle\prod_{i\in I} \pi_i, \sigma, \mathcal{A} \mid \mathcal{B}) = \bigvee_{J\subseteq I} Check(\prod_{j\in J} \pi_j, \sigma, \mathcal{A}) \wedge Check(\prod_{k\in I-J} \pi_k, \sigma, \mathcal{B})$$
$$Check(\tilde{P}, \sigma, \exists x.\mathcal{A}) = \text{let } \{m_1, \ldots, m_k\} = fn(\tilde{P}, \sigma) \cup fn(\mathcal{A}) \text{ in}$$
$$\qquad\qquad \text{let } m_0 \notin \{m_1, \ldots, m_k\} \cup bn(\tilde{P}) \cup dom(\sigma) \text{ be fresh in}$$
$$\qquad\qquad \textstyle\bigvee_{i\in 0..k} Check(\tilde{P}, \sigma, \mathcal{A}\{x \leftarrow m_i\})$$
$$Check(\tilde{P}, \sigma, \Diamond\mathcal{A}) = \textstyle\bigvee_{\langle \tilde{P};\sigma\rangle \to^* \langle \tilde{P}';\sigma'\rangle} Check(\tilde{P}', \sigma', \mathcal{A})$$
$$Check(\tilde{P}, \sigma, \mathord{\diamond\!\!\!\!-}\mathcal{A}) = \textstyle\bigvee_{\langle \tilde{P};\sigma\rangle \downarrow^* \langle \tilde{P}';\sigma\rangle} Check(\tilde{P}', \sigma, \mathcal{A})$$
$$Check(\tilde{P}, \sigma, \mathcal{A}@n) = Check(n[\tilde{P}], \sigma, \mathcal{A})$$

An expression $Check(\tilde{P}, \sigma, \mathcal{A})$ is said to be *normal* if and only if the closure $\langle \tilde{P}; \sigma\rangle$ is normal, $\mathcal{A}$ is a closed formula, and $fn(\mathcal{A}) \cap (bn(\tilde{P}) \cup dom(\sigma)) = \varnothing$. Hence, for the model checking problem $P \models \mathcal{A}$ where $\mathcal{A}$ is a closed formula, the expression $Check(\mathcal{F}(P), \iota, \mathcal{A})$ is normal and moreover we have:

Proposition 3.8. *The model checking algorithm described above preserves the normality of $Check(\tilde{P}, \sigma, \mathcal{A})$.*

Proposition 3.9. *For all processes P and closed formulas $\mathcal{A}$, we have $P \models \mathcal{A}$ if and only if $Check(\mathcal{F}(P), \iota, \mathcal{A}) = \mathbf{T}$.*

Proof. By induction on the structure of the ambient formula $\mathcal{A}$ with appeal to Propositions 3.3 and 3.4 in the cases $\mathord{\diamond\!\!\!\!-}\mathcal{A}$ and $\Diamond\mathcal{A}$, respectively. □

Theorem 3.1. *Model checking the ambient calculus and logic of this paper is decidable in PSPACE.*

Proof. To test for a given process P and formula $\mathcal{A}$ whether $P \models \mathcal{A}$ we simply compute the value of $Check(\mathcal{F}(P), \iota, \mathcal{A})$. The only problem is to implement $Check$ in such a way that it works in polynomial space.
In the case of $\mathbf{T}, \mathbf{0}, n[\mathcal{A}], \mathcal{A}@n, \neg\mathcal{A}$, the algorithm can directly check whether the respective conditions hold. In the case of $\mathcal{A} \vee \mathcal{B}, \mathcal{A} \mid \mathcal{B}, \exists x.\mathcal{A}, \Diamond\mathcal{A}, \text{✧}\mathcal{A}$, we have to be more careful about the space used to compute the value of disjunctions. In a loop we iteratively compute the value of each disjunct, reusing the same space in every iteration. In the case of $\Diamond\mathcal{A}$ the subroutine computing $\bigvee_{\langle \tilde{P};\sigma\rangle \to^* \langle \tilde{P}';\sigma'\rangle} Check(\tilde{P}', \sigma', \mathcal{A})$ could look as follows.

$result \leftarrow \mathbf{F}$
for all $\langle \tilde{P}';\sigma'\rangle$ such that $\langle \tilde{P};\sigma\rangle \to^* \langle \tilde{P}';\sigma'\rangle$
  if $Check(\tilde{P}', \sigma', \mathcal{A}) = \mathbf{T}$ then $result \leftarrow \mathbf{T}$
return$(result)$

By Propositions 3.5 and 3.7, every iteration requires only polynomial space. The cases of $\mathcal{A} \vee \mathcal{B}, \mathcal{A} \mid \mathcal{B}, \exists x.\mathcal{A}, \text{✧}\mathcal{A}$ are similar. Thus, the space $S(k, |\tilde{P}| + |\sigma|)$ used by the algorithm to compute $Check(\tilde{P}, \sigma, \mathcal{A})$ for formulas $\mathcal{A}$ of depth not exceeding k satisfies the inequality

$$S(k+1, |\tilde{P}| + |\sigma|) \leq S(k, |\tilde{P}| + c + |\sigma|) + p(|\tilde{P}| + |\sigma|)$$

for some constant c and some polynomial p (the constant c comes from the fact that in the case of $\mathcal{A} = \mathcal{B}@n$ the size of $n[\tilde{P}]$ is greater than the size of $\tilde{P}$; the polynomial p estimates the space needed for testing reachability etc). Therefore, $S(k, |\tilde{P}| + |\sigma|) \leq k \cdot p(|\tilde{P}| + k \cdot c + |\sigma|)$.
Finally, the fact that $\mathcal{F}(P)$ is polynomial in the size of P and the statement of Proposition 3.9 complete the proof. □

4 Complexity Lower Bounds

Below we present lower bounds on the space complexity of model checking our process calculus against our modal logic, and also for two significant fragments.
The results given here are based on known results about the complexity of decision problems for Quantified Boolean Formulas (QBF). The alternation depth of a formula is the number of alternations between existential and universal quantifiers in its prenex quantification.
Those known results are: (1) deciding the validity problem for a closed quantified Boolean formula φ is PSPACE-complete; (2) deciding the validity problem for a closed quantified Boolean formula φ of alternation depth k whose outermost quantifier is $\exists$ is Σ_k^P-complete [11], where Σ_k^P denotes the k-th level of the polynomial-time hierarchy. In particular, $\Sigma_0^P = P$ and $\Sigma_1^P = NP$.

4.1 The Full Calculus and Logic

We define an encoding of QBF formulas into ambient formulas. (We can assume without loss of generality that these Boolean formulas are in prenex and conjunctive normal

form.) This encoding is then used to prove Theorem 4.1, that the complexity of model checking the ambient logic is PSPACE-hard.
In our encoding, we assume that the truth values tt and ff used in the definition of QBF satisfaction [7] are distinct ambient calculus names.
We also use a derived operator for name equality in the ambient logic [4], $\eta = \mu$, encoded as $\eta[\mathbf{T}]@\mu$.Then $\mathbf{0} \models m = n$ if and only if the names m and n are equal.
We encode the $\forall$ and $\exists$ quantifiers over truth values as follows.

$$\forall x \in \{ff, tt\}.\mathcal{A} \triangleq \forall x.(x = ff \vee x = tt) \Rightarrow \mathcal{A}$$
$$\exists x \in \{ff, tt\}.\mathcal{A} \triangleq \exists x.(x = ff \vee x = tt) \wedge \mathcal{A}$$

Encoding QBF Formulas as Ambient Logic Formulas:

$[\![v]\!] \triangleq (v = tt)$	$[\![\overline{v}]\!] \triangleq (v = ff)$
$[\![\ell_1 \vee \cdots \vee \ell_k]\!] \triangleq [\![\ell_1]\!] \vee \cdots \vee [\![\ell_k]\!]$	$[\![C_1 \wedge \cdots \wedge C_k]\!] \triangleq [\![C_1]\!] \wedge \cdots \wedge [\![C_k]\!]$
$[\![\forall v.\varphi]\!] \triangleq \forall v \in \{ff, tt\}.[\![\varphi]\!]$	$[\![\exists v.\varphi]\!] \triangleq \exists v \in \{ff, tt\}.[\![\varphi]\!]$

The following properties are proved in the extended version of this paper [7].

Lemma 4.1. *Consider a closed quantified boolean formula φ and its encoding $[\![\varphi]\!]$ in the ambient logic. The formula φ is valid if and only if the model checking problem $\mathbf{0} \models [\![\varphi]\!]$ holds.*

Theorem 4.1. *The complexity of model checking the full logic (including name quantification) is PSPACE-hard.*

Proof. Straightforward from Lemma 4.1 since for the fixed ambient process $\mathbf{0}$ solving the model checking problem $\mathbf{0} \models \varphi$ is PSPACE-hard. So in fact the expression complexity, that is, the complexity of checking formulas against a fixed process, is PSPACE-hard. □

The theorem above holds for any fragment of the logic including boolean connectives, name quantification, and the location and location adjunct modalities, and for any fragment of the calculus including ambients. This might suggest that the complexity of the model checking problem comes from the quantification in the logic. It is not the case: the problem remains as complex even if we remove quantification from the logic and communication or mobility from the calculus. This suggests there is little chance of finding interesting fragments of the calculus and the logic that would admit a faster model checking algorithm.

4.2 Mobile Ambients without I/O, No Quantifiers

In this section, we study the complexity of the model checking problem for the fragment of the ambient calculus without I/O and the fragment of the logic without quantification. For every QBF variable, v, we assume that v, v' and v'' are distinct ambient calculus names.

Encoding QBF Formulas as Ambient Processes and Formulas:

$[\![v]\!] = v[pos[\mathbf{0}] \mid v'[\mathbf{0}]] \mid \mathbf{T}$
$[\![\overline{v}]\!] = v[neg[\mathbf{0}] \mid v'[\mathbf{0}]] \mid \mathbf{T}$
$[\![\ell_1 \vee \cdots \vee \ell_k]\!] = [\![\ell_1]\!] \vee \cdots \vee [\![\ell_k]\!]$

$[\![C_1 \wedge \cdots \wedge C_k]\!] = (end[\mathbf{0}], [\![C_1]\!] \wedge \cdots \wedge [\![C_k]\!])$

$[\![\forall v.\varphi]\!] = (v'[in\ v.n[out\ v'.out\ v.P]], \Box((n[\mathbf{T}] \mid \mathbf{T}) \Rightarrow \mathcal{A}))$ where $(n[P], \mathcal{A}) = [\![\varphi]\!]$
$[\![\exists v.\varphi]\!] = (v'[in\ v.n[out\ v'.out\ v.P]], \Diamond((n[\mathbf{T}] \mid \mathbf{T}) \wedge \mathcal{A}))$ where $(n[P], \mathcal{A}) = [\![\varphi]\!]$

$enc(\varphi) = (v_1[pos[\mathbf{0}]] \mid v_1[neg[\mathbf{0}]] \mid \cdots \mid v_n[pos[\mathbf{0}]] \mid v_n[neg[\mathbf{0}]] \mid P, \mathcal{A})$
where $(P, \mathcal{A}) = [\![\varphi]\!]$ and $\varphi = Q_1 v_1 . \ldots . Q_n v_n . C_1 \wedge \cdots \wedge C_k$
where each $Q_i \in \{\exists, \forall\}$.

In the encoding $enc(\varphi)$ above, the parallel composition $v_1[pos[\mathbf{0}]] \mid \ldots \mid v_n[neg[\mathbf{0}]]$ represents the sequence $v_1, \ldots v_n$ of (uninstantiated) boolean variables and P is a process that instantiates them. An instantiated variable v_i is represented by a subprocess $v_i[pos[\mathbf{0}] \mid v_i'[\mathbf{0}]] \mid v_i[neg[\mathbf{0}]]$ (if its value is tt) or $v_i[pos[\mathbf{0}]] \mid v_i[neg[\mathbf{0}] \mid v_i'[\mathbf{0}]]$ (if its value is ff). The process P first instantiates v_1 by choosing one of the ambients $v_1[pos[\mathbf{0}]]$ or $v_1[neg[\mathbf{0}]]$ nondeterministically, going inside it, leaving the token $v_1'[\mathbf{0}]$ inside the chosen ambient and then returning to the top level. It then iteratively instantiates the variables $v_2, \ldots, v_n$ in the same way. The formula $n[\mathbf{T}] \mid \mathbf{T}$ in the context of the encoding for a quantified variable v_i above (where n is v_{i+1} or end for $i = n$) expresses that instantiation of v_i has finished but instantiation of n has yet to start; thus $\Box(n[\mathbf{T}] \mid \mathbf{T} \ldots)$ and $\Diamond(n[\mathbf{T}] \mid \mathbf{T} \ldots)$ express, respectively, universal and existential quantifications over instantiations of v_i. For more details of the encoding, including examples and the proof of the lemma below, we refer to the full version of this paper [7].

Lemma 4.2. *Assume φ is a closed quantified Boolean formula, and that $(P, \mathcal{A}) = enc(\varphi)$. Then $P \models \mathcal{A}$ if and only if φ is valid.*

Theorem 4.2. *The complexity of model checking mobile ambients without I/O against the quantifier-free logic is PSPACE-hard.*

Proof. Straightforward from the PSPACE-completeness of the validity for QBF and from Lemma 4.2, taking into account that for $enc(\varphi) = (P, \mathcal{A})$, both P and $\mathcal{A}$ are of polynomial size with respect to φ. □

4.3 Immobile Ambients with I/O, No Quantifiers

In this section, we study the complexity of the model checking problem for the fragment of the ambient calculus without action prefix. For full details, see the full version.
We consider fixed names end, C, and D. For any QBF variable ambient name v_i', let $Inst(v_i') \triangleq v_i'[\mathbf{T}] \mid \mathbf{T}$ and $Inst^+(v_i') \triangleq v_i'[v_i''[\mathbf{T}] \mid \mathbf{T}] \mid \mathbf{T}$ and for the name end, $Inst(end) \triangleq end[\mathbf{T}] \mid \mathbf{T}$ and $Inst^+(end) \triangleq end[end'[\mathbf{T}] \mid \mathbf{T}] \mid \mathbf{T}$.

Encoding QBF Formulas as Ambient Processes and Formulas:

$$
\begin{array}{l}
[\![v]\!] = v[] \\
[\![\overline{v}]\!] = \overline{v}[] \\
[\![\ell_1 \vee \ldots \vee \ell_k]\!] = D[\mathbf{0}] \mid [\![\ell_1]\!] \mid \ldots \mid [\![\ell_k]\!] \\
\\
enc(C_1 \wedge \ldots \wedge C_k) = (end[C[\,[\![C_1]\!]\,] \mid \ldots \mid C[\,[\![C_k]\!]\,]], \\
\qquad\qquad\qquad\qquad \lozenge((D[\mathbf{0}] \mid \mathbf{T}) \Rightarrow (tt[\mathbf{0}] \mid \mathbf{T}))) \\
enc(\exists v.\varphi) = (v'[\langle tt\rangle \mid \langle ff\rangle \mid (v).(v''[] \mid (\overline{v}).n[P])], \\
\qquad\qquad \mathbf{T} \mid v'[\Diamond(\ (Inst(n) \wedge \neg Inst^{+}(n))\ \wedge \mathcal{A}\ )]) \quad \text{where } enc(\varphi) = (n[P], \mathcal{A}) \\
enc(\forall v.\varphi) = (v'[\langle tt\rangle \mid \langle ff\rangle \mid (v).(v''[] \mid (\overline{v}).n[P])], \\
\qquad\qquad \mathbf{T} \mid v'[\Box(\ (Inst(n) \wedge \neg Inst^{+}(n))\ \Rightarrow \mathcal{A}\ )]) \quad \text{where } enc(\varphi) = (n[P], \mathcal{A})
\end{array}
$$

The idea of the encoding here is quite similar to that from the previous section. A boolean variable v is represented here by two ambients $v[]$ and $\overline{v}[]$, which after the instantiation are named $tt[]$ and $ff[]$. We exploit here the nondeterminism of communication: the variable v reads either the message $\langle tt\rangle$ or $\langle ff\rangle$; then its dual $\overline{v}$ has to read the other one. The names v'_i and v''_i (similar to v'_i in the previous section) are used for distinguishing the moment when the variable v_i is already instantiated but v_{i+1} is not. The formula $\lozenge((D[\mathbf{0}] \mid \mathbf{T}) \Rightarrow (tt[\mathbf{0}] \mid \mathbf{T}))$ requires that in the final state, each ambient representing a clause (that is, an ambient containing $D[\mathbf{0}]$) contains at least one true literal (that is, an ambient $tt[\mathbf{0}]$).

Lemma 4.3. *Assume φ is a closed quantified Boolean formula, and that $(P, \mathcal{A}) = enc(\varphi)$. Then $P \models \mathcal{A}$ if and only if φ is valid.*

Theorem 4.3. *The complexity of model checking immobile ambients with I/O against the quantifier-free logic is PSPACE-hard.*

Proof. This follows from the PSPACE-completeness of validity for QBF, from Lemma 4.3 taking into account that for $enc(\varphi) = (P, \mathcal{A})$, both P and $\mathcal{A}$ are of polynomial size with respect to φ. □

We can strengthen this result by slightly modifying our encoding to obtain a lower bound on the program complexity of the problem, that is, the complexity of model checking processes against a fixed formula.

Theorem 4.4. *For every integer k there exists a formula $\mathcal{A}^{\exists}_k$ such that the complexity of model checking processes against $\mathcal{A}^{\exists}_k$ is Σ^P_k-hard.*

5 Conclusion

We show in this paper that the model checking problem of the replication-free ambient calculus with public names against the ambient logic without composition-adjunct is PSPACE-complete. In order to prove this complexity bound, we have proposed a new representation for processes, called closures, that prevents the exponential blow-up of the size of the problem. We use this representation together with a new algorithm to prove the PSPACE upper bound.

An important property obtained using closures is that there exist at most exponentially many (up to structural congruence) processes reachable from a given ambient process. This result is interesting because it seems very difficult to give a precise estimation of the number of reachable processes without using this new representation. This bound can be used, for instance, to prove that a model checking algorithm previously proposed by Cardelli and Gordon [4] works in single-exponential time on single-exponential space.
Another result given in this paper is an analysis of the complexity of the model checking problem for different interesting subsets of the calculus and the logic. We show that there is little chance to find polynomial algorithms for interesting subproblems. Indeed, model checking remains PSPACE-hard even for quite simple fragments of the calculus (for instance, without communication) and the logic (for instance, without quantification on names).
Possible directions for future work include investigations of the model checking problem for extensions of the logic and the calculus. Recently, Cardelli and Gordon [6] presented an extended version of the logic that allows reasoning about restricted names; it seems that there is no difficulty in extending our algorithm to deal with name restriction.

References

1. L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240:177–213, 2000.
2. L. Cardelli and A.D. Gordon. Types for mobile ambients. In *Proceedings POPL'99*, pages 79–92. ACM, 1999.
3. L. Cardelli and A.D. Gordon. Equational properties of mobile ambients. In *Proceedings FoSSaCS'99*, volume 1578 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 1999. An extended version appears as Technical Report MSR–TR–99–11, Microsoft Research, 1999.
4. L. Cardelli and A.D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings POPL'00*, pages 365–377. ACM, 2000.
5. L. Cardelli and G. Ghelli. A query language for semistructured data based on the ambient logic. To appear in one of the proceedings volumes of ETAPS'01, to accompany an invited talk. Springer, 2001.
6. L. Cardelli and A.D. Gordon. Logical properties of name restriction. In *Proceedings TLCA'01*. Springer, 2001. To appear.
7. W. Charatonik, S. Dal Zilio, A.D. Gordon, S. Mukhopadhyay, and J.-M. Talbot. The complexity of model checking mobile ambients. Technical Report MSR–TR–2001–03, Microsoft Research, 2001.
8. O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.
9. D. Sangiorgi. Extensionality and intensionality of the ambient logics. In *Proceedings POPL'01*. ACM, 2001. To appear.
10. W.J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.
11. L.J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

The Rho Cube

Horatiu Cirstea, Claude Kirchner, and Luigi Liquori

LORIA INRIA INPL ENSMN
54506 Vandoeuvre-lès-Nancy BP 239 Cedex France
{Horatiu.Cirstea,Claude.Kirchner,Luigi.Liquori}@loria.fr
www.loria.fr/{~cirstea,~ckirchne,~lliquori}

Abstract. The *rewriting calculus*, or Rho Calculus (ρCal), is a simple calculus that uniformly integrates abstraction on patterns and non-determinism. Therefore, it fully integrates rewriting and λ-calculus. The original presentation of the calculus was *untyped*. In this paper we present a uniform way to decorate the terms of the calculus with types. This gives raise to a new presentation *à la Church*, together with nine (8+1) type systems which can be placed in a *ρ-cube* that extends the λ-cube of Barendregt. Due to the matching capabilities of the calculus, the type systems use only one abstraction mechanism and therefore gives an original answer to the identification of the standard "λ" and "Π" abstractors.
As a consequence, this brings matching and rewriting as the first class concepts of the Rho-versions of the *Logical Framework* (LF) of Harper-Honsell-Plotkin, and of the *Calculus of Constructions* (CC) of Coquand-Huet.

1 Introduction

The first version of the simply typed *lambda*-calculus was proposed by Church [Chu41] in order to build a calculus having a solid mathematical foundation and a clear relation with logics. Since then, many other type systems have been proposed, and a synthesis, known as the Barendregt's λ-cube, has been realized in [Bar92].

In dependent type theories, the arrow-types of the form $A \to B$ are replaced by dependent products of the form $\Pi X{:}A.B$, with B possibly containing X as a free variable. We have thus an abstraction not only at the level of terms (the "λ" operator) but an abstraction at the level of types as well (the "Π" operator).

Many works can be found in the literature [dB80,KN96,PM97,KBN99,KL01] on the subject of unifying the "λ" and "Π" operators. In particular, a goal of these researches is to replace the $(Appl{-}\lambda)$ rule below, whose meta-level application of substitution can be considered as non aesthetic, with the rule $(Appl'{-}\lambda)$:

$$\frac{\Gamma \vdash A : \Pi X{:}C.D \quad \Gamma \vdash B : C}{\Gamma \vdash AB : D[X/B]}\ (Appl{-}\lambda) \qquad \frac{\Gamma \vdash A : \Pi X{:}C.D \quad \Gamma \vdash B : C}{\Gamma \vdash AB : (\Pi X{:}C.D)B}\ (Appl'{-}\lambda)$$

and to use the reduction on Π-redexes, *i.e.*: $(\Pi X{:}C.D)B \to_{\beta\Pi} D[X/B]$ describing thus the Π-abstractor at the same level as the λ-abstractor.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 168–183, 2001.

The four problems related to Π-reduction are clearly emphasized by Kamareddine *et alt.* [KN96,KBN99]: (1) correctness of type is no longer valid; (2) the system is no longer safe; (3) the type of an expression may be not well-formed; (4) the Π-redexes are not well-formed.

In [KBN99], the authors proposed some solutions for use Π-reductions:

- *Infinite levels of abstraction*, *i.e.* using infinite levels of abstractors of the form "Λ^i", such that we are able to type a function $\Lambda^i X{:}A.B$ with a type $\Lambda^{i+1} X{:}A.C$, where C is the type of B. This approach is essentially inspired by the AUTOMATH project [dB80];
- *Abbreviations*, *i.e.* extending, in typing judgments, the standard contexts Γ with abbreviations of the form $(X{:}A)C$ and adding the rule $(Appl{-}\lambda\Pi)$:

$$\frac{\Gamma, (X{:}A)C \vdash B : D \quad \pi \in \{\lambda, \Pi\}}{\Gamma \vdash (\pi X{:}A.B)C : D[X/C]} \ (Appl{-}\lambda\Pi)$$

  Intuitively, this rule says that if $B : D$ can be typed using the "abbreviation" that X of type A is C (*i.e.* $(X{:}A)C$), then $(\pi X{:}A.B)C : D[X/C]$ can be typed without this abbreviation.

In this paper we propose a different solution to the above four cited problems; our approach is essentially based on the *complete* unification of the two operators "λ" and "Π" into the only abstraction symbol present in the Rho Calculus, *i.e.* the "$\rightarrow$" operator. This unification is, so to speak, *built-in* in the definition of the Rho Calculus itself.

The rewriting calculus, or Rho Calculus, integrates in a uniform way matching, rewriting and non determinism. Its abstraction mechanism is based on the rewrite rule formation: in a ρ-term of the form $l \rightarrow r$, we abstract on the ρ-term l, and it is worth noticing that when l is a variable x this corresponds exactly to the λ-term $\lambda x.r$. When an abstraction $l \rightarrow r$ is applied to the ρ-term u, which is denoted by $(l \rightarrow r)\bullet u$, the evaluation mechanism is based on the binding of the free variables present in l to the appropriate subterms of u. Indeed this binding is realized by matching l against u, and one of the characteristic of the calculus is to possibly use informations in the matching process such as algebraic axioms like associativity or commutativity.

At that stage, non-determinism may come into play since the matching process can return zero, one, or several possibilities. For each of these variable bindings, the value of the variables are propagated in the term r yielding zero, one of several (finite or not) results. In a restricted way this is exactly what happen in the β-redex $(\lambda x.r)\, u$, which is simply denoted in the syntax of the Rho Calculus $(x \rightarrow r)\bullet u$, and where the match is trivially the substitution $\{x/u\}$. Therefore, the Rho Calculus strictly contains the λ-calculus and the possibility to express failure of evaluation or multiplicity of results is directly supported.

The expressiveness of the arrow abstractor together with its matching powered evaluation rule makes very attractive the possibility to uniformly express both abstraction at the level of terms and of types. Therefore, we present and experiment a collection of nine $(8 + 1)$ type systems for the Rho Calculus decorated à la Church, that can be represented in a ρ-cube that extends the

λ-cube of Barendregt. The most powerful type system in our cube (namely the ninth one) is a variant of the plain Calculus of Constructions and it is essentially inspired from the *Extended Calculus of Constructions* (ECC) of Z. Luo [Luo90]. In ECC, indeed, we have an infinite set of sorts, *i.e.* $s \in \{*, \Box_i\}$, with $i \in \mathbb{N}$, and the extra axiom $\vdash \Box_i : \Box_{i+1}$ with $i \in \mathbb{N}$. To sum up, we come up with a rich family of type systems which:

- provide typing mechanisms for the rewriting calculus;
- simplify and embeds the presentation of the λ-cube;
- provide an original and natural solution to the unification of the "λ" and "Π" operators;
- make matching power a truly integrated tool for type theory and therefore naturally integrates rewriting into type theory, and specifically into the calculus of construction;
- open new challenging questions and conjectures about logics for the ρ-cube.

Road map of the paper. In Section 2, we present the syntax of the typed Rho Calculus, the matching relation, some simple matching theories, the operational semantics of the calculus, the type system, and a pictorial classification of the type systems in a ρ-cube. In Section 3, we present several examples of type derivations in some of the most interesting type systems. Section 4 presents some conjectures and we conclude the paper with some future works and perspectives.

The design and study of the ρ-cube are ongoing works. In particular, we emphasize in the last part several questions and conjectures that we are currently working on. The reader will find on the web pages of the authors, the latest state of developments concerning these questions.

2 Syntax Matching and Types

We present here the syntax and operational semantics of the Rho Calculus. This extends with a very general type discipline the calculus presented in [CK99b, CKL00,Cir00].

2.1 Syntax

Notational Conventions. In this paper, the symbols A, B, C range over the set $\mathcal{T}$ of typed terms, the symbols $X^{\mathcal{T}}, Y^{\mathcal{T}}, Z^{\mathcal{T}}, \ldots$ range over the infinite set $\mathcal{V}$ of *typed* variables, the symbols $a^{\mathcal{T}}, b^{\mathcal{T}}, f^{\mathcal{T}} \ldots$ range over the infinite set $\mathcal{C}$ of *typed* constants, the symbol s ranges over the infinite set $\mathcal{S} \equiv \{*, \Box_i\}_{i \in \mathbb{N}}$ of untyped sorts, and the symbols $\alpha^{\mathcal{T}}, \beta^{\mathcal{T}}, \ldots$ range over $\mathcal{C}$ and $\mathcal{V}$. All symbols can be indexed. The symbol $\equiv$ denotes syntactic identity of objects like terms or substitutions. We assume that variables and constants have a unique type, *i.e.* if $X^A, X^B \in \mathcal{V}$ (resp. $a^A, a^B \in \mathcal{C}$), then $A \equiv B$. An application of a constant function, say f^A, to a term B will be denoted by $f(B)$.

We work modulo α-conversion, and we follow the Barendregt convention [Bar84], saying that free and bound variables have different names. The syntax of the pseudo-terms of the Rho Calculus is defined as follows:

$$\mathcal{T} ::= \mathcal{S} \mid a^{\mathcal{T}} \mid X^{\mathcal{T}} \mid \mathcal{T} \to \mathcal{T} \mid \mathcal{T}\bullet\mathcal{T} \mid \qquad \text{plain terms}$$

$$null \mid \mathcal{T}, \mathcal{T} \qquad \text{structured terms}$$

The main intuition behind this syntax is that a rewrite rule $\mathcal{T} \to \mathcal{T}$ is an *abstraction*, the left-hand side of which determines the bound variables and some pattern structure. The application of a ρCal-term on another ρCal-term is represented by "•". The terms can be grouped together into structures built using the "," operator and, according to the theory behind this operator, different structures can be obtained. We will come back to this later but, to support the intuition, let us mention that if we define the "," operator to be associative, then a *list structure* is obtained, while if the "," operator is specified as associative, commutative, and idempotent, then the *set structure* is obtained. The term *null* denotes an empty structure.

As usual, we assume that the application operator "•" associates to the left while the "$\to$" and the "," operators associate to the right. The priority of the application "•" is higher than that of the "$\to$" operator which is in turn of higher priority than the "," operator.

One may notice that in the syntax of ρCal, variables (free and bound) and constants are always annotated with a single and unique type with the only exception of the undecorated term *null*: as it will become clear when presenting the type system, this term can be assigned any type.

Example 1 (ρCal-terms). Some simple examples of ρCal-terms are:

- the term $(f^{A\to A}(X^A) \to X^A)\bullet f^{A\to A}(a^A)$ represents the application of the classical rewrite rule $f^{A\to A}(X^A) \to X^A$ to the term $f^{A\to A}(a^A)$;
- the term $X^A \to Y^B \to X^A$ is the typed λ-term $\lambda X{:}A.\lambda Y{:}B.X$;
- the term (a^A, b^B, c^C) denotes a simple ternary structure;
- the term $(a_1^A \to b_1^B, a_2^A \to b_2^B)$ denotes a record with two fields, a_1 and a_2.

An important parameter of the ρCal is the matching theory $\mathbb{T}$. We assume given a theory $\mathbb{T}$, defined for example equationally, and we define matching problems in this general setting. In what follows we introduce several such theories, some of them being used later for instantiating the general ρCal.

2.2 Typed Matching Theories

An important parameter of the ρCal is the matching theory $\mathbb{T}$. We assume given a theory $\mathbb{T}$ and we define matching problems in this general setting. The theories extend the ones presented in [CKL00] in order to be compatible with the typing discipline. Matching theories are defined equationally via the judgment:

$$\Vdash A =_{\mathbb{T}} B : C$$

As we will see below, this judgment is defined in terms of the typing judgment $\vdash A : B$ defined in Subsection 2.4. Intuitively, its meaning is that A and B are terms of type C and are equivalent in the theory $\mathbb{T}$.

Definition 1 (Matching Theories).

- *the Empty theory* $\mathbb{T}_\emptyset$ *of equality is defined by the following inference rules:*

$$\frac{\Vdash A =_{\mathbb{T}_\emptyset} B : D \quad \Vdash B =_{\mathbb{T}_\emptyset} C : D}{\Vdash A =_{\mathbb{T}_\emptyset} C : D}\ (Trans) \qquad \frac{\Vdash A =_{\mathbb{T}_\emptyset} B : C}{\Vdash B =_{\mathbb{T}_\emptyset} A : C}\ (Symm)$$

$$\frac{\Vdash A =_{\mathbb{T}_\emptyset} B : E \quad \vdash C_{|p} : E \quad \vdash C : D}{\Vdash C[A]_p =_{\mathbb{T}_\emptyset} C[B]_p : D}\ (Ctx) \qquad \frac{\vdash A : B}{\Vdash A =_{\mathbb{T}_\emptyset} A : B}\ (Refl)$$

  where $A[B]_p$ *denotes the term* A *with the term* B *at position* p*, and* $A_{|p}$ *the term at the position* p *of the term* A*.*
  Note that in the theory $\mathbb{T}_\emptyset$ *we do not need an inference rule describing the stability by substitution since all the theories below are defined using inference rules exhibiting all possible instantiations for the terms.*
- *for a given binary symbol* f*, the theory of Commutativity* $\mathbb{T}_{C(f)}$ *is defined by* $\mathbb{T}_\emptyset$ *plus the following inference rule:*

$$\frac{\vdash f^{C\to C\to D}(A\ B) : D}{\Vdash f^{C\to C\to D}(A\ B) =_{\mathbb{T}_{C(f)}} f^{C\to C\to D}(B\ A) : D}\ (Comm)$$

- *for a given binary symbol* f*, the theory of Associativity* $\mathbb{T}_{A(f)}$ *is defined by* $\mathbb{T}_\emptyset$ *plus the following inference rule:*

$$\frac{\vdash f^{D\to D\to D}(f^{D\to D\to D}(A\ B)\ C) : D}{\Vdash f^{D\to D\to D}(f^{D\to D\to D}(A\ B)\ C) =_{\mathbb{T}_{A(f)}} f^{D\to D\to D}(A\ f^{D\to D\to D}(B\ C)) : D}(Assoc)$$

- *for a given binary symbol* f*, the theory of Idempotency* $\mathbb{T}_{I(f)}$ *is defined by* $\mathbb{T}_\emptyset$ *plus the following inference rule:*

$$\frac{\vdash f^{B\to B\to B}(A\ A) : B}{\Vdash f^{B\to B\to B}(A\ A) =_{\mathbb{T}_{I(f)}} A : B}\ (Idem)$$

- *for a given binary symbol* f *and a constant* 0*, the theory of Neutral (left and right) element* $\mathbb{T}_{N(f^0)}$ *is defined by* $\mathbb{T}_\emptyset$ *plus the following inference rules:*

$$\frac{\vdash f^{B\to B\to B}(0^B\ A) : B}{\Vdash f^{B\to B\to B}(0^B\ A) =_{\mathbb{T}_{N(f^0)}} A : B}(0_L) \qquad \frac{\vdash f^{B\to B\to B}(A\ 0^B) : B}{\Vdash f^{B\to B\to B}(A\ 0^B) =_{\mathbb{T}_{N(f^0)}} A : B}(0_R)$$

- *the MultiSet theory,* $\mathbb{T}_{MSet(f,nil)}$*, is obtained by considering the symbol* f *of type* $A \to A \to A$ *as an associative and commutative function symbol with nil as neutral element:*

$$\mathbb{T}_{MSet(f,nil)} = \mathbb{T}_{A(f)} \cup \mathbb{T}_{C(f)} \cup \mathbb{T}_{N(f^{nil})}$$

- *the Set theory,* $\mathbb{T}_{Set(f,nil)}$*, is obtained from* $\mathbb{T}_{MSet(f,nil)}$*, by considering the symbol* f *as an idempotent function symbol:*

$$\mathbb{T}_{Set(f,nil)} = \mathbb{T}_{MSet(f,nil)} \cup \mathbb{T}_{I(f)}$$

As already mentioned, computing the substitutions which solve the matching (in the theory $\mathbb{T}$) from a pattern A to a subject B is a fundamental element of the ρCal. Syntactic matching is formally defined as follows:

Definition 2 (Syntactic Matching). *For a given theory $\mathbb{T}$ over ρCal-terms:*

1. *a* $\mathbb{T}$-match equation *is a formula of the form* $A \ll_{\mathbb{T}} B$;
2. *a* substitution σ *is a solution of the* $\mathbb{T}$*-match equation* $A \ll_{\mathbb{T}} B$ *if* $\sigma A =_{\mathbb{T}} B$;
3. *a* $\mathbb{T}$-matching system *is a conjunction of* $\mathbb{T}$*-match equations;*
4. *a substitution* σ *is a solution of a* $\mathbb{T}$*-matching system if it is a solution of all the* $\mathbb{T}$*-match equations in it;*
5. *a* $\mathbb{T}$*-matching system is* trivial *when all substitutions are solution of it and we denote by* $\mathbb{F}$ *a* $\mathbb{T}$*-matching system without solution;*
6. *we define the function* $\mathcal{S}ol$ *on a* $\mathbb{T}$*-matching system* T *as returning the* $\prec$*-ordered*[1] *list of all* $\mathbb{T}$*-matches of* T *when* T *is not trivial and the list containing only* $\sigma_{\mathbf{id}}$*, where* $\sigma_{\mathbf{id}}$ *is the identity substitution, when* T *is trivial.*

Notice that when the matching algorithm fails to find a solution (*i.e.* returns $\mathbb{F}$), the function $\mathcal{S}ol$ returns the empty list.

Since in general we can consider arbitrary theories over ρCal-terms, $\mathbb{T}$-matching is in general undecidable, even when restricted to first-order equational theories [JK91].

For example, in $\mathbb{T}_\emptyset$, the matching substitution from a ρCal-term A to a ρCal-term B can be computed by the rewrite system presented in Figure 1, where the symbol $\wedge$ is assumed to be associative and commutative, and $\diamond_1$, $\diamond_2$ are either constant symbols or the prefix notations of "," or "•" or "$\rightarrow$".

$$
\begin{array}{lcll}
\diamond_1(A_1 \dots A_n) \ll_{\mathbb{T}_\emptyset} \diamond_2(B_1 \dots B_m) & \rightsquigarrow & \begin{cases} \bigwedge_{i=1\dots n} A_i \ll_{\mathbb{T}_\emptyset} B_i & \text{if } \diamond_1 \equiv \diamond_2 \text{ and } n = m \\ \mathbb{F} & \text{otherwise} \end{cases} \\
(X^C \ll_{\mathbb{T}_\emptyset} A) \wedge (X^C \ll_{\mathbb{T}_\emptyset} B) & \rightsquigarrow & \begin{cases} X^C \ll_{\mathbb{T}_\emptyset} A & \text{if } \Vdash A =_{\mathbb{T}_\emptyset} B : C \\ \mathbb{F} & \text{otherwise} \end{cases} \\
A \ll_{\mathbb{T}_\emptyset} X^B & \rightsquigarrow & \mathbb{F} & \text{if } A \not\in \mathcal{V} \\
\mathbb{F} \wedge (A \ll_{\mathbb{T}_\emptyset} B) & \rightsquigarrow & \mathbb{F} &
\end{array}
$$

Fig. 1. Rules for Typed Syntactic Matching.

Starting from a matching system T, the application of this rule set terminates and returns either $\mathbb{F}$ when there are no substitutions solving the system, or a system T' in "normal form" from which the solution can be trivially inferred [KK99]. This set of rules could be easily extended to matching modulo

[1] We consider a total order $\prec$ on the set of substitutions. Such orderings always exist [CKL00]: one can for example take the lexicographic ordering on the flattened representation of the substitutions. The ordering on substitutions can be seen as a parameter of the ρCal and thus can be customized by the user.

commutativity (it is decidable too but the number of matches can then be exponential in the size of the initial problem). It is also decidable for associativity-commutativity [Hul79] but in the general case matching could be as difficult as unification [Bür89].

Example 2 (Matching).

1. in $\mathbb{T}_\emptyset$, if a and b are two different constants, the equation $a^A \ll_{\mathbb{T}_\emptyset} b^B$ has no solution, and thus $\mathcal{S}ol(a^A \ll_{\mathbb{T}_\emptyset} b^B)$ returns the empty list of substitutions;
2. in $\mathbb{T}_\emptyset$, $a^A \ll_{\mathbb{T}_\emptyset} a^A$ is solved by all substitutions, and thus $\mathcal{S}ol(a^A \ll_{\mathbb{T}_\emptyset} a^A)$ returns the list with only one element, $\sigma_{\mathbf{id}}$;
3. in $\mathbb{T}_\emptyset$, the equation $f(X^A\ g(X^A\ Y^D)) \ll_{\mathbb{T}_\emptyset} f(a^A\ g(a^A\ d^D))$ (f, g stand for $f^{A\to B\to C}, g^{A\to D\to B}$) has for solution the substitution $\sigma \equiv [X^A/a^A\ \ Y^D/d^D]$;
4. in $\mathbb{T}_{C(f)}$ the equation $f(X^A\ Y^A) \ll_{\mathbb{T}_{C(f)}} f(a_1^A\ a_2^A)$ (f stands for the a commutative symbol $f^{A\to A\to B}$) has the two solutions $\sigma_1 \equiv [X^A/a_1^A\ \ Y^A/a_2^A]$ and $\sigma_2 \equiv [X^A/a_2^A\ \ Y^A/a_1^A]$, and therefore the matching equation $\mathcal{S}ol(f(X^A\ Y^A) \ll_{\mathbb{T}_{C(f)}} f(a_1^A\ a_2^A))$ is solved by the list $\sigma_1\ \sigma_2$;
5. in $\mathbb{T}_{AN} = \mathbb{T}_{A(,)} \cup \mathbb{T}_{N(,^{null})}$, the equation $(X^A, a^A, Y^A) \ll_{\mathbb{T}_{AN}} (a_1^A, a^A, a_2^A, a_3^A)$ has for solution the substitution $\sigma \equiv [X^A/a_1^A\ \ Y^A/(a_2^A, a_3^A)]$, while the two equations $(X^A, a^A, Y^A) \ll_{\mathbb{T}_{AN}} (null, a^A)$ and $(X^A, a^A, Y^A) \ll_{\mathbb{T}_{AN}} a^A$ have both the same solution $\sigma \equiv [X^A/null\ \ Y^A/null]$, since $\Vdash a^A =_{\mathbb{T}_{AN}} (null, a^A) =_{\mathbb{T}_{AN}} (null, a^A, null) : A$;
6. in $\mathbb{T}_{ACN} = \mathbb{T}_{A(,)} \cup \mathbb{T}_{C(,)} \cup \mathbb{T}_{N(,^{null})}$, the equation $(X^A, a^A) \ll_{\mathbb{T}_{ACN}} (a_1^A, a^A, a_2^A, a_2^A)$ has for solution the substitution $\sigma \equiv [X^A/(a_1^A, a_2^A, a_3^A)]$, since $\Vdash (a_1^A, a^A, a_2^A, a_3^A) =_{\mathbb{T}_{ACN}} (a_1^A, a_2^A, a_3^A, a^A) : A$.

2.3 Operational Semantics

Using the same notations as [CKL00], to which the reader is referred for further discussion of the concepts, the notion of reduction on terms in $\mathbb{T}$ is uniformly defined as follows: for a given ordering $\prec$ on substitutions (which is left implicit in the notation), and a theory $\mathbb{T}$, the operational semantics is defined by the computational rules given in Figure 2. The central idea of the main rule of the

$$
\begin{array}{lrcl}
(\rho) & (A \to B)\bullet C & \mapsto_{\mathbb{T}} & \begin{cases} null & \text{if } A \ll_{\mathbb{T}} C \text{ has no solution} \\ \sigma_1 B, \ldots, \sigma_n B & \text{if } \sigma_i \in \mathcal{S}ol(A \ll_{\mathbb{T}} C),\ \sigma_i \prec \sigma_{i+1},\ n \leq \infty \end{cases} \\
(\epsilon) & (A, B)\bullet C & \mapsto_{\mathbb{T}} & A\bullet C, B\bullet C \\
(\nu) & null\bullet A & \mapsto_{\mathbb{T}} & null
\end{array}
$$

Fig. 2. Evaluation rules of the ρCal.

calculus (ρ) is that the application of a rewrite rule $A \to B$ at the root (also called top) position of a term C, consists in computing all the solutions of the matching equation $(A \ll_{\mathbb{T}} C)$ in the theory $\mathbb{T}$, and applying all the substitutions

from the $\prec$-ordered list returned by the function $\mathcal{S}ol(A \ll_{\mathbb{T}} C)$ to the term B. When there is no solution for the matching equation $(A \ll_{\mathbb{T}} C)$ the special constant *null* is obtained as result of the application. Notice that there could be an infinity of solutions to the matching problem $(A \ll_{\mathbb{T}} C)$, but in this paper we restrict ourselves to the case where the matching problem is finitary.

It is important to remark that if A is a variable, then the (ρ) rule corresponds exactly to the (β) rule of the λ-calculus, and variables manipulations in substitutions take places externally, using α-conversion and Barendregt's convention if necessary.

The rules (ϵ) and (ν) deal with the distributivity of the application on the structures whose constructors are "," and *null*.

With respect to the previous presentation of the Rho Calculus given in [CK99b,CK99a], we have modified the notation of the application operator, simplified the evaluation rules, and generalized to deal with generic result structures.

When the theory $\mathbb{T}$ is clear from the context, its denotation will be omitted.

Definition 3. *We denote by $=_\rho$ the reflexive, symmetric, transitive, and contextual closure of the reduction $\mapsto_{\mathbb{T}}$ over a theory $\mathbb{T}$. The relation $=_\rho$ is a congruence relation.*

When working modulo reasonably powerful theories $\mathbb{T}$, the evaluation rules of the ρCal are confluent [Cir00,CKL00].

Theorem 1 (Confluence in $\mathbb{T}_\emptyset$). *Given a term A such that all its abstractions contain no arrow in the first argument, if $A \mapsto\!\!\!\!\twoheadrightarrow_{\mathbb{T}_\emptyset} B$ and $A \mapsto\!\!\!\!\twoheadrightarrow_{\mathbb{T}_\emptyset} C$ then there exists a term D such that $B \mapsto\!\!\!\!\twoheadrightarrow_{\mathbb{T}_\emptyset} D$ and $C \mapsto\!\!\!\!\twoheadrightarrow_{\mathbb{T}_\emptyset} D$.*

2.4 The Type System

A statement is of the form $A : B$, with $A, B \in \mathcal{T}$. Judgments are defined as $\vdash A : B$, *i.e.* without any kind of context information: this was possible thanks to the explicit type-decoration for *all* constants and variables (generically denoted by α). We call A the subject and B the predicate of the judgment, respectively. The type rules in Figure 3 are given in three groups: general rules for applications/structures (common to all systems), and one specific rule schema (characteristic to each system) parameterized by a suitable choice of sorts.

Our type system presents in fact nine type systems instead of the usual eight present in the λ-cube; the eight type systems of the ρ-cube (natural counterparts of the λ-cube) are defined by taking the general rules plus the specific subset of rules $\{(*,*),(*,\Box_0),(\Box_0,*),(\Box_0,\Box_0)\}$. In the more general case, many other subsets of specific rules can be obtained by considering all the following pairs of sorts: $\{(*,*),(*,\Box_i),(\Box_i,*),(\Box_i,\Box_j)\}$ with $i,j \in \mathbb{N}$. This (possibly infinite) subset of rules gives raise to the most powerful type system ρECC, which uses the full typing power of the ECC of Z. Luo. Figure 3 describes also all the systems and collects it in a graphical ρ-cube. The system $\rho \rightarrow$ is essentially a variant of the system presented in [Cir00,CK00].

General Rules for Applications

$$\frac{}{\vdash * : \Box_0}\ (Axiom_*) \qquad \frac{i \in \mathbb{N}}{\vdash \Box_i : \Box_{i+1}}\ (Axiom_\Box) \qquad \frac{\vdash A : s}{\vdash \alpha^A : A}\ (Start)$$

$$\frac{\vdash B : C \quad \vdash A \to C : s}{\vdash A \to B : A \to C}\ (Abs) \qquad \frac{\vdash A : C \to D \quad \vdash C : E \quad \vdash B : E}{\vdash A\bullet B : (C \to D)\bullet B}\ (Appl)$$

$$\frac{\vdash A : C \quad \vdash B : s \quad B =_\rho C}{\vdash A : B}\ (Conv)$$

General Rules for Structures

$$\frac{\vdash A : s}{\vdash null : A}\ (Null) \qquad \frac{\vdash A : C \quad \vdash B : C}{\vdash A, B : C}\ (Struct)$$

Specific Rules

$$\frac{\vdash A : C \quad \vdash C : s_1 \quad \vdash B : s_2}{\vdash A \to B : s_2}\ (s_1, s_2)$$

System	Set of specific rules $(i, j \in \mathbb{N})$
$\rho \to$	$(*,*)$
$\rho 2$	$(*,*)$ $(\Box_0,*)$
$\rho P = \rho LF$	$(*,*)$ $(*,\Box_0)$
$\rho P2$	$(*,*)$ $(\Box_0,*)$ $(*,\Box_0)$
$\rho\underline{\omega}$	$(*,*)$ $(\Box_0,\Box_0)$
$\rho\omega$	$(*,*)$ $(\Box_0,*)$ $(\Box_0,\Box_0)$
$\rho P\omega$	$(*,*)$ $(*,\Box_0)$ $(\Box_0,\Box_0)$
$\rho P\underline{\omega} = \rho CC$	$(*,*)$ $(\Box_0,*)$ $(*,\Box_0)$ $(\Box_0,\Box_0)$
ρECC	$(*,*)$ $(\Box_i,*)$ $(*,\Box_i)$ $(\Box_i,\Box_j)$

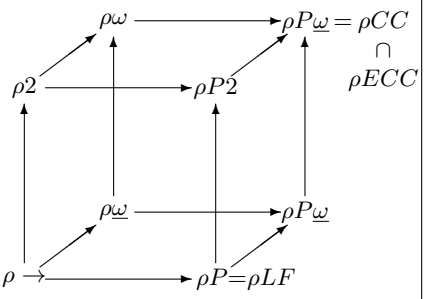


Fig. 3. The type system rules, the generic rules, and the ρ-cube.

In order to help the reader in understanding the type system, we give here a small explication for all the type rules and a brief comparison with the rules of the λ-cube:

- $(Axiom_*), (Axiom_\Box)$. Those two rules are used in the leaves of any well-typed derivation; in particular the rule $(Axiom_\Box)$ allows one to obtain a *type hierarchy* similar to that of Martin-Löf type theory [Mar84];
- $(Start)$. Only well decorated constants or variables are well typed: note that the type is "written" in the term itself, as for example in X^A, or a^A;
- (Abs). This rule represents the counterpart of the λ-cube rule $(Abs{-}\lambda)$ which could be written in our syntax as $(Abs{-}\lambda\rho)$:

$$\frac{\Gamma, X{:}D \vdash B : C \quad \Gamma \vdash \Pi X{:}D.C : s}{\Gamma \vdash \lambda X{:}D.B : \Pi X{:}D.C}\,(Abs{-}\lambda) \qquad \frac{\vdash B : C \quad \vdash X^D \to C : s}{\vdash X^D \to B : X^D \to C}\,(Abs{-}\lambda\rho)$$

The main differences comparing to the corresponding rule in the ρ-cube are:

1. the "λ" and "Π" operators are unified into the "$\rightarrow$" operator: in fact the Π-operator can be considered as a limited form of the "$\rightarrow$" abstractor, where the pattern used for the matching is always a variable X^D. In the (*Abs*) rule the pattern A can be more sophisticated than a simple variable and hence the result type of an abstraction is $A \rightarrow C$;
2. since all variables (free and bound) are decorated by their types, the notion of context Γ is useless, and therefore we do not need in the first premise of the rule to enrich the context Γ with the declaration of the bound variable X^D;
3. as in the λ-cube, the second premise is a call to a specific rule (s_1, s_2) that allows one to discern between the different type systems of the ρ-cube;

– (*Appl*). This rule represents the counterpart of the λ-cube rule ($Appl{-}\lambda$) which could be written in our syntax as ($Appl{-}\lambda\rho$):

$$\frac{\Gamma \vdash A : \Pi X{:}E.D \quad \Gamma \vdash B : E}{\Gamma \vdash AB : D[X/B]}\ (Appl{-}\lambda) \qquad \frac{\vdash A : X^E \rightarrow D \quad \vdash B : E}{\vdash A{\bullet}B : D[X/B]}(Appl{-}\lambda\rho)$$

The main differences comparing to the corresponding rule in the ρ-cube are:

1. in the first premise, the term A is typed with a term $C \rightarrow D$ instead of a Π-term, where C can be *any* ρ-term;
2. in order to type-check the type of the argument with the domain-type of the function, we need to check both the type of the pattern C (since it is not necessarily a variable) and of the argument B; this task is done by the second and the third premises;
3. the result type of the application will be in turn the type $(C \rightarrow D){\bullet}B$, *i.e.* a type not in normal-form. Note here the difference with respect to the solution adopted in the λ-cube, where the result type was $D[X/B]$: this solution takes into account, in a very natural way, all the type systems placed in the right-hand side of the ρ-cube, *i.e.* the ones with dependent-types, and enlightens the higher-order nature of the "$\rightarrow$" operator;

– ($Conv$). This rule (not syntax-directed) has the same shape as the correspondent rule in the λ-cube. When building derivations, this rule should be used in order to reduce the type $(C \rightarrow D){\bullet}B$ to an equivalent type;
– ($Null$). The untyped constant *null* is typed with any (well-formed) type;
– ($Struct$). This rule for structures enforces the type of A to be the same as the type of B; essentially we guarantee that all the terms in a structure have exactly the same type. A more flexible rule that deals with terms having a different type in the same structure would be ($Struct'$):

$$\frac{\vdash A : C \quad \vdash B : D}{\vdash A, B : C, D}\ (Struct')$$

This rule would be essential for typing the *object-oriented* features of the ρCal [CKL00]. This topic is subject to a current parallel study.

– (s_1, s_2). This rule is usually invoked in order to demonstrate the second premise of the (*Abs*) rule: it is a rule schema since it represents in fact an infinite set of type rules, depending on the shape of the sorts s_1 and s_2.

Remark. Note that the more general type system ρECC differs from the original Luo's ECC since we do not introduce:

1. Σ-types (or, strong *sum-types*) $\Sigma X{:}A.B$;
2. a *type-cumulativity* order $\preceq$ on types, re-paraphrased in the ρCal as the smallest partial order over ρ-terms w.r.t. conversion $\simeq$[2] such that: (a) $* \preceq \Box_0 \preceq \Box_1 \preceq \ldots$ and (b) $A \simeq A' \wedge B \preceq B' \Longrightarrow A \to B \preceq A' \to B'$;
3. the "subtyping-like" rule (*cum*) $(\vdash A : B \ \& \ \vdash C : \Box_i \ \& \ B \preceq C \implies \vdash A : C)$.

3 Examples

In this section we present some judgments and (by lack of space) a few type derivations in some representative type systems of the ρ-cube, namely:

- in the system $\rho \to$, the simply typed Rho Calculus (terms dependent on terms);
- in the system $\rho 2$, the polymorphic (in the sense of Girard's system F ($\lambda 2$) [Gir86]) Rho Calculus (terms dependent on types);
- in the system ρLF, the simply typed Rho Calculus with dependent-types: this system is the counterpart of the system LF, *i.e.* the *Logical Framework* [HHP92] (types dependent on terms);
- in the system $\rho\underline{\omega}$, (types depending on types);
- in the system ρECC, the counterpart of the system ECC of Z. Luo.

Most of the examples are inspired (or re-arranged) from [Bar92]. When not explicitly mentioned, we denote the symbol $\Box_0$ by $\Box$.

Example 3 (The Type System $\rho \to$). In this system the following can be derived when we consider the constants int^* and 3^{int^*}:

$$\dfrac{\dfrac{\dfrac{\vdash X^{int^*} : int^* \quad \vdash int^* : *}{\vdash X^{int^*} : int^* \qquad \vdash X^{int^*} \to int^* : *}}{\vdash X^{int^*} \to X^{int^*} : X^{int^*} \to int^*} \qquad \vdash X^{int^*} : int^* \quad \vdash 3^{int^*} : int^*}{\dfrac{\vdash (X^{int^*} \to X^{int^*})\bullet 3^{int^*} : (X^{int^*} \to int^*)\bullet 3^{int^*} \qquad (1,2)}{\vdash (X^{int^*} \to X^{int^*})\bullet 3^{int^*} : int^*}}$$

where (1) is $\vdash int^* : *$, and (2) is $(X^{int^*} \to int^*)\bullet 3^{int^*} =_\rho int^*$, and the last applied rule is $(Conv)$. Proceeding in a similar way, it is not difficult to derive in $\rho \to$ (we write f, g, h instead of $f^{A\to A}, g^{A\to A}, h^{A\to A\to A}$):

$$\vdash X^A \to A : * \qquad \vdash (f(X^A) \to X^A, g(X^A) \to X^A)\bullet f(a^A) : A$$
$$\vdash (f(X^A) \to X^A)\bullet f(a_1^A) : A \qquad \vdash (h(X^A\ Y^A) \to (X^A, Y^A))\bullet h(a_1^A\ a_2^A) : A$$

[2] $\simeq$ denotes the equivalence generated by $\preceq$.

Example 4 (The Type System ρLF). In this system the following can be derived:

$$\dfrac{\dfrac{\dfrac{\vdash X^{int^*} : int^* \quad \vdash int^* : * \quad \vdash * : \Box}{\vdash X^{int^*} \to * : \Box}}{\vdash f^{X^{int^*} \to *} : X^{int^*} \to * \quad \vdash X^{int^*} : int^* \quad \vdash 3^{int^*} : int^*} \qquad \dfrac{}{(X^{int^*} \to *)\bullet 3^{int^*} =_\rho *}\ (1)}{\vdash f^{X^{int^*} \to *}\bullet 3^{int^*} : (X^{int^*} \to *)\bullet 3^{int^*}}$$

$$\vdash f^{X^{int^*} \to *}(3^{int^*}) \equiv f^{X^{int^*} \to *}\bullet 3^{int^*} : *$$

where (1) is $\vdash * : \Box$ the last applied rule is $(Conv)$. Observe that the judgment $\vdash X^{int^*} \to * : \Box$ can be derived thanks to the specific rule $(*, \Box)$.

In a similar way, we can easily derive the following judgments (here p, q are used in $\vdash p^{A \to *}(X^A) : *$ and $\vdash q^{A \to *}(X^A) : *$[3])

$$\vdash X^A \to Y^{p(X^A)} \to q(X^A) : *^4 \qquad \vdash X^A \to Y^{p(X^A)} \to p(X^A) : *^5$$
$$\vdash X^A \to Y^{p(X^A)} \to Y^{p(X^A)} : X^A \to Y^{p(X^A)} \to p(X^A)^6$$

Example 5 (The Type Systems ρ2, ρω, ρECC).

in $\rho 2$ $\quad \vdash \bot \equiv X^* \to X^* : *$
$\vdash Y^* \to Z^\bot \to (Z^\bot \bullet Y^*) : Y^* \to Z^\bot \to Y^*$[7]
$\vdash X^* \to Y^{X^*} \to Y^{X^*} : X^* \to Y^{X^*} \to X^*$ (the polymorphic identity)
$\vdash (X^* \to f(Y^{X^*}) \to Y^{X^*})\bullet int^* \bullet f(3^{int^*}) : int^* \vdash X^* \to * : \Box$

in $\rho\underline{\omega}$ $\quad \vdash X^* \to X^* : X^* \to *$
$\vdash (X^* \to X^*)\bullet int^* : (X^* \to *)\bullet int^* =_\rho *$

in ρECC $\vdash X^\Box \to \Box : \Box_1$ and $\vdash X^\Box \to X^\Box : X^\Box \to \Box$
$\vdash (X^\Box \to X^\Box)\bullet Y^\Box : (X^\Box \to \Box)\bullet Y^\Box =_\rho \Box$

3.1 The Failure of the Unicity of Typing

When dealing with typed calculi, one interesting property is the *unicity of typing* (up to conversion), which can be rephrased in our Rho Calculus as follows:

$$\vdash A : B \quad \wedge \quad \vdash A : B' \quad \Longrightarrow \quad B =_\rho B'$$

This property is verified in all systems defined in the λ-cube (or derivates); in ECC [Luo90], thanks to the type cumulability relation and the (cum) type rule, a weaker property, *i.e.* the *minimum type property*, holds:

$$\exists B!\ \forall B' \quad \vdash A : B \quad \wedge \quad \vdash A : B' \quad \Longrightarrow \quad B \preceq B'$$

[3] Following the Curry-Howard isomorphism, if A is considered as a set, $X \in A$, and p, q are predicates on A, then $p(X^A), q(X^A)$ are types, considered as propositions.

[4] This proposition states that the predicate p (on A) considered as a set is included in the predicate q.

[5] This proposition states the reflexivity of the inclusion.

[6] The subject of this statement gives a "proof" of the reflexivity of the inclusion.

[7] In this case, the predicate of the judgment considered as proposition says: *ex falso sequitur quodlibet*, *i.e.* anything follows from a false judgment: the subject of this judgment is its proof.

The type systems in the right-hand side of the ρ-cube does not satisfies those two properties. If we take, for example in ρLF, the ρ-term $X^{Y^*} \to Y^*$, can be typed with both $*$ and $X^{Y^*} \to *$. Since in the Rho Calculus the $\to$ unifies the λ-cube symbols "λ" and "Π", it is natural to interpret the term $\vdash X^{Y^*} \to Y^*$ in two different ways:

$$\vdash X^{Y^*} \to Y^* : \begin{cases} * & \textit{i.e.} \text{ the term is an "arrow-type"} \\ X^{Y^*} \to * & \textit{i.e.} \text{ the term is a "constructor for dependent-types"} \end{cases}$$

In fact the translation of the above term into the λ-cube is ambiguous:

$$[\![X^{Y^*} \to Y^*]\!] = \begin{cases} \Pi X{:}Y.Y \text{ and } Y : * \vdash_{\lambda\to} \Pi X{:}Y.Y : * \\ \lambda X{:}Y.Y \text{ and } Y : * \vdash_{\lambda LF} \lambda X{:}Y.Y : \Pi X{:}Y.* \end{cases}$$

However, we conjecture that unicity of typing is valid in the left-hand side type systems of the cube, *i.e.* the ones without types dependent on terms.

4 Some Conjectures and Conclusion

4.1 Conjectures

The full proof-theoretical work for the ρ-cube has to be done: however, to stimulate the interested reader, we would like to sketch some conjectures that should be proved in a future work. We denote $\vdash_{\mathcal{S}}$ as the symbol of derivability in one of the type systems $\mathcal{S}$ of the ρ-cube.

Property 1 (Consistency). The ρ-cube is logically consistent, *i.e.* given a closed term A, we have $\not\vdash_{\mathcal{S}} A : \bot$.

Property 2 (Correctness of Typing (in ρECC)). If $\vdash_{\mathcal{S}} A : B$, then $\vdash_{\rho ECC} B : C$.

This property states in fact that we can find a correct derivation in ρECC for any predicate of a judgment in $\mathcal{S}$. It is crucial to prove subject reduction.

Property 3 (Weak Uniqueness of Typing). If $\vdash_{\mathcal{S}} A : B$ and $\vdash_{\mathcal{S}} A : B'$ and $B \neq_\rho s \neq_\rho B'$, then $B =_\rho B'$.

This property states that, for the simple terms of the ρCal, we can find a principal type. It is crucial for the decidability of type checking.

Property 4 (Subject Reduction). If $\vdash_{\mathcal{S}} A : B$, and $A \mapsto\!\!\!\!\twoheadrightarrow C$, then $\vdash_{\rho ECC} C : B$ and there exists B' such that $\vdash_{\mathcal{S}} C : B'$ and $B \mapsto\!\!\!\!\twoheadrightarrow B'$.

This property proves that types are preserved over reduction.

Property 5 (Strong Normalization of Typable Terms). If $\vdash_{\mathcal{S}} A : B$, then A and B are strongly normalizing.

This property states that every typable term normalizes. It will be proved following either the lines of [CH88,Coq91], or in a modular fashion, by first translating the result of strong normalization for $\lambda\omega$ [Gir72] into $\rho\omega$, and then using a translation function from $\rho\omega$ into ρECC, in the style of [HHP92,GN91].

Property 6 (Decidability of Type Checking). Given a Theory $\mathbb{T}$ and a type system $\vdash_{\mathcal{S}}$, it is decidable whether $\vdash_{\mathcal{S}} A : B$ for a given A, and B.

This property is probably true for the theory $\mathbb{T}_{\emptyset}$ and it heavily depends on the decidability of the theory $\mathbb{T}$ considered. In fact, the $(Conv)$ type rule uses the relation $=_{\rho}$, which in turn depends on the theory $\mathbb{T}$ considered, which in turn depends on a suitable type derivation (see Definition 2.2). This sort of "circularity" might be a possible source of undecidability.

4.2 Conclusion

In this work we have defined a very powerful collection of nine (8+1) type systems whose main originality consists in using only one abstractor, namely the "$\rightarrow$" operator of the Rho Calculus, and matching as a primitive mechanism, leading thus to a behavior similar to the Π-reduction for types. We have enlighten with several examples the usefulness of such a calculus, its expressive power as well as its ability to avoid the drawbacks of the previous systems differentiating term-abstractors (*i.e.* "λ") from type-abstractors (*i.e.* "Π"). The full proof-theoretical work for the ρ-cube is an ongoing work in progress.

This work opens many questions that we intend to solve now. They go from standard (and important) ones like proving subject reduction and strong normalization of typable terms, to extensions of the type system in order to take heterogeneous structures into account, and to the study of the more general setting of Pure Type Systems [Ber88,Ber90]. We would also like to explore, following the Curry-Howard isomorphism, the possibility to define a "logical" ρ-cube.

Acknowledgement. The authors would like to thank Gilles Barthe and Fairouz Kamareddine for pointing out some crucial papers and the anonymous referees for their useful comments. We would also like to thank all the members of the ELAN group for their comments and interactions on the topics of the Rho Calculus.

References

[Bar84] H. Barendregt. *Lambda Calculus: its Syntax and Semantics.* North Holland, 1984.

[Bar92] H. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, volume II, pages 118–310. Oxford University Press, 1992.

[Ber88] S. Berardi. Towards a Mathematical Analysis of Type Dependence in Coquand–Huet Calculus of Constructions and the Other Systems in Barendregt's Cube. Technical report, Dept. of Computer Science, Carnegie Mellon University, and Dip. di Matematica, Università di Torino, 1988.

[Ber90] S. Berardi. *Type Dependence and Constructive Cathematics*. PhD thesis, Dipartimento di Matematica, Università di Torino, 1990.

[Bür89] H.-J. Bürckert. Matching–A Special Case of Unification? *Journal of Symbolic Computation*, 8(5):523–536, 1989.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[Chu41] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1941.

[Cir00] H. Cirstea. *Calcul de Réécriture : Fondements et Applications*. Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 2000.

[CK99a] H. Cirstea and C. Kirchner. An Introduction to the Rewriting Calculus. Research Report RR-3818, INRIA, 1999.

[CK99b] H. Cirstea and C. Kirchner. Combining Higher-Order and First-Order Computation Using ρ-calculus: Towards a Semantics of ELAN. In *Frontiers of Combining Systems 2*, pages 95–120. Wiley, 1999.

[CK00] H. Cirstea and C. Kirchner. The Typed Rewriting Calculus. In *Third International Workshop on Rewriting Logic and Application*, 2000.

[CKL00] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. Research Report A00-RR-363, LORIA, 2000. Submitted.

[Coq91] T. Coquand. Metamathematical Investigations of a Calculus of Constructions. In *Logic and Computer Science*, pages 91–122. Academic Press, 1991.

[dB80] N. G. de Bruijn. A Survey of the Project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[Gir72] J.Y. Girard. *Interpretation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. PhD thesis, Université Paris VII, 1972.

[Gir86] J.Y. Girard. The System F of Variable Types, Fifteen Years Later. *Theoretical Computer Science*, 45:159–192, 1986.

[GN91] H. Geuvers and M.J. Nederhof. A Modular Proof of Strong Normalization for the Calculus of Constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.

[HHP92] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, 1992.

[Hul79] J.-M. Hullot. Associative-Commutative Pattern Matching. In *Proc. of IJCAI*, 1979.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving Equations in Abstract Algebras: A Rule-based Survey of Unification. In *Computational Logic. Essays in Honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, 1991.

[KBN99] F. Kamareddine, R. Bloo, and R. Nederpelt. On π-conversion on the λ-cube and the Combination with Abbreviations. *Annals of Pure and Applied Logics*, 97(1-3):27–45, 1999.

[KK99] C. Kirchner and H. Kirchner. Rewriting, Solving, Proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[KL01] F. Kamareddine and T. Laan. A Correspondence between Martin-Löf Type Theory, the Ramified Theory of Types and Pure Type Systems. *Journal of Logic, Language and Information*, 2001. To appear.

[KN96] F. Kamareddine and R. Nederpelt. Canonical Typing and π-conversion in the λ-cube. *Journal of Functional Programming*, 6(2):85–109, 1996.

[Luo90] Z. Luo. ECC: An Extended Calculus of Constructions. In *Proceedings of LICS*, pages 385–395, 1990.

[Mar84] P. Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, Naples, 1984.

[PM97] S.L. Peyton Jones and E. Meijer. Henk: a Typed Intermediate Language. In *Types in Compilation Workshop*, 1997.

Type Inference with Recursive Type Equations

Mario Coppo

Dipartimento di Informatica, Università di Torino,
Corso Svizzera 185, 10149 Torino, Italy
coppo@di.unito.it
http://www.di.unito.it/∼coppo

Abstract. This paper discusses decidability and existence of principal types in type assignment systems for λ-terms in which types are considered modulo an equivalence relation induced by a set of type equations. There are two interesting ways of defining such equivalence, an initial and a final one. A suitable transformation will allow to treat both in an uniform way.

Keywords: Recursive types, type assignment.

1 Introduction

In this paper we study type inference problems in the basic type inference system for the λ-calculus ([5], [8]) extended with recursive type definitions. For example, assuming the existence of a type constant *nil* for the one element type and of type constructors + for disjoint union and $\times$ for cartesian product, the type of integer lists can be specified by the equation

$$int\text{–}list = nil + (int \times int\text{–}list)$$

An alternative way of denoting the same type is via the use of an explicit operator μ or recursion over types. In this case the former type is denoted by $\mu t.nil + (int \times t)$. Although the use of recursive equations is closer to the way in which recursive types are introduced in programming languages like ML ([9]) most of the technical literature on the subject has been focused on the μ-notation (see e.g. [12]). Although the two notations are equivalent in many contexts, there are interesting aspects in which they are not such. Taking recursive definitions we specify in advance a finite number of recursive types that we can use for typing a term, whereas taking μ-types we are allowed to use arbitrary recursive types. The introduction of recursive types via recursive definitions rises questions about typability of terms of the kind:
- Given a term M and a set $\mathcal{R}$ of recursive definitions can M be typed from R?
- Can the notion of principal type scheme ([8]) be generalized to type assignment with respect to $\mathcal{R}$?

While some general results about typability of terms using μ-types are well established in the literature (see for instance [3]) there seems to be much less about recursive definitions.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 184–198, 2001.

Following [7], we start by considering the notion of "type structure" which is a set T of types equipped with a congruence relation $\simeq$ with respect to the $\rightarrow$ type constructor. Type structures are usually defined starting from a set of equations between types of the shape $B = C$, where B and C are simple type expressions. There are two basic ways of defining an equivalence relation from a set of type equations: an initial one in which equivalence is defined via simple equational reasoning and a final one which amounts to consider equivalent two types if they have the same interpretation as infinite trees. The latter is, for instance, the notion of type equivalence determined by the type checking algorithms of most programming languages, starting from ALGOL60. Tree equivalence is also the notion of equality induced by the interpretations of types in continuous models (see [3]). We will define a transformation to reduce tree equivalence to the equational one. This allows to treat both equivalences in an uniform way.

In this paper we take $\rightarrow$ as the only type constructor, but this is enough to define quite interesting type assignments. For example it is well known that, assuming a type c such that $c = c \rightarrow A$ (where A is any type) one can assign type $(A \rightarrow A) \rightarrow A$ to the fixed point combinator $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, permitting to type recursive functions over values without assuming an "ad-hoc" constant for this. Other type constructors, as those used in the introductory example, can be added easily.

The basic definitions and properties about type structures and type equalities are given in sections 2 while type assignment is introduced in section 3. Section 4 is about some basic properties of type equivalence. The results concerning type assignment are given in section 5. The basic source for this paper is [11] in which some basic properties of recursive definition are established. Type inference, however, is not discussed in that paper.

2 Type Structures and Type Constraints

Following [7], we start by defining the notion of type structure. This notion was first motivated by Scott [10] and formally developed in Breazu Tannen and Meyer [2].

Type Structures

The main feature of recursive types is that one makes identifications between them. In a type structure types are taken modulo a congruence relation.

Definition 1. *Let* T *be a set of syntactic objects (types) closed under the* $\rightarrow$ *type constructor. A* type structure *over* T *is a pair* $\mathcal{T} = \langle \mathsf{T}, \simeq \rangle$ *where* $\simeq$ *is a congruence over* T *(i.e. an equivalence relation such that* $A \simeq A'$ *and* $B \simeq B'$ *implies* $A \rightarrow A' \simeq B \rightarrow B'$*).*

We mainly consider a set T of types built from a denumerable set $\mathbf{A}$ of *atomic* types by the $\rightarrow$ type constructor[1]. We will sometime write $\mathsf{T}(\mathbf{A})$ to make explicit

[1] There are other interesting sets of types that naturally build a type structure like that of μ-types defined in the introduction.

the set of atomic types we have started from. The notion of atomic type here is only a syntactic one: an atomic type can be equivalent via $\simeq$ to a non atomic one.

The set T of simple types is isomorphic to *free* type structure $\langle \mathsf{T}, \equiv \rangle$, that will be identified with T itself.

Definition 2.
(i) *A mapping* $h : \mathsf{T} \to \mathsf{T}$ *is a* type homomorphism *if* $h(A \to B) = h(A) \to h(B)$.
(ii) *Let* $\mathcal{T} = \langle \mathsf{T}, \simeq \rangle$ *and* $\mathcal{T}' = \langle \mathsf{T}, \simeq' \rangle$ *be type structures. A type homomorphism* $h : \mathsf{T} \to \mathsf{T}$ *is a* type structure homomorphism *(written also* $h : \mathcal{T} \to \mathcal{T}'$*) if* $A \simeq B \ \Rightarrow \ h(A) \simeq' h(B)$.

Remark 1. Note that if $\mathcal{T} = \langle \mathsf{T}(\mathbf{A}), \simeq \rangle$, any type homomorphism $h : \mathcal{T} \to \mathcal{T}'$, where $\mathcal{T}'$ is any other type structure, is uniquely determined by its restriction on the atomic types in $\mathbf{A}$. We will then define an homomorphism h by giving only its value on the atomic types.

A substitution (in the usual sense) is a type homomorphism. A substitution is a homomorphism between free type structures but not, in general, between arbitrary ones. We denote with $[a_1 := A_1, \ldots, a_n := A_n]$ the type homomorphism $h : \mathsf{T} \to \mathsf{T}$ determined by $h(a_1) = A_1, h(a_n) = A_n$ and $h(a) = a$ for all other atomic types $a \notin \{a_1, \ldots, a_n\}$. In this case e say that h is a *finite* type homomorphism.

Type structure homomorphisms are closed under composition (indeed type structures with homomorphisms form a category).

Definition 3 (Invertibility). *Let* $\langle \mathsf{T}, \simeq \rangle$ *be a type structure. Then* $\simeq$ *is said to be* invertible *if* $(A \to B) \simeq (A' \to B') \ \Rightarrow \ A \simeq A'$ *and* $B \simeq B'$.
We also say that the relation $\simeq$ *over* T *is* invertible.

Invertibility holds, for instance, in a free type structure.

Type Constraints and Recursive Definitions

A very natural way of generating type structures is to assume a (finite) set of equations between types and to take the congruence generated by it via equational reasoning. A stronger way of generating a congruence between types (interpreting them in the domain of trees) will be considered in the next section.

Definition 4. *A* system of type constraints *over* T *is a set* $\mathcal{C} = \{A_i = B_i \mid 1 \leq i \leq n\}$ *of formal equations between types in* T.

A system of type constraints determines, via standard equational reasoning, a congruence between types.

Definition 5. *Let* $\mathcal{C} = \{A_i = B_i \mid 1 \leq i \leq n\}$*, be a system of type constrains over* T*. The congruence relation determined by* $\mathcal{C}$ *(notation* $=_{\mathcal{C}}$*) is the least relation satisfying the following axioms and rules:*

$$(eq)\quad \vdash A =_{\mathcal{C}} B \ \text{ if } \ A = B \in \mathcal{C} \qquad (ident)\ \vdash A =_{\mathcal{C}} A$$

$$(symm)\ \frac{\vdash A =_{\mathcal{C}} B}{\vdash B =_{\mathcal{C}} A} \qquad (trans)\ \frac{\vdash A =_{\mathcal{C}} B \qquad \vdash B =_{\mathcal{C}} C}{\vdash A =_{\mathcal{C}} C}$$

$$(\rightarrow)\ \frac{\vdash A =_{\mathcal{C}} A' \qquad \vdash B =_{\mathcal{C}} B'}{\vdash A \rightarrow B =_{\mathcal{C}} A' \rightarrow B'}$$

We call $=_{\mathcal{C}}$ *the* equational *equivalence over* T *induced by* $\mathcal{C}$*. Let* $\mathcal{T}_{\mathcal{C}}$ *do denote the type structure* $\langle \mathsf{T}, =_{\mathcal{C}} \rangle$*.*

The relation $=_{\mathcal{C}}$ is the minimal congruence over T generated by the equations in $\mathcal{C}$. A basic related notion is the following.

Definition 6. *A type structure* $\mathcal{T} = \langle \mathsf{T}', \simeq \rangle$ solves *a system of type constraints* $\mathcal{C}$ *over* T *if there is a type structure homomorphism* $h : \mathcal{T}_{\mathcal{C}} \rightarrow \mathcal{T}$*. We also say that* h solves $\mathcal{C}$ *in* $\mathcal{T}$*.*

Note that h solves $\mathcal{C}$ in $\mathcal{T} = \langle \mathsf{T}', \simeq \rangle$ iff $h(A) \simeq h(B)$ for all $A = B \ \in \ \mathcal{C}$.

In the definition of recursive types we are particularly interested in the type structures generated by a system of type constraints of the shape $c = C$ where c is an atom and C is a non atomic type expression. This corresponds to the idea of defining type c as equivalent to a type expression C containing possibly c itself. More formally:

Definition 7 (Simultaneous recursion).

(i) A system of type constraints $\mathcal{R}$ *over* T *is a* simultaneous recursion *(s.r. for short) if it has the form* $\mathcal{R} = \{c_i = C_i \mid 1 \leq i \leq n\}$*. where*

1. *for all* $1 \leq i \leq n$*,* c_i *is atomic and* $C_i \in \mathsf{T}$*;*
2. $c_i \not\equiv c_j$ *for all* $i \neq j$*;*
3. *if* $C_i \equiv c_j$ *for some* $1 \leq i, j \leq n$ *then* $i < j$*.*

We call $\{c_1, \ldots, c_n\}$ *the* domain *of* $\mathcal{R}$*, and denote it by* $dom(\mathcal{R})$*. The atoms* $c_1, \ldots, c_n$ *are called the* indeterminates *of the s.r..*

(ii) A s.r. is proper *if* C_i *is not atomic for all* $1 \leq i \leq n$*.*

Example 1. Let $\mathcal{R}_1 = \{c_1 = t \rightarrow c_1\}$ where t is a type variable is a s.r. defining a type c such that $c_1 =_{\mathcal{R}_1} t \rightarrow c_1 =_{\mathcal{R}_1} t \rightarrow t \rightarrow c_1 \ldots$ and so on. Define now $\mathcal{R}_2 = \{c_2 = t \rightarrow t \rightarrow c_2\}$. Note that $c_2 \neq_{\mathcal{R}_2} t \rightarrow c_2$. Indeed if we take $\mathcal{R}_3 = \mathcal{R}_1 \cup \mathcal{R}_2$ we have that $c_1 \neq_{\mathcal{R}_3} c_2$.

In general an s.r. satisfies point 3. above if it can be defined a total order $<$ on its domain such that if there is an equation $c = c' \in \mathcal{R}$ then $c < c'$. This restriction is introduced to rule out circular equations like $c = c$ or $c_1 = c_2, c_2 =$

c_1 which have no interesting meaning. Notice that in a s.r. we can eliminate any equation of the shape $c_i = c_j$ $(i \neq j)$ simply by replacing c_i by c_j in $\mathcal{R}$ (or vice versa).

We call *unfolding* the operation of replacing c_i by C_i in a type and *folding* the reverse operation. So, two types are weakly equivalent w.r.t. $\mathcal{C}$ if they can be transformed one into the other by a finite number of applications of the operations folding and unfolding.

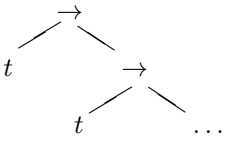
Definition 8. *Let $\mathcal{R}$ be a s.r. over T. The* kernel *of $\mathcal{R}$ is the set $ker(\mathcal{R}) \subseteq \mathcal{R}$ containing all equations $c = C \in \mathcal{R}$ such that $c =_{\mathcal{R}} A$ for some non atomic type A containing at least one occurrence in c.*

It is a simple exercise to design an algorithm to test whether an indeterminate c is in the kernel of $\mathcal{R}$. The kernel of $\mathcal{R}$ determines essentially the set of new types defined via $\mathcal{R}$. An equation $c = C$ which is not in $ker(\mathcal{R})$ is simply a way of giving "name" c to type C, but is uninteresting from our point of view.

The Tree Equivalence of Types

Consider Example 1 above. Types like c_1 and c_2, although not equivalent with respect to $=_{\mathcal{R}_3}$, seem to express the same informal behaviour. Indeed c_1 and c_2 represent to the same (infinite) type when the they are "pushed down" by repeated steps of unfolding. Another notion of equivalence between recursive type expressions can be defined by regarding them as finitary descriptions of infinite trees: this approach is followed systematically in [3] and is indeed the most popular way of considering type equivalence in programming languages ([12]).

For the basic notions about infinite and regular trees we refer to [4]). Recall that an infinite trees is *regular* if it has only a finite number of subtrees. An example of regular tree is the following tree α_0:



Indeed α_0 has only two subtrees, t and α_0 itself.

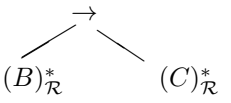
Let $\mathsf{Tr}^R(\mathbf{A})$ denote the set of regular trees defined starting from a set $\mathbf{A}$ of atomic types ($\mathbf{A}$ will be omitted when understood). Let $\mathsf{Tr}^F(\mathbf{A}) \subseteq \mathsf{Tr}^R(\mathbf{A})$ be the set of *finite* trees. $\mathsf{Tr}^F(\mathbf{A})$ and $\mathsf{T}(\mathbf{A})$ are isomorphic and will be identified.

Note that $\langle \mathsf{Tr}^R, \equiv \rangle$ is a type structure, that we identify with Tr^R. It is immediate to verify that Tr^R is invertible.

There is a standard way of interpreting recursive types as infinite trees.

Definition 9. *Let $\mathcal{R}$ be a s.r. over T. Let $(-)^*_{\mathcal{R}} : \mathcal{T}_{\mathcal{R}} \to \mathsf{Tr}^R$ be the type structure homomorphism defined in the following way:*

1. *If $A \equiv a$ for some $a \in \mathbf{A}$ such that $a \notin dom(\mathcal{R})$ then $(A)^*_{\mathcal{R}}$ is a leaf a.*
2. *If $A \equiv B \to C$ for some $B, C \in \mathsf{T}$ then $(A)^*_{\mathcal{R}}$ is*



3. *If $A \equiv c$ for some $c \in dom(\mathcal{R})$ such that $c = C \in \mathcal{R}$ then $(c)^*_{\mathcal{R}} = (C)^*_{\mathcal{R}}$.*

Note that this is not an inductive definition but a co-inductive one since C in case 3. is in general more complex than c. Note that, owing to condition 3. of Definition 7, $(A)^*_{\mathcal{R}}$ is always well defined.

It is well known ([4]) that if $\mathcal{R}$ is finite $(A)^*_{\mathcal{R}}$ is a regular tree for all $A \in \mathsf{T}$. The map $(-)^*_{\mathcal{R}}$ induces the following notion of equivalence between types.

Definition 10. (i) *Let $\mathcal{R}$ be a s.r over T. Let $=^*_{\mathcal{R}}$ be the relation over T defined by $A =^*_{\mathcal{R}} B$ if $(A)^*_{\mathcal{R}} = (B)^*_{\mathcal{R}}$. We call $=^*_{\mathcal{R}}$ the* tree equivalence *over T induced by $\mathcal{R}$.*

(ii) *Let $\mathcal{T}^*_{\mathcal{R}} = \langle \mathsf{T}, =^*_{\mathcal{R}} \rangle$ be the type structure determined by $=^*_{\mathcal{R}}$.*

It is easy to see that $=^*_{\mathcal{R}}$ is a congruence with respect to $\to$ and hence that $\langle \mathsf{T}, =^*_{\mathcal{R}} \rangle$ is well defined and $=_{\mathcal{R}} \subseteq =^*_{\mathcal{R}}$. Then there is a (unique) homomorphism $\mathsf{i}^* : \mathcal{T}_{\mathcal{R}} \to \mathcal{T}^*_{\mathcal{R}}$ which coincides with the identity at the type level.

Referring to example 1 it is easy to see that $c_2 =^*_{\mathcal{R}_2} t \to c_2$ and that $c_1 =^*_{\mathcal{R}_3} c_2$.

Clearly $=^*_{\mathcal{R}}$ has a more semantic nature than $=_{\mathcal{R}}$ and is indeed the type equivalence induced by the interpretation of types in continuous models ([3]). Axiomatizations of $=^*_{\mathcal{R}}$ are usually given using some co-induction principle (see for instance [1]). However $=^*_{\mathcal{R}}$ can be also characterized as the equational theory of a suitable s.r. $\mathcal{R}^*$.

Theorem 1. *Let $\mathcal{R}$ be an s.r. over T. Then there is s.r. $\mathcal{R}^*$ such that:*

(i) *For all $A, B \in \mathsf{T}$ $A =^*_{\mathcal{R}} B$ iff $A =_{\mathcal{R}^*} B$.*

(ii) *$\mathcal{T}^*_{\mathcal{R}}$ is isomorphic to $\mathcal{T}_{\mathcal{R}^*}$.*

We can give here only an example of this construction. The complete proof will be contained in a forthcoming paper. Let $\mathcal{R} = \{c_1 = t \to t \to c_1\}$ we proceed through the following steps:

1. We first transform $\mathcal{R}$ in its *flat* form $\mathsf{f}(\mathcal{R})$, in which every equation is of the shape $c = a$ where $a \in \mathbf{A}$ or $c = d \to e$ where both c and d are indeterminates. This may require the introduction of new indeterminates. In the case of our example we have $\mathsf{f}(\mathcal{R}_1) = \{c_1 = d_1 \to d_2, d_1 = t, d_2 = d_3 \to c_1, d_3 = t\}$, where d_1, d_2, d_3 are new indeterminates.

2. Then define by an iterative construction a relation $\approxeq$ on the indeterminates of $\mathsf{f}(\mathcal{R})$ such that $a \approxeq b$ if and only if a and b have the same infinite unfolding (this is the crucial step of the construction). In our example we get $c_1 \approxeq d_2$ and $d_1 \approxeq d_3$.

3. Now apply to $\mathsf{f}(\mathcal{R})$ a transformation $\mathsf{c}(-)$ that identifies (by adding suitable equations) all the indeterminates that are equated by $\cong$. In our case we get $\mathsf{c}(\mathsf{f}(\mathcal{R})) = \{c_1 = d_1 \to c_1, d_2 = c_1, d_1 = t, d_3 = d_1\}$.

4. Lastly apply to $\mathsf{c}(\mathsf{f}(\mathcal{R}))$ a transformation $\mathsf{s}(-)$ which removes all the indeterminates introduced by $\mathsf{f}(-)$ in step 1. and define $\mathcal{R}^* = \mathsf{s}(\mathsf{c}(\mathsf{f}(\mathcal{R})))$. In our case we get $\mathcal{R}^* = \{c_1 = t \to c_1\}$.

It is easy to see that $\mathcal{R}^*$ has the same indeterminates of $\mathcal{R}$ and satisfies Theorem 1.

3 λ-Calculi with Recursive Types

Type assignment in a generic type structure $\mathcal{T} = \langle \mathsf{T}, \simeq \rangle$ is defined by introducing a rule to consider types modulo the equivalence determined by $\mathcal{T}$.

The judgements of the inference system are of the shape $\Gamma \vdash_{\mathcal{T}} M : A$ where Γ denotes a *type environment*, i.e. a set of assumptions of the shape $x : A$, where x is a term variable and A a type.

Definition 11. *Let $\mathcal{T} = \langle \mathsf{T}, \simeq \rangle$ be a Type structure and $A, B \in \mathsf{T}$. Then the inference rules for assigning types to λ-terms from $\mathcal{T}$ are the following:*

$$(ax)\ \Gamma, x : A \vdash_{\mathcal{T}} x : A \qquad (\to E)\ \frac{\Gamma \vdash_{\mathcal{T}} M : A \to B \qquad \Gamma \vdash_{\mathcal{T}} N : A}{\Gamma \vdash_{\mathcal{T}} (M\ N) : B}$$

$$(\to I)\ \frac{\Gamma, x : A \vdash_{\mathcal{T}} M : B}{\Gamma \vdash_{\mathcal{T}} \lambda x.M : A \to B} \qquad (equiv)\ \frac{\Gamma \vdash_{\mathcal{T}} M : A \quad A \simeq B}{\Gamma \vdash_{\mathcal{T}} M : B}$$

If $\mathcal{C}$ is a system of type constraints, we simply write $\Gamma \vdash_{\mathcal{C}} M : A$ for $\Gamma \vdash_{\mathcal{T}_{\mathcal{C}}} M : A$ and $\Gamma \vdash^*_{\mathcal{C}} M : A$ instead of $\Gamma \vdash_{\mathcal{T}^*_{\mathcal{C}}} M : A$ (in case $\mathcal{C}$ is a s.r.).

Type assignment in different type structures can be related via the notion of type structure homomorphism. The following property can easily be proved by induction on derivations.

Lemma 1. *Let $h : \mathcal{T} \to \mathcal{T}'$ be a type structure homomorphism. Then*

$$\Gamma \vdash_{\mathcal{T}} M : A \quad \Rightarrow \quad h(\Gamma) \vdash_{\mathcal{T}'} M : h(A)$$

where $h(\Gamma)$ is obtained by applying h to all type occurrences in Γ.

For example since there is a homomorphism $\mathsf{i}^* : \langle \mathsf{T}, =_{\mathcal{R}} \rangle \to \langle \mathsf{T}, =^*_{\mathcal{R}} \rangle$ we have that any term which can be typed in $\mathcal{T}_{\mathcal{R}}$ can also be typed (with the same types) in $\mathcal{T}^*_{\mathcal{R}}$.

A remarkable property of type assignment is subject reduction which states that typings of terms are preserved by β-reduction. The following property characterizing the type systems in which subject reduction hold has been proved by R. Statman in [11].

Theorem 2. *A type assignment system $\vdash_{\mathcal{T}}$ has the subject reduction property iff $\mathcal{T}$ is invertible.*

We will see in the next section that if $\mathcal{C}$ is a system of type constraints $=_{\mathcal{C}}$ is invertible iff $\mathcal{C}$ is equivalent to a s.r.. Then it is mostly interesting to study assignment in type structures determined by a s.r..

4 Some Properties of Simultaneous Recursions

In this section we state some properties of the equivalence between types induced by a s.r. or by a systems of type constraints. We are mainly concerned in studying the solvability of a system of type constraints in a s.r.. Many results of this section are essentially due to R. Statman ([11]).

The Term Rewriting System Associated to a S.R.

Definition 12. (i) *Let $\mathcal{C}$ and $\mathcal{C}'$ be two systems of type constraints over the same set $\mathsf{T}_{\mathbf{A}}$ of types. $\vdash_{\mathcal{C}} \mathcal{C}'$ means that $\vdash_{\mathcal{C}} A{=}B$ for all equations $A{=}B \in \mathcal{C}'$.*
(ii) *$\mathcal{C}$ and $\mathcal{C}'$ are* equivalent *(notation $\mathcal{C} \sim \mathcal{C}'$) if $\vdash_{\mathcal{C}} \mathcal{C}'$ and $\vdash_{\mathcal{C}'} \mathcal{C}$.*

Note that $\mathcal{C}$ is equivalent to $\mathcal{C}'$ if $=_{\mathcal{C}}$ and $=_{\mathcal{C}'}$ coincide.

Given a s.r. $\mathcal{R}$ we define now a Church-Rosser and strongly normalizing term rewriting system which generates $=_{\mathcal{R}}$. The construction is a slight modification of the original one given by Statman [11], to which we refer for the proofs. A more structured but less direct approach is given in Marz [7].

Definition 13. *Let $\mathcal{R} = \{c_i = C_i \mid 1 \leq i \leq n\}$ be an s.r. over T. The rewriting system Trs($\mathcal{R}$) contains a rewriting rules $C_i \rightsquigarrow c_i$ for all $c_i = C_i \in \mathcal{R}$.*

Note that $=_{\mathcal{R}}$ is the convertibility relation over $\mathsf{T} \times \mathsf{T}$ generated by Trs($\mathcal{R}$).

It is easy to see that Trs($\mathcal{R}$) is strongly normalizing, because each contraction either decreases the size of the type to which it is applied or is of the shape $c_j \rightsquigarrow c_i$ where $i < j$. However, $\rightsquigarrow$ it is not, in general, Church-Rosser.

Example 2. Let $\mathcal{R} = \{c_0 = c_0 \to c_2,\ c_1 = (c_0 \to c_2) \to c_2,\ c_2 = c_0 \to c_1\}$. Then Trs($\mathcal{R}$) consists of the rules $\{c_0 \to c_2 \rightsquigarrow c_0,\ (c_0 \to c_2) \to c_2 \rightsquigarrow c_1,\ c_0 \to c_1 \rightsquigarrow c_2\}$. Observe that the l.h.s. of the first equation is a subterm of the l.h.s. of the second one. In particular $(c_0 \to c_2) \to c_2$ can be reduced both to c_1 and to $c_0 \to c_2$ which further reduces to c_0: it has then two distinct normal forms c_1 and c_0. Therefore Trs($\mathcal{R}$) is not confluent.

Expressions like $c_0 \to c_2$ and $(c_0 \to c_2) \to c_2$ (called *critical pairs* in the literature on term rewriting systems [6]) destroy confluence. By a well known result of Knuth and Bendix (see [6, Corollary 2.4.14]) a strongly normalizing term rewriting system without critical pairs is Church-Rosser.

The following algorithm, which amounts to Knuth-Bendix completion (see [6]), transforms any s.r. into an equivalent one without critical pairs which has, therefore, the Church-Rosser property.

Definition 14 (Completion of $\mathcal{R}$).
(i) *Let $\mathcal{R}$ be a s.r.. We define by induction on n a sequence of s.r. $\mathcal{R}_n$ ($n \geq 0$). Let $\mathcal{R}_0 = \mathcal{R}$. Define $\mathcal{R}_{n+1}$ from $\mathcal{R}_n$ ($n \geq 0$) in the following way:*

1. if there exists a pair of equations $c_i = C_i, c_j = C_j \in \mathcal{R}_n$ such that C_j is not atomic and is a proper subexpression of C_i take

$$\mathcal{R}_{n+1} = (\mathcal{R}_n - \{c_i = C_i\}) \cup \{c_i = C_i^*\}$$

where C_i^* is the result of replacing all occurrences of C_j in C_i by c_j.

2. If there exist two equations $c_i = C, c_j = C \in \mathcal{R}_n$ then, assuming $i < j$, take

$$\mathcal{R}_{n+1} = \mathcal{R}_n[c_i := c_j] \cup \{c_i = c_j\}.$$

3. otherwise take $\mathcal{R}_{n+1} = \mathcal{R}_n$.

(ii) *Let N be the least n such that $\mathcal{R}_{n+1} = \mathcal{R}_n$ (this value must certainly exist since, in both steps 1 and 2 the total number of symbols in $\mathcal{R}_n$ strictly decreases). Let $Trs^+(\mathcal{R}) = Trs(\mathcal{R}_N)$.*

It is easy to prove by induction on n that for all $n \geq 0$ $\mathcal{R}_n$ is a s.r. equivalent to $\mathcal{R}$. Then $\mathrm{Trs}(\mathcal{R}_N)$ has no critical pairs (by construction) and generates the same equality as $\mathcal{R}$.

Applying this construction to Example 2 get $N = 2$ and the resulting s.r. is $\mathcal{R}_2 = \{c_1 = c_0 \to c_2,\ c_2 = c_0 \to c_0,\ c_0 = c_1\}$ and then we get
$\mathrm{Trs}^+(\mathcal{R}) = \{c_0 \to c_2 \rightsquigarrow c_1,\ c_0 \to c_0 \rightsquigarrow c_2,\ c_1 \rightsquigarrow c_0\}$.

Lemma 2. *Let $\mathcal{R}$ be a s.r.. Then $Trs^+(\mathcal{R})$ is strongly normalizing and Church-Rosser.*

Corollary 1. *Let $\mathcal{R}$ be a s.r.. Then $=_{\mathcal{R}}$ is decidable.* □

Using $\mathrm{Trs}^+(\mathcal{R})$ it can be proved the invertibility of $=_{\mathcal{R}}$ (see [11] for details).

Theorem 3. *Let $\mathcal{R}$ be a s.r.. Then $=_{\mathcal{R}}$ is invertible.*

Solving a S.R. in Another

It is sometimes useful to force a type system generated by a set of type constraints $\mathcal{C}$ to have the invertibility property.

Definition 15. *Let $\mathcal{C}$ be a system of type constraints over $\mathsf{T}_{\mathbf{A}}$. The relation $=_{\mathcal{C}}^{\mathrm{inv}}$ is defined by adding to the axioms and rules of Definition 5 the following rule for invertibility*

$$(inv)\ \frac{\vdash A_1 \to A_2 =_{\mathcal{C}}^{\mathrm{inv}} B_1 \to B_2}{\vdash A_i =_{\mathcal{C}}^{\mathrm{inv}} B_i}\ (i = 1, 2)$$

Then $=_{\mathcal{C}}^{\mathrm{inv}}$ is the least invertible congruence containing $=_{\mathcal{C}}$.

If A is type expression its *length* (denoted $|A|$) is defined as the number of symbols occurring in it. The following construction is essentially taken from ([11]).

Definition 16. *Let $\mathcal{C}$ a system of type constraints. As in Definition 14 we define by induction on n a sequence of sets of equations $\mathcal{C}_n$ ($n \geq 0$). We can assume without less of generality that there is a total order between the atomic types occurring in $\mathcal{C}$, that will be denoted by $c_1, c_2, \ldots$). Let $\mathcal{C}_0 = \mathcal{C}$. Define $\mathcal{C}_{n+1}$ from $\mathcal{C}_n$ ($n \geq 0$) in the following way:*

1. If there is an equation $A \to B = C \to D \in \mathcal{C}_n$ then take

$$\mathcal{C}_{n+1} = \mathcal{C}_n - \{A \to B = A' \to B'\} \cup \{A = A', B = B'\};$$

2. If there are two equations $c = A \to B, c = A' \to B' \in \mathcal{C}_n$, assuming $|A \to B| \leq |A' \to B'|$, then take

$$\mathcal{C}_{n+1} = \mathcal{C}_n - \{c = A' \to B'\} \cup \{A = A', B = B'\};$$

3. Otherwise take $\mathcal{C}_{n+1} = \mathcal{C}_n$.

Let N be the least n such that $\mathcal{C}_{n+1} = \mathcal{C}_n$ (it is easy to prove that value must certainly exist). Note that we can write $\mathcal{C}_N = \mathcal{R} \cup \mathcal{E}$ where $\mathcal{R}$ contains all equations of the shape $c = C$ such that C is not atomic and $\mathcal{E}$ is a set of equations of the shape $c_i = c_j$ such that both c_i and c_j are atomic. Now take the equivalence classes of the atomic types occurring in $\mathcal{E}$ with respect to the $\mathcal{E}$ itself (seen as a set of type constraints). For each equivalence class c with respect to $\mathcal{E}$ with more than one element choose as representative the element $c_j \in \mathsf{c}$ with the greatest index, replace with c_j each occurrence of an element of c in $\mathcal{R}$ and add to $\mathcal{R}$ an equations $c = c_j$ for each element $c \in \mathsf{c}$ different from c_j.

Lastly define $\mathcal{C}^{\mathrm{inv}}$ as the so obtained s.r..

It is immediate to prove, by induction on n, that for all $n \geq 0$ $\mathcal{C}_n \vdash \mathcal{C}$ and $\mathcal{C} \vdash^{\mathrm{inv}} \mathcal{C}_n$. We have immediately the following result.

Lemma 3. *Let $\mathcal{C}$ a system of type constraints. Then $\mathcal{C} \vdash^{\mathrm{inv}} \mathcal{C}^{\mathrm{inv}}$ and $\mathcal{C}^{\mathrm{inv}} \vdash \mathcal{C}$ (i.e. $\mathcal{C}^{\mathrm{inv}}$ is equivalent to $\mathcal{C}$ plus invertibility).*

From Theorem 3 we have immediately that a system of type constraint is invertible iff it is equivalent to a s.r.

Let now $\mathcal{C}$ be a system of type constraints over T. We say that a s.r. $\mathcal{R}$ over T' *solves* $\mathcal{C}$ if $\langle \mathsf{T}', =_{\mathcal{R}} \rangle$ solves $\mathcal{C}$ (see Definition 6). Another immediate consequence of Theorem 3 is the following.

Lemma 4. *Let $\mathcal{C}$ be a system of type constraints and $\mathcal{R}$ a s.r.. Then $h : \langle \mathsf{T}, =_{\mathcal{C}} \rangle \to \langle \mathsf{T}, =_{\mathcal{R}} \rangle$ iff $h : \langle \mathsf{T}, =^{\mathrm{inv}}_{\mathcal{C}} \rangle \to \langle \mathsf{T}, =_{\mathcal{R}} \rangle$, i.e. $\mathcal{R}$ solves $\mathcal{C}$ iff $\mathcal{R}$ solves $\mathcal{C}^{\mathrm{inv}}$.*

Owing to lemma 4 we are mainly interested to study solvability of a s.r. into another one. The following lemma is an extension of a result proved in [11].

Lemma 5. *Let $\mathcal{R} = \{c_i = C_i \mid 1 \leq i \leq n\}$ be a s.r. over T and let $a_1, \ldots, a_p$ ($p \geq 0$) the other variables occurring in $ker(\mathcal{R})$. If $\mathcal{S}$ is another s.r. over T any homomorphism $h : \langle \mathsf{T}, =_{\mathcal{R}} \rangle \to \langle \mathsf{T}, =_{\mathcal{S}} \rangle$ (solving $\mathcal{R}$ in $\mathcal{S}$) can be written as*

$$h = \overline{\overline{h}} \circ \overline{h}$$

where $\overline{h} : \langle \mathsf{T}, =_{\mathcal{R}} \rangle \to \langle \mathsf{T}, =_{\mathcal{S}} \rangle$ and $\overline{\overline{h}} : \langle \mathsf{T}, =_{\mathcal{S}} \rangle \to \langle \mathsf{T}, =_{\mathcal{S}} \rangle$ are such that:

1. $\overline{h} = [c_1 := B_1, \ldots, c_n := B_n, a_1 := A_1, \ldots, a_p := A_p]$ *is a finite homomorphism which solves* $\mathcal{R}$ *in* $\mathcal{S}$ *where both* $B_1, \ldots, B_n$ *and* $A_1, \ldots, A_n$ *are subexpressions of some* $Trs^+(\mathcal{R})-$*redex.*

2. $h' \circ \overline{h}$ *solves* $\mathcal{R}$ *in* $\mathcal{S}$ *for any homomorphism* $h' : \langle \mathsf{T}, =_{\mathcal{S}} \rangle \to \langle \mathsf{T}, =_{\mathcal{S}} \rangle$*;*
We say that $\overline{h}$ *is an* essential *solution of* $\mathcal{R}$ *in* $\mathcal{S}$.

Example 3. (i) Let $\mathcal{R}_0 = \{a = a \to b\}$ be an s.r.. We want to find a solution of $\mathcal{R}_0$ in the s.r. $\mathcal{R}_1 = \{c = c \to c\}$. By point 1. of the Lemma both $h(a)$ and $h(b)$ must be subterms of some redex in $\mathrm{Trs}^+(\mathcal{R}_1)$. Note that the only redex in $\mathcal{R}_1$ is $c \to c$. Indeed choosing $h_k(a) = h_k(b) = (c \to c)$ we have a solution of $\mathcal{R}_0$ in $\mathcal{R}_1$.

(ii) Let now $\mathcal{R}'_1 = \{c = c \to c,\ c' = c' \to c'\}$ and note that $c \neq_{\mathcal{R}'_1} c'$. So, besides h_k, also the homomorphism h'_k defined by $h'_k(a) = h'_k(b) = (c' \to c')$ solves $\mathcal{R}_0$ in $\mathcal{R}'_1$. But note that $h_k \neq h'_k$.

Since there are a finite number of $\mathrm{Trs}^+(\mathcal{R})-$redexes (the l.h.s. of the equations left in $\mathcal{R}_N$) we have immediately the following corollary.

Corollary 2. *It is decidable whether a s.r.* $\mathcal{R}$ *solves a system* $\mathcal{C}$ *of equations over* $\mathsf{T}_{\mathbf{A}}$.

Statman [11] has shown that the solvability of a s.r. in another s.r. is in general a NP-complete problem. A last property will be needed in the following section.

Lemma 6. *Let* $\mathcal{R}$ *be a s.r. over* T *and* $\mathcal{P} = \{\langle A_i, B_i \rangle \mid 1 \leq i \leq n\}$ *be a set of pair of types in* T*. Then it is decidable whether there is a type structure homomorphism* $h : \mathcal{T}_{\mathcal{R}} \to \mathcal{T}_{\mathcal{R}}$ *such that* $h(A_i) = B_i$ *for all* $1 \leq i \leq n$ *(in this case we say that* h solves $\mathcal{P}$*).*

Proof. Let's define a sequence $\mathcal{P}_m$ $(m \geq 0)$ where $\mathcal{P}_m$ is either a set of pair of types or FAIL. Let $\mathcal{P}_0 = \mathcal{P}$. Then define $\mathcal{P}_{m+1}$ from $\mathcal{P}_m$ in the following way:

a. If there is a pair $\langle A_1 \to A_2, B \rangle \in \mathcal{P}_m$ we have the following cases.
   1. If $B \neq_{\mathcal{R}} B_1 \to B_2$ for some types B_1 and B_2 then $\mathcal{P}_{m+1} = \text{FAIL}$.
   2. Otherwise let $\mathcal{P}_{m+1} = \mathcal{P}_m - \{\langle A_1 \to A_2, B \rangle\} \cup \{\langle A_l, B_l \rangle \mid l = 1, 2\}$.

b. Otherwise take all pairs $\langle c, A \rangle \in \mathcal{P}_m$ such that c is an indeterminate of $\mathcal{R}$ and set $\mathcal{P}_{m+1} = \mathcal{P}_m \cup \{\langle C, A \rangle \mid \langle c, A \rangle \in \mathcal{P}_m \text{ and } c = C \in \mathcal{R}\}$.

Let's say that $\mathcal{P}_m$ is *flatten* if it contains only pairs of the shape $\langle a, A \rangle$ where a is atomic. Now it is easy to prove that either $\mathcal{P}_m = \text{FAIL}$ for some $m > 0$ or there is an $N > 0$ such that

1. $\mathcal{P}_N$ is flatten.
2. for all $n > N$ such that $\mathcal{P}_n$ is flatten we have that $\mathcal{P}_n = \mathcal{P}_N$.

The proof of this follows from the observation that all types occurring $\mathcal{P}_m$ for $m \geq 0$ must be subtypes of some type occurring in $\mathcal{P}$ or in some C_i.
It is immediate to prove, by induction on m, that there is a solution of $\mathcal{P}$ iff there is a solution of $\mathcal{P}_m$.

Now for each $a \in \mathbf{A}$ let

$$\mathcal{B}_a = \{B \mid a = B \in \mathcal{P}_N\} \cup \{C \mid a \in dom(\mathcal{R}) \text{ and } a = C \in \mathcal{R}\}.$$

Let $e_1, \ldots, e_m$ $(1 \leq m \leq n)$ denote the atoms $a \in \mathbf{A}$ such that $\mathcal{B}_a \neq \emptyset$. Now for all e_j $(1 \leq j \leq m)$ and for all $B', B'' \in \mathcal{B}_{e_j}$ check that $B' =_{\mathcal{R}} B''$. If this is true then $h = [e_1 := B_1, \ldots, e_m := B_m]$, where $B_i \in \mathcal{B}_{e_i}$ for all $1 \leq i \leq n$, solves $\mathcal{P}$. Otherwise there is no h which solves it.

Example 4. Let $\mathcal{R} = \{c_1 = c_2 \to c_1, c_2 = c_1 \to c_2\}$ and let $\mathcal{P} = \{\langle c_1, c_2 \rangle\}$. Then we have $\mathcal{P}_1 = \{\langle c_2 \to c_1, c_2 \rangle\}$ and $\mathcal{P}_2 = \{\langle c_2, c_1 \rangle, \langle c_1, c_2 \rangle\}$

It is easy to see that $\mathcal{P}_5 = \mathcal{P}_2$ and then $N = 2$. So let h be defined as $h(c_1) = c_2$ and $h(c_2) = c_1$. It is immediate to verify that $h(c_i) = h(C_i)$ for $i = 1, 2$. Then h solves $\mathcal{P}$. Note that setting only $h(c_1) = c_2$ would not give the correct solution. This shows the necessity of step b in the definition of h.

5 Finding Types

We consider now some natural problems about the typability of a λ-term in an inference system with recursive types. The questions that we will consider are the following. Let M be a given untyped λ-term:

1. Does it exist a s.r. $\mathcal{R}$, a type environment Γ and a type A such that $\Gamma \vdash_{\mathcal{R}} M : A$?
2. Given a s.r. $\mathcal{R}$ does there exist a type environment Γ and a type A such that $\Gamma \vdash_{\mathcal{R}} M : A$?
3. Given a s.r. $\mathcal{R}$, a type environment Γ and a type A does $\Gamma \vdash_{\mathcal{R}} M : A$ hold?

The same questions can be stated for the corresponding systems with tree equivalence, i.e. with $\vdash^*_{\mathcal{R}}$ instead of $\vdash_{\mathcal{R}}$. We will prove that all these questions are decidable for both notions of equivalence.

Theorem 1 which reduces tree equivalence to the equational one allows to threat both equivalences in a uniform way. The decidability of question 1. was well known (see for instance [3] or [7]) but there seems to be no published proof of the decidability of 2. and 3..

Note that in this paper recursive types are not considered in polymorphic sense. For instance an equation like $c = t \to c$ represent only itself and not all its possible instances via t. It would be interesting to allow s.r.s to contain schemes like $\forall t.c[t] = t \to c[t]$ rather then simple equations, but this is left for further research.

To keep technical details simple we investigate these problems for pure λ-terms (without term constants). All results however can be generalized in a quite straightforward way to terms including constants (see remark 2).

Using type structure homomorphisms we can define, in a quite natural way, a notion of principal type scheme for type assignments with respect to arbitrary invertible type structures. In the following definition we assume, without loss of generality, that all free and bound variables in a term have distinct names.

Definition 17. *Let M be a pure λ-term. The system of type constraints $\mathcal{C}_M$, the basis Γ_M and type T_M are defined as follows.*
Let $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ where $\mathcal{S}_1$ is set of all bound and free variables of M and $\mathcal{S}_2$

is the set of all occurrences of subterms of M which are not variables. Take a set $\mathbf{I}_M$ of atomic types such that there is a bijection $\pi : \mathcal{S} \to \mathbf{I}_M$. Then define a system of type constraints $\mathcal{C}_M$ over $\mathsf{T}_{\mathbf{I}_M}$ in the following way.

(a) For every $\lambda x.P \in \mathcal{S}_2$ put $\pi(\lambda x.P) = \pi(x) \to \pi(P)$ in $\mathcal{C}_M$;

(b) For every $P = (P_1 P_2)$ put $\pi(P_1) = \pi(P_2) \to \pi(P)$ in $\mathcal{C}_M$.

Now let $\Gamma_M = \{x : \pi(x) \mid x \text{ is free in } M\}$ and $T_M = \pi(M)$.

Lemma 7. *Let M be λ-term. Then $\Gamma_M \vdash_{\mathcal{C}_M} M : T_M$.*

Example 5. Consider the term $\lambda x.(xx)$ and assume $\pi(x) = i_1$, $\pi(xx) = i_2$, $\pi(\lambda x.(xx)) = i_3$. Then we have $\mathcal{C}_{\lambda x.(xx)} = \{i_1 = i_1 \to i_2,\ i_3 = i_1 \to i_2\}$. Moreover we have $\Gamma_{\lambda x.(xx)} = \emptyset$, $T_{\lambda x.(xx)} = i_3$.

Note that $\mathcal{C}_M$ is a system of type constraints but not, in general, a s.r.. In fact in general we can have many equations with the same left hand side.

Indeed to type any pure λ-term it would be enough to take a s.r. with only one equation $\mathcal{R}_0 = \{c = c \to c\}$. However the typings obtained assuming $\mathcal{R}_0$ are trivial and not interesting. $\mathcal{C}_M$, instead, gives the weakest type constraints necessary to type M, and then gives more informations about the structure of M. But there are obviously terms which can be typed only with respect to $\mathcal{R}_0$, like $(\lambda x.(xx))(\lambda y.y)$ (we leave to the reader to show this).
We will remark (see Remark 2) that the problem of typability is not more trivial when term and type constants are considered.

For the proof of this theorem, which was more or less known as folklore of the subject, we refer to [7].

Theorem 4. *Let M be a Λ-term and $\mathcal{T} = \langle \mathsf{T}, \simeq \rangle$ be a type structure. Then*

$$\Gamma \vdash_{\mathcal{T}} M : A$$

iff there is a type structure homomorphism

$$h : \langle \mathsf{T}_{\mathbf{I}_M}, =_{\mathcal{C}_M} \rangle \to \mathcal{T}$$

such that $A \simeq h(T_M)$ and $h(\Gamma_M)$ is a subset (modulo $\simeq$) of Γ.

Theorem 4 is a straightforward generalization of the Principal Type Theorem for Simple Curry's assignment system. The terms for which $\mathcal{C}_M$ can be solved in an empty s.r. (i.e. no equation between types assumed) are exactly these typable in Curry's system. A generalization of Theorem 4 is the following.

Theorem 5. (i) *Let M be a Λ-term and $\mathcal{R}$ a s.r. over T. Then there is a finite set $H = \{h^{\mathcal{R}}_{M,1}, \ldots, h^{\mathcal{R}}_{M,k}\}$ $(k \geq 0)$ of type structure homomorphisms $h^{\mathcal{R}}_{M,i} : \langle \mathsf{T}(\mathbf{I}_M), =_{\mathcal{C}_M} \rangle \to \mathcal{T}_{\mathcal{R}}$ such that:*

1) $h^{\mathcal{R}}_{M,i}(\Gamma_M) \vdash_{\mathcal{R}} M : h^{\mathcal{R}}_{M,i}(T_M)$

2) $\Gamma \vdash_{\mathcal{R}} M : A$ for some environment Γ and type A iff for some $1 \leq i \leq k$ there is a type structure homomorphism $h' : \mathcal{T}_{\mathcal{R}} \to \mathcal{T}_{\mathcal{R}}$ such that $A =_{\mathcal{R}} h' \circ h^{\mathcal{R}}_{M,i}(T_M)$ and $h' \circ h^{\mathcal{R}}_{M,i}(\Gamma_M)$ is a subset (modulo $=_{\mathcal{R}}$) of Γ.

(ii) *The same property holds for $\vdash^*_{\mathcal{R}}$, replacing $\mathcal{T}_{\mathcal{R}}$ with $\mathcal{T}^*_{\mathcal{R}}$ and $=_{\mathcal{R}}$ with $=^*_{\mathcal{R}}$.*

Proof. (i) If $\mathsf{T} = \mathsf{T}(\mathbf{A})$ we can assume without loss of generality that $\mathbf{I}_M \subseteq \mathbf{A}$. The proof follows by Theorem 4 and Lemma 5, observing that any homomorphism $h : \langle \mathsf{T}, =_{\mathcal{C}_M} \rangle \to \mathcal{T}_{\mathcal{R}}$ gives a solution of $\mathcal{C}_M$ in $\mathcal{R}$ and then, by Lemma 4, of ${\mathcal{C}_M}^{\text{inv}}$ in $\mathcal{R}$. So we take H as the set of the essential solutions of ${\mathcal{C}_M}^{\text{inv}}$ in $\mathcal{R}$.
(ii) Apply (i) by replacing $\vdash^*_{\mathcal{R}}$ with $\vdash_{\mathcal{R}^*}$, where $\mathcal{R}^*$ is built as in the proof of Theorem 1.

Some direct consequences of this theorem are the following.

Theorem 6. (i) *Given a pure λ-term M and an s.r. $\mathcal{R}$ it is decidable whether there exist a type environment Γ and a type A such $\Gamma \vdash_{\mathcal{R}} M : A$.*

(ii) *Given a pure λ-term M, an s.r. $\mathcal{R}$, a type context Γ and a type A it is decidable whether $\Gamma \vdash_{\mathcal{R}} M : A$*

(iii) *Both i) and ii) hold also for $\vdash^*_{\mathcal{R}}$ (i.e. replacing $\vdash_{\mathcal{R}}$ by $\vdash^*_{\mathcal{R}}$).*

Proof. (i) By Theorem 5 we have that a term can be types with respect to $\mathcal{R}$ only if $\mathcal{C}_M$ can be solved in $\mathcal{R}$. The property then follows by Lemma 2
(ii) Assume $\Gamma \vdash_{\mathcal{R}} M : A$. We can assume without loss of generality that Γ contains all and only the variables free in M. By Lemma 4 and Theorem 5 there is a type structure homomorphism $h' \circ h : \langle \mathsf{T}, \mathcal{C}_M \rangle \to \mathcal{T}_{\mathcal{R}}$ such that $A \simeq h' \circ h(T_M)$ and $h' \circ h(\Gamma_M) =_{\mathcal{R}} \Gamma$, where $h \in H$, the set of essential solutions of $\mathcal{C}_M$ in $\mathcal{R}$, and $h' : \mathcal{T}_{\mathcal{R}} \to \mathcal{T}_{\mathcal{R}}$. Referring to Definition 17, let $\mathcal{X} = \{i \mid x \in FV(M) \text{ and } x : A_i \in \Gamma \text{ and } \pi(x) = i\}$. Then h' must be such that:

- $h'(h(T_M)) = A$;
- $h'(h(i)) = A_i$ for all $i \in \mathcal{X}$.

We get the result from Lemma 6 and the fact that H has a finite number of elements.
(iii) Immediate from Theorem 1.

Remark 2 (About Term Constants). If M can contain occurrences of term constants then we must assume a set K of constants types, like the type **int** of integers. Moreover to each term constants c we associate a type $\tau(\mathsf{c})$ which is usually a constant of first order type (for instance if 3 is a numeral $\tau(3) = \mathbf{int}$). It is standard to assume that constant types can be equivalent only to themselves, and one is not allowed to equate them to other types, for instance **int** to an arrow type. We say that $\mathcal{C}_M$ is *consistent* if it does not imply any equation $\kappa = C$ where κ is a constant type and C is a either a different constant type or a non atomic expression. Then as a consequence of Theorem 4 a term M can be typed (w.r.t. some type structure) iff $\mathcal{C}_M$ is consistent.

We can easily take that into account constants in the construction of Definition 17 by adding to $\mathcal{C}_M$ an equation $\pi(\mathsf{c}) = \tau(\mathsf{c})$ for each constant c occurring in M. To check now that $\mathcal{C}_M$ is consistent it is enough to build ${\mathcal{C}_M}^{\text{inv}}$. We can easily prove that $\mathcal{C}_M$ is consistent iff for each atomic constant type κ there are in ${\mathcal{C}_M}^{\text{inv}}$ equations $\kappa = a_1, a_1 = a_2, \ldots, a_n = C$ where C is either a constant different from κ or a non atomic type expression. It is then decidable if $\mathcal{C}_M$ is consistent.

All the results given this section still hold if we consider terms with constants (in the previous sense). In some cases this makes these results more interesting. For instance Problem 1. (to decide whether a given term has a type w.r.t. some s.r.) is not trivial any more since there are terms, like (3 3), which have no type w.r.t. any s.r.. By the subject reduction theorem, moreover, we still have that a term to which it can be given a type (in any consistent s.r.) can not produce bad applications during its evaluation.

Acknowledgments. The author acknowledge H. Barendregt, W. Dekkers and the anonymous referees for their helpful suggestions.

References

1. M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In P. de Groote and J. R. Hindley, editors, *Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 63–81. Springer-Verlag, 1997.
2. V. Breazu-Tannen and A. Meyer. Lambda calculus with constrained types. In R. Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 23–40. Springer-Verlag, 1985.
3. F. Cardone and M. Coppo. Type inference with recursive types. Syntax and Semantics. *Information and Computation*, 92(1):48–80, 1991.
4. B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
5. J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.
6. J.W. Klop. Term rewriting systems. In Dov M. Gabbay S. Abramsky and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 1–116. Oxford University Uress, New York, 1992.
7. M. Marz. An algebraic view on recursive types. *Applied Categorical Structures*, 7(12):147–157, 1999.
8. R. Milner. A Theory of Type Polimorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
9. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
10. D. Scott. Some philosophical issues concerning theories of combinators. In C. Böhm, editor, *Lambda calculus and computer science theory*, volume 37 of *Lecture Notes in Computer Science*, pages 346–366. Springer-Verlag, 1975.
11. R. Statman. Recursive types and the subject reduction theorem. Technical Report 94–164, Carnegie Mellon University, 1994.
12. B. Pierce V. Gapeyev, M. Levin. Recursive subtyping revealed. In *Proceedings fifth ACM SIGPLAN International Conference on Functional Programming*, pages 221–232. ACM press, 2000.

On the Modularity of Deciding Call-by-Need

Irène Durand[1] and Aart Middeldorp[2⋆]

[1] Université de Bordeaux I
33405 Talence, France
idurand@labri.u-bordeaux.fr

[2] Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba 305-8573, Japan
ami@is.tsukuba.ac.jp

Abstract. In a recent paper we introduced a new framework for the study of call by need computations. Using elementary tree automata techniques and ground tree transducers we obtained simple decidability proofs for a hierarchy of classes of rewrite systems that are much larger than earlier classes defined using the complicated sequentiality concept. In this paper we study the modularity of membership in the new hierarchy. Surprisingly, it turns out that none of the classes in the hierarchy is preserved under signature extension. By imposing various conditions we recover the preservation under signature extension. By imposing some more conditions we are able to strengthen the signature extension results to modularity for disjoint and constructor-sharing combinations.

1 Introduction

The seminal work of Huet and Lévy [9] on optimal normalizing reduction strategies for orthogonal rewrite systems marks the beginning of the quest for decidable subclasses of (orthogonal) rewrite systems that admit a computable call by need strategy for deriving normal forms. Call by need means that the strategy may only contract *needed* redexes, i.e., redexes that are contracted in every normalizing rewrite sequence. Huet and Lévy showed that for the class of orthogonal rewrite systems every term not in normal form contains a needed redex and repeated contraction of needed redexes results in a normal form if the term under consideration has a normal form. However, neededness is in general undecidable. In order to obtain a decidable approximation to neededness Huet and Lévy introduced in the second part of [9] the subclass of *strongly sequential* systems. In a strongly sequential system at least one of the needed redexes in every reducible term can be effectively computed. Moreover, Huet and Lévy showed that strong sequentiality is a decidable property of orthogonal rewrite systems. Several authors ([2,6,10,13,15,17,20,19]) studied extensions of the class of strong sequential rewrite systems that preserve its good properties.

⋆ Partially supported by the Grant-in-Aid for Scientific Research C(2) 11680338 of the Ministry of Education, Science, Sports and Culture of Japan.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 199–213, 2001.

In a previous paper (Durand and Middeldorp [6]) we presented a uniform framework for decidable call by need based on approximations. We introduced classes $\mathcal{CBN}_\alpha$ of rewrite systems parameterized by approximation mappings α. In [6] we identified the properties an approximation mapping α has to satisfy in order that the resulting class $\mathcal{CBN}_\alpha$ is decidable and every rewrite system in that class admits a computable normalizing call by need strategy. We showed moreover that our classes are much larger than the corresponding classes based on the difficult sequentiality concept.

Not much is known about the complexity of the problem of deciding membership in one of the classes that guarantees a computable call by need strategy to normal form. Comon [2] showed that strong sequentiality of a left-linear rewrite system can be decided in exponential time. Moreover, for left-linear rewrite systems satisfying the additional syntactic condition that whenever two proper subterms of left-hand sides are unifiable one of them matches the other, strong sequentiality can be decided in polynomial time. The class of forward-branching systems (Strandh [18]), a proper subclass of the class of orthogonal strongly sequential systems, coincides with the class of transitive systems (Toyama *et al.* [21]) and can be decided in quadratic time (Durand [5]). For classes higher in the hierarchy only double exponential upper bounds are known ([7]).

Consequently, it is of obvious importance to have results available that enable to split a rewrite system into smaller components such that membership in $\mathcal{CBN}_\alpha$ of the components implies membership of the original system in $\mathcal{CBN}_\alpha$. Such *modularity* results have been extensively studied for basic properties like confluence and termination, see [8,14,16] for overviews.

The simplest kind of modularity results are concerned with enriching the signature. Most properties of rewrite systems are preserved under *signature extension*. Two notable exceptions are the normal form property and the unique normal form property (with respect to reduction), see Kennaway *et al.* [11]. Also some properties dealing with ground terms are not preserved under signature extension. For instance, consider the property that every ground term has a normal form, the rewrite system consisting of the single rewrite rule $\mathsf{f}(x) \to \mathsf{f}(x)$, and add a new constant a. (A slightly more interesting example is obtained by adding the rewrite rule $\mathsf{f}(\mathsf{b}) \to \mathsf{b}$.) It turns out that for no α membership in $\mathcal{CBN}_\alpha$ is preserved under signature extension. In this paper we present several sufficient conditions which guarantee the preservation of signature extension. These results are presented in Section 3.

Since preservation under signature extension does not give rise to a very useful technique for splitting a system into smaller components, in Section 4 we consider combinations of systems without common function symbols as well as constructor-sharing combinations.

In the next section we briefly recall the framework of our earlier paper [6] for analyzing call by need computations in term rewriting and we introduce some useful definitions.

2 Preliminaries

We assume the reader is familiar with the basics of term rewriting ([1,4,12]). We recall the following definitions from [6]. We refer to the latter paper for motivation and examples. A term rewrite system (TRS for short) consists of rewrite rules $l \to r$ that satisfy $\mathsf{root}(l) \notin \mathcal{V}$ and $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)$. If the second condition is not imposed we find it useful to speak of extended TRSs (eTRSs). Such systems arise naturally when we approximate TRSs, as explained below.

Let $\mathcal{R}$ be an eTRS over a signature $\mathcal{F}$. The set of ground normal forms of $\mathcal{R}$ is denoted by $\mathsf{NF}(\mathcal{R},\mathcal{F})$. Let $\mathcal{R}_\bullet$ be the eTRS $\mathcal{R} \cup \{\bullet \to \bullet\}$ over the extended signature $\mathcal{F}_\bullet = \mathcal{F} \cup \{\bullet\}$. We say that redex Δ in $C[\Delta] \in \mathcal{T}(\mathcal{F})$ is $\mathcal{R}$-needed if there is no term $t \in \mathsf{NF}(\mathcal{R},\mathcal{F})$ such that $C[\bullet] \to^*_{\mathcal{R}} t$. So to determine $\mathcal{R}$-neededness of a redex Δ in $C[\Delta]$ we replace it by $\bullet$ and check whether we can derive a normal form without $\bullet$. Redex Δ is $\mathcal{R}$-needed only if this is impossible. Note that $\mathsf{NF}(\mathcal{R},\mathcal{F}) = \mathsf{NF}(\mathcal{R}_\bullet,\mathcal{F}_\bullet)$. For orthogonal TRSs $\mathcal{R}$-neededness coincides with neededness. We denote by $\mathsf{WN}(\mathcal{R},\mathcal{F})$ the set of all ground terms in $\mathcal{T}(\mathcal{F})$ that rewrite in $\mathcal{R}$ to a normal form in $\mathsf{NF}(\mathcal{R},\mathcal{F})$. If no confusion can arise, we just write $\mathsf{WN}(\mathcal{R})$.

Let $\mathcal{R}$ and $\mathcal{S}$ be eTRSs over the same signature $\mathcal{F}$. We say that $\mathcal{S}$ approximates $\mathcal{R}$ if $\to^*_{\mathcal{R}} \subseteq \to^*_{\mathcal{S}}$ and $\mathsf{NF}(\mathcal{R},\mathcal{F}) = \mathsf{NF}(\mathcal{S},\mathcal{F})$. An approximation mapping is a mapping α from TRSs to eTRSs with the property that $\alpha(\mathcal{R})$ approximates $\mathcal{R}$, for every TRS $\mathcal{R}$. In the following we write $\mathcal{R}_\alpha$ instead of $\alpha(\mathcal{R})$. The class of *left-linear* TRSs $\mathcal{R}$ such that every reducible term in $\mathcal{T}(\mathcal{F})$ has an $\mathcal{R}_\alpha$-needed redex is denoted by $\mathcal{CBN}_\alpha$. Here $\mathcal{F}$ denotes the signature of $\mathcal{R}$. We assume throughout the paper that ground terms exist. Although not explicitly stated in our earlier paper [6] we assume that $\mathcal{R}$ is a proper TRS (i.e., not an eTRS). For arbitrary left-linear eTRSs $\mathcal{R}$ we introduce a corresponding class $\mathcal{CBN}$ which contains the eTRSs with the property that every reducible ground term has an $\mathcal{R}$-needed redex. So a TRS $\mathcal{R}$ belongs to $\mathcal{CBN}_\alpha$ if and only if $\mathcal{R}_\alpha \in \mathcal{CBN}$.

Next we define the approximation mappings s, nv, and g. Let $\mathcal{R}$ be a TRS. The *strong* approximation $\mathcal{R}_\mathrm{s}$ is obtained from $\mathcal{R}$ by replacing the right-hand side of every rewrite rule by a variable that does not occur in the corresponding left-hand side. The *nv* approximation $\mathcal{R}_\mathrm{nv}$ is obtained from $\mathcal{R}$ by replacing the variables in the right-hand sides of the rewrite rules by pairwise distinct variables that do not occur in the corresponding left-hand sides. An eTRS is called growing if for every rewrite rule $l \to r$ the variables in $\mathcal{V}\mathsf{ar}(l) \cap \mathcal{V}\mathsf{ar}(r)$ occur at depth 1 in l. The *growing* approximation $\mathcal{R}_\mathrm{g}$ is defined as any growing eTRS that is obtained from $\mathcal{R}$ by renaming the variables in the right-hand sides that occur at a depth greater than 1 in the corresponding left-hand sides. In [6] we showed that for a left-linear TRS $\mathcal{R}$ and $\alpha \in \{\mathrm{s},\mathrm{nv},\mathrm{g}\}$ membership of $\mathcal{R}$ in $\mathcal{CBN}_\alpha$ is decidable.

We conclude this preliminary section with some easy definitions. A rewrite rule $l \to r$ of an eTRS is collapsing if r is a variable. A redex with respect to a collapsing rewrite rule is also called collapsing and so is an eTRS that contains a collapsing rewrite rule. A redex is called *flat* if it does not contain smaller redexes. Let $\mathcal{R}$ be a TRS over the signature $\mathcal{F}$. A function symbol in $\mathcal{F}$ is called

defined if it is the root symbol of a left-hand side of a rewrite rule in $\mathcal{R}$. All other function symbols in $\mathcal{F}$ are called constructors. A term without defined symbols is called a constructor term. We say that $\mathcal{R}$ is a constructor system (CS for short) if the arguments of the left-hand side of a rewrite rule are constructor terms.

Let $\mathcal{R}$ be an eTRS over the signature $\mathcal{F}$ and let $\mathcal{G} \subseteq \mathcal{F}$. We denote by $\mathsf{WN}(\mathcal{R}, \mathcal{F}, \mathcal{G})$ the set of terms in $\mathcal{T}(\mathcal{G})$ that have a normal form with respect to $(\mathcal{R}, \mathcal{F})$. The subset of $\mathsf{WN}(\mathcal{R}, \mathcal{F}, \mathcal{G})$ consisting of those terms that admit a normalizing rewrite sequence in $(\mathcal{R}, \mathcal{F})$ containing a root rewrite step is denoted by $\mathsf{WNR}(\mathcal{R}, \mathcal{F}, \mathcal{G})$. If $\mathcal{F} = \mathcal{G}$ then we just write $\mathsf{WNR}(\mathcal{R}, \mathcal{F})$ or even $\mathsf{WNR}(\mathcal{R})$ if the signature is clear from the context. We also find it convenient to write $\mathsf{WN}_\bullet(\mathcal{R}, \mathcal{F}, \mathcal{G})$ for $\mathsf{WN}(\mathcal{R}_\bullet, \mathcal{F}_\bullet, \mathcal{G}_\bullet)$ and $\mathsf{WNR}_\bullet(\mathcal{R}, \mathcal{F}, \mathcal{G})$ for $\mathsf{WNR}(\mathcal{R}_\bullet, \mathcal{F}_\bullet, \mathcal{G}_\bullet)$. A reducible term without needed redexes is called *free*. A *minimal* free term has the property that its proper subterms are not free.

3 Signature Extension

In this section we study the question whether membership in $\mathcal{CBN}_\alpha$ is preserved after adding new function symbols.

Definition 1. *We say that a class $\mathcal{C}$ of eTRSs is* preserved under signature extension *if $(\mathcal{R}, \mathcal{G}) \in \mathcal{C}$ for all $(\mathcal{R}, \mathcal{F}) \in \mathcal{C}$ and $\mathcal{F} \subseteq \mathcal{G}$.*

The results we obtain in this section are summarized below. In the result marked with $(*)$ signature extension only holds if we further require that the set of $\mathcal{R}_{\mathrm{nv}}$-normalizable terms over the original signature is not increased.

approximation	sufficient conditions	Theorem
strong	$\exists$ ground normal form	1
nv	$\exists$ external normal form	2
nv	$\mathcal{R}$ is collapsing or $\mathcal{R}$ is a CS $(*)$	3
growing	$\exists$ external normal form	2

All our proofs follow the same strategy. We consider a TRS $\mathcal{R}$ over a signature $\mathcal{F}$ such that $(\mathcal{R}, \mathcal{F}) \in \mathcal{CBN}_\alpha$. Let $\mathcal{G}$ be an extension of $\mathcal{F}$. Assuming that $(\mathcal{R}, \mathcal{G}) \notin \mathcal{CBN}_\alpha$, we consider a minimal $(\mathcal{R}_\alpha, \mathcal{G})$-free term t in $\mathcal{T}(\mathcal{G})$. By replacing the maximal subterms of t that start with a function symbol in $\mathcal{G} \backslash \mathcal{F}$—such subterms will be called *aliens* or more precisely $\mathcal{G} \backslash \mathcal{F}$-aliens in the sequel—by a suitable term in $\mathcal{T}(\mathcal{F})$, we obtain an $(\mathcal{R}_\alpha, \mathcal{F})$-free term t' in $\mathcal{T}(\mathcal{F})$. Hence $(\mathcal{R}, \mathcal{F}) \notin \mathcal{CBN}_\alpha$, contradicting the assumption.

Our first example shows that $\mathcal{CBN}_{\mathrm{s}}$ is not preserved under signature extension.

Example 1. Consider the TRS

$$\mathcal{R} = \left\{ \begin{array}{rcl} \mathsf{f}(x, \mathsf{g}(y), \mathsf{h}(z)) & \to & x \\ \mathsf{f}(\mathsf{h}(z), x, \mathsf{g}(y)) & \to & x \\ \mathsf{f}(\mathsf{g}(y), \mathsf{h}(z), x) & \to & x \\ \mathsf{a} & \to & \mathsf{a} \end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules. As $\mathsf{NF}(\mathcal{R},\mathcal{F}) = \varnothing$, trivially $(\mathcal{R},\mathcal{F}) \in \mathcal{CBN}_{\mathrm{s}}$. Let $\mathcal{G} = \mathcal{F} \cup \{\mathsf{b}\}$ with b a constant. We have $(\mathcal{R},\mathcal{G}) \notin \mathcal{CBN}_{\mathrm{s}}$ as the term $\mathsf{f}(\mathsf{a},\mathsf{a},\mathsf{a})$ has no $(\mathcal{R}_{\mathrm{s}},\mathcal{G})$-needed redex:

$$\begin{array}{llllllll}
\mathsf{f}(\bullet,\mathsf{a},\mathsf{a}) & \to_{\mathrm{s}} & \mathsf{f}(\bullet,\mathsf{g}(\mathsf{a}),\mathsf{a}) & \to_{\mathrm{s}} & \mathsf{f}(\bullet,\mathsf{g}(\mathsf{a}),\mathsf{h}(\mathsf{a})) & \to_{\mathrm{s}} & \mathsf{b} \\
\mathsf{f}(\mathsf{a},\bullet,\mathsf{a}) & \to_{\mathrm{s}} & \mathsf{f}(\mathsf{h}(\mathsf{a}),\bullet,\mathsf{a}) & \to_{\mathrm{s}} & \mathsf{f}(\mathsf{h}(\mathsf{a}),\bullet,\mathsf{g}(\mathsf{a})) & \to_{\mathrm{s}} & \mathsf{b} \\
\mathsf{f}(\mathsf{a},\mathsf{a},\bullet) & \to_{\mathrm{s}} & \mathsf{f}(\mathsf{g}(\mathsf{a}),\mathsf{a},\bullet) & \to_{\mathrm{s}} & \mathsf{f}(\mathsf{g}(\mathsf{a}),\mathsf{h}(\mathsf{a}),\bullet) & \to_{\mathrm{s}} & \mathsf{b}
\end{array}$$

The above example is interesting since it furthermore shows that $\mathcal{CBN}_{\mathrm{s}}$ *properly* includes the class of strongly sequential TRSs defined by Huet and Lévy [9], contrary to the claim in [6] that these two classes coincide.

One may wonder whether there are any nontrivial counterexamples, where nontrivial means that the set of ground normal forms is nonempty. Surprisingly, the answer is yes, provided we consider an approximation map α that is at least as good as nv.

Example 2. Consider the TRS

$$\mathcal{R} = \left\{ \begin{array}{llllllll}
\mathsf{f}(x,\mathsf{a},\mathsf{b}) & \to & \mathsf{g}(x) & \mathsf{f}(\mathsf{a},\mathsf{a},\mathsf{a}) & \to & \mathsf{g}(\mathsf{a}) & \mathsf{g}(\mathsf{a}) & \to & \mathsf{g}(\mathsf{a}) \\
\mathsf{f}(\mathsf{b},x,\mathsf{a}) & \to & \mathsf{g}(x) & \mathsf{f}(\mathsf{b},\mathsf{b},\mathsf{b}) & \to & \mathsf{g}(\mathsf{a}) & \mathsf{g}(\mathsf{b}) & \to & \mathsf{g}(\mathsf{b}) \\
\mathsf{f}(\mathsf{a},\mathsf{b},x) & \to & \mathsf{g}(x) & \mathsf{e}(x) & \to & x & & &
\end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules. First we show that $(\mathcal{R},\mathcal{F}) \in \mathcal{CBN}_{\mathrm{nv}}$. It is not difficult to show that the only $(\mathcal{R}_{\mathrm{nv}},\mathcal{F})$-normalizable terms are a, b, and $\mathsf{e}(t)$ for every $t \in \mathcal{T}(\mathcal{F})$. Since a and b are normal forms, we only have to show that every $\mathsf{e}(t)$ contains an $(\mathcal{R}_{\mathrm{nv}},\mathcal{F})$-needed redex, which is easy since $\mathsf{e}(t)$ itself is an $(\mathcal{R}_{\mathrm{nv}},\mathcal{F})$-needed redex. Let $\mathcal{G} = \mathcal{F} \cup \{\mathsf{c}\}$ with c a constant. We have $(\mathcal{R},\mathcal{G}) \notin \mathcal{CBN}_{\mathrm{nv}}$ as the term $\mathsf{f}(\mathsf{e}(\mathsf{a}),\mathsf{e}(\mathsf{a}),\mathsf{e}(\mathsf{a}))$ has no $(\mathcal{R}_{\mathrm{nv}},\mathcal{G})$-needed redex:

$$\begin{array}{llllllll}
\mathsf{f}(\bullet,\mathsf{e}(\mathsf{a}),\mathsf{e}(\mathsf{a})) & \to_{\mathrm{nv}} & \mathsf{f}(\bullet,\mathsf{a},\mathsf{e}(\mathsf{a})) & \to_{\mathrm{nv}} & \mathsf{f}(\bullet,\mathsf{a},\mathsf{b}) & \to_{\mathrm{nv}} & \mathsf{g}(\mathsf{c}) \\
\mathsf{f}(\mathsf{e}(\mathsf{a}),\bullet,\mathsf{e}(\mathsf{a})) & \to_{\mathrm{nv}} & \mathsf{f}(\mathsf{b},\bullet,\mathsf{e}(\mathsf{a})) & \to_{\mathrm{nv}} & \mathsf{f}(\mathsf{b},\bullet,\mathsf{a}) & \to_{\mathrm{nv}} & \mathsf{g}(\mathsf{c}) \\
\mathsf{f}(\mathsf{e}(\mathsf{a}),\mathsf{e}(\mathsf{a}),\bullet) & \to_{\mathrm{nv}} & \mathsf{f}(\mathsf{a},\mathsf{e}(\mathsf{a}),\bullet) & \to_{\mathrm{nv}} & \mathsf{f}(\mathsf{a},\mathsf{b},\bullet) & \to_{\mathrm{nv}} & \mathsf{g}(\mathsf{c})
\end{array}$$

For $\alpha = \mathrm{s}$ there is no nontrivial counterexample.

Theorem 1. *The subclass of $\mathcal{CBN}_{\mathrm{s}}$ consisting of all orthogonal TRSs $(\mathcal{R},\mathcal{F})$ such that $\mathsf{NF}(\mathcal{R},\mathcal{F}) \neq \varnothing$ is preserved under signature extension.* □

We refrain from giving the proof at this point since the statement easily follows from Theorem 3 below, whose proof is presented in detail. Our second result states that the subclass of $\mathcal{CBN}$ consisting of all eTRSs $\mathcal{R}$ with the property defined below is preserved under signature extension.

Definition 2. *We say that an eTRS $\mathcal{R}$ has* external *normal forms if there exists a ground normal form which is not an instance of a proper non-variable subterm of a left-hand sides of a rewrite rule in $\mathcal{R}$.*

Note that the TRS of Example 2 lacks external normal forms as both ground normal forms a and b appear in the left-hand sides of the rewrite rules. Further note that it is decidable whether a left-linear TRS has external normal forms by straightforward tree automata techniques. Finally note that the external normal form property is satisfied whenever there exists a constant not occurring in the left-hand sides of the rewrite rules.

Before proving our second result we present a useful lemma which is used repeatedly in the sequel.

Lemma 1. *Let $\mathcal{R}$ be a left-linear eTRS. Every minimal free term belongs to* $\mathsf{WNR}(\mathcal{R})$.

Proof. Let $\mathcal{F}$ be the signature of $\mathcal{R}$ and let $t \in \mathcal{T}(\mathcal{F})$ be a minimal free term. For every redex position p in t we have $t[\bullet]_p \in \mathsf{WN}_\bullet(\mathcal{R})$. Let p' be the minimum position above p at which a contraction takes place in any rewrite sequence from $t[\bullet]_p$ to a normal form in $\mathcal{T}(\mathcal{F})$ and define $P = \{p' \mid p \text{ is a redex position in } t\}$. Let p^* be a minimal position in P. We show that $p^* = \epsilon$. If $p^* > \epsilon$ then we consider the term $t|_{p^*}$. Let q be a redex position in $t|_{p^*}$. There exists a redex position p in t such that $p = p^*q$. We have $t|_{p^*}[\bullet]_q = (t[\bullet]_p)|_{p^*} \in \mathsf{WN}_\bullet(\mathcal{R})$ by the definition of p^*. Since $t|_{p^*}$ has at least one redex, it follows that $t|_{p^*}$ is free. As $t|_{p^*}$ is a proper subterm of t we obtain a contradiction to the minimality of t. Hence $p^* = \epsilon$. So there exists a redex position p in t and a rewrite sequence $A\colon t[\bullet]_p \to^+_{\mathcal{R},\mathcal{F}_\bullet} u \in \mathsf{NF}(\mathcal{R},\mathcal{F})$ that contains a root rewrite step. Because $\mathcal{R}$ is left-linear and $\bullet$ does not occur in the rewrite rules of $\mathcal{R}$, $\bullet$ cannot contribute to this sequence. It follows that if we replace in A every occurrence of $\bullet$ by $t|_p$ we obtain an $(\mathcal{R},\mathcal{F})$-rewrite sequence from t to u with a root rewrite step. □

In particular, minimal free terms are not root-stable.

Theorem 2. *The subclass of $\mathcal{CBN}$ consisting of all left-linear eTRSs with external normal forms is preserved under signature extension.*

Proof. Let $(\mathcal{R},\mathcal{F}) \in \mathcal{CBN}$ and let $c \in \mathsf{NF}(\mathcal{R},\mathcal{F})$ be an external normal form. Let $\mathcal{F} \subseteq \mathcal{G}$. We have to show that $(\mathcal{R},\mathcal{G}) \in \mathcal{CBN}$. Suppose to the contrary that $(\mathcal{R},\mathcal{G}) \notin \mathcal{CBN}$. According to Lemma 1 there exists a term $t \in \mathsf{WNR}(\mathcal{R},\mathcal{G})$ without $(\mathcal{R},\mathcal{G})$-needed redex. Let t' be the term in $\mathcal{T}(\mathcal{F})$ obtained from t by replacing every $\mathcal{G}\backslash\mathcal{F}$-alien by c. Because t is not root-stable and $\mathcal{R}$ left-linear, t' must be reducible. Hence t' contains an $(\mathcal{R},\mathcal{F})$-needed redex Δ, say at position p. Because c is an external normal form, Δ is also a redex in t and hence there exists a rewrite sequence $t[\bullet]_p \to^+_{\mathcal{R},\mathcal{G}_\bullet} u$ with $u \in \mathsf{NF}(\mathcal{R},\mathcal{G})$. If we replace in this rewrite sequence every $\mathcal{G}\backslash\mathcal{F}$-alien by c, we obtain a rewrite sequence $t'[\bullet]_p \to^+_{\mathcal{R},\mathcal{F}_\bullet} u'$. Because c does not unify with a proper non-variable subterm of a left-hand side of a rewrite rule, it follows that $u' \in \mathsf{NF}(\mathcal{R},\mathcal{F})$. Hence Δ is not an $(\mathcal{R},\mathcal{F})$-needed redex in t', yielding the desired contradiction. □

Note that for $\mathcal{CBN}_\mathrm{s}$ the above theorem is a special case of Theorem 1 since the existence of an external normal form implies the existence of a ground normal form.

In the remainder of this section we present a signature extension result for TRSs without external normal form. Such TRSs are quite common (see also Lemma 4 below).

Example 3. Consider the TRS

$$\mathcal{R} = \left\{ \begin{array}{lcll} 0 + y & \to & y & \mathsf{s}(x) + y \to \mathsf{s}(x + y) \\ 0 \times y & \to & 0 & \mathsf{s}(x) \times y \to x \times y + y \end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules. Since every normal form is of the form $\mathsf{s}^n(0)$ for some $n \geqslant 0$, it follows that $\mathcal{R}$ lacks external normal forms.

We start with some preliminary results.

Definition 3. *Let $\mathcal{R}$ be a TRS. Two redexes Δ_1, Δ_2 are called* pattern equal, *denoted by $\Delta_1 \approx \Delta_2$, if they have the same redex pattern, i.e., they are redexes with respect to the same rewrite rule.*

Lemma 2. *Let $\mathcal{R}$ be an orthogonal TRS, $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$, and suppose that $\Delta \approx \Delta'$. If $C[\Delta] \in \mathsf{WN}(\mathcal{R}_\alpha)$ then $C[\Delta'] \in \mathsf{WN}(\mathcal{R}_\alpha)$.*

Proof. Let $C[\Delta] \to^* t$ be a normalizing rewrite sequence in $\mathcal{R}_\alpha$. If we replace every descendant of Δ by Δ' then we obtain a (possibly shorter) normalizing rewrite sequence $C[\Delta'] \to^* t$. The reason is that every descendant Δ'' of Δ satisfies $\Delta'' \approx \Delta$ due to orthogonality and hence if Δ'' is contracted to some term u then Δ rewrites to the same term because the variables in the right-hand sides of the rewrite rules in $\mathcal{R}_\alpha$ are fresh. Moreover, as t is a normal form, there are no descendants of Δ left. Note that the resulting sequence can be shorter since rewrite steps below a descendant of Δ are not mimicked. □

The above lemma does not hold for the growing approximation, as shown by the following example.

Example 4. Consider the TRS $\mathcal{R} = \{\mathsf{f}(x) \to x, \mathsf{a} \to \mathsf{b}, \mathsf{c} \to \mathsf{c}\}$. We have $\mathcal{R}_\mathrm{g} = \mathcal{R}$. Consider the redexes $\Delta = \mathsf{f}(\mathsf{a})$ and $\Delta' = \mathsf{f}(\mathsf{c})$. Clearly $\Delta \approx \Delta'$. Redex Δ admits the normal form b, but Δ' has no normal form.

Lemma 3. *Let $\mathcal{R}$ be an orthogonal TRS over a signature $\mathcal{F}$, $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$, and $\mathcal{F} \subseteq \mathcal{G}$. If $\mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$ then $\mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F}) = \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$.*

Proof. The inclusion $\mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F}) \subseteq \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$ is obvious. For the reverse inclusion we reason as follows. Let $t \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$ and consider a rewrite sequence A in $(\mathcal{R}_\alpha, \mathcal{G}_\bullet)$ that normalizes t. We may write $t = C[t_1, \ldots, t_n]$ such that $t_1, \ldots, t_n$ are the maximal subterms of t that are rewritten in A at their root positions. Hence A can be rearranged into A':

$$t \to^*_{\mathcal{R}_\alpha, \mathcal{G}_\bullet} C[\Delta_1, \ldots, \Delta_n] \to^*_{\mathcal{R}_\alpha, \mathcal{G}_\bullet} C[u_1, \ldots, u_n]$$

for some redexes $\Delta_1, \ldots, \Delta_n$ and normal form $C[u_1, \ldots, u_n] \in \mathcal{T}(\mathcal{G})$. Since the context C cannot contain $\bullet$, all occurrences of $\bullet$ are in the substitution parts of the redexes $\Delta_1, \ldots, \Delta_n$. If we replace in $C[\Delta_1, \ldots, \Delta_n]$ every $\mathcal{G}_\bullet \backslash \mathcal{F}$-alien by some ground term $c \in \mathcal{T}(\mathcal{F})$, we obtain a term $t' = C[\Delta'_1, \ldots, \Delta'_n]$ with $\Delta'_i \in \mathcal{T}(\mathcal{F})$ and $\Delta_i \approx \Delta'_i$ for every i. Repeated application of Lemma 2 yields $t' \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$. Because $\bullet$ cannot contribute to the creation of a normal form, we actually have $t' \in \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G})$ and thus $t' \in \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$ as $t' \in \mathcal{T}(\mathcal{F})$. The assumption yields $t' \in \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F})$. Since $\mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F}) \subseteq \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F})$ clearly holds, we obtain $t' \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F})$. Now, if we replace in the first part of A' every $\mathcal{G} \backslash \mathcal{F}$-alien by c then we obtain a (possibly shorter) rewrite sequence $t \to^*_{\mathcal{R}_\alpha, \mathcal{F}_\bullet} C[\Delta''_1, \ldots, \Delta''_n] \in \mathcal{T}(\mathcal{F}_\bullet)$ with $\Delta_i \approx \Delta''_i$ and thus also $\Delta'_i \approx \Delta''_i$ for every i. Repeated application of Lemma 2 yields $C[\Delta''_1, \ldots, \Delta''_n] \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F})$ and therefore $t \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F})$ as desired. □

We note that for $\alpha = \mathrm{s}$ the preceding lemma is a simple consequence of Lemma 6 below. The restriction to $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$ is essential. For $\mathcal{R}_\mathrm{g}$ we have the following counterexample.

Example 5. Consider the orthogonal TRS

$$\mathcal{R} = \left\{ \begin{array}{lcllcl} \mathsf{f}(x, \mathsf{a}, y) & \to & \mathsf{g}(y) & \mathsf{h}(\mathsf{a}) & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{a}, \mathsf{b}(x), y) & \to & \mathsf{a} & \mathsf{h}(\mathsf{b}(x)) & \to & \mathsf{i}(x) \\ \mathsf{f}(\mathsf{b}(x), \mathsf{b}(y), z) & \to & \mathsf{a} & \mathsf{i}(\mathsf{a}) & \to & \mathsf{a} \\ \mathsf{g}(\mathsf{a}) & \to & \mathsf{b}(\mathsf{g}(\mathsf{a})) & \mathsf{i}(\mathsf{b}(x)) & \to & \mathsf{b}(\mathsf{a}) \\ \mathsf{g}(\mathsf{b}(x)) & \to & \mathsf{g}(\mathsf{a}) & \mathsf{j}(x, \mathsf{a}) & \to & \mathsf{f}(x, \mathsf{h}(\mathsf{b}(\mathsf{a})), \mathsf{a}) \\ & & & \mathsf{j}(x, \mathsf{b}(y)) & \to & \mathsf{f}(x, \mathsf{h}(\mathsf{b}(\mathsf{a})), y) \end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules and let $\mathcal{S} = \mathcal{R}_\mathrm{g}$. Let $\mathcal{G} = \mathcal{F} \cup \{\mathsf{c}\}$ with c a constant. With some effort one can check that $\mathsf{WN}(\mathcal{S}, \mathcal{F}) = \mathsf{WN}(\mathcal{S}, \mathcal{G}, \mathcal{F})$. However, $\mathsf{WN}_\bullet(\mathcal{S}, \mathcal{F})$ is different from $\mathsf{WN}_\bullet(\mathcal{S}, \mathcal{G}, \mathcal{F})$ as witnessed by the term $t = \mathsf{j}(\bullet, \mathsf{b}(\mathsf{a}))$. In $(\mathcal{S}_\bullet, \mathcal{G}_\bullet)$ we have $t \to \mathsf{f}(\bullet, \mathsf{h}(\mathsf{b}(\mathsf{a})), \mathsf{c}) \to^+ \mathsf{f}(\bullet, \mathsf{a}, \mathsf{c}) \to \mathsf{g}(\mathsf{c})$, hence $t \in \mathsf{WN}_\bullet(\mathcal{S}, \mathcal{G}, \mathcal{F})$, but one easily verifies that t does not have a normal form in $(\mathcal{S}_\bullet, \mathcal{F}_\bullet)$.

Lemma 4. *The set of ground normal forms of a CS without external normal forms coincides with the set of ground constructor terms.*

Proof. Clearly every ground constructor term is a normal form. Suppose there exists a ground normal form that contains a defined function symbol. Since every subterm of a normal form is a normal form, there must be a ground normal form t whose root symbol is a defined function symbol. By definition of external normal form, t must be an instance of a proper non-variable subterm of a left-hand side l. This implies that a proper subterm of l contains a defined function symbol, contradicting the assumption that the TRS under consideration is a CS. □

Lemma 5. *Let $(\mathcal{R}, \mathcal{F})$ and $(\mathcal{S}, \mathcal{G})$ be orthogonal TRSs and $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$ such that $(\mathcal{R}, \mathcal{F}) \subseteq (\mathcal{S}, \mathcal{G})$ and $\mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F})$. If $t \in \mathsf{WNR}(\mathcal{S}_\alpha, \mathcal{G})$ and $\mathsf{root}(t) \in \mathcal{F}$ then there exists a flat redex Ξ in $\mathcal{T}(\mathcal{F})$. Moreover, if $\mathcal{R}_\alpha$ is collapsing then we may assume that Ξ is $\mathcal{R}_\alpha$-collapsing.*

Proof. From $t \in \mathsf{WNR}(\mathcal{S}_\alpha, \mathcal{G})$ we infer that $t \to^*_{\mathcal{S}_\alpha,\mathcal{G}} \Delta$ for some redex $\Delta \in \mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G})$. By considering the first such redex it follows that Δ is a redex with respect to $(\mathcal{R}_\alpha, \mathcal{G})$. If we replace in Δ the subterms below the redex pattern by an arbitrary ground term in $\mathcal{T}(\mathcal{F})$ then we obtain a redex $\Delta' \in \mathcal{T}(\mathcal{F})$ with $\Delta \approx \Delta'$. Lemma 2 yields $\Delta' \in \mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G})$ and thus $\Delta' \in \mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F})$. Hence $\mathsf{NF}(\mathcal{R}, \mathcal{F}) = \mathsf{NF}(\mathcal{R}_\alpha, \mathcal{F}) \neq \varnothing$. Therefore, using orthogonality, we obtain a flat redex $\Xi \in \mathcal{T}(\mathcal{F})$ by replacing the variables in the left-hand side of any rewrite rule in $\mathcal{R}$ by terms in $\mathsf{NF}(\mathcal{R}, \mathcal{F})$. If $\mathcal{R}_\alpha$ is a collapsing then we take any $\mathcal{R}_\alpha$-collapsing rewrite rule. □

We are now ready for the final signature extension result of this section. The condition $\mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$ expresses that the set of $\mathcal{R}_\alpha$-normalizable terms in $\mathcal{T}(\mathcal{F})$ is not enlarged by allowing terms in $\mathcal{T}(\mathcal{G})$ to be substituted for the variables in the rewrite rules. We stress that this condition is decidable for left-linear $\mathcal{R}$ and $\alpha \in \{\mathrm{s}, \mathrm{nv}, \mathrm{g}\}$ by standard tree automata techniques.

Theorem 3. *Let $\mathcal{R}$ be an orthogonal TRS over a signature $\mathcal{F}$, $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$, and $\mathcal{F} \subseteq \mathcal{G}$ such that $\mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$. If $(\mathcal{R}, \mathcal{F}) \in \mathcal{CBN}_\alpha$ and $\mathcal{R}$ is a CS or $\mathcal{R}_\alpha$ is collapsing then $(\mathcal{R}, \mathcal{G}) \in \mathcal{CBN}_\alpha$.*

Proof. If $(\mathcal{R}, \mathcal{F})$ has external normal forms then the result follows from Theorem 2. So we assume that $(\mathcal{R}, \mathcal{F})$ lacks external normal forms. We also assume that $\mathcal{R} \neq \varnothing$ for otherwise the result is trivial. Suppose to the contrary that $(\mathcal{R}, \mathcal{G}) \notin \mathcal{CBN}_\alpha$. According to Lemma 1 there exists a term $t \in \mathsf{WNR}(\mathcal{R}_\alpha, \mathcal{G})$ without $(\mathcal{R}_\alpha, \mathcal{G})$-needed redex. Lemma 5 (with $\mathcal{R} = \mathcal{S}$) yields a flat redex $\Xi \in \mathcal{T}(\mathcal{F})$. If $\mathcal{R}_\alpha$ is collapsing then we may assume that Ξ is $\mathcal{R}_\alpha$-collapsing. Let t' be the term in $\mathcal{T}(\mathcal{F})$ obtained from t by replacing every $\mathcal{G}\backslash\mathcal{F}$-alien by Ξ. Let P be the set of positions of those aliens. Since t' is reducible, it contains an $(\mathcal{R}_\alpha, \mathcal{F})$-needed redex, say at position q. We show that $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$. We consider two cases.

1. Suppose that $q \in P$. Since $t \in \mathsf{WNR}(\mathcal{R}_\alpha, \mathcal{G})$, $t \to^*_{\mathcal{R}_\alpha,\mathcal{G}} \Delta$ for some redex $\Delta \in \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}) \subseteq \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$. Since the root symbol of every alien belongs to $\mathcal{G}\backslash\mathcal{F}$, aliens cannot contribute to the creation of Δ and hence we may replace them by arbitrary terms in $\mathcal{T}(\mathcal{G}_\bullet)$ and still obtain a redex that is pattern equal to Δ. We replace the alien at position q by $\bullet$ and every alien at position $p \in P \setminus \{q\}$ by $t'|_p$. This gives $t'[\bullet]_q \to^*_{\mathcal{R}_\alpha,\mathcal{G}_\bullet} \Delta'$ with $\Delta' \approx \Delta$. Lemma 2 yields $\Delta' \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$ and hence $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$.
2. Suppose that $q \notin P$. Since Ξ is flat, it follows by orthogonality that q is also a redex position in t. Since t is an $(\mathcal{R}_\alpha, \mathcal{G})$-free term, $t[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$. We distinguish two further cases.
   a) First assume that $\mathcal{R}$ is a CS. Since t is not root-stable, its root symbol must be defined. From Lemma 4 we learn that a ground normal form of $\mathcal{R}$ (and thus of $\mathcal{R}_\alpha$) cannot contain defined symbols. Hence any $(\mathcal{R}_\alpha, \mathcal{G}_\bullet)$-rewrite sequence that normalizes $t[\bullet]_q$ contains a root step and thus $t[\bullet]_q \in \mathsf{WNR}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$. Hence $t[\bullet]_q \to^*_{\mathcal{R}_\alpha,\mathcal{G}_\bullet} \Delta' \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$ for some

redex $\Delta' \in \mathcal{T}(\mathcal{G}_\bullet)$. Similar to case (1) above, we replace the $\mathcal{G}\backslash\mathcal{F}$-aliens in $t[\bullet]_q$ by Ξ. This yields $t'[\bullet]_q \to^*_{\mathcal{R}_\alpha,\mathcal{G}_\bullet} \Delta''$ with $\Delta'' \approx \Delta'$. Lemma 2 yields $\Delta'' \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{G})$ and hence $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{G})$.

b) Next assume that $\mathcal{R}_\alpha$ is collapsing. Because Ξ is a collapsing redex, we have $\Xi \to_{\mathcal{R}_\alpha,\mathcal{G}} t|_p$ for all $p \in P$. Hence $t'[\bullet]_q \to^*_{\mathcal{R}_\alpha,\mathcal{G}_\bullet} t[\bullet]_q$ and thus $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{G})$.

As $t' \in \mathcal{T}(\mathcal{F})$, we have $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{G},\mathcal{F})$ and thus $t'[\bullet]_q \notin \mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{F})$ by Lemma 3, contradicting the assumption that q is the position of an $(\mathcal{R}_\alpha,\mathcal{F})$-needed redex in t'. □

It can be shown that all conditions in the above theorem are necessary. Below we show the necessity of the $\mathsf{WN}(\mathcal{R}_\alpha,\mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha,\mathcal{G},\mathcal{F})$ condition for noncollapsing CSs $\mathcal{R}_\alpha$. Due to lack of space we omit the other (complicated) counterexamples.

Example 6. Consider the orthogonal noncollapsing CS

$$\mathcal{R} = \left\{ \begin{array}{lcllcl} \mathsf{f}(x,\mathsf{a},\mathsf{b}) & \to & \mathsf{g}(x) & \mathsf{g}(\mathsf{a}) & \to & \mathsf{g}(\mathsf{a}) \\ \mathsf{f}(\mathsf{b},x,\mathsf{a}) & \to & \mathsf{g}(x) & \mathsf{g}(\mathsf{b}) & \to & \mathsf{g}(\mathsf{a}) \\ \mathsf{f}(\mathsf{a},\mathsf{b},x) & \to & \mathsf{g}(x) & \mathsf{h}(x) & \to & \mathsf{i}(x) \\ \mathsf{f}(\mathsf{a},\mathsf{a},\mathsf{a}) & \to & \mathsf{a} & \mathsf{i}(\mathsf{a}) & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{b},\mathsf{b},\mathsf{b}) & \to & \mathsf{a} & \mathsf{i}(\mathsf{b}) & \to & \mathsf{b} \end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules and let $\mathcal{S} = \mathcal{R}_{\mathrm{nv}}$. Let $\mathcal{G} = \mathcal{F} \cup \{\mathsf{c}\}$ with c a constant. Note that $\mathsf{WN}(\mathcal{S},\mathcal{F}) \neq \mathsf{WN}(\mathcal{S},\mathcal{G},\mathcal{F})$ as witnessed by the term $\mathsf{f}(\mathsf{a},\mathsf{a},\mathsf{b})$. With some effort one can check that $(\mathcal{R},\mathcal{F}) \in \mathcal{CBN}$. However, $(\mathcal{R},\mathcal{G}) \notin \mathcal{CBN}$ as $\mathsf{f}(\mathsf{h}(\mathsf{a}),\mathsf{h}(\mathsf{a}),\mathsf{h}(\mathsf{a}))$ lacks $(\mathcal{S},\mathcal{G})$-needed redexes.

We show that Theorem 1 is a special case of Theorem 3 by proving that for $\alpha = \mathrm{s}$ the condition $\mathsf{WN}(\mathcal{R}_\alpha,\mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha,\mathcal{G},\mathcal{F})$ is a consequence of $\mathsf{NF}(\mathcal{R},\mathcal{F}) \neq \varnothing$.

Lemma 6. *Let $\mathcal{R}$ be an eTRS over a signature $\mathcal{F}$. If $\mathsf{NF}(\mathcal{R},\mathcal{F}) \neq \varnothing$ then $\mathsf{WN}(\mathcal{R}_{\mathrm{s}},\mathcal{F}) = \mathcal{T}(\mathcal{F})$.*

Proof. If $\mathsf{NF}(\mathcal{R},\mathcal{F}) \neq \varnothing$ then there must be a constant $c \in \mathsf{NF}(\mathcal{R},\mathcal{F})$. Define the TRS $\mathcal{R}' = \{l \to c \mid l \to r \in \mathcal{R}\}$ over the signature $\mathcal{F}$. Clearly $\to_{\mathcal{R}'} \subseteq \to_{\mathrm{s}}$. Consider a precedence (i.e., a well-founded proper order on $\mathcal{F}$) $>$ with $f > c$ for every function symbol $f \in \mathcal{F}$ different from c. The TRS $\mathcal{R}'$ is compatible with the induced recursive path order $>_{\mathrm{rpo}}$ ([3]) and thus terminating. Since $\mathcal{R}_{\mathrm{s}}$ and $\mathcal{R}'$ have the same normal forms, it follows that $\mathcal{R}_{\mathrm{s}}$ is weakly normalizing. □

Proof of Theorem 1 Let $\mathcal{R}$ be an orthogonal TRS over a signature $\mathcal{F}$ such that $(\mathcal{R},\mathcal{F}) \in \mathcal{CBN}_{\mathrm{s}}$. Let $\mathcal{F} \subseteq \mathcal{G}$. We have to show that $(\mathcal{R},\mathcal{G}) \in \mathcal{CBN}_{\mathrm{s}}$. Since $\mathcal{R}_{\mathrm{s}}$ is collapsing (if $\mathcal{R} \neq \varnothing$; otherwise $\mathcal{R}$ is a CS), the result follows from Theorem 3 provided that $\mathsf{WN}(\mathcal{R}_{\mathrm{s}},\mathcal{F}) = \mathsf{WN}(\mathcal{R}_{\mathrm{s}},\mathcal{G},\mathcal{F})$. From Lemma 6 we obtain $\mathsf{WN}(\mathcal{R}_{\mathrm{s}},\mathcal{F}) = \mathcal{T}(\mathcal{F})$ and $\mathsf{WN}(\mathcal{R}_{\mathrm{s}},\mathcal{G},\mathcal{F}) = \mathsf{WN}(\mathcal{R}_{\mathrm{s}},\mathcal{G}) \cap \mathcal{T}(\mathcal{F}) = \mathcal{T}(\mathcal{G}) \cap \mathcal{T}(\mathcal{F}) = \mathcal{T}(\mathcal{F})$. □

We conclude this section by remarking that we have to use Theorem 3 only once. After adding a single new function symbol we obtain an external normal form and hence we can apply Theorem 2 for the remaining new function symbols.

4 Modularity

The results obtained in the previous section form the basis for the modularity results presented in this section. We first consider disjoint combinations.

Definition 4. *We say that a class $\mathcal{C}$ of TRSs is* modular (for disjoint combinations) *if $(\mathcal{R} \cup \mathcal{R}', \mathcal{F} \cup \mathcal{F}') \in \mathcal{C}$ for all $(\mathcal{R}, \mathcal{F}), (\mathcal{R}', \mathcal{F}') \in \mathcal{C}$ such that $\mathcal{F} \cap \mathcal{F}' = \varnothing$.*

To simplify notation, in the remainder of this section we write $\mathcal{S}$ for $\mathcal{R} \cup \mathcal{R}'$ and $\mathcal{G}$ for $\mathcal{F} \cup \mathcal{F}'$. The condition in Theorem 2 is insufficient for modularity as shown by the following example.

Example 7. Consider the TRS

$$\mathcal{R} = \left\{ \begin{array}{lcl} \mathsf{f}(x, \mathsf{a}, \mathsf{b}) & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{b}, x, \mathsf{a}) & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{a}, \mathsf{b}, x) & \to & \mathsf{a} \end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules and the TRS $\mathcal{R}' = \{ \mathsf{g}(x) \to x \}$ over the signature $\mathcal{F}'$ consisting of a constant c in addition to g. Both TRSs have external normal forms and belong to $\mathcal{CBN}_{\mathrm{nv}}$, as one easily shows. Their union does not belong to $\mathcal{CBN}_{\mathrm{nv}}$ as the term $\mathsf{f}(\mathsf{g}(\mathsf{a}), \mathsf{g}(\mathsf{a}), \mathsf{g}(\mathsf{a}))$ has no $(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$-needed redex:

$$\begin{array}{llllllll} \mathsf{f}(\bullet, \mathsf{g}(\mathsf{a}), \mathsf{g}(\mathsf{a})) & \to_{\mathrm{nv}} & \mathsf{f}(\bullet, \mathsf{a}, \mathsf{g}(\mathsf{a})) & \to_{\mathrm{nv}} & \mathsf{f}(\bullet, \mathsf{a}, \mathsf{b}) & \to_{\mathrm{nv}} & \mathsf{a} \\ \mathsf{f}(\mathsf{g}(\mathsf{a}), \bullet, \mathsf{g}(\mathsf{a})) & \to_{\mathrm{nv}} & \mathsf{f}(\mathsf{b}, \bullet, \mathsf{g}(\mathsf{a})) & \to_{\mathrm{nv}} & \mathsf{f}(\mathsf{b}, \bullet, \mathsf{a}) & \to_{\mathrm{nv}} & \mathsf{a} \\ \mathsf{f}(\mathsf{g}(\mathsf{a}), \mathsf{g}(\mathsf{a}), \bullet) & \to_{\mathrm{nv}} & \mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{a}), \bullet) & \to_{\mathrm{nv}} & \mathsf{f}(\mathsf{a}, \mathsf{b}, \bullet) & \to_{\mathrm{nv}} & \mathsf{a} \end{array}$$

If we forbid collapsing rules like $\mathsf{g}(x) \to x$, modularity holds. The following theorem can be proved along the lines of the proof of Theorem 2; because there are no collapsing rules and the eTRSs are left-linear, aliens cannot influence the possibility to perform a rewrite step in the non-alien part of a term.

Theorem 4. *The subclass of $\mathcal{CBN}$ consisting of all noncollapsing left-linear eTRSs with external normal forms is modular.* □

We find it convenient to separate the counterpart of Theorem 3 into two parts. The next two lemmata are used in both proofs. The nontrivial proof of the first one is omitted due to lack of space.

Lemma 7. *Let $(\mathcal{R}, \mathcal{F})$ and $(\mathcal{R}', \mathcal{F}')$ be disjoint TRSs. If $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$ then $\mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$.* □

Lemma 8. *Let $(\mathcal{R}, \mathcal{F})$ and $(\mathcal{R}', \mathcal{F}')$ be disjoint orthogonal TRSs and $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$. If $\mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F})$ then $\mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F}) = \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F})$.*

Proof. The previous lemma yields $\mathsf{WN}(\mathcal{R}_\alpha,\mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha,\mathcal{G},\mathcal{F})$. From Lemma 3 we obtain $\mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{F}) = \mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{G},\mathcal{F})$. Another application of the previous lemma yields the desired $\mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{F}) = \mathsf{WN}_\bullet(\mathcal{S}_\alpha,\mathcal{G},\mathcal{F})$. □

Theorem 5. *Let $(\mathcal{R},\mathcal{F})$ and $(\mathcal{R}',\mathcal{F}')$ be disjoint orthogonal noncollapsing CSs such that $\mathsf{WN}(\mathcal{R}_{\mathrm{nv}},\mathcal{G},\mathcal{F}) = \mathsf{WN}(\mathcal{R}_{\mathrm{nv}},\mathcal{F})$ and $\mathsf{WN}(\mathcal{R}'_{\mathrm{nv}},\mathcal{G},\mathcal{F}') = \mathsf{WN}(\mathcal{R}'_{\mathrm{nv}},\mathcal{F}')$. If $(\mathcal{R},\mathcal{F}),(\mathcal{R}',\mathcal{F}') \in \mathcal{CBN}_{\mathrm{nv}}$ then $(\mathcal{S},\mathcal{G}) \in \mathcal{CBN}_{\mathrm{nv}}$.*

Proof. We assume that both $\mathcal{R}$ and $\mathcal{R}'$ are nonempty, for otherwise the result follows from Theorem 3. Suppose to the contrary that $(\mathcal{S},\mathcal{G}) \notin \mathcal{CBN}_{\mathrm{nv}}$. According to Lemma 1 there exists a term $t \in \mathsf{WNR}(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$ without $(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$-needed redex. Assume without loss of generality that $\mathsf{root}(t) \in \mathcal{F}$. Lemma 5 yields a flat redex $\Xi \in \mathcal{T}(\mathcal{F})$. Let t' be the term in $\mathcal{T}(\mathcal{F})$ obtained from t by replacing every $\mathcal{G}\backslash\mathcal{F}$-alien by Ξ. Let P be the set of positions of those aliens. Since t' is reducible, it contains an $(\mathcal{R}_{\mathrm{nv}},\mathcal{F})$-needed redex, say at position q. We show that $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$. We consider two cases.

1. Suppose that $q \in P$. Since $t \in \mathsf{WNR}(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$, $t \to^*_{\mathcal{S}_{\mathrm{nv}},\mathcal{G}} \Delta$ for some redex $\Delta \in \mathsf{WN}(\mathcal{S}_{\mathrm{nv}},\mathcal{G}) \subseteq \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$. Since $\mathcal{S}_{\mathrm{nv}}$ is noncollapsing and the root symbol of every alien belongs to $\mathcal{G}\backslash\mathcal{F}$, aliens cannot contribute to the creation of Δ and hence we may replace them by arbitrary terms in $\mathcal{T}(\mathcal{G}_\bullet)$ and still obtain a redex that is pattern equal to Δ. We replace the alien at position q by $\bullet$ and every alien at position $p \in P \setminus \{q\}$ by $t'|_p$. This gives $t'[\bullet]_q \to^*_{\mathcal{S}_{\mathrm{nv}},\mathcal{G}_\bullet} \Delta'$ with $\Delta' \approx \Delta$. Lemma 2 yields $\Delta' \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$ and hence $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$.
2. Suppose that $q \notin P$. Since Ξ is flat, it follows by orthogonality that q is also a redex position in t. Since t is an $(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$-free term, $t[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$. Since t is not root-stable, its root symbol must be defined. From Lemma 4 we learn that a ground normal form of $\mathcal{S}$ (and thus of $\mathcal{S}_{\mathrm{nv}}$) cannot contain defined symbols. Hence any $(\mathcal{S}_{\mathrm{nv}},\mathcal{G}_\bullet)$-rewrite sequence that normalizes $t[\bullet]_q$ contains a root step and thus $t[\bullet]_q \in \mathsf{WNR}_\bullet(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$. Hence $t[\bullet]_q \to^*_{\mathcal{S}_{\mathrm{nv}},\mathcal{G}_\bullet} \Delta \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$ for some redex $\Delta \in \mathcal{T}(\mathcal{G}_\bullet)$. We replace the $\mathcal{G}\backslash\mathcal{F}$-aliens in $t[\bullet]_q$ by Ξ. Since $\mathcal{S}$ is noncollapsing, this yields $t'[\bullet]_q \to^*_{\mathcal{S}_{\mathrm{nv}},\mathcal{G}_\bullet} \Delta'$ with $\Delta' \approx \Delta$. Lemma 2 yields $\Delta' \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$ and hence $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}},\mathcal{G})$.

As $t' \in \mathcal{T}(\mathcal{F})$, we have $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}},\mathcal{G},\mathcal{F})$ and thus $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_{\mathrm{nv}},\mathcal{F})$ by Lemmata 7 and 8, contradicting the assumption that q is the position of an $(\mathcal{R}_{\mathrm{nv}},\mathcal{F})$-needed redex in t'. □

Theorem 6. *Let $(\mathcal{R},\mathcal{F})$ and $(\mathcal{R}',\mathcal{F}')$ be disjoint orthogonal TRSs and $\alpha \in \{\mathrm{s},\mathrm{nv}\}$ such that $\mathsf{WN}(\mathcal{R}_\alpha,\mathcal{G},\mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha,\mathcal{F})$ and $\mathsf{WN}(\mathcal{R}'_\alpha,\mathcal{G},\mathcal{F}') = \mathsf{WN}(\mathcal{R}'_\alpha,\mathcal{F}')$. If $(\mathcal{R},\mathcal{F}),(\mathcal{R}',\mathcal{F}') \in \mathcal{CBN}_\alpha$ and both $\mathcal{R}_\alpha$ and $\mathcal{R}'_\alpha$ are collapsing then $(\mathcal{S},\mathcal{G}) \in \mathcal{CBN}_\alpha$.*

Proof. We assume that both $\mathcal{R}$ and $\mathcal{R}'$ are nonempty, for otherwise the result follows from Theorem 3. Suppose to the contrary that $(\mathcal{S},\mathcal{G}) \notin \mathcal{CBN}_\alpha$. According to Lemma 1 there exists a term $t \in \mathsf{WNR}(\mathcal{S}_\alpha,\mathcal{G})$ without $(\mathcal{S}_\alpha,\mathcal{G})$-needed redex. Assume without loss of generality that $\mathsf{root}(t) \in \mathcal{F}'$. Lemma 5 yields a

flat $\mathcal{R}'_\alpha$-collapsing redex $\Xi \in \mathcal{T}(\mathcal{F}')$. Let t' be the term in $\mathcal{T}(\mathcal{F}')$ obtained from t by replacing every $\mathcal{G}\backslash\mathcal{F}'$-alien by Ξ. Let P be the set of positions of those aliens. Since t' is reducible, it contains an $(\mathcal{R}'_\alpha, \mathcal{F}')$-needed redex, say at position q. We show that $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$. Because Ξ is a collapsing redex, we have $\Xi \to_{\mathcal{R}_\alpha,\mathcal{G}} t|_p$ for all $p \in P$. Hence $t' \to^*_{\mathcal{R}_\alpha,\mathcal{G}_\bullet} t$ and thus, by orthogonality, $t'[\bullet]_q \to^*_{\mathcal{R}_\alpha,\mathcal{G}_\bullet} t[\bullet]_q$. Hence it suffices to show that $t[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$. We distinguish two cases.

1. Suppose that $q \in P$. Since $t \in \mathsf{WNR}(\mathcal{S}_\alpha, \mathcal{G})$, $t \to^*_{\mathcal{S}_\alpha,\mathcal{G}} \Delta$ for some redex $\Delta \in \mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G}) \subseteq \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$. We distinguish two further cases.
   a) If $t|_q$ is a normal form then it cannot contribute to the creation of Δ and hence by replacing it by $\bullet$ we obtain $t[\bullet]_q \to^*_{\mathcal{S}_\alpha,\mathcal{G}} \Delta'$ with $\Delta \approx \Delta'$. Lemma 2 yields $\Delta' \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$ and thus $t[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$.
   b) Suppose $t|_q$ is reducible. Because t is a minimal free term, $t|_q$ contains an $(\mathcal{S}_\alpha, \mathcal{G})$-needed redex, say at position q'. So $t|_q[\bullet]_{q'} \notin \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$. In particular, $t|_q[\bullet]_{q'}$ does not $(\mathcal{S}_\alpha, \mathcal{G})$-rewrite to a collapsing redex, for otherwise it would rewrite to a normal form in one extra step. Hence the root symbol of every reduct of $t|_q[\bullet]_{q'}$ belongs to $\mathcal{F}$. Since qq' is not the position of an $(\mathcal{S}_\alpha, \mathcal{G})$-needed redex in t, $t[\bullet]_{qq'} \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$. Since any normalizing $(\mathcal{S}_\alpha, \mathcal{G})$-rewrite sequence must contain a rewrite step at a position above q, we may write $t[\bullet]_{qq'} \to^*_{\mathcal{S}_\alpha,\mathcal{G}} C[\Delta'] \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$ such that Δ' is the first redex above position q. Since $\mathsf{root}(\Delta') \in \mathcal{F}'$, the subterm $t|_q[\bullet]_{q'}$ of $t[\bullet]_{qq'}$ does not contribute to the creation of Δ' and hence $t[\bullet]_q \to^*_{\mathcal{S}_\alpha,\mathcal{G}} C[\Delta'']$ with $\Delta'' \approx \Delta'$. Lemma 2 yields $C[\Delta''] \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$ and thus $t[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$.
2. Suppose that $q \notin P$. Since Ξ is flat, it follows by orthogonality that q is also a redex position in t. Since t is an $(\mathcal{S}_\alpha, \mathcal{G})$-free term, $t[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$.

As $t' \in \mathcal{T}(\mathcal{F}')$, we have $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F}')$ and thus $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}'_\alpha, \mathcal{F}')$ by Lemmata 7 and 8, contradicting the assumption that q is the position of an $(\mathcal{R}'_\alpha, \mathcal{F}')$-needed redex in t'. □

It is rather surprising that the presence of collapsing rules helps to achieve modularity; for most properties of TRSs collapsing rules are an obstacle for modularity (see e.g. Middeldorp [14]).

The next result is the modularity counterpart of Theorem 1. It is an easy corollary of the preceding theorem.

Theorem 7. *The subclass of $\mathcal{CBN}_\mathrm{s}$ consisting of all orthogonal TRSs $(\mathcal{R}, \mathcal{F})$ such that $\mathsf{NF}(\mathcal{R}, \mathcal{F}) \neq \varnothing$ is modular.*

Klop and Middeldorp [13] showed the related result that strong sequentiality is a modular property of orthogonal TRSs. We already remarked in the preceding section that $\mathcal{CBN}_\mathrm{s}$ properly includes all strongly sequential TRSs. Actually, in [13] it is remarked that it is sufficient that the left-hand sides of the two strongly sequential rewrite systems do not share function symbols. One easily verifies that for our modularity results it is sufficient that $\mathcal{R}_\alpha$ and $\mathcal{R}'_\alpha$ do not share function symbols. Actually, we can go a step further by considering so-called

constructor-sharing combinations. In such combinations the participating systems may share constructors but not defined symbols. It can be shown that the results obtained in this section extend to constructor-sharing combinations, provided we strengthen the requirements in Theorems 4, 5, and 6 by forbidding the presence of *constructor-lifting* rules. A rewrite rule $l \to r$ is called constructor-lifting if $\mathsf{root}(r)$ is a shared constructor. The necessity of this condition is shown in the following examples.

Example 8. Consider the TRS

$$\mathcal{R} = \left\{ \begin{array}{lcl} \mathsf{f}(x, \mathsf{c}(\mathsf{a}), \mathsf{c}(\mathsf{b})) & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{c}(\mathsf{b}), x, \mathsf{c}(\mathsf{a})) & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{c}(\mathsf{a}), \mathsf{c}(\mathsf{b}), x) & \to & \mathsf{a} \end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules and the TRS $\mathcal{R}' = \{ \mathsf{g}(x) \to \mathsf{c}(x) \}$ over the signature $\mathcal{F}'$ consisting of a constant d in addition to g and c. Both TRSs have external normal forms, lack collapsing rules, and belong to $\mathcal{CBN}_{\mathrm{nv}}$. Their union does not belong to $\mathcal{CBN}_{\mathrm{nv}}$ as the term $\mathsf{f}(\mathsf{g}(\mathsf{a}), \mathsf{g}(\mathsf{a}), \mathsf{g}(\mathsf{a}))$ has no $(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$-needed redex. Note that $\mathcal{R}$ and $\mathcal{R}'$ share the constructor c and hence $\mathsf{g}(x) \to \mathsf{c}(x)$ is constructor-lifting.

A simple modification of the above example shows the necessity of forbidding constructor-lifting rules in Theorem 5 even if we require that the constituent CSs lack external normal forms.

Example 9. Consider the TRSs

$$\mathcal{R} = \left\{ \begin{array}{lcllcl} \mathsf{f}(x, \mathsf{a}, \mathsf{b}) & \to & \mathsf{c}(\mathsf{g}(x)) & \mathsf{g}(x) & \to & \mathsf{g}(\mathsf{a}) \\ \mathsf{f}(\mathsf{b}, x, \mathsf{a}) & \to & \mathsf{c}(\mathsf{g}(x)) & \mathsf{h}(x) & \to & x \\ \mathsf{f}(\mathsf{a}, \mathsf{b}, x) & \to & \mathsf{c}(\mathsf{g}(x)) & & & \end{array} \right\}$$

and

$$\mathcal{R}' = \left\{ \begin{array}{rcl} \mathsf{i}(\mathsf{a}) & \to & \mathsf{a} \\ \mathsf{i}(\mathsf{c}(x)) & \to & x \end{array} \right\}$$

over the signatures $\mathcal{F}$ and $\mathcal{F}'$ consisting of function symbols that appear in their respective rewrite rules. The two TRSs are obviously collapsing and share the constructors a and c. One easily verifies that both TRSs belong to $\mathcal{CBN}_{\mathrm{nv}}$ and that $\mathsf{WN}(\mathcal{R}_{\mathrm{nv}}, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_{\mathrm{nv}}, \mathcal{F})$ and $\mathsf{WN}(\mathcal{R}_{\mathrm{nv}}, \mathcal{G}, \mathcal{F}') = \mathcal{T}(\mathcal{F}') = \mathsf{WN}(\mathcal{R}'_{\mathrm{nv}}, \mathcal{F}')$. However, the union of the two TRSs does not belong to $\mathcal{CBN}_{\mathrm{nv}}$ as the term $\mathsf{i}(\mathsf{f}(\Delta, \Delta, \Delta))$ with any collapsing redex Δ has no $(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$-needed redex.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. H. Comon. Sequentiality, monadic second-order logic and tree automata. *Information and Computation*, 157:25–51, 2000.

3. N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.
4. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.
5. I. Durand. Bounded, strongly sequential and forward-branching term rewriting systems. *Journal of Symbolic Computation*, 18:319–352, 1994.
6. I. Durand and A. Middeldorp. Decidable decidable call by need computations in term rewriting (extended abstract). In *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *LNAI*, pages 4–18, 1997.
7. I. Durand and A. Middeldorp. On the complexity of deciding call-by-need. Technical Report 1194-98, LaBRI, Université de Bordeaux I, 1998.
8. B. Gramlich. *Termination and Confluence Properties of Structured Rewrite Systems*. PhD thesis, Universität Kaiserslautern, 1996.
9. G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems, I and II. In *Computational Logic, Essays in Honor of Alan Robinson*, pages 396–443. The MIT Press, 1991. Original version: Report 359, Inria, 1979.
10. F. Jacquemard. Decidable approximations of term rewriting systems. In *Proceedings of the 7th International Conference on Rewriting Techniques and Applications*, volume 1103 of *LNCS*, pages 362–376, 1996.
11. R. Kennaway, J.W. Klop, R. Sleep, and F.-J. de Vries. Comparing curried and uncurried rewriting. *Journal of Symbolic Computation*, 21(1):15–39, 1996.
12. J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Vol. 2*, pages 1–116. Oxford University Press, 1992.
13. J.W. Klop and A. Middeldorp. Sequentiality in orthogonal term rewriting systems. *Journal of Symbolic Computation*, 12:161–195, 1991.
14. A. Middeldorp. *Modular Properties of Term Rewriting Systems*. PhD thesis, Vrije Universiteit, Amsterdam, 1990.
15. T. Nagaya and Y. Toyama. Decidability for left-linear growing term rewriting systems. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *LNCS*, pages 256–270, 1999.
16. E. Ohlebusch. *Modular Properties of Composable Term Rewriting Systems*. PhD thesis, Universität Bielefeld, 1994.
17. M. Oyamaguchi. NV-sequentiality: A decidable condition for call-by-need computations in term rewriting systems. *SIAM Journal on Computation*, 22:114–135, 1993.
18. R. Strandh. Classes of equational programs that compile into efficient machine code. In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, volume 355 of *LNCS*, pages 449–461, 1989.
19. T. Takai, Y. Kaji, and H. Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In *Proceedings of the 11th International Conference on Rewriting Techniques and Applications*, volume 1833 of *LNCS*, pages 246–260, 2000.
20. Y. Toyama. Strong sequentiality of left-linear overlapping term rewriting systems. In *Proceedings of the 7th IEEE Annual Symposium on Logic in Computer Science*, pages 274–284, 1992.
21. Y. Toyama, S. Smetsers, M. van Eekelen, and R. Plasmeijer. The functional strategy and transitive term rewriting systems. In *Term Graph Rewriting: Theory and Practice*, pages 61–75. Wiley, 1993.

Synchronized Tree Languages Revisited and New Applications

Valérie Gouranton[1], Pierre Réty[1], and Helmut Seidl[2]

[1] LIFO, Université d'Orléans, France.
{gouranto,rety}@lifo.univ-orleans.fr
[2] Dept. of Computer Science, University of Trier, Germany.
seidl@psi.uni-trier.de

Abstract. We present a new formulation for tree-tuple synchronized languages, much simpler than the existing one. This new formulation allows us to prove stronger structural results. As a consequence, synchronized languages give rise to new applications:

- to rewriting: given tree languages L_1 (synchronized), L_2 (regular), $Rel(L_1) \subseteq L_2$ is decidable for several rewrite-like relations Rel.
- to concurrency: we prove decidability of the logic EF for a process calculus allowing some bounded form of communication. Consequently, the absence of deadlocks is decidable.

Keywords: tree-tuple language, rewriting, concurrency.

1 Introduction

In the field of tree[1] languages, let us consider the so-called *tree-tuple synchronized languages*, i.e. the languages generated by *Tree-Tuple Synchronized Grammars* (TTSG for short). TTSG's have been introduced to solve some equational unification [4,5] and disunification [6] problems. They have next been applied to logic program validation [8], and one-step rewriting theory [7]. Before going on, it is necessary to recall what a TTSG is, using an example. The following TTSG contains four packs of synchronized productions:

$$\begin{cases} X \Rightarrow f(X, X') \\ Y \Rightarrow f(Y, Y') \end{cases} \quad \begin{cases} X \Rightarrow b \\ Y \Rightarrow b \end{cases} \quad \begin{cases} X' \Rightarrow f(X, X') \\ Y' \Rightarrow f(Y, Y') \end{cases} \quad \begin{cases} X' \Rightarrow b \\ Y' \Rightarrow b \end{cases}$$

The first pack means that if X is derived into $f(X, X')$, then Y must be derived into $f(Y, Y')$ at the same time. Therefore if X appears in a given tree and Y does not, the tree cannot be derived by the first pack. If the axiom is the pair of non-terminals (X, Y), a possible derivation is:

$$(X, Y) \Rightarrow (f(X, X'),\ f(Y, Y')) \Rightarrow (f(X, f(X, X')),\ f(Y, f(Y, Y')))$$

Now there is an ambiguity: which X should be synchronized (derived at the same time) with which Y? To remove this ambiguity, a control (an integer) is attached to each non-terminal, along the derivation, and is increased to a new

[1] Trees are first-order terms.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 214–229, 2001.

value whenever a pack of productions is applied. And the rule is : only non-terminals having the same control can be derived at the same time. Thus the above derivation is actually (For readability, control is written as non-terminal index) :

$$(X_0, Y_0) \Rightarrow (f(X_1, X_1'),\ f(Y_1, Y_1')) \Rightarrow (f(X_1, f(X_2, X_2')),\ f(Y_1, f(Y_2, Y_2')))$$

Now, an X and a Y can be derived at the same time only if they appear at identical positions in the two trees. Thus, this grammar generates the pairs of identical terms.

To get something interesting and more general for applications, control is indispensable. Unfortunately, this simple control is not enough when computing the natural join of two tree-tuple synchronized languages : we get a TTSG that needs a more complicated control (pairs of integers). Worse, when computing several joins incrementally, each step needs a control more complicated than the previous one (tuples of integers, see [5] for details). In this paper :

- We present a simple formalism which essentially is equivalent to TTSG's of level 1 (i.e., those with an integer as control) but elegantly circumvents the use of controls. The idea is that in the new formalism, a non-terminal will represent tuples of synchronizable[2] trees (instead of single trees) from which we will extract components, as needed. This new grammar formalism is given in the form of constraint systems for which we consider their least fix-point solutions. In other words, we adopt the bottom-up point of view instead of the top-down one[3].
- Instead of formally proving the equivalence of the two formalisms, we prefer to present the vastly simplified proofs of existing results using the new formalism, and also take advantage of its simplicity by proving new results. In particular, we show closure by union and cartesian product (existing results), as well as closure by projection (new) and, as our key technical result, also closure under join (new[4]). Moreover, we present a linear time emptiness for constraints (an exponential-time pumping technique, very complicated because of control, was used for TTSG's), and a rather efficient membership test (comparable to [12] but much simpler).
- Thanks to the new results, we get new applications :
  - to rewriting : given tree languages L_1 (synchronized, then non-necessarily regular), L_2 (regular), we prove that $Rel(L_1) \subseteq L_2$ is decidable for one-step, parallel, one-pass, one-pass root-started rewritings.
  - to concurrency : we extend the work of [10] on PA-processes by introducing bounded communications. We get a new concurrency formalism for which we show that the whole logic EF is decidable. In particular, we are able to verify that a given process is deadlock free.

[2] I.e. that had identical control values.

[3] Due to the essential equivalence, the reader may still also interpret constraint systems as grammars.

[4] This amounts to show that TTSG's of level $n > 1$ are actually useless.

2 Constraint Systems for Tuple Synchronized Languages

Example 2.1. This is the example given in the introduction, but now expressed with a constraint system. In the signature $\Sigma = \{f^{\setminus 2}, b^{\setminus 0}\}$ let $L_{id} = \{(t,t) \mid t \in T_\Sigma\}$ be the set of pairs of identical terms. L_{id} can be defined by the following grammar, given in the form of a constraint system:

$$\begin{array}{l} X_{id} \supseteq (b, b) \\ X_{id} \supseteq (f(1_1, 2_1),\ f(1_2, 2_2))\ (X_{id}, X_{id}) \end{array}$$

where $1_1, 2_1, 1_2, 2_2$ abbreviate pairs (for readability). For example 2_1 means $(2,1)$, which denotes the first component of the second argument (the second X_{id}). Note that since 1_1 and 1_2 come from the same X_{id}, they represent two identical terms, in other words they are linked (synchronized), whereas for example 1_1 and 2_1 are independent.

Example 2.2. Now if we consider the slightly different constraint system :

$$\begin{array}{l} X_{sym} \supseteq (b, b) \\ X_{sym} \supseteq (f(1_1, 2_1),\ f(2_2, 1_2))\ (X_{sym}, X_{sym}) \end{array}$$

we get the set $L_{sym} = \{(t, t_{sym}) \mid t_{sym}$ is the symmetric of $t\}$.

Example 2.3. In the signature $\Sigma = \{s^{\setminus 1}, b^{\setminus 0}\}$ let $L_{dble} = \{(s^n(b), s^{2n}(b))\}$. It can be defined by the constraint system :

$$\begin{array}{l} X_{dble} \supseteq (b, b) \\ X_{dble} \supseteq (s(1_1),\ s(s(1_2)))\ X_{dble} \end{array}$$

General Formalization

Assume we are given a (universal) index set N for tuple components. For $I \subseteq N$ and any set M, the set of I-tuples $a : I \to M$ is denoted by M^I. Often, we also write $a = (a_i)_{i \in I}$ provided $a(i) = a_i$ which for $I = \{1, \ldots, k\} \subseteq \mathbb{N}$, is also written as $a = (a_1, \ldots, a_k)$.

Different tree-tuple languages may refer to tuples of different length, or, to different index sets. Our constraint variables represent tree-tuple languages. Consequently, they have to be equipped with the intended index set. Such an assignment is called *classification*. Accordingly, a *classified* set of tuple variables (over N) is a pair $(\mathcal{X}, \rho)$ where $\rho : \mathcal{X} \to 2^N$ assigns to each variable X a subset of indices. This subset is called the *class* of X. For convenience and whenever ρ is understood, we omit ρ and denote the classified set $(\mathcal{X}, \rho)$ by $\mathcal{X}$. The maximal cardinality of the classes in $\mathcal{X}$ is also called the *width* of $\mathcal{X}$. In particular, in example 2.1, $N = \{1, 2\}$, $\mathcal{X} = \{X_{id}\}$, and $\rho(X_{id}) = \{1, 2\}$. Thus, the width of $\mathcal{X}$ is 2.

A constraint system for tree-tuple languages consists of a classified set $(\mathcal{X}, \rho)$ of constraint variables, together with a set $\mathcal{E}$ of inequations of the form

$$X \supseteq \square(X_1, \ldots, X_k) \qquad (1)$$

where $X, X_1, \ldots, X_k \in \mathcal{X}$ and $\Box$ is an operator mapping the concatenation of tuples for the variables X_i to tuples for X. More precisely, let

$$J = \{(i, x) \mid 1 \leq i \leq k, x \in \rho(X_i)\} \tag{2}$$

denote the *disjoint union* of the index sets corresponding to the variables X_i (in example 2.1, $J = \{(1,1), (1,2), (2,1), (2,2)\}$ abbreviated into $\{1_1, 1_2, 2_1, 2_2\}$). Then $\Box$ denotes a mapping $T_\Sigma^J \to T_\Sigma^{\rho(X)}$. Each component of this mapping is specified through a tree expression t which may access the components of the argument tuple and apply constructors from signature Σ. Thus, t can be represented as an element of $T_\Sigma(J)$ where $T_\Sigma(J)$ denotes all trees over Σ which additionally may contain nullary symbols from the index set J.

Consider, e.g., the second constraint in example 2.1. There, the first component of the operator is given by $t = f(1_1, 2_1)$.

The mapping induced by such a tree t then is defined by

$$t(s_j)_{j \in J} = t\{j \mapsto s_j\}_{j \in J}$$

for every $(s_j)_{j \in J} \in T_\Sigma^J$. Accordingly, $\Box$ is given by a tuple

$$\Box \in T_\Sigma(J)^{\rho(X)}$$

Let us collect a set of useful special forms of constraint systems. The constraint (1) is called

- *non-copying* iff no index $j \in J$ occurs twice in $\Box$;
- *irredundant* iff $\Box$ refers to each argument at least once, i.e., $\Box$ contains at least one occurrence of some (j, x_j) for each j;
- *empty* iff $\Box$ contains no constructor application, i.e., $\Box_x \in J$ for all $x \in \rho(X)$;
- *Greibach* iff each component $\Box_x$ either is contained in J or is a single constructor application $a(j_1, \ldots, j_n)$, $a \in \Sigma, j_1, \ldots, j_n \in J$;
- *regular*[5] iff each component of $\Box$ is a single constructor application of the form $\Box_x = a((1, x), \ldots, (n, x))$ for each $x \in \rho(X)$.

The whole constraint system is called non-copying, (irredundant, non-empty, Greibach, regular) iff each constraint in $\mathcal{E}$ is so.

For example, the constraint systems that define L_{id} and L_{sym} are non-copying, irredundant, and Greibach. Moreover L_{id} is regular. On the other hand, L_{dble} is not Greibach. It is always possible to write any constraint system in Greibach form. Thus the Greibach form of L_{dble} is :

$$\begin{array}{ll} X_{dble} \supseteq (b, b) & \\ X_{dble} \supseteq (s(1_1),\ s(1_2))\ Y & \qquad Y \supseteq (1_1,\ s(1_2))\ X_{dble} \end{array}$$

A variable assignment is a mapping σ assigning to each variable $X \in \mathcal{X}$ a subset of $T_\Sigma^{\rho(X)}$ (i.e. a set of tuples of ground terms). Variable assignment σ satisfies the constraint (1) iff

$$\sigma(X) \supseteq \{\Box(t_1, \ldots, t_k) \mid t_i \in \sigma(X_i)\} \tag{3}$$

[5] This corresponds to the notion of regular relation, i.e. the tree language over the product-alphabet obtained by overlapping the tuple components is regular.

Note here that the $\Box$-operator is applied, in fact, to the cartesian product of the argument sets $\sigma(X_i)$. In particular, the tuples inside the argument sets are kept "synchronized" while tuples from different argument sets may be arbitrarily combined.

The variable assignment σ is a *solution* of the constraint system iff it satisfies all constraints in the system. Since, the operator application in (3) is monotonic, even continuous (w.r.t. set inclusion of tuple languages) we conclude that each constraint system has a unique least solution.

3 Properties

3.1 Deciding Emptiness

The fix-point characterization of languages defined by tuple language constraint systems, allows us immediately to derive a polynomial emptiness test. In order to do that, we first observe that we can always remove from a constraint $X \supseteq \Box\,(X_1, \ldots, X_k)$ all arguments X_j which do not contribute to the components of X. Consequently, we have:

Proposition 3.1. *For every constraint system $\mathcal{C}$ we can construct in linear time a irredundant constraint system $\mathcal{C}'$ such that $\mathcal{C}$ and $\mathcal{C}'$ have the same least solution.*

For irredundant systems, however, emptiness can be decided along the same lines as for tree automata. For the constraint system $\mathcal{C} = (\mathcal{X}, \mathcal{E})$ we (conceptually) introduce a new ranked alphabet Δ which contains a new letter a_r for each constraint r in $\mathcal{E}$ where a_r is of rank k provided the right-hand of r refers to k variables. Then the *uninterpreted* regular constraint system $\mathcal{C}_0 = (\mathcal{X}_0, \mathcal{E}_0)$ of $\mathcal{C}$ is obtained as follows:

- $\mathcal{X}_0$ is obtained from $\mathcal{X}$ by forgetting the classification.
- $\mathcal{E}_0$ is obtained from $\mathcal{E}$ by replacing each constraint r of the form $X \supseteq \Box\,(X_1, \ldots, X_k)$ with the constraint $X \supseteq a_r\,(X_1, \ldots, X_k)$.

Theorem 3.2. *Let $\mathcal{C}$ denote a constraint system and $\mathcal{C}_0$ the uninterpreted regular constraint system of $\mathcal{C}$. Furthermore, let σ and σ_0 denote the least solutions of $\mathcal{C}$ and $\mathcal{C}_0$, respectively. Then the following holds:*

1. *If $\mathcal{C}$ is irredundant, then for every $X \in \mathcal{X}$, $\sigma\, X \neq \emptyset$ iff $\sigma_0\, X \neq \emptyset$.*
2. *The set $\{X \in \mathcal{X} \mid \sigma\, X \neq \emptyset\}$ can be computed in time linear in the size of $\mathcal{C}$.*

Indeed, theorem 3.2 does not come as a surprise. Taking a closer look at our definition of tuple languages, reveals that these can be viewed as polynomial systems over a specific algebra of graphs (cf. [2]).

3.2 Basic Closure Properties

We are actually interested in non-copying constraint systems, which are enough to formalize tree-tuple synchronized languages, because there is no copying in TTSG's. Let $\mathcal{L}$ denote the class of languages defined by non-copying constraint systems. We now collect some basic closure properties of languages in $\mathcal{L}$. Obviously, $\mathcal{L}$ is closed under union and cartesian product.

Proposition 3.3. *Assume that $L_1, L_2 \in \mathcal{L}$ are tree-tuple languages of ranks I_1 and I_2, respectively. Then we have:*

- *If $I_1 = I_2$, then $L_1 \cup L_2$ is in $\mathcal{L}$, and*
- *If $I_1 \cap I_2 = \emptyset$, then $L_1 \times L_2 \in \mathcal{L}$ where for $I = I_1 \cup I_2$, the* cartesian product *$L_1 \times L_2$ is defined as the set of all I-tuples*

$$L_1 \times L_2 = \{t \in T_\Sigma^I \mid t|_{I_i} \in L_i\}$$

The class of tree-tuple languages defined through constraint systems is *not* closed under intersection – even with a regular relation. The reason intuitively is that through intersection, we can construct the set of all complete binary trees which cannot be defined by any constraint system as considered here.

Finally, let us remark that $\mathcal{L}$ is closed under projections.

Proposition 3.4. *Assume that $L \subseteq T_\Sigma^I$ and $A \subseteq I$. then also the projection $\pi_A(L)$ of L onto A is in $\mathcal{L}$ where the* projection *$\pi_A(L)$ is defined by*

$$\pi_A(L) = \{t|_A \mid t \in L\}$$

Although not stated explicitly in the propositions 3.3 and 3.4, the constructions for union and cartesian product can be implemented in linear lime. The same clearly also holds for projection.

3.3 Join and Membership

Another operation on languages of tree-tuples is the *join* operation. Assume that I_1, I_2 are two index sets with $I_1 \cap I_2 = \{x\}$. The *join* of two languages $L_i \subseteq T_\Sigma^{I_i}$ is defined by

$$L_1 \bowtie L_2 = \{t \in T_\Sigma^I \mid t|_{I_i} \in L_i\}$$

where $I = I_1 \cup I_2$. In case that the index sets are disjoint, we may select elements $x_1 \in I_1$ and $x_2 \in I_2$ which are to be identified. In order to do so, we construct the cartesian product of the two languages, select all tuples t where $t_{x_1} = t_{x_2}$ and then project onto the set $I = I_1 \cup I_2 \backslash \{x_2\}$. This operation is denoted by $L_1 \bowtie_{x_1,x_2} L_2$. Note that this operation can also be implemented by renaming the component x_2 of the language L_2 to x_1 and then apply the ordinary join. In general, the family $\mathcal{L}$ will not be closed under joins. An exception is the case where $I_1 = \{x\}$ and L_1 is a regular set of trees. Then $L_1 \bowtie L_2 = \{t \in L_2 \mid t_x \in L_1\}$, and we find:

Proposition 3.5. *If $L_1 \subseteq T_\Sigma^{\{x\}}$ is regular, and $L \subseteq T_\Sigma^I$ is in $\mathcal{L}$ with $x \in I$, then $L' = L_1 \bowtie L$ is in $\mathcal{L}$ as well. Given a finite tree automaton of size m for L_1 and a constraint system of size n for L using a classified set of variables of width r, the construction of a constraint system for L' can be performed in time $\mathcal{O}(n \cdot (m+1)^r)$.*

Prop. 3.5 can be used to compute the *intersection* of cartesian products of regular sets with languages from $\mathcal{L}$.

Proposition 3.6. *Assume $I = \{x_1, \ldots, x_k\}$ for pairwise different indices x_i, $L_i \subseteq T_\Sigma^{\{x_i\}}, i = 1, \ldots, k$, are regular sets, and $L \subseteq T_\Sigma^I$ is in $\mathcal{L}$. Then the intersection $L' = (L_1 \times \ldots \times L_k) \cap L$ is in $\mathcal{L}$ as well. Given finite tree automata of joint size m for the L_i and a constraint system of size n for L using a classified set of variables of width r, the construction of a constraint system for L' can be performed in time $\mathcal{O}(n \cdot (m+1)^r)$.*

Indeed, prop. 3.6 follows directly from prop. 3.5, since

$$L' = L_1 \bowtie (L_2 \bowtie \ldots (L_k \bowtie L) \ldots)$$

In order to derive the given complexity bound we have to recall that each component of a class on the right-hand side can be used at most once on the left hand side – implying that, when keeping track of the states of components, we only have to track the state of one of the r automata for each component. Nonetheless, as the width of the set of variables of the constraint system for L occurs in the exponent of the complexity estimation, we conclude that the corresponding construction will only be feasible for "small" index sets I. Sadly enough, the complexity estimation cannot be easily improved. In order to see this, recall that there is a simple constraint system for the language

$$Id^k = \{(t, \ldots, t) \mid t \in T_\Sigma\} \subseteq T_\Sigma^k$$

The intersection of Id^k with the cartesian product $L_1 \times \ldots \times L_k$ of regular tree languages L_i is nonempty iff $L_1 \cap \ldots \cap L_k \neq \emptyset$. Deciding the emptiness of the intersection of a sequence of regular tree languages, however, is well-known to be complete for DEXPTIME [13]. Now since emptiness for our constraint systems can be decided in linear time, we have succeeded in reducing a DEXPTIME complete problem to our intersection construction.

Prop. 3.6, however, offers a conceptually simple method for solving the membership problem. Since singleton sets of trees are trivially regular, we obtain:

Theorem 3.7. *Given a language $L \subseteq T_\Sigma^I$ in $\mathcal{L}$ and a tree-tuple $t \in T_\Sigma^I$, it can be decided whether or not $t \in L$. Assuming that L is given through a constraint system of size n using a classified set of variables of width r, membership of t in L can be decided in time $\mathcal{O}(n \cdot (|t|+1)^r)$.*

Let us now return to prop. 3.5. The basic idea of the construction used in the proof there consists in the observation that, while building up tuples according to a constraint system, we very well may keep track of the states of a finite tree automaton on all components of classes that contribute to a given component.

Indeed, this idea can be generalized. A component x of a class X will be considered as *free* if its values trees are generated by the constraint system in a "regular way". By this we mean that no use is ever made of several components from the same variable, i.e., "internal synchronization".

Example 3.8. Consider the constraint system :

$$X \supseteq (a,\ f(1_1, 1_2))\ X_{id}$$

where X_{id} is defined as previously. In X, the second component is not free, since both arguments of f come from the same variable implying that their values are potentially inter-related and indeed, the projection of X onto its second argument is not regular. On the other hand, the first component is trivially free. Now let us add the constraint :

$$Y \supseteq s(1_2)\ X$$

Then the component of Y is not free either because the second component of X is used to define Y which is not free.

In order to make this notion formal, assume that we are given a constraint system $(\mathcal{X}, \mathcal{E})$ which is Greibach and non-copying. The relation $\mathsf{free} \subset N \times \mathcal{X}$ is given as the maximal relation $\mathcal{F}$ such that for each $(x, X) \in \mathcal{F}$, $x \in \rho(X)$ and for all constraints $X \supseteq \Box(X_1, \ldots, X_k)$ in $\mathcal{E}$ and $t = \Box_x$ the following holds:

1. If $(i, y) \in I$ occurs in t then $(y, X_i) \in \mathcal{F}$;
2. If $(i_1, y_1) \neq (i_2, y_2)$ occur in t then $i_1 \neq i_2$.

Since properties (1) and (2) are preserved by unions, free is well-defined. If $(x, X) \in \mathsf{free}$, we also say that component x is *free* in X. The second property above then essentially implies that there are no "internal synchronizations" in free components.

We call a component x free w.r.t. a language L (from $\mathcal{L}$) if there is a constraint system $(\mathcal{X}, \mathcal{E})$ with least solution σ such that $L = \sigma\, X$ for some $X \in \mathcal{X}$ where x is free in X.

Proposition 3.9. *If x is free in the language L (from $\mathcal{L}$), then the projection $\pi_{\{x\}}(L)$ of L onto the component x is a regular tree language.*

Here is the main result of this section.

Theorem 3.10. *Assume $L_i \subseteq T_\Sigma^{I_i}$, $i = 1, 2$, are in $\mathcal{L}$ where I_1 and I_2 are disjoint. Then the join $L_1 \bowtie_{x_1,x_2} L_2$ is in $\mathcal{L}$ whenever x_1 is free in L_1. Moreover if x_3 is free in L_2, then x_3 is still free in $L_1 \bowtie_{x_1,x_2} L_2$.*

A proof of this theorem can be found in the full version [3].

4 Application to Rewriting

For a binary relation on terms Rel and two regular tree languages L_1, L_2, the decidability question $Rel(L_1) \subseteq L_2$ has already been studied in [14] assuming Rel is a rewrite-like relation. Now, if $Rel \in \mathcal{L}$ (i.e. is defined by a non-copying constraint system), we also get a decidability result (the proof is in [3]).

Proposition 4.1. *If $L_1, Rel \in \mathcal{L}$ and L_2 is regular, then*

$$\left.\begin{array}{r} L_1 \text{ is regular} \\ \text{or the first component of } Rel \text{ is free}^6 \end{array}\right\} \implies Rel(L_1) \subseteq L_2 \text{ is decidable}$$

Note that this result can be used incrementally to compose relations. If the first component of each relation $Rel_1, \ldots, Rel_n$ is free, then $Rel_1(\ldots\ldots(Rel_n(L_1)\ldots) \subseteq L_2$ is still decidable.

Concerning rewrite-like relations, we can easily encode one-step rewriting, parallel rewriting by non-copying constraint systems (see the full version [3]). If the rewrite system is entirely linear we can also encode one-pass rewriting and one-pass root-started rewriting. Thus, if L_1 is regular, we get the same decidability results as in [14] (or weaker because of linearity). On the other hand, if L_1 is not regular, for example if L_1 is composed of the instances of a non-linear term, we get new results, assuming however left-linearity for one-step rewriting and parallel rewriting, to ensure the freeness of the first component of Rel.

Moreover, assuming left-linearity, we can compose n times the relation one-or-zero-step rewriting with itself, and we get the decidability of $\xrightarrow{\leq n}(L_1) \subseteq L_2$, where $\xrightarrow{\leq n}$ is the rewrite relation in no more than n steps.

5 Application to Concurrency

We introduce a new concurrency formalism, the *Bounded-Communication Process Calculus* (BCPC). The idea is : when running two processes in parallel, they cannot communicate to each other if either (or both) has performed too many statements (actions), because it lasts too long. In other words, communication channels are not kept indefinitely, and nothing new is tried after the time limit. We think that this formalism could apply to check some properties in communication protocols.

Since everything is assumed to be time-limited, and to simplify the formalization, we consider that process derivation rules are folded into bigger ones, that simulate the time limit. For example, given the derivation rules $\{x \xrightarrow{a} y \| z,\ y \xrightarrow{b} t\}$ and considering that communication is not allowed beyond two actions, the derivation rules are replaced by $x \xrightarrow{c} t \| z$ where c is a new action representing the sequence $a.b$. Thus, the process $x \| t'$ derives in one step, with

[6] I.e. the projection of Rel on the first component is a regular language.

action c, into $(t\|z)\|t'$, and $t\|z$ will not be allowed to communicate with t' because we consider that applying one rule means that the time limit is reached, i.e. the new subterm created by the rule cannot communicate with the rest of the world anymore. Obviously, the user may fold the rules as he likes, and may for instance consider that all actions do not have the same durations.

Starting from a process name x, we are interested in testing properties on processes derived from x. Properties are expressed by means of the temporal logic CTL on an infinite structure[7], restricted to operators EF and EX, and we show that the model-checking problem is decidable. As a consequence, the absence of deadlocks is decidable.

To prove this, we express the transitive closure $\rightarrow^*$ of the process-derivation relation by a constraint system[8], and apply constraint system properties. For readability, we first express $\rightarrow^*$ for the PA formalism (i.e. without communication), which we then extend to BCPC by introducing communication (also called synchronization).

5.1 PA

PA introduced in [1] is a process algebra which permits non-determinism, sequential and parallel compositions, and recursion.

Syntax and semantics. *Act* is a set of *action names*, $Act = \{a, b, c, \ldots\}$. *Const* is a finite set of *process constants* (or process names), $Const = \{x, y, z, \ldots\}$. T is the set $\{t_1, t_2, \ldots\}$ of PA-terms defined by the following equation :

$$t_1, t_2 ::= 0 \ \mid \ x \ \mid \ t_1 \parallel t_2 \ \mid \ t_1 \ . \ t_2$$

where $x \in Const$. The interpretation of the syntax expressions is the following : 0 represents the process which performs no events, $t_1 \parallel t_2$ the parallel composition, and $t_1 \ . \ t_2$ represents the sequential composition.

A PA declaration is a finite family of recursive process rewrite rules :

$$\Delta = \{x_i \stackrel{a_i^j}{\rightarrow} t_i^j \mid x_i \in Const,\ t_i^j \in T,\ i = 1, \ldots, n,\ j = 1, \ldots, k_i\}$$

We define $\Delta(x) = \{t \mid (x \stackrel{a}{\rightarrow} t) \in \Delta\}$. $\Delta(x) = \emptyset$ denotes that there is no rule $(x \stackrel{a}{\rightarrow} t)$ belonging to Δ.

A family Δ of process rewrite rules determines a labeled transition relation $\rightarrow_\Delta \subseteq T \times Act \times T$. We omit the Δ and we note $t \stackrel{a}{\rightarrow} t'$ for $(t, a, t') \in \rightarrow$ with $t, t' \in T$ and $a \in Act$. The semantics is defined in Figure 1 by a Structural Operational Semantics [11]. In the parallel composition $\parallel$, the two processes t_1 and t_2 are evaluated independently. In the sequential composition $t_1 \ . \ t_2$, t_1 is first computed, and then t_2 is evaluated when the process t_1 is terminated. $Const(t)$ denotes the set of process constants occurring in term t. The predicate *Finished* represents the information of process termination. A term t is finished when it contains no process constants : $Finished(t) = (Const(t) = \emptyset)$.

[7] States are process terms.

[8] For concision, in this section constraint system stands for non-copying constraint system.

$$x \xrightarrow{a} t \quad \text{if } (x \xrightarrow{a} t) \in \Delta$$

$$\frac{t_1 \xrightarrow{a} t_1'}{t_1 \parallel t_2 \xrightarrow{a} t_1' \parallel t_2} \qquad \frac{t_2 \xrightarrow{a} t_2'}{t_1 \parallel t_2 \xrightarrow{a} t_1 \parallel t_2'}$$

$$\frac{t_1 \xrightarrow{a} t_1'}{t_1 \,.\, t_2 \xrightarrow{a} t_1' \,.\, t_2} \qquad \frac{t_2 \xrightarrow{a} t_2'}{t_1 \,.\, t_2 \xrightarrow{a} t_1 \,.\, t_2'} \quad \text{if } Finished(t_1)$$

Fig. 1. The semantics

Constraint system. We use the standard notation $\rightarrow^*$ for transitive closure of the rewrite relation (labels are omitted). The set $Pre^*(t)$ denotes the iterated predecessors of the term $t : Pre^*(t) = \{t' \in T \mid t' \xrightarrow{*} t\}$ and the set $Post^*(t)$ denotes the iterated successors of the term $t : Post^*(t) = \{t' \in T \mid t \xrightarrow{*} t'\}$.

The constraint system G (axiom A) for $\rightarrow^*$ is presented[9] in Figure 2. We note $L(N)$ the language generated from the non-terminal N. $L(A)$ represents the language of pairs (t, t') where $t \in T$ and $t' \in Post^*(t)$. In this definition, we use $Sub(\Delta) = \{s \mid s \text{ is a subterm of } t, t \in \Delta(x), x \in Const\}$. We use the non-terminal A_t to generate $Post^*(t)$. For each $x \in Const$, the non-terminal X is introduced to define $Post^+(x)$. The non-terminals F, F_X and F_t play the same part as A, X and A_t except that they express only terminated processes (without constants).

$$\left.\begin{array}{l} A \supseteq (0,0) \\ A \supseteq (x,x) \\ A \supseteq (x,X) \end{array}\right\} \forall x \in Const$$

$$A \supseteq (1_1 \parallel 2_1, 1_2 \parallel 2_2)(A,A)$$
$$A \supseteq (1_1 \,.\, 2_1, 1_2 \,.\, 2_2)(A,Id)$$
$$A \supseteq (1_1 \,.\, 2_1, 1_2 \,.\, 2_2)(F,A)$$

$$X \supseteq A_t \quad \forall t \,:\, (x \xrightarrow{a} t) \in \Delta$$

$$\left.\begin{array}{l} A_0 \supseteq 0 \\ A_x \supseteq x \\ A_x \supseteq X \end{array}\right\} \forall x \in Const$$

$$A_{t_1 \parallel t_2} \supseteq A_{t_1} \parallel A_{t_2} \quad \forall\, t_1 \parallel t_2 \in Sub(\Delta)$$

$$\left.\begin{array}{l} A_{t_1 \,.\, t_2} \supseteq A_{t_1} \,.\, t_2 \\ A_{t_1 \,.\, t_2} \supseteq F_{t_1} \,.\, A_{t_2} \end{array}\right\} \forall\, t_1 \,.\, t_2 \in Sub(\Delta)$$

$$F \supseteq (0,0)$$
$$F \supseteq (x, F_X) \quad \forall x \in Const$$
$$F \supseteq (1_1 \parallel 2_1, 1_2 \parallel 2_2)(F,F) \quad \forall\, t_1 \parallel t_2 \in Sub(\Delta)$$
$$F \supseteq (1_1 \,.\, 2_1, 1_2 \,.\, 2_2)(F,F) \quad \forall\, t_1 \,.\, t_2 \in Sub(\Delta)$$

$$F_X \supseteq F_t \quad \forall t \,:\, (x \xrightarrow{a} t) \in \Delta$$
$$F_0 \supseteq 0$$
$$F_x \supseteq F_X$$
$$F_{t_1 \parallel t_2} \supseteq F_{t_1} \parallel F_{t_2} \quad \forall\, t_1 \parallel t_2 \in Sub(\Delta)$$
$$F_{t_1 \,.\, t_2} \supseteq F_{t_1} \,.\, F_{t_2} \quad \forall\, t_1 \,.\, t_2 \in Sub(\Delta)$$

Fig. 2. The constraint system G for $\rightarrow^*$

[9] Shortened notations are used, like $A \supseteq (x, X)$ which means $A \supseteq (x, 1_1)X$.

Theorem 5.1. *G generates exactly $\rightarrow^*$ (i.e. $L(A) = \{(t, t') \mid t \rightarrow^* t'\}$). Moreover G is regular.*

As a consequence, we get the regularity of $\rightarrow^*$, already proved in [9]. The proof of this theorem can be achieved using the following lemma :

Lemma 5.2.

$$L(X) = \{t' \mid x \rightarrow^+ t'\} \qquad L(F) = \{(t, t') \mid t \rightarrow^* t' \wedge Finished(t')\}$$
$$L(A_t) = \{t' \mid t \rightarrow^* t'\} = Post^*(t) \qquad L(F_X) = \{t' \mid x \rightarrow^* t' \wedge Finished(t')\}$$
$$L(F_t) = \{t' \mid t \rightarrow^* t' \wedge Finished(t')\}$$

The proof can be made by classical induction or using the results of [10,9]. In [10], tree automata techniques are used to compute $Post^*(t)$ and $Pre^*(t)$. Two families of tree languages are defined, as the least solution of a recursive equation set. They define $L'_t = Post^*(t)$ and L''_t is the restriction of L'_t to terminated processes. It is clear that we have $L(A_t) = L'_t$ and $L(F_t) = L''_t$. The non-terminal A is represented by $[I, R]$ and F by $[I', RT]$. In [9], automata for tree languages are replaced by automata for tree relations. A notion of cost is introduced. A regular tree constraint system for $Post^*(X)$ is defined. F is represented by I and A by F.

5.2 BCPC

Now $Act = \{a, b, c, \ldots, \overline{a}, \overline{b}, \overline{c}, \ldots\}$, $\Delta = \Delta_1 \cup \Delta_2$ where Δ_2 is a set of synchronized rules :

$$\Delta_1 = \{x_i \xrightarrow{a_i^j} t_i^j \mid x_i \in Const,\ t_i^j \in T,\ i = 1, \ldots, n,\ j = 1, \ldots, k_i\}$$

$$\Delta_2 = \{ \begin{cases} y_i \xrightarrow{a_i^j} t_{i,1}^j \\ z_i \xrightarrow{\overline{a_i^j}} t_{i,2}^j \end{cases} \mid \begin{array}{l} y_i, z_i \in Const,\ t_{i,1}^j, t_{i,2}^j \in T, \\ i = 1, \ldots, n,\ j = 1, \ldots, p_i \end{array} \}$$

and

$$\Delta_1(x) = \{t \mid (x \xrightarrow{a} t) \in \Delta_1\} \qquad \Delta_2(y, z) = \{(t_1, t_2) \mid \begin{cases} y \xrightarrow{a} t_1 \\ z \xrightarrow{\overline{a}} t_2 \end{cases} \in \Delta_2\}$$

A synchronization is only allowed within a rule right-hand-side. In this paper, to avoid a conflict with the semantics of sequential composition, we assume that a synchronization is only allowed within a right-hand-side subterm that contains no sequential composition. We hope that in further work, by refining the semantics of synchronized process-derivation as well as the constraint system for $\rightarrow^*$, we will manage to remove this restriction.

Let Rhs be the set of the greatest[10] subterms of right-hand-sides of rules in Δ that contain no sequential composition. The semantics of $\rightarrow_\Delta$ is defined by Figure 1 (where a may also be replaced by ϵ) and :

$$s \xrightarrow{\epsilon} s[y/t_1, z/t_2] \quad \text{if} \begin{cases} y \xrightarrow{a} t_1 \\ z \xrightarrow{\overline{a}} t_2 \end{cases} \in \Delta_2 \text{ and } s \in Rhs$$

where $s[y/t_1, z/t_2]$ is obtained by replacing y by t_1 and z by t_2 in s.

[10] To avoid redundancies.

Example. For readability we first present an example. Consider

$$\Delta_1 = \{x \xrightarrow{a} y \| z,\ z \xrightarrow{a} y_2,\ y_1 \xrightarrow{a} 0\},\ \Delta_2 = \begin{cases} y \xrightarrow{a} (0\ .\ y_1) \\ z \xrightarrow{\overline{a}} 0 \end{cases}$$

Then $Rhs = \{y \| z\}$. The constraint system G' for $\rightarrow^*$ is presented in Figure 3. G' also includes the rules of Figure 2. The non-terminal YZ permits a synchronization of y and z.

Theorem 5.3. *G' generates exactly $\rightarrow^*$.*

Note that G' is not regular because constraints $A \supseteq \ldots$ and $F \supseteq \ldots$ introduce internal synchronizations. The proof comes from Lemma 5.2 and Lemma 5.4.

Lemma 5.4.

$L(YZ) = \{(t'_1, t'_2) \mid y \parallel z \rightarrow^+ t'_1 \parallel t'_2\}$
$L(H_{YZ}) = \{(t'_1, t'_2) \mid\ y \parallel z \rightarrow^* t'_1 \parallel t'_2\ \wedge\ Finished(t'_1)\ \wedge\ Finished(t'_2)\}$
$L(B_{yz}) = \{(t'_1, t'_2) \mid y \parallel z \rightarrow^* t'_1 \parallel t'_2\},\ L(B_{t_1 t_2}) = \{(t'_1, t'_2) \mid t_1 \rightarrow^* t'_1 \wedge t_2 \rightarrow^* t'_2\}$

The proof can be made by induction on the length of $\rightarrow^*$.

$$A \supseteq (y \parallel z, 1_1 \parallel 1_2)YZ$$

$$YZ \supseteq B_{t_1 t_2}\ \ \forall t_1, t_2 : \begin{cases} y \xrightarrow{a} t_1 \\ z \xrightarrow{\overline{a}} t_2 \end{cases} \in \Delta_2$$

$$B_{t_1 t_2} \supseteq (A_{t_1}, A_{t_2})$$

$$F \supseteq (y \parallel z, 1_1 \parallel 1_2)H_{YZ}$$

$$H_{YZ} \supseteq H_{t_1 t_2}\ \ \forall t_1, t_2\ :\ \begin{cases} y \xrightarrow{a} t_1 \\ z \xrightarrow{\overline{a}} t_2 \end{cases} \in \Delta_2$$

$$H_{t_1 t_2} \supseteq (F_{t_1}, F_{t_2})$$
$$A_{y \| z} \supseteq (1_1 \parallel 1_2)YZ$$

$$F_{y \| z} \supseteq (1_1 \parallel 1_2)H_{YZ}$$

Fig. 3. The additional constraints of G' for $\rightarrow^*$

The following process derivations:

$$x \rightarrow y \parallel z \rightarrow (0\ .\ y_1) \parallel 0 \rightarrow (0\ .\ 0) \parallel 0 \qquad x \rightarrow y \parallel z \rightarrow y \parallel y_2$$

are expressed by G' in the following way:

$A \supseteq (x, x)$ $\quad A \supseteq (x, X)$ $\quad X \supseteq A_{y \| z}$ because $(x \xrightarrow{a} y \parallel z) \in \Delta_1$
$A_{y \| z} \supseteq A_y \parallel A_z$ $\quad A_y \supseteq y$ $\quad A_z \supseteq z$ $\quad A_z \supseteq Z$
$Z \supseteq A_{y_2}$ because $(z \xrightarrow{a} y_2) \in \Delta_1$ $\quad A_{y_2} \supseteq y_2$

So, we can prove : $A \supseteq (x, x),\ (x, y \parallel z),\ (x, y \parallel y_2)$

$$A_{y\parallel z} \supseteq (1_1 \parallel 1_2)YZ \qquad YZ \supseteq B_{0.y_1\ 0} \text{ because } \begin{cases} y \xrightarrow{a} 0 \cdot y_1 \\ z \xrightarrow{\bar{a}} 0 \end{cases} \in \Delta_2$$

$$B_{0.y_1\ 0} \supseteq (A_{0.y_1}, A_0) \qquad A_{0.y_1} \supseteq A_0 \cdot y_1 \quad A_{0.y_1} \supseteq F_0 \cdot A_{y_1}$$

$$A_0 \supseteq 0 \quad F_0 \supseteq 0 \qquad A_{y_1} \supseteq y_1 \quad A_{y_1} \supseteq Y_1$$

$Y_1 \supseteq 0$ because $(y_1 \xrightarrow{a} 0) \in \Delta_1$

So, we can prove : $A \supseteq (x, 0 \cdot y_1 \parallel 0),\ (x, 0 \cdot 0 \parallel 0)$

General case. See the full version [3].

Model-checking. The model-checking problem solved in [10] for PA is still decidable for BCPC.

We consider a set $Prop = \{P_1, P_2, \ldots\}$ of *atomic propositions*. For $P \in Prop$, let $Mod(P)$ denote the set of PA-processes for which P holds. $Mod(P)$ is always supposed to be a regular tree-language.

The EF-logic has the following syntax :

$$\varphi ::= P \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \mathsf{EX}_\varphi \mid \mathsf{EF}_\varphi$$

and semantics :

$$t \models P \Leftrightarrow t \in Mod(P) \qquad t \models \mathsf{EX}_\varphi \Leftrightarrow t' \models \varphi \text{ for some } t \to t'$$

$$t \models \neg\varphi \Leftrightarrow t \not\models \varphi \qquad t \models \mathsf{EF}_\varphi \Leftrightarrow t' \models \varphi \text{ for some } t \to^* t'$$

$$t \models \varphi \wedge \varphi' \Leftrightarrow t \models \varphi \text{ and } t \models \varphi'$$

Definition 5.5. *The* model-checking problem *consists in testing whether* $t \models \varphi$ *for given* t *and* φ.

If we define $Mod(\varphi) = \{s \in T \mid s \models \varphi\}$, the model-checking problem for s and φ amounts to test whether $t \in Mod(\varphi)$. Trivially, Mod satisfies :

$$Mod(\neg\varphi) = T - Mod(\varphi) \qquad Mod(\mathsf{EX}_\varphi) = Pre(Mod(\varphi))$$

$$Mod(\varphi \wedge \varphi') = Mod(\varphi) \cap Mod(\varphi') \qquad Mod(\mathsf{EF}_\varphi) = Pre^*(Mod(\varphi))$$

Theorem 5.6. *For every EF-formula* φ, $Mod(\varphi)$ *is a regular language.*

Proof. See the full version [3].

This result may seem surprising, because it uses $\to^*$ as an intermediate language, which is a non-regular synchronized language. It comes from the fact that the first component of G' is free, i.e. the projection of $\to^*$ on the first component is regular.

So, and thanks to membership test, the model-checking problem is decidable. As a consequence, the absence of deadlocks is decidable. Indeed, starting from a process t, there is a deadlock iff there exists an iterated successor t' of t s.t. t' has no successors and t' is not a finished process. Thus, there is no deadlocks iff $t \models \neg\mathsf{EF}(\neg\mathsf{EX}(True) \wedge \neg Finished)$, where $Mod(True) = T$ and $Mod(Finished) = \{t \in T \mid Finished(t)\}$ are regular languages.

6 Further Work

Synchronized languages defined by constraint systems are not closed by intersection, and therefore not closed by complement[11]. In further work, we will define a subclass closed by intersection, and that preserves all properties of synchronized languages. This should give rise to further applications.

BCPC allows rendez-vous of only two processes. This can be trivially extended to finitely many processes, since we can handle tuples of any size. Removing the restriction on Rhs is more difficult, but we hope for it. On the other hand, we can test properties on processes, not on actions. However, it should be possible to take actions into account by using triple languages for $\stackrel{a}{\rightarrow}$ and $\stackrel{a}{\rightarrow}^*$.

References

1. J.C.M. Baeten and W.P. Weijland. Process algebra. In *Cambridge Tracts in Theoretical Computer Science*, volume 18, 1990.
2. B. Courcelle. The Monadic Second-Order Logic of Graphs I, Recognizable Sets of Finite Graphs. In *Inf. Comp.*, volume 85, pages 12–75, 1990.
3. V. Gouranton, P. Réty, and H. Seidl. Synchronized Tree Languages Revisited and New Applications. Research Report 2000-16, LIFO, 2000. http://www.univ-orleans.fr/SCIENCES/LIFO/Members/rety/publications.html.
4. S. Limet and P. Réty. E-Unification by Means of Tree Tuple Synchronized Grammars. In *Proceedings of 6th Colloquium on Trees in Algebra and Programming*, volume 1214 of *LNCS*, pages 429–440. Springer-Verlag, 1997.
5. S. Limet and P. Réty. E-Unification by Means of Tree Tuple Synchronized Grammars. *Discrete Mathematics and Theoritical Computer Science (http://dmtcs.loria.fr/)*, 1:69–98, 1997.
6. S. Limet and P. Réty. Solving Disequations modulo some Class of Rewrite Systems. In *Proceedings of 9th Conference on Rewriting Techniques and Applications, Tsukuba (Japon)*, volume 1379 of *LNCS*, pages 121–135. Springer-Verlag, 1998.
7. S. Limet and P. Réty. A New Result about the Decidability of the Existential One-step Rewriting Theory. In *Proceedings of 10th Conference on Rewriting Techniques and Applications, Trento (Italy)*, volume 1631 of *LNCS*. Springer-Verlag, 1999.
8. S. Limet and F. Saubion. On partial validation of logic programs. In M. Johnson, editor, *proc of the 6th Conf. on Algebraic Methodology and Software Technology, Sydney (Australia)*, volume 1349 of *LNCS*, pages 365–379. Springer Verlag, 1997.
9. D. Lugiez and P. Schnoebelen. Decidable firt-order transition logics for PA-processes. In Springer, editor, *ICALP*, LNCS, Geneva, Switzerland, July 2000.
10. D. Lugiez and P. Schnoebelen. The regular viewpoint on PA-processes. *Theoretical Computer Science*, 2000.
11. G.D. Plotkin. A structural approach of operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.
12. F. Saubion and I. Stéphan. On Implementation of Tree Synchronized Languages. In *Proceedings of 10th Conference on Rewriting Techniques and Applications, Trento (Italy)*, LNCS. Springer-Verlag, 1999.

[11] Closure by union and complement implies closure by intersection.

13. H. Seidl. Haskell Overloading is DEXPTIME Complete. *Information Processing Letters*, 52(2):57–60, 1994.
14. F. Seynhaeve, S. Tison, and M. Tommasi. Homomorphisms and concurrent term rewriting. In G. Ciobanu and G. Paun, editors, *Proceedings of the twelfth International Conference on Fundamentals of Computation theory*, number 1684 in Lecture Notes in Computer Science, pages 475–487, Iasi, Romania, 1999.

Computational Completeness of Programming Languages Based on Graph Transformation

Annegret Habel[1] and Detlef Plump[2]

[1] Fachbereich Informatik, Universität Oldenburg
Postfach 2503, D-26111 Oldenburg, Germany
habel@informatik.uni-oldenburg.de

[2] Department of Computer Science, The University of York
Heslington, York YO10 5DD, United Kingdom
det@cs.york.ac.uk

Abstract. We identify a set of programming constructs ensuring that a programming language based on graph transformation is computationally complete. These constructs are (1) nondeterministic application of a set of graph transformation rules, (2) sequential composition and (3) iteration. This language is minimal in that omitting either sequential composition or iteration results in a computationally incomplete language. By computational completeness we refer to the ability to compute every computable partial function on labelled graphs. Our completeness proof is based on graph transformation programs which encode arbitrary graphs as strings, simulate Turing machines on these strings, and decode the resulting strings back into graphs.

1 Introduction

The use of graphs to represent and visualise complex structures is ubiquitous in computer science, and often these structures occur in contexts where they have to be dynamically changed. Functional and logic programming languages, on the other hand, are successful examples of high-level programming languages based on rules. Thus a natural idea is to design programming languages based on graph transformation rules, in order to combine the strengths of graphs and rule-based programming.

Existing programming languages of this type include PROGRES [SWZ99], AGG [ERT99], GAMMA [FM98], GRRR [Rod98] and DACTL [GKS91]. These languages have in common that they are based on graph transformation rules, but they vary strongly with respect to both the formalisms underlying the rules and the available constructs for controlling rule applications. In view of the variety of control mechanisms, the question arises what programming constructs are really needed on top of graph transformation rules to obtain a computationally complete language. By computational completeness we mean the ability to compute every computable partial function on labelled graphs. Identifying such a kernel language for graph transformation will benefit to both the understanding of existing languages and the design of new programming languages of this kind.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 230–245, 2001.

In this paper we show that three programming constructs suffice to guarantee computational completeness: (1) nondeterministic application of a rule from a set of graph transformation rules (according to the so-called double-pushout approach), (2) sequential composition and (3) iteration in the form that rules are applied as long as possible. This language is not only complete but also minimal in that omitting either sequential composition or iteration makes the language computationally incomplete.

One may wonder why plain sets of graph transformation rules with the semantics "apply as long as possible" are not computationally complete. Indeed it is not difficult to simulate Turing machines by sets of graph transformation rules (in the double-pushout approach), but this only means that all computations on *representations* of graphs can be modelled. The ability to transform a string representation of a graph G into a string representation of the graph $f(G)$, where f is some graph function, does not imply that there is a set of rules transforming G directly into $f(G)$.

So what is different to the case of string rewriting, where sets of rules do suffice to compute all computable functions on strings? (See, for example, Lewis' and Papadimitriou's concept of a grammatically computable function [LP98].) The point is that in a string-based model, prior to computations input strings are provided with some context of auxiliary symbols which must not occur in inputs but which can be used in the rules. This context allows to control the application of rules, ensuring that computations have universal power. It is open whether there is a similar concept that makes sets of graph transformation rules universally powerful. The problem is that in contrast to strings, arbitrary graphs do not possess distinguished points for attaching context.

For the programming language introduced below we do not assume that input graphs come in any particular format, the idea is rather to provide just enough control constructs to ensure computational completeness. Our completeness proof is based on the sequential composition of three programs: the first encodes arbitrary graphs as certain strings, the second simulates Turing machines on these strings, and the third decodes the resulting strings back into graphs. The strings for representing graphs are similar to those of Uesu [Ues78] who showed that graph grammars according to the double-pushout approach can generate all recursively enumerable sets of labelled graphs.

Finally we show that our programming language is minimal, by proving that the function converse which swaps sources and targets of all edges in a graph cannot be computed if either sequential composition or iteration is missing.

2 Rules

This section recalls the application of graph transformation rules according to the "double-pushout" approach. Details and pointers to the literature can be found in [HMP99].

A *label alphabet* $\mathcal{C} = \langle \mathcal{C}_V, \mathcal{C}_E \rangle$ is a pair of finite sets of *node labels* and *edge labels*. A *graph* over $\mathcal{C}$ is a system $G = (\mathrm{V}_G, \mathrm{E}_G, \mathrm{s}_G, \mathrm{t}_G, \mathrm{l}_G, \mathrm{m}_G)$ consisting of

two finite sets V_G and E_G of *nodes* (or *vertices*) and *edges*, two *source* and *target functions* $\mathrm{s}_G, \mathrm{t}_G \colon \mathrm{E}_G \to \mathrm{V}_G$, and two *labelling functions* $\mathrm{l}_G \colon \mathrm{V}_G \to \mathcal{C}_V$ and $\mathrm{m}_G \colon \mathrm{E}_G \to \mathcal{C}_E$.

A *graph morphism* $g \colon G \to H$ between two graphs G and H consists of two functions $g_V \colon \mathrm{V}_G \to \mathrm{V}_H$ and $g_E \colon \mathrm{E}_G \to \mathrm{E}_H$ that preserve sources, targets, and labels, that is, $\mathrm{s}_H \circ g_E = g_V \circ \mathrm{s}_G$, $\mathrm{t}_H \circ g_E = g_V \circ \mathrm{t}_G$, $\mathrm{l}_H \circ g_V = \mathrm{l}_G$, and $\mathrm{m}_H \circ g_E = \mathrm{m}_G$. A morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective), and an *isomorphism* if it is both injective and surjective. In the latter case G and H are *isomorphic*, which is denoted by $G \cong H$. A morphism g is an *inclusion* if $g_V(v) = v$ and $g_E(e) = e$ for all $v \in V_G$ and $e \in E_G$.

A *rule* $r = \langle L \leftarrow K \rightarrow R \rangle$ consists of two graph morphisms with a common domain K, which is the *interface* of r. We throughout assume that $K \to L$ is an inclusion. The *application* of r to a graph G amounts to the following steps:

(1) Find an injective graph morphism $g \colon L \to G$ satisfying the *dangling condition*: No edge in $G - g(L)$ is incident to a node in $g(L) - g(K)$.
(2) Remove $g(L) - g(K)$ from G, yielding a graph D, a graph morphism $K \to D$ which is the restriction of g, and the inclusion $D \to G$.
(3) Construct the pushout of $K \to D$ and $K \to R$, yielding a graph H and graph morphisms $D \to H$ and $R \to H$. (See [Ehr79] or the appendix of [HMP99] for the construction of graph pushouts.)

This construction yields the pushout diagrams (1) and (2) in Figure 1. Roughly, H is obtained from the intermediate graph D by merging items according to the morphism $K \to R$ (in case this morphism is not injective) and adding the items of R that are not in the image of K.

The transformation of G into H is denoted by $G \Rightarrow_{r,g} H$. We write $G \Rightarrow_r H$ to express that there is a graph morphism g such that $G \Rightarrow_{r,g} H$. Given a set $\mathcal{R}$ of rules, $G \Rightarrow_{\mathcal{R}} H$ means that there is a rule r in $\mathcal{R}$ such that $G \Rightarrow_r H$. So the relation $\Rightarrow_{\mathcal{R}}$ is nondeterministic with respect to both the rule chosen from $\mathcal{R}$ and the position in the given graph where this rule is applied.

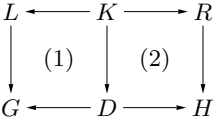


Fig. 1. A transformation step in form of a double-pushout

We will use graph transformation to compute relations on abstract graphs rather than on concrete graphs as above, so we identify isomorphic graphs and lift transformation steps to isomorphism classes of graphs. An *abstract graph* over a label alphabet $\mathcal{C}$ is an isomorphism class of graphs over $\mathcal{C}$. We write $[G]$ for the isomorphism class of a graph G and denote by $\mathcal{A}_{\mathcal{C}}$ the set of all abstract graphs

over $\mathcal{C}$. The relation $\Rightarrow_{\mathcal{R}}$ is lifted to $\mathcal{A}_{\mathcal{C}}$ by: $[G] \Rightarrow_{\mathcal{R}} [H]$ if $G \Rightarrow_{\mathcal{R}} H$. This yields a well-defined relation since, by the definition of transformation steps as double-pushouts, we have for all graphs G, G', H and H' over $\mathcal{C}$: $G' \cong G \Rightarrow_{\mathcal{R}} H \cong H'$ implies $G' \Rightarrow_{\mathcal{R}} H'$.

3 Programs

The programs we are going to define are based on sets of graph transformation rules. In this paper we do not address the issue how to represent rules syntactically, we rather assume that sets of rules, single rules and graphs have names to which programs can refer.

Definition 1 (Program). *Programs* over a label alphabet $\mathcal{C}$ are inductively defined as follows:

(1) Every finite set $\mathcal{R}$ of rules over $\mathcal{C}$ is a program.
(2) If P_1 and P_2 are programs, then $\langle P_1; P_2 \rangle$ is a program.
(3) If P is a program according to (1) or (2), then $P\!\downarrow$ is a program.

Programs according to (1) are *elementary*, the program $\langle P_1; P_2 \rangle$ is the *sequential composition* of P_1 and P_2, and $P\!\downarrow$ is the *iteration* of P. Programs of the form $\langle P_1; \langle P_2; P_3 \rangle\rangle$ and $\langle \langle P_1; P_2 \rangle; P_3 \rangle$ are considered as equal; by convention, both can be written as $\langle P_1; P_2; P_3 \rangle$.

Next we provide programs with a relational input/output semantics. Given a binary relation $\rightarrow$ on a set S, we denote by $\rightarrow^+$ the transitive closure of $\rightarrow$ and by $\rightarrow^*$ the reflexive-transitive closure. An element a in S is a *normal form* with respect to $\rightarrow$ if there is no b in S such that $a \rightarrow b$.

Definition 2 (Semantics). Given a program P over a label alphabet $\mathcal{C}$, the *semantics* of P is a binary relation $\rightarrow_P$ on $\mathcal{A}_{\mathcal{C}}$ which is inductively defined as follows:

(1) $\rightarrow_P = \Rightarrow_{\mathcal{R}}$ if P is an elementary program $\mathcal{R}$.
(2) $\rightarrow_{\langle P_1;P_2 \rangle} = \rightarrow_{P_1} \circ \rightarrow_{P_2}$.
(3) $\rightarrow_{P\downarrow} = \{\langle G, H \rangle \mid G \rightarrow_P^* H$ and H is a normal form with respect to $\rightarrow_P\}$.

Consider now subalphabets $\mathcal{C}_1$ and $\mathcal{C}_2$ of $\mathcal{C}$ and a relation $Rel \subseteq \mathcal{A}_{\mathcal{C}_1} \times \mathcal{A}_{\mathcal{C}_2}$. We say that P *computes* Rel if $Rel = \rightarrow_P \cap (\mathcal{A}_{C_1} \times \mathcal{A}_{\mathcal{C}_2})$, that is, if Rel coincides with the semantics of P restricted to $\mathcal{A}_{\mathcal{C}_1}$ and $\mathcal{A}_{C_2}$. The same applies to partial functions $f\colon \mathcal{A}_{\mathcal{C}_1} \rightarrow \mathcal{A}_{C_2}$, which are just special relations.

Example 1 (Functions computed by programs).

1. Given a graph K in $\mathcal{A}_{\mathcal{C}}$, the constant function $\mathrm{const}_K\colon \mathcal{A}_{\mathcal{C}} \rightarrow \mathcal{A}_{\mathcal{C}}$ with $\mathrm{const}_K(G) = K$ for all $G \in \mathcal{A}_{\mathcal{C}}$ is computed by the program

$$\texttt{Const}_K = \langle \texttt{Delete}\!\downarrow;\ \texttt{Add}_K \rangle,$$

where `Delete` is an elementary program deleting nodes and edges (with arbitrary labels in $\mathcal{C}$), and $\texttt{Add}_K$ is the elementary program consisting of the single rule $\langle \emptyset \leftarrow \emptyset \rightarrow K \rangle$.

2. The function converse: $\mathcal{A}_\mathcal{C} \to \mathcal{A}_\mathcal{C}$ swaps source and target of each edge in a graph. It is computed by the program

$$\texttt{Converse} = \langle \texttt{Redirect}{\downarrow};\ \texttt{Relabel}{\downarrow} \rangle$$

over the label alphabet $\langle \mathcal{C}_V, \mathcal{C}_E \cup (\mathcal{C}_E \times \{'\}) \rangle$. The rules of `Converse` are shown in Figure 2.[1] Note that the redirected edges temporarily get auxiliary labels to prevent further redirection. After termination of `Redirect` $\downarrow$ all edges get their original labels by the subprogram `Relabel`$\downarrow$.

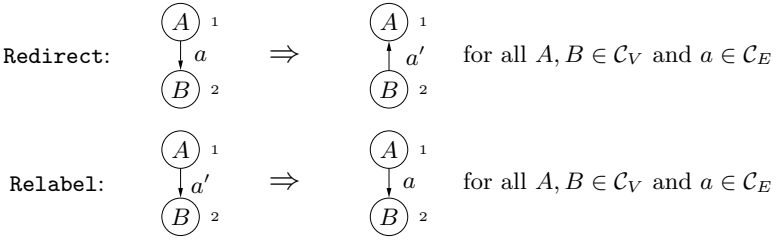


Fig. 2. The rules of the program `Converse`

In the proof of our completeness result in Section 6, we will use a program scheme $\texttt{Ite}(K, P_1, P_2)$ which checks whether the input graph equals K and executes P_1 or P_2 depending on whether the check is successful or not. More precisely, the semantics is given by $G \to_{\texttt{Ite}(K,P_1,P_2)} H$ if and only if $G = K$ and $G \to_{P_1} H$ or $G \neq K$ and $G \to_{P_2} H$. The scheme is defined by

$$\texttt{Ite}(K, P_1, P_2) = \langle \texttt{Check}(K);\ \langle \texttt{Delete}_1;\ P_1 \rangle{\downarrow};\ \langle \texttt{Delete}_2;\ P_2 \rangle{\downarrow} \rangle,$$

where $\texttt{Check}(K)$ copies the input graph G and reduces the copy to a node with label 1 if $G = K$, and to a node with label 2 otherwise. For $i = 1, 2$, $\texttt{Delete}_i$ deletes a node with label i. If $\texttt{Check}(K)$ yields 1, then $\langle \texttt{Delete}_1;\ P_1 \rangle$ can be executed only once because the node with label 1 is deleted and $\langle \texttt{Delete}_2;\ P_2 \rangle$ is executed zero times because there is no node with label 2. Vice versa, if $\texttt{Check}(K)$ yields a node with label 2, then $\langle \texttt{Delete}_1;\ P_1 \rangle$ is executed zero times and $\langle \texttt{Delete}_2;\ P_2 \rangle$ is executed once. We omit the rules of this program scheme for space reasons.

4 Computable Graph Functions

In this section we introduce the notion of a computable partial function on abstract graphs, by using Turing computability on strings and an encoding of abstract graphs as strings. This is consistent with Weihrauch's concept of *(strong) relative computability* [Wei87].

[1] In order to present rules concisely, we show only the left- and right-hand sides. The interfaces consist of the numbered nodes of the left-hand sides and have no edges.

We start by defining graph expressions as certain well-formed strings and a surjective partial function gra from strings to abstract graphs which assigns to every graph expression an abstract graph. To this end, let $\mathcal{C}$ be a label alphabet and set $\Sigma = \mathcal{C}_{\mathrm{V}} \cup \mathcal{C}_{\mathrm{E}} \cup \{1, 2, \#\}$. We assume $\mathcal{C}_V \cap \mathcal{C}_E = \emptyset$ and that 1, 2 and # do not occur in $\mathcal{C}_{\mathrm{V}}$ and $\mathcal{C}_{\mathrm{E}}$.

Definition 3 (Graph expression). The set Exp of *graph expressions* over Σ and the graph $w^{\square}$ represented by a graph expression w are inductively defined as follows:

(1) The empty string λ is in Exp and $\lambda^{\square} = \emptyset$.
(2) For all $A \in \mathcal{C}_{\mathrm{V}}$, $\#A1\# \in$ Exp and $\#A1\#^{\square}$ is the graph consisting of a single node 1 with label A.
(3) If $v\#w \in$ Exp and $A \in \mathcal{C}_{\mathrm{V}}$, then $v\#A1^n\#w \in$ Exp with $n = |\mathrm{V}_{v\#w^{\square}}| + 1$ and $v\#A1^n\#w^{\square}$ is obtained from $v\#w^{\square}$ by adding a node n with label A.
(4) If $v\#w \in$ Exp, $F \in \mathcal{C}_{\mathrm{E}}$ and $v\#w$ contains substrings $A1^m\#$ and $B1^n\#$, then $v\#A2^mF2^nB\#w \in$ Exp and $v\#A2^mF2^nB\#w^{\square}$ is obtained from $v\#w^{\square}$ by adding an edge $|\mathrm{E}_{v\#w^{\square}}| + 1$ which has label F, source node m and target node n.

A substring $\#A1^n\#$ in a graph expression represents a node with name n and label A, while a substring $\#A2^mF2^nB\#$ stands for an edge with label F, source node m and target node n. Note that the order of nodes and edges in a graph expression is arbitrary and that a graph $w^{\square}$ has the node set $\{1, 2, \ldots, |\mathrm{V}_{w^{\square}}|\}$.

Definition 4 (Representation of abstract graphs). The partial function gra: $\Sigma^* \to \mathcal{A}_{\mathcal{C}}$ is defined as follows:

$$\mathrm{gra}(w) = \begin{cases} [w^{\square}] & \text{if } w \text{ is a graph expression,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The function gra is surjective since every isomorphism class of graphs contains a graph represented by a graph expression.

Let now $\mathcal{C}_1$ and $\mathcal{C}_2$ be any subalphabets of $\mathcal{C}$ and define for $i = 1, 2$, $\Sigma_i = \mathcal{C}_{iV} \cup \mathcal{C}_{iE} \cup \{1, 2, \#\}$.

Definition 5 (Computable graph function). A partial function $f: \mathcal{A}_{\mathcal{C}_1} \to \mathcal{A}_{\mathcal{C}_2}$ on abstract graphs is *computable* if there exists a computable partial function $f': \Sigma_1^* \to \Sigma_2^*$ on strings such that for every graph expression w,

$$f(\mathrm{gra}(w)) = \mathrm{gra}(f'(w))$$

and $\mathrm{gra}(w) \not\in \mathrm{Dom}(f)$ implies $w \not\in \mathrm{Dom}(f')$.[2]

Thus f is computable if there is a computable function f' on strings such that for every abstract graph G for which f is defined and every graph expression w denoting G, f' is defined for w and yields a graph expression denoting $f(G)$. This situation is illustrated in Figure 3. Moreover, f' is not defined on graph expressions denoting graphs on which f is not defined.

[2] We denote by $\mathrm{Dom}(f)$ the domain of a partial function f.

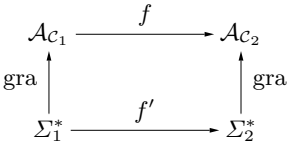


Fig. 3. Computability of a graph function f

5 Simulation of Turing Machines

In the next section we will show that every computable graph function can be computed by a program. An essential argument will be that every computable partial function on strings can be computed by a graph transformation program working on so-called *string graphs*. The string graph of a string $w = a_1 a_2 \dots a_n$ is the abstract graph shown in Figure 4 and is denoted by $w^\bullet$. (It is understood that $w^\bullet$ consists of a single node if w is the empty string.)

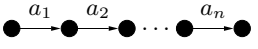


Fig. 4. The string graph for $a_1 a_2 \dots a_n$

Theorem 1. *For every computable partial function $f\colon \Sigma_1^* \to \Sigma_2^*$ there is a program P_f over a label alphabet $\mathcal{C}$ with $\mathcal{C}_1, \mathcal{C}_2 \subseteq \mathcal{C}$ such that for all w in Σ_1^* and G in $\mathcal{AC}$,*

$$w^\bullet \to_{P_f} G \text{ if and only if } G = f(w)^\bullet.$$

To prove Theorem 1 we will simulate Turing machines by programs, so we first recall the definition of Turing machines and their computed functions (using a version similar to that in [LP98]).

A *Turing machine* is a system $M = \langle Q, \Gamma, \delta, q_0, \Box \rangle$ where Q is a finite set of states, Γ is a finite set of tape symbols, $\Box \in \Gamma$ is the blank symbol, δ is a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{r, n, l\}$ called the transition function and $q_0 \in Q$ is the start state.

A *configuration* of M is a string $c = uqav$ such that $q \in Q$, $a \in \Gamma$ and $u, v \in \Gamma^*$. The start configuration of M with respect to a string w in Σ^* is given by $\alpha(w) = q_0 w$ if w is not empty and by $q_0 \Box$ otherwise. Given a configuration $c = uqav$, we write $c \vdash_M c'$ and call c' the successor configuration of c if c' is given by

$$c' = \begin{cases} ua'q'v & \text{if } \delta(q,a) = (q',a',r) \text{ and } v \neq \lambda, \\ ua'q'\Box & \text{if } \delta(q,a) = (q',a',r) \text{ and } v = \lambda, \\ uq'a'v & \text{if } \delta(q,a) = (q',a',n), \\ u'q'ba'v & \text{if } \delta(q,a) = (q',a',l) \text{ and } u = u'b, \\ q'\Box a'v & \text{if } \delta(q,a) = (q',a',l) \text{ and } u = \lambda. \end{cases}$$

A configuration is final if it has no successor configuration. The *result* of a configuration $c = uqv$ is the string $\omega(uqv) = \bar{u}\bar{v}$ where $\bar{u}$ is the shortest string with $u = \Box \ldots \Box\bar{u}$ and $\bar{v}$ is the shortest string with $v = \bar{v}\Box \ldots \Box$.

Let Σ_1 and Σ_2 be subsets of $\Gamma - \{\Box\}$. The partial function $f_M \colon \Sigma_1^* \to \Sigma_2^*$ *computed* by M is given by $f_M(v) = w$ if there is a final configuration c such that $\alpha(v) \vdash_M^* c$, $\omega(c) = w$ and $w \in \Sigma_2^*$, and undefined otherwise.

Proof of Theorem 1. For every computable partial function $f \colon \Sigma_1^* \to \Sigma_2^*$ there exists a Turing machine M that computes f. So we have to show that for every Turing machine M there exists a program $\texttt{Turing}(M)$ that transforms string graphs into string graphs such that for all v in Σ_1^* and w in Σ_2^*, $f_M(v) = w$ if and only if $v^\bullet \to_{\texttt{Turing}(M)} w^\bullet$. To this end, let

$$\texttt{Turing}(M) = \langle \texttt{Initiate}; \texttt{Simulate}{\downarrow}; \texttt{Eliminate}{\downarrow}; \texttt{Finish} \rangle$$

where `Initiate` is an elementary program attaching a state node labelled with the start state to an input string, `Simulate` $\downarrow$ simulates the working of M, `Eliminate`$\downarrow$ removes all blanks from the final configuration, and `Finish` deletes the state node. The rules of $\texttt{Turing}(M)$ are given in Figure 5. □

6 Computational Completeness

We are now ready to state our main result, namely that every computable partial function on abstract graphs is computed by a program in the programming language defined in Section 3.

Theorem 2. *For every computable partial function $f \colon \mathcal{A}_{\mathcal{C}_1} \to \mathcal{A}_{\mathcal{C}_2}$ there exists a program that computes f.*

Proof. Let $f \colon \mathcal{A}_{\mathcal{C}_1} \to \mathcal{A}_{\mathcal{C}_2}$ be computable. Then, by Definition 5, there is a computable partial function $f' \colon \Sigma_1^* \to \Sigma_2^*$ such that for every graph expression w, $f(\text{gra}(w)) = \text{gra}(f'(w))$. Let $P_{f'}$ be the program of Theorem 1 which simulates f', and let `Encode` and `Decode` be the programs of Lemmata 1 and 2 below. Without loss of generality, we may assume that `Encode`, $P_{f'}$, and `Decode` are programs over a common label alphabet $\mathcal{C}$. We show that for all G in $\mathcal{A}_{\mathcal{C}_1}$ and H in $\mathcal{A}_{\mathcal{C}_2}$,

$$G \to_{\langle \texttt{Encode};\, P_{f'};\, \texttt{Decode} \rangle} H \text{ if and only if } f(G) = H.$$

"Only if": Let $G \to_{\langle \texttt{Encode};\, P_{f'};\, \texttt{Decode} \rangle} H$. Then there are G_1 and G_2 in $\mathcal{A}_{\mathcal{C}}$ such that $G \to_{\texttt{Encode}} G_1 \to_{P_{f'}} G_2 \to_{\texttt{Decode}} H$. By Lemma 1 we have $G_1 = w^\bullet$ for

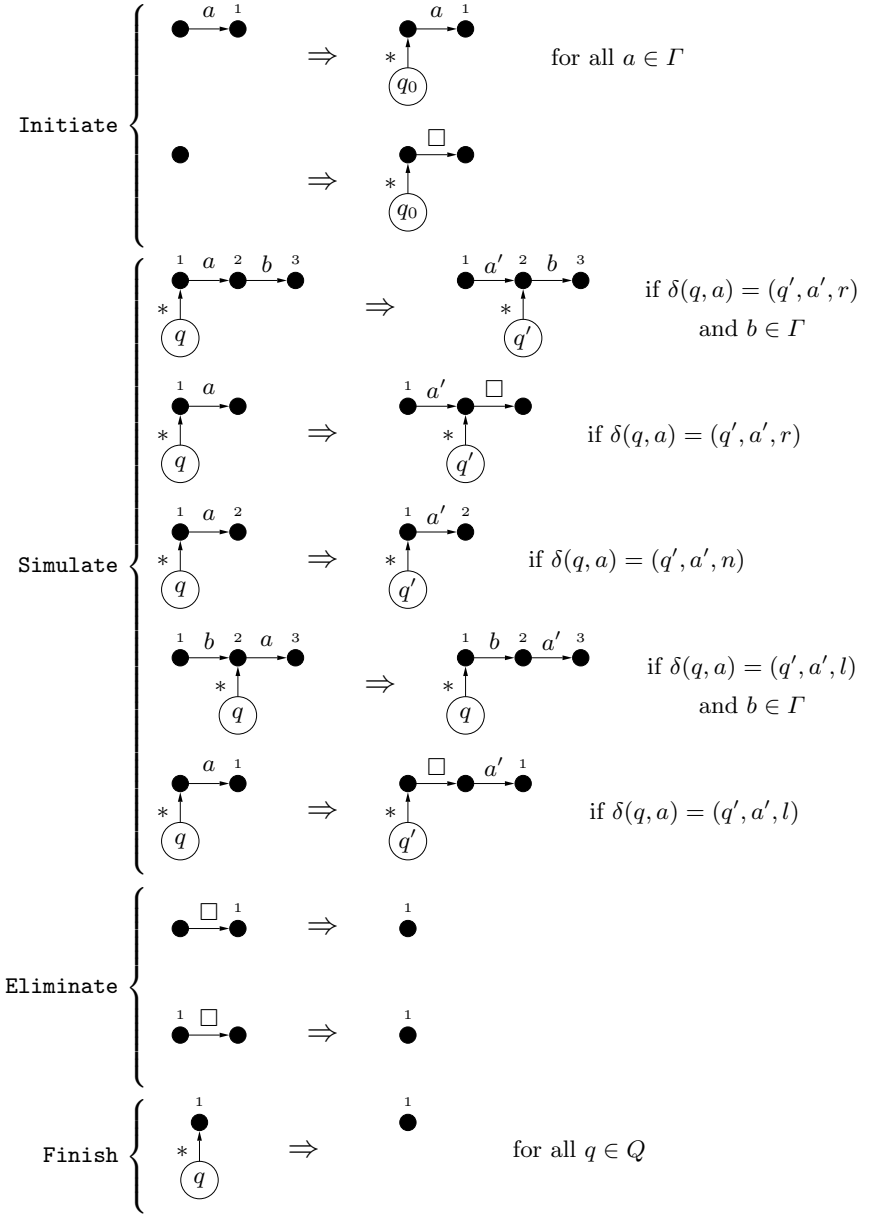


Fig. 5. The rules of the program `Turing`(M)

some graph expression w with $\mathrm{gra}(w) = G$. Then $G_2 = f'(w)^\bullet$ by Theorem 1. Hence $w \in \mathrm{Dom}(f')$ and, by Definition 5, $\mathrm{gra}(w) \in \mathrm{Dom}(f)$ and $f(\mathrm{gra}(w)) = \mathrm{gra}(f'(w))$. So $f'(w)$ is a graph expression and hence, by Lemma 2, $\mathrm{gra}(f'(w)) = H$. Thus $f(G) = f(\mathrm{gra}(w)) = \mathrm{gra}(f'(w)) = H$.

"If": Let $f(G) = H$, and let w be a graph expression with $\mathrm{gra}(w) = G$. Then, by Lemma 1, $G \to_{\texttt{Encode}} w^\bullet$. By the computability of f, $f(\mathrm{gra}(w)) = \mathrm{gra}(f'(w))$. Thus $w \in \mathrm{Dom}(f')$ and $f'(w)$ is a graph expression. By Theorem 1, $w^\bullet \to_{P_{f'}} f'(w)^\bullet$. By Lemma 2, $\mathrm{gra}(f'(w)) = f(\mathrm{gra}(w)) = f(G) = H$ implies $f'(w)^\bullet \to_{\texttt{Decode}} H$. Thus $G \to_{\texttt{Encode}} w^\bullet \to_{P_{f'}} f'(w)^\bullet \to_{\texttt{Decode}} H$. □

The rest of this section is devoted to Lemmata 1 and 2 which give programs for encoding abstract graphs as graph expressions, and decoding graph expressions back into abstract graphs. Let $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}$ be any label alphabets such that $\mathcal{C}_{1V} \cup \mathcal{C}_{2V} \cup \mathcal{C}_{1V} \times \{'\} \cup \{\bullet, *\} \subseteq \mathcal{C}_V$ and $\mathcal{C}_{1E} \cup \mathcal{C}_{2E} \cup \mathcal{C}_{1V} \cup \{1, 2, 2', \#, =, ?, >, \bowtie\} \subseteq \mathcal{C}_E$, where we assume that the symbols $\bullet$, $*$, $2'$, $=$, $?$, $>$ and $\bowtie$ do not occur in $\mathcal{C}_{iV}$ and $\mathcal{C}_{iE}$ $(i = 1, 2)$.

Lemma 1. *There is a program* Encode *such that for all G in $\mathcal{A}_{\mathcal{C}_1}$ and H in $\mathcal{A}_{\mathcal{C}}$,*

$$G \to_{\texttt{Encode}} H \text{ if and only if } G = \mathrm{gra}(w) \text{ and } H = w^\bullet \text{ for some graph}$$

expression w.

Proof. The program Encode is given by

$$\texttt{Encode} = \texttt{Ite}(\emptyset, \texttt{Encode}_1, \texttt{Encode}_2)$$

where $\emptyset$ is the empty graph and $\texttt{Encode}_1$ and $\texttt{Encode}_2$ encode the empty graph and non-empty abstract graphs, respectively. While $\texttt{Encode}_1$ just creates a single node with label $\bullet$, $\texttt{Encode}_2$ consists of three subprograms:

$$\texttt{Encode}_2 = \langle \texttt{Prepare};\ \texttt{Bundle};\ \texttt{Compose} \rangle.$$

Prepare prepares an abstract graph for encoding by representing node labels as edge labels and decorating each node by a chain of edges labelled with 1. The program Bundle transforms an abstract graph into a bundle of string graphs each of which represents a node or an edge of the original graph. Compose composes the string graphs in the bundle by connecting them with #-labelled edges and attaching a #-edge at the begin and the end of the resulting graph.

The program Prepare consists of four subprograms:[3]

$$\texttt{Prepare} = \langle \texttt{Choose};\ \texttt{Inc}{\downarrow};\ \texttt{Relabel}{\downarrow};\ \texttt{Stop} \rangle {\downarrow}\,.$$

Here Choose selects a labelled node, relabels it into $\bullet$, attaches a 1-labelled edge, and decorates the source of this edge by a loop with the original node label. The

[3] Prepare contains rules that *relabel* nodes, that is, rules in which the node labelling function of the interface is partial. These rules can be simulated by programs with ordinary rules. We omit the details for space reasons; they will be given in a long version of this paper.

program $\texttt{Inc}\downarrow$ attaches a chain of 1-labelled edges to a loop-marked node and marks the begin of the chain by a loop. The length of the chain coincides with the number of not yet relabelled nodes in the current graph. Note that visited nodes temporarily get auxiliary labels to prevent further visiting. After termination of $\texttt{Inc}\downarrow$ the nodes get their original labels be the subprogram $\texttt{Relabel}\downarrow$. The elementary program `Stop` replaces a loop by an ordinary edge.

The program `Bundle` is defined by

$$\texttt{Bundle} = \langle \texttt{Separate};\ \texttt{Copy}\downarrow;\ \texttt{Redirect}\downarrow\rangle\downarrow,$$

where `Separate` separates an edge from the graph and initiates copying, $\texttt{Copy}\downarrow$ copies the information about the source and the target of an edge, and $\texttt{Redirect}\downarrow$ redirects edges such that a bundle of string graphs is obtained.

The program `Compose` is given by

$$\texttt{Compose} = \langle\langle \texttt{Initiate};\ \texttt{Search}\downarrow\rangle\downarrow; \texttt{Extend}_1; \texttt{Extend}_2\rangle.$$

Here `Initiate` initiates the connection of two string graphs by connecting the begin nodes of two different string graphs with a #-labelled edge. The program $\texttt{Search}\downarrow$ searches for the end of the first string graph and redirects the source of the #-labelled edge to the end of it. Finally, $\texttt{Extend}_1$ and $\texttt{Extend}_2$ add a node and a #-labelled edge at the begin and the end of the string graph.

The rules of $\texttt{Encode}_2$ are given in Figure 6. By inspecting the rules, it is not difficult to check that `Encode` behaves as stated in the proposition. □

Lemma 2. *There is a program* `Decode` *such that for every graph expression* w *in* Σ^* *and every abstract graph* G *in* $\mathcal{A}_{\mathcal{C}_2}$,

$$w^\bullet \to_{\texttt{Decode}} G \text{ if and only if } \operatorname{gra}(w) = G.$$

Proof. Let `Decode` be the program

$$\texttt{Decode} = \texttt{Ite}(\bullet, \texttt{Decode}_1, \texttt{Decode}_2)$$

where $\bullet$ is the abstract graph consisting of a single node labelled with $\bullet$, and $\texttt{Decode}_1$ and $\texttt{Decode}_2$ are programs for decoding $\bullet$ and string graphs representing nonempty graph expressions, respectively. While $\texttt{Decode}_1$ consists of a single rule deleting $\bullet$, $\texttt{Decode}_2$ consists of three subprograms:

$$\texttt{Decode}_2 = \langle \texttt{Decompose};\ \texttt{Interweave};\ \texttt{MakeNode}\rangle.$$

`Decompose` decomposes the string graph of a nonempty graph expression into a bundle of string graphs representing nodes and edges, `Interweave` interweaves this bundle into an edge-labelled graph, and `MakeNode` transforms this graph into a graph with node and edge labels and removes auxiliary information.

The program `Decompose` is given by

$$\texttt{Decompose} = \langle \texttt{Cut}_1;\ \texttt{Cut}_2;\ \texttt{Deco}\downarrow\rangle,$$

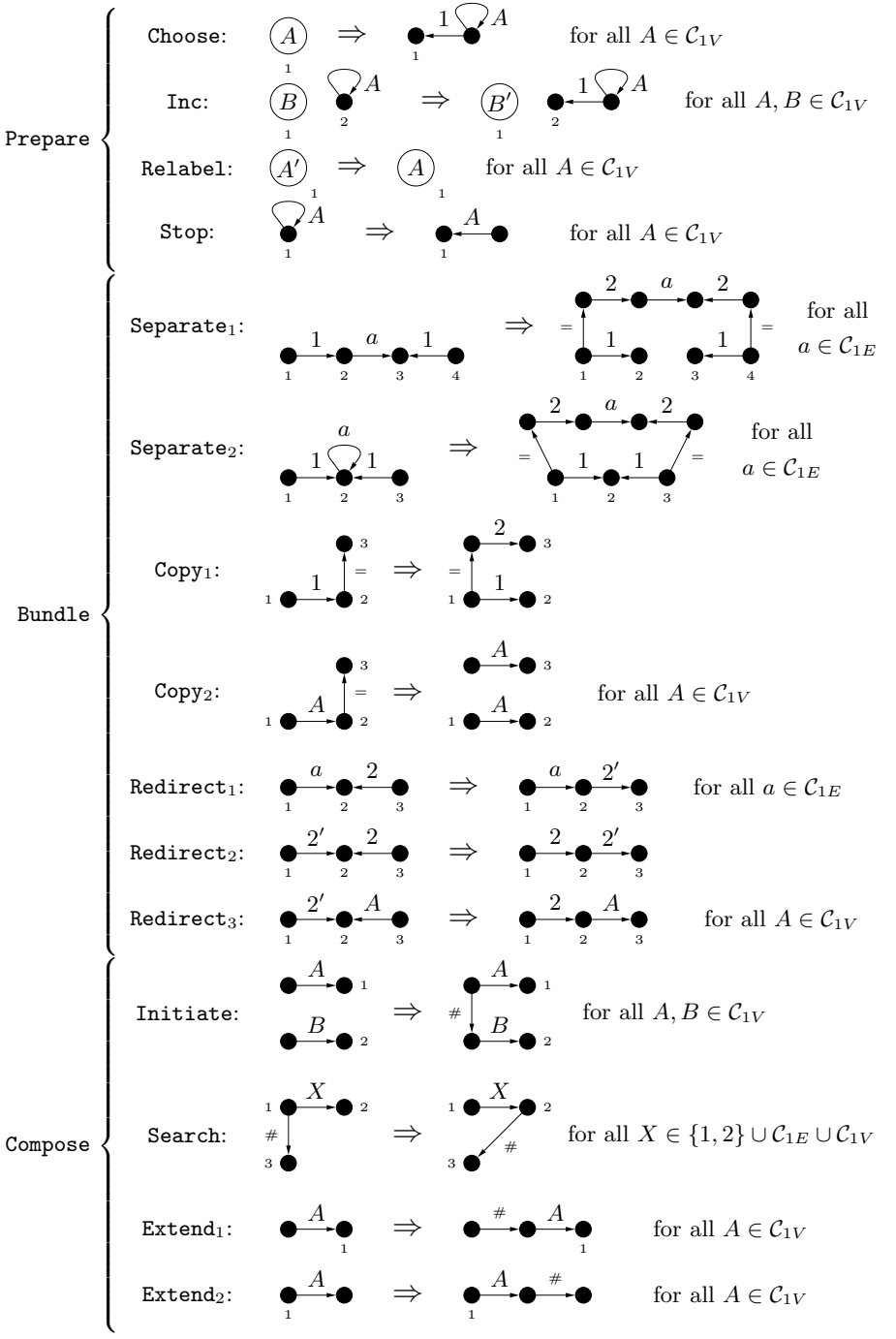


Fig. 6. The rules of the program Encode$_2$

where $\texttt{Cut}_1$ and $\texttt{Cut}_2$ remove the outer #-labelled edges together with the outermost nodes, and $\texttt{Deco}\downarrow$ removes the inner #-labelled edges. This results in a bundle of string graphs representing nodes and edges.

The program `Interweave` is defined as follows:

$$\texttt{Interweave} = \langle \texttt{Redirect}\downarrow;\ \texttt{MarkAll};\ \texttt{Checkid}\downarrow \rangle.$$

Here $\texttt{Redirect}\downarrow$ redirects the edges representing the target node of an edge, `MarkAll` with

$$\texttt{MarkAll} = \langle \texttt{Select};\ \texttt{Connect}\downarrow;\ \texttt{Relabel}\downarrow \rangle \downarrow$$

marks all pairs of representations which have to be checked with respect to coincidence, and `Checkid` with

$$\texttt{Checkid} = \langle \texttt{Initiate}; \texttt{Compare}\downarrow; \texttt{Delete}\downarrow;\ \langle \texttt{Ident};\ \texttt{GarColl}\downarrow \rangle \downarrow \rangle$$

selects a pair and compares the representations. If the representations do not coincide, it finishes the comparison by deleting the comparison edge. Otherwise, it identifies two nodes[4] and performs "garbage collection".

Finally, `MakeNode` is given by

$$\texttt{MakeNode} = \langle \texttt{MakeNode}_1;\ \texttt{MakeNode}_2\downarrow \rangle \downarrow,$$

where $\texttt{MakeNode}_1$ transforms edges representing nodes into nodes and removes auxiliary edges and nodes, and $\texttt{MakeNode}_2\downarrow$ removes further auxiliary information from the graph.

The rules of $\texttt{Decode}_2$ are given in Figure 7. By inspecting the rules one can see that `Decode` transforms graph expressions in form of string graphs into abstract graphs, and that it is correct in the sense that for every graph expression w in Σ^* and every abstract graph G in $\mathcal{A}_{\mathcal{C}_2}$, $w^\bullet \rightarrow_{\texttt{Decode}} G$ if and only if $\mathrm{gra}(w) = G$. □

7 Minimality

Our programming language defined in Section 3 is minimal in that omitting either sequential composition or iteration results in a computationally incomplete language. For the proof of this fact we call a function $f\colon \mathcal{A}_\mathcal{C} \to \mathcal{A}_\mathcal{C}$ *cyclic* if there are some G in $\mathcal{A}_\mathcal{C}$ and $n \geq 2$ such that $f(G) \neq G$ and $f^n(G) = G$.

Lemma 3. *No cyclic function is computable by a program of the form* $P\downarrow$.

Proof. Let $f\colon \mathcal{A}_\mathcal{C} \to \mathcal{A}_\mathcal{C}$ be cyclic and consider some G in $\mathcal{A}_\mathcal{C}$ and $n \geq 2$ such that $f(G) \neq G$ and $f^n(G) = G$. Suppose that there is a program $P\downarrow$ such that $\rightarrow_{P\downarrow} = f$. Then $f^{n-1}(G) \rightarrow_{P\downarrow} G$ and hence G is a normal form with respect to $\rightarrow_P$. But $G \rightarrow_{P\downarrow} f(G)$ and $G \neq f(G)$ imply $G \rightarrow_P^+ f(G)$. Thus G cannot be a normal form, a contradiction. □

[4] `Ident` contains a rule $\langle L \leftarrow K \rightarrow R \rangle$ where $K \rightarrow R$ is not injective, but it can be replaced by a program in which all rules have injective morphisms. We omit this program for space reasons.

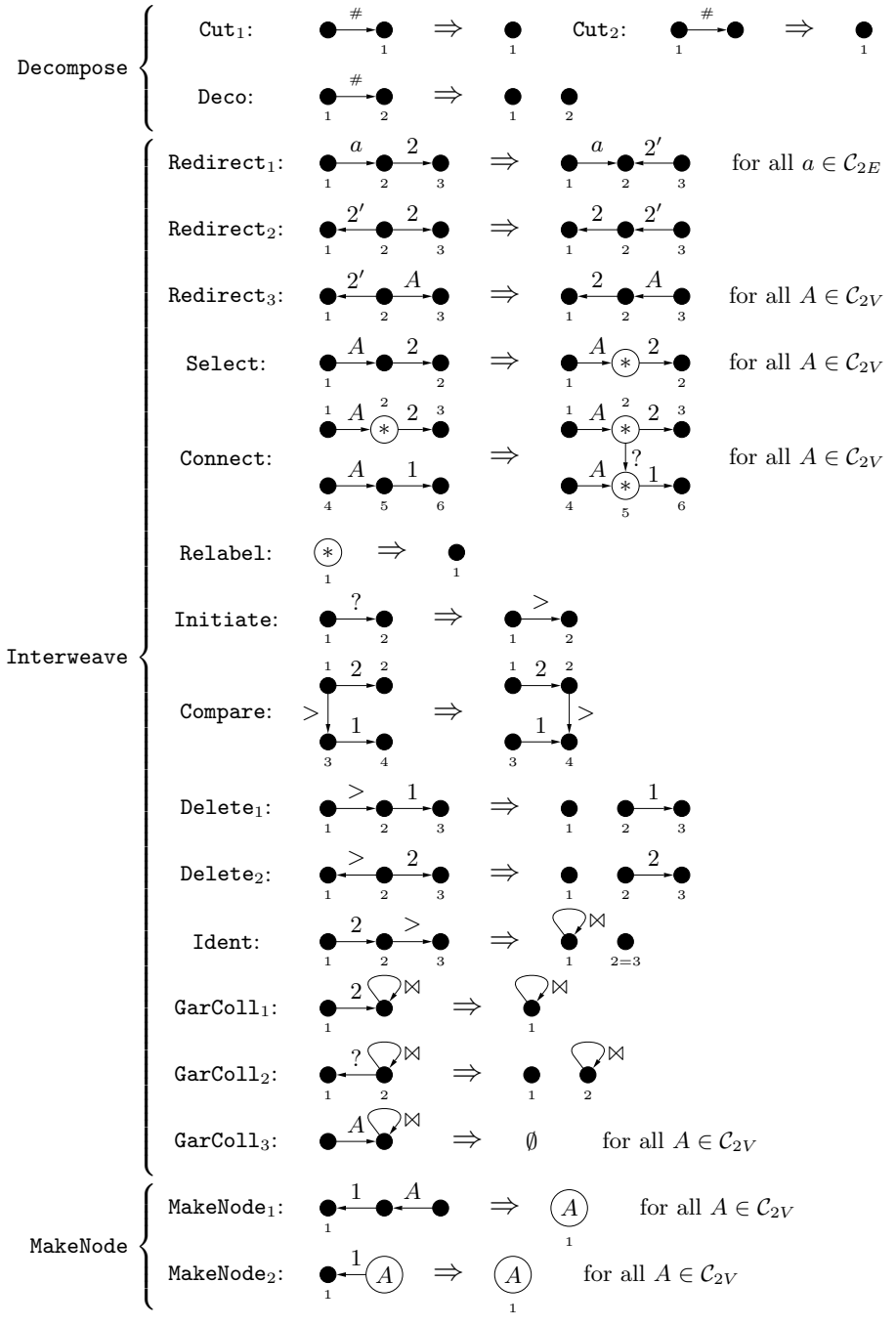


Fig. 7. The rules of the program `Decode`$_2$

For example, the function converse: $\mathcal{A}_\mathcal{C} \to \mathcal{A}_\mathcal{C}$ discussed in Example 1.2 is cyclic since converse(converse(G)) = G for every G in $\mathcal{A}_\mathcal{C}$. Hence a program computing this function cannot have an outermost iteration construct.

Theorem 3. *The set of programs without sequential composition is computationally incomplete.*

Proof. The function converse: $\mathcal{A}_\mathcal{C} \to \mathcal{A}_\mathcal{C}$ of Example 1.2 is computable and cyclic. By Lemma 3, a program P without sequential composition that computes this function has to be elementary. Let n be the largest number of edges occurring in the left-hand side of any rule in P. Consider $[G]$ in $\mathcal{A}_\mathcal{C}$ with $\mathrm{V}_G = \{0, 1, \ldots, n{+}2\}$, $\mathrm{E}_G = \{1, \ldots, n+2\}$ and for $i = 1, \ldots, n+2$, $\mathrm{s}_G(i) = 0$ and $\mathrm{t}_G(i) = i$. Now if $[G] \to_P [H]$, then H contains at least two edges with a common source node. So $[H] \neq \text{converse}([G])$ and hence P does not compute converse. □

An argument similar to the one in the above proof also shows that programs without iteration are computationally incomplete. For, it is clear that a program $\mathcal{R}_1; \ldots; \mathcal{R}_n$ whose component programs are elementary cannot convert graphs of arbitrary size.

It is worth mentioning that Theorem 3 has an alternative proof showing that programs without sequential composition cannot compute any function $f: \mathcal{A}_\mathcal{C} \to \mathcal{A}_\mathcal{C}$ satisfying (1) $f(\emptyset) \neq \emptyset$ and (2) for every $n \geq 0$ there is a graph G such that $\text{size}(G) + n < \text{size}(f(G))$. So the class of functions not computable without sequential composition does not just contain cyclic functions. The proof of this fact will be given in a long version of this paper.

8 Conclusion

We have answered the question what programming constructs are needed on top of (double-pushout) graph transformation rules to obtain a computationally complete programming language. It turned out that sequential composition and iteration of programs suffice for this purpose. Moreover, we have shown that omitting either of these two constructs makes the language incomplete.

These results should help to better understand the semantics and power of existing programming languages based on graph transformation rules, and they should also be useful for the design of new languages of this kind. In particular, due to the simplicity of our language, it should be feasible to prove computational completeness for a language in question by translating our programs into semantically equivalent programs of that language.

References

[Ehr79] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In *Proc. Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of Lecture Notes in Computer Science, 1–69. Springer-Verlag, 1979.

[ERT99] Claudia Ermel, Michael Rudolf, and Gabi Taentzer. The AGG approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 14, 551–603. World Scientific, 1999.

[FM98] Pascal Fradet and Daniel Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2–3):263–289, 1998.

[GKS91] John Glauert, Richard Kennaway, and Ronan Sleep. Dactl: An experimental graph rewriting language. In *Proc. Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 378–395. Springer-Verlag, 1991.

[HMP99] Annegret Habel, Jürgen Müller, Detlef Plump. Double-pushout graph transformation revisited. Bericht Nr. 7/99, Fachbereich Informatik, Universität Oldenburg, 1999. Revised version to appear in *Mathematical Structures in Computer Science*.

[LP98] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, second edition, 1998.

[Rod98] Peter J. Rodgers. A graph rewriting programming language for graph drawing. In *Proc. 14th IEEE Symposium on Visual Languages*. IEEE Computer Society Press, 1998.

[SWZ99] Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 13, 487–550. World Scientific, 1999.

[Ues78] Tadahiro Uesu. A system of graph grammars which generates all recursively enumerable sets of labelled graphs. *Tsukuba J. Math.* 2, 11–26, 1978.

[Wei87] Klaus Weihrauch. *Computability*. Volume 9 of EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1987.

Axioms for Recursion in Call-by-Value
(Extended Abstract)

Masahito Hasegawa and Yoshihiko Kakutani

Research Institute for Mathematical Sciences, Kyoto University
{hassei,kakutani}@kurims.kyoto-u.ac.jp

Abstract. We propose an axiomatization of fixpoint operators in typed call-by-value programming languages, and give its justifications in two ways. First, it is shown to be sound and complete for the notion of uniform T-fixpoint operators of Simpson and Plotkin. Second, the axioms precisely account for Filinski's fixpoint operator derived from an iterator (infinite loop constructor) in the presence of first-class controls, provided that we define the uniformity principle on such an iterator via a notion of effect-freeness (centrality). We also investigate how these two results are related in terms of the underlying categorical models.

1 Introduction

While the equational theories of *fixpoint operators* in call-by-name programming languages and in domain theory have been extensively studied and now there are some canonical axiomatizations (including the *iteration theories* [1] and *Conway theories*, equivalently *traced cartesian categories* [9] – see [18] for the latest account), there seems no such widely-accepted result in the context of *call-by-value (cbv) programming languages*. In this paper we propose a candidate of such an axiomatization, which consists of three simple axioms.

A type-indexed family of closed values $\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau} : ((\sigma \to \tau) \to \sigma \to \tau) \to \sigma \to \tau$ is called a *stable uniform call-by-value fixpoint operator* if the following conditions are satisfied:

1. (cbv fixpoint) For any value $F : (\sigma \to \tau) \to \sigma \to \tau$
   $\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau} F \;=\; \lambda x^{\sigma}.F\,(\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau} F)\,x$
2. (stability) For any value $F : (\sigma \to \tau) \to \sigma \to \tau$
   $\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau} F \;=\; \mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\,(\lambda f^{\sigma\to\tau}.\lambda x^{\sigma}.F\,f\,x)$
3. (uniformity) For values $F : (\sigma \to \tau) \to \sigma \to \tau$, $G : (\sigma' \to \tau') \to \sigma' \to \tau'$ and $H : (\sigma \to \tau) \to \sigma' \to \tau'$, if $H(\lambda x^{\sigma}.M\,x) = \lambda y^{\sigma'}.H\,M\,y$ holds for any $M : \sigma \to \tau$ (such an H is called *rigid*) and $H \circ F = G \circ H$ holds, then $H\,(\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau} F) \;=\; \mathsf{fix}^{\mathsf{v}}_{\sigma'\to\tau'} G$

The first axiom is known as the *call-by-value fixpoint equation*; the eta-expansion in the right-hand-side means that $\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau} F$ is equal to a value. The second axiom

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 246–260, 2001.

says that, though the functionals F and $\lambda f.\lambda x.F\, f\, x$ may behave differently, their fixpoints, when applied to values, satisfy the same fixpoint equation and cannot be distinguished. The last axiom is a call-by-value variant of Plotkin's *uniformity principle*; here the *rigid functionals* (the word "rigid" was coined by Filinski in [4]) play the rôle of *strict functions* in the uniformity principle for the call-by-name fixpoint operators. Intuitively, a rigid functional uses its argument exactly once, and it does not matter whether the argument is evaluated beforehand or evaluated at its actual use. Our uniformity axiom can be justified by the fact that $H\,(\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\,F)$ satisfies the same fixpoint equation as $\mathsf{fix}^{\mathsf{v}}_{\sigma'\to\tau'}\,G$ when H is rigid and $H \circ F = G \circ H$ holds:

$$\begin{aligned} H\,(\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\,F) &= H\,(\lambda x^{\sigma}.F\,(\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\,F)\,x) && \text{cbv fixpoint equation for } \mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\,F \\ &= \lambda y^{\sigma'}.H\,(F\,(\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\,F))\,y && H \text{ is rigid} \\ &= \lambda y^{\sigma'}.G\,(H\,(\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\,F))\,y && H \circ F = G \circ H \end{aligned}$$

We give two main results on these axioms.

1. The λ_c-*calculus* (*computational lambda calculus*) [12] with a stable uniform cbv fixpoint operator is sound and complete for the models based on the notion of *uniform T-fixpoint operators* of Simpson and Plotkin [18].
2. In the *call-by-value* $\lambda\mu$-*calculus* [17] (= the λ_c-calculus plus *first-class continuations*) there is a bijective correspondence between stable uniform cbv fixpoint operators and *uniform iterators*, via Filinski's construction of *recursion from iteration* [4].

In fact, we distill our axioms from the uniform T-fixpoint operators, so the first result is not an unexpected one. A surprise is the second one, in that the axioms precisely account for Filinski's cbv fixpoint operator derived from an *iterator* (infinite loop constructor) and first-class continuations, provided that we refine Filinski's notion of uniformity, for which the distinction between values and effect-free programs [19,7] is essential.

So here is an interesting coincidence of a category-theoretic axiomatics (of Simpson and Plotkin) with a program construction (of Filinski). However, we also show that, after sorting out the underlying categorical semantics, Filinski's construction combined with the *Continuation-Passing Style (CPS) translation* can be understood within the abstract setting of Simpson and Plotkin.

Construction of this paper. In Section 2 we recall the λ_c-calculus and the call-by-value $\lambda\mu$-calculus, which will be used as our working languages in this paper. Section 3 demonstrates how our axioms are used for establishing the Filinski's correspondence between recursion and iteration (which can be seen as a syntactic proof of the second main result). Up to this section, all results are presented in an entirely syntactic manner. In Section 4 we start to look at the semantic counterpart of our axiomatization, by recalling the categorical models of the λ_c-calculus and the call-by-value $\lambda\mu$-calculus. We then recall the notion of uniform T-fixpoint operators on these models in Section 5, and explain how

our axioms are distilled from the uniform T-fixpoint operators (the first main result). In Section 6, we specialise the result in the previous section to the models of the call-by-value $\lambda\mu$-calculus, and give a semantic proof of the second main result. Section 7 gives some concluding remarks.

2 The Call-by-Value Calculi

The λ_c-calculus (computational lambda calculus) [12], an improvement of the call-by-value λ-calculus [14], is sound and complete for

1. categorical models based on *strong monads* [12]
2. Continuation-Passing Style translation into the $\lambda\beta\eta$-calculus [16]

and has been proved useful for reasoning about call-by-value programs. In particular, it can be seen as the theoretical backbone of (the typed version of) the theory of *A-normal forms* [6], which enables us to optimise call-by-value programs directly without performing the CPS translation.

For these reasons, we take the λ_c-calculus as a basic calculus for typed call-by-value programming languages. We also use an extension of the λ_c-calculus with first-class controls, called the call-by-value $\lambda\mu$-calculus, for which the soundness and completeness results mentioned above have been extended by Selinger [17].

2.1 The λ_c-Calculus

The syntax, typing rules and axioms on the well-typed terms of the λ_c-calculus are summarised in Figure 1. The types, terms and typing judgements are those of the standard simply typed lambda calculus (including the unit $\top$ and binary products $\times$). c^σ ranges over the constants of type σ. As an abbreviation, we write $\mathsf{let}\ x^\sigma\ \mathsf{be}\ M\ \mathsf{in}\ N$ for $(\lambda x^\sigma.N)M$. $\mathrm{FV}(M)$ denotes the set of free variables in M. The crucial point is that we have the notion of values, and the axioms are designed so that the above-mentioned completeness results hold. Below we may call a term a value if it is provably equal to a value defined by the grammar.

Centre and focus. In call-by-value languages, we often regard values as representing effect-free (finished or suspended) computation. While this intuition is valid, the converse may not always be justified; in fact, the answer depends on the computational effects under consideration [7]. In a λ_c-theory (where we may have additional constructs and axioms), we say that a term $M : \sigma$ is *central* if it commutes with any other computational effect, that is,

$$\mathsf{let}\ x^\sigma\ \mathsf{be}\ M\ \mathsf{in}\ \mathsf{let}\ y^\tau\ \mathsf{be}\ N\ \mathsf{in}\ L\ =\ \mathsf{let}\ y^\tau\ \mathsf{be}\ N\ \mathsf{in}\ \mathsf{let}\ x^\sigma\ \mathsf{be}\ M\ \mathsf{in}\ L\ :\ \theta$$

holds for any $N : \tau$ and $L : \theta$, where x and y are not free in M and N. In addition, we say that $M : \sigma$ is *focal* if it is central and moreover *copyable* and *discardable*, i.e., $\mathsf{let}\ x^\sigma\ \mathsf{be}\ M\ \mathsf{in}\ \langle x, x\rangle = \langle M, M\rangle : \sigma \times \sigma$ and $\mathsf{let}\ x^\sigma\ \mathsf{be}\ M\ \mathsf{in}\ * = * : \top$ hold. It is worth emphasising that a value is always focal, but the converse is not true. A detailed analysis of these concepts in several λ_c-theories is found in [7]; see also discussions in Section 4.

Types $\sigma, \tau \quad ::= b \mid \sigma \to \tau \mid \top \mid \sigma \times \tau$ where b ranges over base types
Terms $M, N ::= x \mid c^\sigma \mid \lambda x^\sigma.M \mid M\,N \mid * \mid \langle M, N\rangle \mid \pi_1\,M \mid \pi_2\,M$
Values $V, U \quad ::= x \mid c^\sigma \mid \lambda x^\sigma.M \mid * \mid \langle V, U\rangle \mid \pi_1\,V \mid \pi_2\,V$

Typing Rules:

$$\frac{}{\Gamma \vdash x : \sigma}\ x{:}\sigma \in \Gamma \qquad \frac{}{\Gamma \vdash c^\sigma : \sigma} \qquad \frac{\Gamma, x:\sigma \vdash M : \tau}{\Gamma \vdash \lambda x^\sigma.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}$$

$$\frac{}{\Gamma \vdash * : \top} \qquad \frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash \langle M, N\rangle : \sigma \times \tau} \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \pi_1\,M : \sigma} \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \pi_2\,M : \tau}$$

Axioms:

$$\begin{array}{lll}
\mathsf{let}\ x^\sigma\ \mathsf{be}\ V\ \mathsf{in}\ M = M[V/x] & & \\
\lambda x^\sigma.V\,x = V & & (x \not\in \mathrm{FV}(V)) \\
V = * & & (V : \top) \\
\pi_i\langle V_1, V_2\rangle = V_i & & \\
\langle \pi_1\,V, \pi_2\,V\rangle = V & & \\
\mathsf{let}\ x^\sigma\ \mathsf{be}\ M\ \mathsf{in}\ x = M & & \\
\mathsf{let}\ y^\tau\ \mathsf{be}\ (\mathsf{let}\ x^\sigma\ \mathsf{be}\ L\ \mathsf{in}\ M)\ \mathsf{in}\ N = \mathsf{let}\ x^\sigma\ \mathsf{be}\ L\ \mathsf{in}\ \mathsf{let}\ y^\tau\ \mathsf{be}\ M\ \mathsf{in}\ N & & (x \not\in \mathrm{FV}(N)) \\
M\,N = \mathsf{let}\ f^{\sigma\to\tau}\ \mathsf{be}\ M\ \mathsf{in}\ \mathsf{let}\ x^\sigma\ \mathsf{be}\ N\ \mathsf{in}\ f\,x & & (M : \sigma\to\tau, N : \sigma) \\
\langle M, N\rangle = \mathsf{let}\ x^\sigma\ \mathsf{be}\ M\ \mathsf{in}\ \mathsf{let}\ y^\tau\ \mathsf{be}\ N\ \mathsf{in}\ \langle x, y\rangle & & (M : \sigma, N : \tau) \\
\pi_i\,M = \mathsf{let}\ x^{\sigma\times\tau}\ \mathsf{be}\ M\ \mathsf{in}\ \pi_i\,x & & (M : \sigma \times \tau)
\end{array}$$

Fig. 1. The λ_c-calculus

2.2 The Call-by-Value $\lambda\mu$-Calculus

Our call-by-value $\lambda\mu$-calculus, summarised in Figure 2, is the version due to Selinger [17]. We regard it as an extension of the λ_c-calculus with first-class continuations and sum types (the empty type $\bot$ and binary sums $+$). We write $\neg\sigma$ for the type $\sigma \to \bot$ ("negative type").

Remark 1. We have chosen the cbv $\lambda\mu$-calculus as our working language firstly because we intend the results in this paper to be compatible with the duality result of the second author [11] (see Section 7) which is based on Selinger's work on the $\lambda\mu$-calculus [17], and secondly because it has a well-established categorical semantics, again thanks to Selinger. However our results are not specific to the $\lambda\mu$-calculus; they apply also to any other language with similar semantics – for example, we could have used Hofmann's axiomatization of control operators [10]. Also, strictly speaking, the inclusion of sum types (coproducts) is not necessary in the main development of this paper, though they enable us to describe iterators more naturally (as general feedback operators, see Remark 2 in Section 3) and are also used in some principles on iterators like diagonal property (see Section 7), and crucially needed for the duality result in [17,11].

The typing judgements take the form $\Gamma \vdash M : \sigma \mid \Delta$ where $\Delta = \alpha_1 : \tau_1, \ldots, \alpha_n : \sigma_n$ is a sequence of *names* (ranged over by α, β,...) with their types. A judgement $x_1 : \sigma_1, \ldots, x_m : \sigma_m \vdash M : \tau \mid \alpha_1 : \tau_1, \ldots, \alpha_n : \tau_n$ represents

$$
\begin{array}{lll}
\text{Types } \sigma,\tau & ::= \cdots \mid \bot \mid \sigma+\tau & \\
\text{Terms } M,N & ::= \cdots \mid [\alpha]M \mid \mu\alpha^{\sigma}.M \mid [\alpha,\beta]M \mid \mu(\alpha^{\sigma},\beta^{\tau}).M & \\
\text{Values } V,U & ::= \cdots \mid \mu(\alpha^{\sigma},\beta^{\tau}).[\alpha]V \mid \mu(\alpha^{\sigma},\beta^{\tau}).[\beta]V & \text{where } \alpha,\beta \not\in \mathrm{FN}(V)
\end{array}
$$

Additional Typing Rules:

$$
\frac{\Gamma \vdash M : \sigma \mid \Delta}{\Gamma \vdash [\alpha]M : \bot \mid \Delta}\ \alpha{:}\sigma \in \Delta
\qquad
\frac{\Gamma \vdash M : \bot \mid \alpha : \sigma, \Delta}{\Gamma \vdash \mu\alpha^{\sigma}.M : \sigma \mid \Delta}
$$

$$
\frac{\Gamma \vdash M : \sigma+\tau \mid \Delta}{\Gamma \vdash [\alpha,\beta]M : \bot \mid \Delta}\ \alpha{:}\sigma, \beta{:}\tau \in \Delta
\qquad
\frac{\Gamma \vdash M : \bot \mid \alpha : \sigma, \beta : \tau, \Delta}{\Gamma \vdash \mu(\alpha^{\sigma},\beta^{\tau}).M : \sigma+\tau \mid \Delta}
$$

Additional Axioms:

$$
\begin{array}{lll}
V\,(\mu\alpha^{\sigma}.M) & = \mu\beta^{\tau}.M[[\beta](V(-))/[\alpha](-)] & (V : \sigma \to \tau) \\
[\alpha'](\mu\alpha^{\sigma}.M) & = M[\alpha'/\alpha] & \\
\mu\alpha^{\sigma}.[\alpha]M & = M & (\alpha \not\in \mathrm{FN}(M)) \\
[\alpha',\beta'](\mu(\alpha^{\sigma},\beta^{\tau}).M) & = M[\alpha'/\alpha,\beta'/\beta] & \\
\mu(\alpha^{\sigma},\beta^{\tau}).[\alpha,\beta]M & = M & (\alpha,\beta \not\in \mathrm{FN}(M)) \\
[\alpha]M & = M & (M : \bot) \\
[\alpha]M & = \mathsf{let}\ x^{\sigma}\ \mathsf{be}\ M\ \mathsf{in}\ [\alpha]x & (M : \sigma) \\
[\alpha,\beta]M & = \mathsf{let}\ x^{\sigma+\tau}\ \mathsf{be}\ M\ \mathsf{in}\ [\alpha,\beta]x & (M : \sigma+\tau)
\end{array}
$$

Fig. 2. The call-by-value $\lambda\mu$-calculus

a well-typed term M with at most m *free variables* $x_1,\ldots,x_m$ and n *free names* $\alpha_1,\ldots,\alpha_n$. We write $\mathrm{FN}(M)$ for the set of free names in M. In this judgement, M can be thought as a proof of the sequent $\sigma_1,\ldots,\sigma_m \vdash \tau, \tau_1,\ldots,\tau_n$ or the proposition $(\sigma_1 \wedge \ldots \wedge \sigma_m) \to (\tau \vee \tau_1 \vee \ldots \vee \tau_n)$ in the classical propositional logic. Among the additional axioms, the first one involves the *mixed substitution* $M[C(-)/[\alpha](-)]$ for a term M, a context $C(-)$ and a name α, which is the result of recursively replacing any subterm of the form $[\alpha]N$ by $C(N)$ and any subterm of the form $[\alpha_1,\alpha_2]N$ (with $\alpha=\alpha_1$ or $\alpha=\alpha_2$) by $C(\mu\alpha.[\alpha_1,\alpha_2]N)$. See [17] for further details on these syntactic conventions.

Centre and focus. In the presence of first-class controls, central and focal terms coincide [19,17], and enjoy a simple characterisation (*thunkability* [19]).

Lemma 1. *In a cbv $\lambda\mu$-theory, the following conditions on a term $M : \sigma$ are equivalent.*

1. *M is central.*
2. *M is focal.*
3. *(thunkability)* $\mathsf{let}\ x^{\sigma}\ \mathsf{be}\ M\ \mathsf{in}\ \lambda k^{\neg\sigma}.k\,x = \lambda k^{\neg\sigma}.k\,M : \neg\neg\sigma$ *holds.*

We also note that central terms and values agree at function types [17].

Lemma 2. *In a cbv $\lambda\mu$-theory, a term $M : \sigma \to \tau$ is central if and only if it is a value, i.e., $M = \lambda x^{\sigma}.M\,x$ holds.*

3 Recursion from Iteration

For grasping the rôle of our axioms, it is best to look at the actual construction in the second main result: the correspondence of recursors and iterators under the presence of first-class continuations due to Filinski [4]. For ease of presentation, we write $g \circ f$ for the composition $\lambda x.g\,(f\,x)$ of values f and g, and id_σ for $\lambda x^\sigma.x$.

A type-indexed family of closed values $\mathsf{loop}_\sigma : (\sigma \to \sigma) \to \neg\sigma$ is called a *uniform iterator* if the following conditions are satisfied:

1. (iteration) For any value $f : \sigma \to \sigma$, $\quad \mathsf{loop}_\sigma\, f \;=\; \lambda x^\sigma.\mathsf{loop}_\sigma\, f\,(f\,x)$
2. (uniformity) For values $f : \sigma \to \sigma$, $g : \sigma' \to \sigma'$ and $h : \sigma \to \sigma'$, if h is *total* and $h \circ f = g \circ h$ holds, then $\;(\mathsf{loop}_{\sigma'}\, g) \circ h = \mathsf{loop}_\sigma\, f$

where a value $h : \sigma \to \tau$ is called *total* if $h\,v : \tau$ is central (see Section 2) for any value $v : \sigma$. The word "total" is due to Filinski [4], though in his original definition $h\,v$ is asked to be a *value* rather than a central term.[1]

Remark 2. The expressive power of an iterator is not so weak, as we can derive a general *feedback operator* $\mathsf{feedback}_{\sigma,\tau} : (\sigma \to \sigma + \tau) \to \sigma \to \tau$ from an iterator using sums and first-class controls, which satisfies (with a syntax sugar for sums) $\mathsf{feedback}_{\sigma,\tau}\, f\, a \;=\; \mathsf{case}\; f\, a\; \mathsf{of}\; (\mathsf{in}_1\, x^\sigma \Rightarrow \mathsf{feedback}_{\sigma,\tau}\, f\, x \,|\, \mathsf{in}_2\, y^\tau \Rightarrow y)$ for values $f : \sigma \to \sigma + \tau$ and $a : \sigma$.

Surprisingly, in the presence of first-class continuations, there is a bijective correspondence between the stable uniform cbv fixpoint operators and the uniform iterators. We recall the construction which is essentially the same as that in [4]. Sample codes are found in Figure 3 (in SML/NJ [8]) and 4 (in the cbv $\lambda\mu$-calculus).

The construction is divided into two parts. For the first part, we introduce a pair of "inside-out" (contravariant) constructions

$$\begin{cases} \mathsf{step}_{\sigma,\tau} \;:\; (\neg\tau \to \neg\sigma) \to \sigma \to \tau \\ \mathsf{pets}_{\sigma,\tau} = \lambda f^{\sigma\to\tau}.\lambda k^{\neg\tau}.\lambda x^\sigma.k\,(f\,x) \;:\; (\sigma \to \tau) \to \neg\tau \to \neg\sigma \end{cases}$$

so that $\mathsf{step}_{\sigma,\tau} \circ \mathsf{pets}_{\sigma,\tau} = \mathsf{id}_{\sigma\to\tau}$ and $\mathsf{pets}_{\sigma,\tau} \circ \mathsf{step}_{\sigma,\tau} = \lambda F^{\neg\tau\to\neg\sigma}.\lambda k^{\neg\tau}.\lambda x^\sigma.F\,k\,x$ hold; here we need first-class continuations to implement $\mathsf{step}_{\sigma,\tau}$. We are then able to see that, if loop is a uniform iterator, the composition

$$\mathsf{loop}_\sigma \circ \mathsf{step}_{\sigma,\sigma} \;:\; (\neg\sigma \to \neg\sigma) \to \neg\sigma$$

yields a stable uniform fixpoint operator restricted on the negative types $\neg\sigma$. In particular, the cbv fixpoint axiom is verified as (by noting the equation $k^{\neg\tau}\,(\mathsf{step}_{\sigma,\tau}\, F^{\neg\tau\to\neg\sigma}\, x^\sigma) \;=\; F\,k\,x$)

$$\begin{aligned} (\mathsf{loop}_\sigma \circ \mathsf{step}_{\sigma,\sigma})\, F &= \mathsf{loop}_\sigma\,(\mathsf{step}_{\sigma,\sigma}\, F) \\ &= \lambda x^\sigma.\mathsf{loop}_\sigma\,(\mathsf{step}_{\sigma,\sigma}\, F)\,(\mathsf{step}_{\sigma,\sigma}\, F\, x) \\ &= \lambda x^\sigma.F\,(\mathsf{loop}_\sigma\,(\mathsf{step}_{\sigma,\sigma}\, F))\, x \\ &= \lambda x^\sigma.F\,((\mathsf{loop}_\sigma \circ \mathsf{step}_{\sigma,\sigma})\, F)\, x \end{aligned}$$

[1] Our use of the word "total" can be misleading, as there is a more general and perhaps more sensible notion of "totality" used in Thielecke's analysis [20]. However in this paper we put our priority on the compatibility with Filinski's development in [4].

Conversely, if $\mathsf{fix}^{\mathsf{v}}$ is a stable uniform fixpoint operator,

$$\mathsf{fix}^{\mathsf{v}}_{\neg\sigma} \circ \mathsf{pets}_{\sigma,\sigma} \ : \ (\sigma \to \sigma) \to \neg\sigma$$

gives a uniform iterator:

$$\begin{aligned}(\mathsf{fix}^{\mathsf{v}}_{\neg\sigma} \circ \mathsf{pets}_{\sigma,\sigma})\, f &= \mathsf{fix}^{\mathsf{v}}_{\neg\sigma}\,(\mathsf{pets}_{\sigma,\sigma}\, f)\\ &= \lambda x^{\sigma}.(\mathsf{pets}_{\sigma,\sigma}\, f)\,(\mathsf{fix}^{\mathsf{v}}_{\neg\sigma}\,(\mathsf{pets}_{\sigma,\sigma}\, f))\, x\\ &= \lambda x^{\sigma}.(\lambda k^{\neg\sigma}.\lambda y^{\sigma}.k\,(f\,y))\,(\mathsf{fix}^{\mathsf{v}}_{\neg\sigma}\,(\mathsf{pets}_{\sigma,\sigma}\, f))\, x\\ &= \lambda x^{\sigma}.(\mathsf{fix}^{\mathsf{v}}_{\neg\sigma}\,(\mathsf{pets}_{\sigma,\sigma}\, f))\,(f\, x)\\ &= \lambda x^{\sigma}.(\mathsf{fix}^{\mathsf{v}}_{\neg\sigma} \circ \mathsf{pets}_{\sigma,\sigma})\, f\,(f\, x)\end{aligned}$$

One direction of the bijectivity of these constructions is guaranteed by the stability axiom (while the other direction follows from $\mathsf{step}_{\sigma,\sigma} \circ \mathsf{pets}_{\sigma,\sigma} = \mathsf{id}_{\sigma\to\sigma}$):

$$\mathsf{fix}^{\mathsf{v}}_{\neg\sigma} \circ \mathsf{pets}_{\sigma,\sigma} \circ \mathsf{step}_{\sigma,\sigma} = \lambda F.\mathsf{fix}^{\mathsf{v}}_{\neg\sigma}(\lambda k.\lambda x.F\,k\,x) = \lambda F.\mathsf{fix}^{\mathsf{v}}_{\neg\sigma}\, F = \mathsf{fix}^{\mathsf{v}}_{\neg\sigma}$$

We note that step sends a rigid function to a total one, while pets sends a total function to a rigid one, and moreover they are (contravariantly) functorial. These facts imply that the two notions of uniformity for recursors and iterators are in perfect harmony.

The second part is to reduce fixpoints on an arrow type $\sigma \to \tau$ to those on a negative type $\neg(\sigma \times \neg\tau)$. This is possible because we can implement an isomorphism (again using first-class continuations)

$$\mathsf{switch}_{\sigma,\tau} \ : \ \neg(\sigma \times \neg\tau) \xrightarrow{\simeq} \sigma \to \tau$$

which is rigid ($\mathsf{switch}_{\sigma,\tau}\,(\lambda x^{\sigma}.M\,x) = \lambda y^{\sigma\times\neg\tau}.\,\mathsf{switch}_{\sigma,\tau}\, M\, y$ holds) and we have

$$\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\, F \ = \ \mathsf{switch}_{\sigma,\tau}\,(\mathsf{fix}^{\mathsf{v}}_{\neg(\sigma\times\neg\tau)}\,(\mathsf{switch}^{-1}_{\sigma,\tau} \circ F \circ \mathsf{switch}_{\sigma,\tau}))$$

by the uniformity ($\mathsf{switch}_{\sigma,\tau} \circ (\mathsf{switch}^{-1}_{\sigma,\tau} \circ F \circ \mathsf{switch}_{\sigma,\tau}) = F \circ \mathsf{switch}_{\sigma,\tau}$). So we conclude that, under the presence of first-class continuations, stable uniform cbv fixpoint operators are precisely those derived from uniform iterators, and vice versa:

$$\begin{aligned}\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\, F \ &= \ \mathsf{switch}_{\sigma,\tau}(\mathsf{loop}_{\sigma\times\neg\tau}(\mathsf{step}_{\sigma\times\neg\tau,\sigma\times\neg\tau}(\mathsf{switch}^{-1}_{\sigma,\tau} \circ F \circ \mathsf{switch}_{\sigma,\tau})))\\ \mathsf{loop}_{\sigma}\, f \ &= \ \mathsf{fix}^{\mathsf{v}}_{\neg\sigma}(\mathsf{pets}_{\sigma,\sigma}\, f)\end{aligned}$$

Behind syntax. As noted by Filinski, the CPS-transform of an iterator is a usual (call-by-name) fixpoint operator on the types of the form R^A in the target $\lambda\beta\eta$-calculus, where R is the answer type. If we let T be the continuation monad $R^{R^{(-)}}$, then the uniform T-fixpoint operator of Simpson and Plotkin [18] precisely amounts to the uniform fixpoint operator on the types R^A.

Since our first main result (Section 5, Theorem 1) is that the stable uniform cbv fixpoint operator is sound and complete for uniform T-fixpoint operators, it turns out that Filinski's construction combined with the CPS translation can be regarded as a consequence of the general categorical axiomatics. By specialising it to the setting with a continuation monad, we obtain a semantic version of the *recursion from iteration* construction (Section 6, Theorem 2 and 3).

```
(* an empty type "bot" with an initial map "abort" A : bot -> 'a *)
datatype bot = VOID of bot;
fun A (VOID v) = A v;
(* the C operator, C : (('a -> bot) -> bot) -> 'a *)
fun C f = SMLofNJ.Cont.callcc
            (fn k => A (f (fn x => (SMLofNJ.Cont.throw k x) : bot)));

(* basic combinators *)
fun step F x = C (fn k => F k x);
fun pets f k x = k (f x) : bot;
fun switch l x = C (fn q => l (x,q));
fun switch_inv f (x, k) = k (f x) : bot;
(* step : (('a -> bot) -> 'b -> bot) -> 'b -> 'a
   pets : ('a -> 'b) -> ('b -> bot) -> 'a -> bot
   switch : ('a * ('b -> bot) -> bot) -> 'a -> 'b
   switch_inv : ('a -> 'b) -> 'a * ('b -> bot) -> bot *)

(* an iterator, loop : ('a -> 'a) -> 'a -> bot *)
fun loop f x = loop f (f x) : bot;

(* recursion from iteration *)
fun fix F = switch (loop (step (switch_inv o F o switch)));
(* fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b *)
```

Fig. 3. Coding in SML/NJ (versions based on SML '97)

4 Categorical Semantics

4.1 Models of the λ_c-Calculus

Let $\mathcal{C}$ be a category with finite products and a strong monad $T = (T, \eta, \mu, \theta)$. We write $\mathcal{C}_T$ for the Kleisli category of T, and $J : \mathcal{C} \to \mathcal{C}_T$ for the associated left adjoint functor. We assume that $\mathcal{C}$ has Kleisli exponentials, i.e., for every X in $\mathcal{C}$ the functor $J((-) \times X) : \mathcal{C} \to \mathcal{C}_T$ has a right adjoint $X \Rightarrow (-) : \mathcal{C}_T \to \mathcal{C}$. This gives the structure for modelling computational lambda calculus [12]. Following Moggi, we call such a structure a *computational model*.

$$
\begin{aligned}
\mathsf{step}_{\sigma,\tau} &= \lambda F^{\neg\tau\to\neg\sigma}.\lambda x^{\sigma}.\mu\beta^{\tau}.F(\lambda y^{\tau}.[\beta]y)x &&: (\neg\tau \to \neg\sigma) \to \sigma \to \tau \\
\mathsf{pets}_{\sigma,\tau} &= \lambda f^{\sigma\to\tau}.\lambda k^{\neg\tau}.\lambda x^{\sigma}.k(f\,x) &&: (\sigma \to \tau) \to \neg\tau \to \neg\sigma \\
\mathsf{switch}_{\sigma,\tau} &= \lambda l^{\neg(\sigma\times\neg\tau)}.\lambda x^{\sigma}.\mu\beta^{\tau}.l\langle x, \lambda y^{\tau}.[\beta]y\rangle &&: \neg(\sigma \times \neg\tau) \to \sigma \to \tau \\
\mathsf{switch}^{-1}_{\sigma,\tau} &= \lambda f^{\sigma\to\tau}.\lambda\langle x^{\sigma}, k^{\neg\tau}\rangle.k(f\,x) &&: (\sigma \to \tau) \to \neg(\sigma \times \neg\tau)
\end{aligned}
$$

Fig. 4. Coding in the call-by-value $\lambda\mu$-calculus

4.2 Models of the Call-by-Value $\lambda\mu$-Calculus

Let $\mathcal{C}$ be a distributive category, i.e., a category with finite products and coproducts so that $(-) \times A : \mathcal{C} \to \mathcal{C}$ preserves finite coproducts for each A. We call an object R a *response object* if there exists an exponential R^A for each A, i.e., $\mathcal{C}(- \times A, R) \simeq \mathcal{C}(-, R^A)$ holds. Given such a structure, we can model the cbv $\lambda\mu$-calculus in the Kleisli category $\mathcal{C}_T$ of the strong monad $T = R^{R^{(-)}}$ [17]. Following Selinger, we call $\mathcal{C}$ a *response category* and the Kleisli category $\mathcal{C}_T$ a *category of continuations* and write $R^{\mathcal{C}}$ for $\mathcal{C}_T$ (though in [17] a category of continuations means the opposite of $R^{\mathcal{C}}$).

4.3 Centre and Focus

We have already seen the notion of centre and focus in the λ_c-calculus and the cbv $\lambda\mu$-calculus in a syntactic form (Section 2). However, these concepts originally arose from the analysis on the category-theoretic models given as above. Following the discovery of the *premonoidal structure* on the Kleisli category part $\mathcal{C}_T$ ($R^{\mathcal{C}}$) of these models [15], Thielecke [19] proposed a *direct* axiomatization of $R^{\mathcal{C}}$ not depending on the base category $\mathcal{C}$ (which may be seen as a chosen category of "values") but on the subcategory of "effect-free" morphisms of $R^{\mathcal{C}}$, which is the *focus* (equivalently *centre*) of $R^{\mathcal{C}}$. Führmann [7] carries out further study on models of the λ_c-calculus along this line.

For lack of space we do not describe the details of these analyses. However, we will soon see that these concepts naturally arise in our analysis of the uniformity principles for recursors and iterators. In particular, a total value $h : \sigma \to \tau$ (equivalently the term $x : \sigma \vdash h\, x : \tau$) precisely corresponds to the central morphisms in the semantic models. In the case of the models of the cbv $\lambda\mu$-calculus, the centre can be characterised in terms of the category of algebras, for which our uniformity principles are defined; that is, we have

Proposition 1. *$f \in R^{\mathcal{C}}(A, B) \simeq \mathcal{C}(R^B, R^A)$ is central if and only if its counterpart in $\mathcal{C}$ is an algebra morphism from the canonical algebra structure on R^B (see Section 6) to that on R^A.*

We note that this result has been observed in various forms in [19,17,7].

5 Uniform T-Fixpoint Operators

In this section we shall consider a computational model with the base category $\mathcal{C}$ and a strong monad T.

Definition 1. *[18] A* T-fixpoint operator on $\mathcal{C}$ *is a family of functions*

$$(-)^* : \mathcal{C}(TX, TX) \to \mathcal{C}(1, TX)$$

such that, for any $f : TX \to TX$, $f \circ f^* = f^*$ holds. It is called uniform *if, for any $f : TX \to TX$, $g : TY \to TY$ and $h : TX \to TY$, $h \circ \mu = \mu \circ Th$ and $g \circ h = h \circ f$ imply $g^* = h \circ f^*$.*

Thus a T-fixpoint operator is given as a fixpoint operator restricted on the objects of the form TX. However, this is sufficient to model a call-by-value fixpoint operator. To see this, suppose that we are given an object A with an arrow $\alpha : TA \to A$ so that $\alpha \circ \eta = id$ (in fact it is more natural to ask (A, α) to be a T-algebra, see Proposition 2 below). Given $f : A \to A$, we have $\alpha \circ (\eta \circ f \circ \alpha)^* : 1 \to A$ and

$$\begin{aligned} \alpha \circ (\eta \circ f \circ \alpha)^* &= \alpha \circ \eta \circ f \circ \alpha \circ (\eta \circ f \circ \alpha)^* \\ &= f \circ \alpha \circ (\eta \circ f \circ \alpha)^* \end{aligned}$$

Therefore we can extend $(-)^*$ to be a fixpoint operator on A.

Definition 2. *[18] Suppose that $\mathcal{S}$ and $\mathcal{D}$ are categories with finite products and the same objects, and $I : \mathcal{S} \to \mathcal{D}$ is a functor which strictly preserves finite products and is the identity on objects. A* parameterized fixpoint operator on $\mathcal{D}$ *is a family of functions $(-)^\dagger : \mathcal{D}(X \times A, A) \to \mathcal{D}(X, A)$ which is natural in X and satisfies $f^\dagger = f \circ \langle id_X, f^\dagger \rangle$. It is* parametrically uniform with respect to $I : \mathcal{S} \to \mathcal{D}$ *if, for any $f : X \times A \to A$, $g : X \times B \to B$ in $\mathcal{D}$ and $h : A \to B$ in $\mathcal{S}$, $Ih \circ f = g \circ (id_X \times Ih)$ implies $g^\dagger = Ih \circ f^\dagger$.*

Proposition 2. *[18] Let $\mathcal{C}^T$ be the category of T-algebras and algebra morphisms. Let $\mathcal{D}$ be the category whose objects are T-algebras and hom-sets are given by $\mathcal{D}((A, \alpha), (B, \beta)) = \mathcal{C}(A, B)$, and let $I : \mathcal{C}^T \to \mathcal{D}$ be the inclusion. Then a uniform T-fixpoint operator on $\mathcal{C}$ induces a parametrically uniform parameterized fixpoint operator on $\mathcal{D}$ with respect to $I : \mathcal{C}^T \to \mathcal{D}$, and vice versa.*

(The reader is invited to check that the standard domain-theoretic situations arise by taking T as the lifting monad on a category of predomains.) In particular, Kleisli exponentials $X \Rightarrow Y$ fit in this scheme, where the algebra structure $\alpha_{X,Y} : T(X \Rightarrow Y) \to X \Rightarrow Y$ is given as the adjoint mate of

$$T(X \Rightarrow Y) \times X \xrightarrow{\theta} T((X \Rightarrow Y) \times X) \xrightarrow{T\mathsf{ev}} T^2Y \xrightarrow{\mu} TY$$

where $\mathsf{ev} : (X \Rightarrow Y) \times X \to TY$ is the counit of the adjunction. We note that $\eta \circ \alpha_{X,Y} : T(X \Rightarrow Y) \to T(X \Rightarrow Y)$ corresponds to an eta-expansion in the λ_c-calculus. That is, if a term $\Gamma \vdash M : X \to Y$ represents an arrow $f : A \to T(X \Rightarrow Y)$ in $\mathcal{C}$, then $\Gamma \vdash \lambda x^X.M\,x : X \to Y$ represents $\eta \circ \alpha_{X,Y} \circ f : A \to T(X \Rightarrow Y)$. This observation is frequently used in distilling the axioms of the stable uniform cbv fixpoint operators below.

5.1 Axiomatization in the λ_c-Calculus

Using the λ_c-calculus as an internal language of $\mathcal{C}_T$, the equation $f^* = f \circ f^*$ on $X \Rightarrow Y$ can be represented as

$$F^* \;=\; \lambda x^X.F\,F^*\,x \quad \text{where } F = \lambda f^{X \to Y}.\lambda x^X.F\,f\,x : (X \to Y) \to X \to Y$$

The side condition $F = \lambda f^{X \to Y}.\lambda x^X.F\,f\,x$ means that F corresponds to an arrow in $\mathcal{C}(X \Rightarrow Y, X \Rightarrow Y)$, not $\mathcal{C}_T(X \Rightarrow Y, X \Rightarrow Y)$. However, the operator

$(-)^* : \mathcal{C}(X \Rightarrow Y, X \Rightarrow Y) \to \mathcal{C}(1, X \Rightarrow Y)$ can be equivalently axiomatized by a slightly different operator

$$(-)^\ddagger : \mathcal{C}(X \Rightarrow Y, T(X \Rightarrow Y)) \to \mathcal{C}(1, X \Rightarrow Y)$$

subject to $f^\ddagger = \alpha_{X,Y} \circ f \circ f^\ddagger$, with an additional condition $f^\ddagger = (\eta \circ \alpha_{X,Y} \circ f)^\ddagger$. In fact, we can define such a $(-)^\ddagger$ as $(\alpha_{X,Y} \circ (-))^*$ and conversely $(-)^*$ by $(\eta \circ (-))^\ddagger$, and it is easy to see that these are in bijective correspondence. The condition $f^\ddagger = \alpha_{X,Y} \circ f \circ f^\ddagger$, equivalently $\eta \circ f^\ddagger = \eta \circ \alpha_{X,Y} \circ f \circ f^\ddagger$, is axiomatized in the λ_c-calculus as (by recalling that $\eta \circ \alpha_{X,Y} \circ (-)$ gives an eta-expansion)

$$F^\ddagger \;=\; \lambda x.F\, F^\ddagger\, x \text{ for any value } F : (X \to Y) \to X \to Y$$

which is precisely the cbv fixpoint axiom. The additional condition $f^\ddagger = (\eta \circ \alpha_{X,Y} \circ f)^\ddagger$ is axiomatized as

$$F^\ddagger = (\lambda f.\lambda x.F\, f\, x)^\ddagger \quad \text{where } F \text{ is a value}$$

This is no other than the stability axiom. We thus obtain the first two axioms of our stable uniform cbv fixpoint operators, which are precisely modelled by T-fixpoint operators.

5.2 Uniformity Axiom

Finally, we shall see how the uniformity condition on T-fixpoint operators can be represented in the λ_c-calculus. By Proposition 2, we define uniformity with respect to $I : \mathcal{C}^T \to \mathcal{D}$; that is, we regard $H \in \mathcal{C}(X \Rightarrow Y, X' \Rightarrow Y')$ as "strict" (or "rigid" in our terminology) if it is an algebra morphism from $(X \Rightarrow Y, \alpha_{X,Y})$ to $(X' \Rightarrow Y', \alpha_{X',Y'})$.[2] Spelling out this condition, we ask H to satisfy $H \circ \alpha_{X,Y} = \alpha_{X',Y'} \circ T(H)$, equivalently $T(H) \circ \eta \circ \alpha_{X,Y} = \eta \circ \alpha_{X',Y'} \circ T(H)$. In terms of the λ_c-calculus, this means that an eta-expansion commutes with the application of H; therefore, in the λ_c-calculus, we ask $H : (X \to Y) \to X' \to Y'$ to be a value such that

$$H(\lambda x^X.M\, x) \;=\; \lambda y^{X'}.H\, M\, y \;:\; X' \to Y'$$

holds for any $M : X \to Y$. We have called such an H rigid, and defined the uniformity condition with respect to such rigid functionals.

Theorem 1. *The computational models with a uniform T-fixpoint operator provide a sound and complete class of models of the computational lambda calculus with a stable uniform call-by-value fixpoint operator.*

[2] A characterisation of rigid functions (on computation types) in the same spirit is given in Filinski's thesis [5] (Section 2.2.2) though unrelated to the uniformity of fixpoint operators.

6 Recursion from Iteration Revisited

6.1 Iteration in the Category of Continuations

Let $\mathcal{C}$ be a response category with a response object R. An *iterator* on the category of continuations $R^{\mathcal{C}}$ is a family of functions $(-)_* : R^{\mathcal{C}}(A, A) \to R^{\mathcal{C}}(A, 0)$ so that $f_* = f_* \circ f$ holds for $f \in R^{\mathcal{C}}(A, A)$. Spelling out this definition in $\mathcal{C}$, to give an iterator on $R^{\mathcal{C}}$ is to give a family of functions $(-)^* : \mathcal{C}(R^A, R^A) \to \mathcal{C}(1, R^A)$ so that $f^* = f \circ f^*$ holds for $f \in \mathcal{C}(R^A, R^A)$. Thus an iterator on $R^{\mathcal{C}}$ (hence in the cbv $\lambda\mu$-calculus) is no other than a fixpoint operator on $\mathcal{C}$ (hence the target call-by-name calculus) restricted on objects of the form R^A ("negative objects").

Example 1. We give a simple-minded model of the cbv $\lambda\mu$-calculus with an iterator. Let $\mathcal{C}$ be the category of ω-cpos (possibly without bottom) and continuous maps, and let R be an ω-cpo with bottom. Since $\mathcal{C}$ is a cartesian closed category with finite coproducts, it serves as a response category with the response object R. Moreover there is a least fixpoint operator on the negative objects R^A because R^A has a bottom element, thus we have an iterator on $R^{\mathcal{C}}$ (which in fact is a unique uniform iterator in the sense below).

6.2 Relation to Uniform T-Fixpoint Operators

For any object A, the negative object R^A canonically has a T-algebra structure $\alpha_A = \lambda m^{R^{R^{R^A}}}.\lambda x^A.m\,(\lambda f^{R^A}.f\,x) : R^{R^{R^A}} \to R^A$ for the monad $T = R^{R^{(-)}}$. Thus the consideration on the uniform T-fixpoint operators applies to this setting: if this computational model has a uniform T-fixpoint operator, then we have a fixpoint operator on negative objects, hence we can model an iterator of the cbv $\lambda\mu$-calculus in the category of continuations.

Conversely, if we have an iterator on $R^{\mathcal{C}}$, then it corresponds to a fixpoint operator on negative objects in $\mathcal{C}$, which of course include objects of the form $TA = R^{R^A}$. Therefore we obtain a T-fixpoint operator. It is then natural to expect that, if the iterator satisfies a suitable uniformity condition, then it bijectively corresponds to a uniform T-fixpoint operator. This uniformity condition on an iterator must be determined again with respect to the category of algebras $\mathcal{C}^T$. So we regard $h \in R^{\mathcal{C}}(A, B) \simeq \mathcal{C}(R^B, R^A)$ as "strict" ("total" in our terminology) when its counterpart in $\mathcal{C}(R^B, R^A)$ is an algebra morphism from (R^B, α_B) to (R^A, α_A), i.e., $h \circ \alpha_B = \alpha_A \circ R^{R^h}$ holds in $\mathcal{C}$. We say that an iterator $(-)_*$ on $R^{\mathcal{C}}$ is *uniform* if $f_* = g_* \circ h$ holds for $f : A \to A$, $g : B \to B$ and total $h : A \to B$ such that $h \circ f = g \circ h$.

Theorem 2. *Given a response category $\mathcal{C}$ with a response object R, to give a uniform $R^{R^{(-)}}$-fixpoint operator on $\mathcal{C}$ is to give a uniform iterator on $R^{\mathcal{C}}$.*

Fortunately, the condition to be an algebra morphism is naturally represented in a cbv $\lambda\mu$-theory. A value $h : A \to B$ represents an algebra morphism if and only if

$$x : A \vdash \mathsf{let}\ y^B\ \mathsf{be}\ h\,x\ \mathsf{in}\ \lambda k^{\neg B}.k\,y\ =\ \lambda k^{\neg B}.k(h\,x)\ :\ \neg\neg B$$

holds – in fact, the CPS translation of this equation is no other than the equation $h \circ \alpha_B = \alpha_A \circ R^{R^h}$. By Lemma 1, in a cbv $\lambda\mu$-theory, this requirement is equivalent to saying that $h\,x$ is a central term for each value x (this also implies Proposition 1). Therefore we obtain the uniformity condition for an iterator in Section 3.

Theorem 3. *In a cbv $\lambda\mu$-theory, there is a bijective correspondence between the stable uniform cbv fixpoint operators and the uniform iterators.*

6.3 On Filinski's Uniformity

In [4] Filinski introduced uniformity principles for both cbv fixpoint operators and iterators, for establishing a bijective correspondence between them. While his definitions turn out to be sufficient for his purpose, in retrospect they seem to be somewhat ad hoc and are strictly weaker than our uniformity principles. Here we give a brief comparison.

First, Filinski calls a value $h : \sigma \to \tau$ "total" when $h\,v$ is a *value* for each value $v : \sigma$. However, while a value is always central, the converse is not true. Note that, while the notion of centre is uniquely determined for each cbv $\lambda\mu$-theory (and category of continuations), the notion of value is not canonically determined (a category of continuations can arise from different response categories [17]). Since the uniformity principle is determined not in terms of the base category $\mathcal{C}$ but in terms of the category of algebras $\mathcal{C}^T$, it seems natural that it corresponds to the notion of centre which is determined not by $\mathcal{C}$ but by $\mathcal{C}_T$.

Second, Filinski calls a value $H : (\sigma \to \tau) \to \sigma' \to \tau'$ "rigid" when there are total $h_1 : \sigma' \to \tau \to \tau'$ and $h_2 : \sigma' \to \sigma$ such that

$$H = \lambda f^{\sigma\to\tau}.\lambda y^{\sigma'}.h_1\, y\, (f\, (h_2\, y)) : (\sigma \to \tau) \to \sigma' \to \tau'$$

holds. It is easily checked that if H is rigid in the sense of Filinski, it is also rigid in our sense – but the converse does not hold, even if we change the notion of total values to ours (for instance, $\mathsf{switch}_{\sigma,\tau}$ in Section 3 is not rigid in the sense of Filinski). By closely inspecting the correspondence of rigid functionals and total functions via the $\mathsf{step}/\mathsf{pets}$ and switch constructions, we can strengthen Filinski's formulation to match ours:

Proposition 3. *In a cbv $\lambda\mu$-theory, $H : (\sigma \to \tau) \to \sigma' \to \tau'$ is rigid if and only if there are total $h_1 : \sigma' \to \tau \to \tau'$ and $h_2 : (\sigma' \times \neg\tau') \to \sigma$ such that $H = \lambda f^{\sigma\to\tau}.\lambda y^{\sigma'}.\mu\gamma^{\tau'}.[\gamma](h_1\, y\, (f\, (h_2\, \langle y, \lambda z^{\tau'}.[\gamma]z\rangle)))$ holds.*

This subsumes Filinski's rigid functionals as special cases where h_2 does not use the second argument.

Remark 3. Filinski's uniformity principle in [4] takes the following form: if H is rigid and $H \circ (\lambda f.\lambda x.F\, f\, x) = G \circ H$ holds, then $H\,(\mathsf{fix}^{\mathsf{v}} F) = \mathsf{fix}^{\mathsf{v}}\, G$. It follows that this condition is equivalent to our stability and uniformity axioms.

7 Conclusion and Further Work

We have proposed an axiomatization of fixpoint operators in typed call-by-value programming languages, and have shown that it can be justified in two different ways: as a sound and complete axiomatization for uniform T-fixpoint operators of Simpson and Plotkin [18], and also by Filinski's bijective correspondence between recursion and iteration under the presence of first-class continuations [4]. We also have shown that these results are closely related, by inspecting the semantic structure behind Filinski's construction, which turns out to be a special case of the uniform T-fixpoint operators.

Towards practical principles for call-by-value recursion. We think that our axioms are reasonably simple, and we expect they can be a practical tool for reasoning about call-by-value programs involving recursion, just in the same way as the equational theory of the computational lambda calculus is the theoretical basis of the theory of A-normal forms [16,6].

It is an interesting challenge to strengthen the axioms in some systematic ways. For instance, by adding other natural axioms on an iterator under the presence of first-class controls, one may derive the corresponding axioms on the cbv fixpoint operator. In particular, we note that the *dinaturality* $\mathsf{loop}\,(g \circ f) = \mathsf{loop}\,(f \circ g) \circ f$ on an iterator loop precisely amounts to the axiom $\mathsf{fix}^{\mathsf{v}}\,(G \circ F) = \lambda x.G\,(\mathsf{fix}^{\mathsf{v}}\,(F \circ G))\,x$ on the corresponding cbv fixpoint operator $\mathsf{fix}^{\mathsf{v}}$. Similarly, the *diagonal property* on the iterator $\mathsf{loop}\,(\lambda x.\mu\alpha.[\alpha,\alpha](f\,x)) = \mathsf{loop}\,(\lambda x.\mu\alpha.\mathsf{loop}\,(\lambda y.\mu\beta.[\alpha,\beta](f\,y))\,x)$ corresponds to that on the fixpoint operator $\mathsf{fix}^{\mathsf{v}}\,(\lambda f.F\,f\,f) = \mathsf{fix}^{\mathsf{v}}\,(\lambda f.\mathsf{fix}^{\mathsf{v}}\,(\lambda g.F\,f\,g))$. These can be seen axiomatizing the call-by-value counterpart of the Conway theories [1,9]. One may further consider the call-by-value version of the Bekič property (another equivalent axiomatization of these properties [9]) along this line, which could be used for reasoning about mutual recursion.

Another promising direction is the approach based on *fixpoint objects* [2], as a uniform T-fixpoint operator is canonically derived from a fixpoint object whose universal property implies strong proof principles. For instance, in Example 1, a uniform iterator is unique because the monad $R^{R^{(-)}}$ has a fixpoint object. For the setting with first-class controls, it might be fruitful to study the implications of the existence of a fixpoint object of continuation monads.

Relating recursion in call-by-name and in call-by-value. The results reported here can be nicely combined with *Filinski's duality* [3] between call-by-value and call-by-name languages with first-class control primitives. In his MSc thesis [11], the second author demonstrates that recursion in the *call-by-name $\lambda\mu$-calculus* [13] exactly corresponds to iteration in the call-by-value $\lambda\mu$-calculus via this duality, by extending Selinger's work [17]. Together with the observation in this paper, we obtain a bijective correspondence between call-by-name recursion and call-by-value recursion, which seems to open a way to relate the reasoning principles on recursive computations under these two calling strategies.

Acknowledgements. We thank Shin-ya Katsumata for helpful discussions, and the anonymous reviewers for numerous suggestions.

References

1. Bloom, S. and Esik, Z. (1993) *Iteration Theories*. EATCS Monographs on Theoretical Computer Science, Springer-Verlag.
2. Crole, R.L. and Pitts, A.M. (1992) New foundations for fixpoint computations: FIX-hyperdoctrines and the FIX-logic. *Inform. and Comput.* **98**(2), 171–210.
3. Filinski, A. (1989) Declarative continuations: an investigation of duality in programming language semantics. In *Proc. Category Theory and Computer Science*, Springer Lecture Notes in Comput. Sci. **389**, pp. 224–249.
4. Filinski, A.(1994) Recursion from iteration. *Lisp and Symbolic Comput.***7**(1),11–38.
5. Filinski, A. (1996) *Controlling Effects*. PhD thesis, Carnegie Mellon University, CMU-CS-96-119.
6. Flanagan, C., Sabry, A., Duba, B.F. and Felleisen, M. (1993) The essence of compiling with continuations. In *Proc. ACM Conference on Programming Languages Design and Implementation*, pp. 237–247.
7. Führmann, C. (2000) *The Structure of Call-by-Value*. PhD thesis, University of Edinburgh.
8. Harper, R., Duba, B.F. and MacQueen, D. (1993) Typing first-class continuations in ML. *J. Funct. Programming* **3**(4), 465–484.
9. Hasegawa, M. (1997) *Models of Sharing Graphs: A Categorical Semantics of* let *and* letrec. PhD thesis, University of Edinburgh, ECS-LFCS-97-360; also in Distinguished Dissertation Series, Springer-Verlag, 1999.
10. Hofmann, M. (1995) Sound and complete axiomatisations of call-by-value control operators. *Math. Structures Comput. Sci.* **5**(4), 461–482.
11. Kakutani, Y. (2001) *Duality between Call-by-Name Recursion and Call-by-Value Iteration*. MSc thesis, Kyoto University.
12. Moggi, E. (1989) Computational lambda-calculus and monads. In *Proc. 4th Annual Symposium on Logic in Computer Science*, pp. 14–23.
13. Parigot, M. (1992) $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proc. International Conference on Logic Programming and Automated Reasoning*, Springer Lecture Notes in Comput. Sci. **624**, pp. 190–201.
14. Plotkin, G.D. (1975) Call-by-name, call-by-value, and the λ-calculus. *Theoret. Comput. Sci.* **1**(1), 125–159.
15. Power, A.J. and Robinson, E.P. (1997) Premonoidal categories and notions of computation. *Math. Structures Comput. Sci.* **7**(5), 453–468.
16. Sabry, A. and Felleisen, M. (1992) Reasoning about programs in continuation-passing style. In *Proc. ACM Conference on Lisp and Functional Programming*, pp. 288–298; extended version in *Lisp and Symbolic Comput.* **6**(3/4), 289–360, 1993.
17. Selinger, P. (2001) Control categories and duality: on the categorical semantics of the lambda-mu calculus. To appear in *Math. Structures Comput. Sci.*
18. Simpson, A.K. and Plotkin, G.D. (2000) Complete axioms for categorical fixed-point operators. In *Proc. 15th Annual Symposium on Logic in Computer Science*.
19. Thielecke, H. (1997) *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, ECS-LFCS-97-376.
20. Thielecke, H. (1999) Using a continuation twice and its implications for the expressive power of call/cc. *Higher-Order and Symbolic Comput.* **12**(1), 47–73.

Class Analysis of Object-Oriented Programs through Abstract Interpretation

Thomas Jensen and Fausto Spoto

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{jensen,spoto}@irisa.fr

Abstract. We use abstract interpretation to define a uniform formalism for presenting and comparing class analyses for object-oriented languages. We consider three domains for class analysis derived from three techniques present in the literature, viz., rapid type analysis, a simple dataflow analysis and constraint-based 0-CFA analysis. We obtain three static analyses which are provably correct and whose abstract operations are provably optimal. Moreover, we prove that our formalisation of the 0-CFA analysis is more precise than that of the dataflow analysis.

Keywords: *Abstract interpretation, class analysis, object-oriented languages, domain theory, semantics.*

1 Introduction

Class analysis, one of the most important analyses for object-oriented languages, computes the set of classes that an expression can have at run-time [1,2,10,11, 12]. It can serve to prove *type safety* by guaranteeing that methods are only invoked on objects that implement such a method. In certain cases it allows one to optimise virtual method invocations into direct calls to the code implementing the method. It is also used for building a precise call graph for a program which in turn enables other analyses. A variety of class analyses have been proposed (sometimes called receiver class analysis, type analysis, *etc.*), often using different analysis frameworks. This complicates the comparison of the analyses w.r.t. their precision. The complex question of proving the analyses correct is not always addressed.

Cousot [7] shows how *abstract interpretation* [8,9] can organise eleven different type systems for functional languages into a lattice that allows one to state their correctness and establish their relative precision. Here, we apply abstract interpretation to classify and prove correct class analyses for object-oriented languages. Abstract interpretation is a technique for the systematic construction of semantics and semantics-based static analyses for programming languages. Given a concrete semantics over a concrete domain and an *abstraction function* from the concrete to an abstract semantic domain, abstract interpretation shows how to define an abstract semantics over the abstract domain such that the result

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 261–275, 2001.

of the analysis is provably *correct* w.r.t. the concrete one. The abstraction function can be seen as a specification of the program property of interest, and the abstract semantics as an analyser for that property. Furthermore, since abstract interpretation is a framework for relating semantic definitions, it can be used to compare the relative precision of two analyses. We present a framework for the static analysis of object-oriented programs. It gives semantics to a simple object-oriented language by specifying an algebra of states. The operations of this algebra are reminiscent of Java bytecode operations. Different static analyses are obtained by abstract interpretation of these operations. This approach allows one to make the following three contributions:

- We define three abstractions of the above algebra, namely, three domains for class analysis which use the same abstract information as [2], [10] and [11]. We obtain formal reconstructions of these analyses in a common framework.
- We use abstract interpretation to derive optimal (and hence correct) abstract operations for our three domains. Note that only the analysis in [11] has been provided with an (informal) correctness proof.
- We show that our reconstruction of the analysis in [11] is *provably* more precise than that of the analysis in [10].

We emphasise that we are not questioning the correctness of the original analyses. Nor are we aiming at defining new, more powerful class analyses (although the present framework could be used for this). Our goal is to set up a framework for studying the common structure of class analyses, and to facilitate their comparison and proof of correctness by means of abstract interpretation.

The paper is organised as follows. Section 2 presents notation and preliminaries. Section 3 shows our framework for the analysis of object-oriented languages, the states and their operations. Sections 4, 5 and 6 define the analyses in [2], [10] and [11], respectively, as abstract interpretations of the sets of states of Section 3. Section 7 compares the precision of our three analyses. Section 8 concludes. Due to space limitations, some proofs have been omitted.

2 Preliminaries

We recall some notions of abstract interpretation [8,9]. In the following, a total function is denoted by $\mapsto$and a partial function by $\to$. Let $(C, \leq)$ and $(A, \preceq)$ be two posets (the concrete and the abstract domain). A *Galois connection* is a pair of monotonic maps $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ such that for each $x \in C$ we have $x \leq (\gamma \circ \alpha)(x)$, and for each $y \in A$ we have $(\alpha \circ \gamma)(y) \preceq y$. A *Galois insertion* is a Galois connection where $\alpha \circ \gamma$ is the identity map. This means that the abstract domain does not contain *useless* elements. The map α (γ) is called the *abstraction* (*concretisation*) function. The abstraction map uniquely determines the concretisation map and vice versa.

Let $f : C^n \to C$ be an operator. We say that $\hat{f} : A^n \to A$ is *correct* w.r.t. f if and only if for all $y_1, \ldots, y_n \in A$ we have $\alpha(f(\gamma(y_1), \ldots, \gamma(y_n))) \preceq$

$\hat{f}(y_1, \ldots, y_n)$. For each operator f, there exists an *optimal* (most precise) correct abstract operator $\hat{f}$ defined as $\hat{f}(y_1, \ldots, y_n) = \alpha(f(\gamma(y_1), \ldots, \gamma(y_n)))$. This means that $\hat{f}$ is the most precise abstraction of f which can be defined on A.

In the theory of abstract interpretation, the *semantics* of a program is the fixpoint of a continuous operator $f : C \mapsto\!\!\!\!\rightarrow C$, where C is the computational domain [6]. Its *collecting version* [8] works over *properties* of C, i.e., over $\wp(C)$ and is the fixpoint of the powerset extension of f. If f is defined as composition of suboperations, their powerset extensions *and* $\cup$ induce the extension of f. The $\cup$ operation merges the semantics of the branches of a conditional. Note that the powerset extension of a predicate over C is a constant of $\wp(C)$. In Props. 2, 4 and 6, we compute the optimal abstractions of the powerset extension of every small operation g above as $\alpha \circ g \circ \gamma$, consistently with the previous paragraph.

Upper closure operators (*uco*'s) are monotonic, extensive and idempotent maps. They are isomorphic to Galois insertions [9]. The abstract domain induced by an uco ρ over the concrete domain L (for our purposes, the semantic domain $\wp(C)$ of a collecting semantics) is its image $\rho(L)$, i.e., the set of its fixpoints. This image is a *Moore family* of L, i.e., a non-empty meet-closed subset of L. Conversely, any Moore family of L is the image of some uco on L. Thus, in order to define a Galois insertion on L, i.e., an abstraction of L, we can equivalently define a Moore family on L, whose induced uco is the abstraction map. In Props. 1, 3 and 5 we use this technique to prove that we deal with Galois insertions.

We denote by $[v_1 \mapsto\!\!\!\!\rightarrow t_1, \ldots, v_n \mapsto\!\!\!\!\rightarrow t_n]$ a function f whose domain is $\{v_1, \ldots, v_n\}$ and such that $f(v_i) = t_i$ for $i = 1, \ldots, n$. Its update is $f[d_1/w_1, \ldots, d_m/w_m]$, where the domain of the function can be potentially enlarged. Its domain and codomain are $\mathsf{dom}(f)$ and $\mathsf{codom}(f)$, respectively. By $f|_s$ we denote the restriction of f to $s \subseteq \mathsf{dom}(f)$, by $f|_{-s}$ its restriction to $\mathsf{dom}(f) \setminus s$. A definition like $S = \langle a, b \rangle$, with a and b meta-variables, defines the selectors $s.a$ and $s.b$ for $s \in S$. For instance, Def. 2 defines $o.\kappa$ and $o.\phi$, for $o \in Obj$. If $\langle D, \leq \rangle$ is a poset and $d \in D$, we define $\downarrow(d) = \{d' \in D \mid d' \leq d\}$.

3 Semantic Domains and Operations

We describe here the semantic domain of program states and their operations. A transition trace semantics (where the state is a stack of activation frames) can be defined either in the style of Bertelsen's small-step operational semantics [3] or the Abstract State Machines of Börger and Schulte [4]. Our set of operations is not sufficiently complete to give semantics for a concrete language like Java, (neither static fields and methods nor interfaces are considered). However it is representative of the set of operations that would be required for such a task. The correctness of the whole abstract interpretation follows by fairly standard means from the correctness of the abstraction of the operations [8,9].

For simplicity, integers and booleans are the only basic types. There are no static fields or methods. Methods are uniquely identified by a *Method_ref* (for instance, their fully qualified name, like in Java), and classes by a *Class_ref*. We define $Type = \{int, bool\} \cup Class\_ref$, $Int = \mathbb{Z}$ and $Bool = \{true, false\}$.

In the following, we assume that the class structure of a program is specified by some $d \in Decl$, $p \in Pars$ and $n \in Name$ (see below), which give the classes, the parameters of the methods and their names, respectively, together with a subtype relation $\leq$ on $Type$. For simplicity, we do not allow basic types to have subtypes. By *instance variables* we refer to fields. A *program variable* is a local variable, a parameter of a method or `this`. By *class variables* we denote the variables and fields which can reference objects.

Definition 1. *Let Id be an infinite set of* identifiers*, with* $\texttt{this} \in Id$*. We define*

$$Typing = \left\{ \tau : Id \to Type \;\middle|\; \begin{array}{l} \mathsf{dom}(\tau) \text{ is finite} \\ \text{if } \texttt{this} \in \mathsf{dom}(\tau) \text{ then } \tau(\texttt{this}) \in Class\_ref \end{array} \right\}$$

$$Class = Typing \times (Id \to Method\_ref) \qquad Decl = (Class\_ref \rightharpoonup Class)$$

$$Pars = (Method\_ref \rightharpoonup \mathsf{seq}(Id) \times Typing) \quad Name = (Method\_ref \rightharpoonup Id)\ .$$

If $cl \in Class$, we name its components as $cl = \langle \tau, \nu \rangle$. If $p \in Pars$, $n \in Name$ and $mr \in Method\_ref$ then $p(mr) = \langle s, \tau \rangle$, where $\texttt{this}, n(mr) \in s$ and $\mathsf{dom}(\tau) = s$.

The hypothesis that $n(mr) \in s$ means that the name of a method is among its local variables. Namely, it is used to store its result, like in Pascal procedures. Typings map variables to types, but `this` can be bound only to objects. The indirect definition of classes through $Decl$ allows one to define two classes with the same structure. This structure is a pair whose first component is the typing of the fields defined or inherited by the class, and the second the table of the methods defined or inherited by the class. The map $Pars$ binds every method reference to the typing (its signature) and its list of parameters. This list, whose elements are the domain of the typing, is useful since it yields their order in the definition of the method. The map $Name$ binds a method reference to its name.

Example 1. Assume that we have two classes `a` and `b`, subclass of `a`, and that `a` (and, hence, `b`) has a field `n` of class `a`. We model this situation as $Class\_ref = \{\kappa_a, \kappa_b\}$, $[\kappa_a \rightharpoonup a, \kappa_b \rightharpoonup b] \in Decl$, $a = b = \langle [\texttt{n} \rightharpoonup \kappa_a], [] \rangle$ and $\kappa_b \leq \kappa_a$. Note how a and b have the same structure but have different class references.

Frames yield values to the variables addressable at a given program point. The special variable `this` cannot be unbound. *Memories* map locations to objects. *Objects* contain their *Class_ref* and the frame of their instance variables.

Definition 2. *Let Loc be an infinite set of* locations*. Let $Value = Int + Bool + Loc + \{nil\}$. Given $\tau \in Typing$, we define frames, memories and objects as in Fig. 1. Given $\mu_1, \mu_2 \in Memory$, we say that μ_2 is an* update *of μ_1, written $\mu_1 \lhd \mu_2$, if $\mathsf{dom}(\mu_1) \subseteq \mathsf{dom}(\mu_2)$ and for every $l \in \mathsf{dom}(\mu_1)$ we have $\mu_1(l).\kappa = \mu_2(l).\kappa$.*

Def. 2 states that the types of the variables in a frame are consistent with its typing, but class variables are just required to be bound to *nil* or a location (only a location for `this`). In order to guarantee that they contain objects of a class compatible with the typing, we define states, i.e., pairs of frame and memory.

$$Frame_\tau = \left\{ \phi \;\middle|\; \begin{array}{l} \phi \in \mathsf{dom}(\tau) \rightarrowtail Value \text{ and for every } v \in \mathsf{dom}(\tau) \\ \text{if } \tau(v) = int \text{ then } \phi(v) \in Int \\ \text{if } \tau(v) = bool \text{ then } \phi(v) \in Bool \\ \text{if } \tau(v) \in Class\_ref \text{ then } \phi(v) \in \{nil\} \cup Loc \\ \text{if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ then } \phi(\mathtt{this}) \in Loc \end{array} \right\}$$

$$Memory = \left\{ \begin{array}{l} \mu \in Loc \to Obj \\ \mathsf{dom}(\mu) \text{ is finite} \end{array} \right\} \qquad Obj = \left\{ \langle \kappa, \phi \rangle \;\middle|\; \begin{array}{l} \kappa \in Class\_ref \\ \phi \in Frame_{d(\kappa).\tau} \end{array} \right\}$$

Fig. 1. Frames, memories and objects.

$$\begin{array}{rl}
\mathsf{nop}_\tau & : State_\tau \rightarrowtail State_\tau \\
\mathsf{get_int}^i_\tau & : State_\tau \rightarrowtail State_{\tau[int/res]} \quad \text{with } res \not\in \mathsf{dom}(\tau),\ i \in Int \\
\mathsf{get_nil}^{cr}_\tau & : State_\tau \rightarrowtail State_{\tau[cr/res]} \quad \text{with } res \not\in \mathsf{dom}(\tau),\ cr \in Class_ref \\
\mathsf{get_bool}^b_\tau & : State_\tau \rightarrowtail State_{\tau[bool/res]} \quad \text{with } res \not\in \mathsf{dom}(\tau),\ b \in Bool \\
\mathsf{get_var}^v_\tau & : State_\tau \rightarrowtail State_{\tau[\tau(v)/res]} \quad \text{with } v \in \mathsf{dom}(\tau),\ res \not\in \mathsf{dom}(\tau) \\
\mathsf{get_field}^f_\tau & : State_\tau \to State_{\tau[d(\tau(res)).\tau(f)/res]} \\
& \quad \text{with } res \in \mathsf{dom}(\tau),\ \tau(res) \in Class_ref, f \in \mathsf{dom}(d(\tau(res)).\tau) \\
\mathsf{put_var}^v_\tau & : State_\tau \rightarrowtail State_{\tau|_{-res}} \quad \text{with } res \in \mathsf{dom}(\tau),\ v \in \mathsf{dom}(\tau),\ v \neq res,\ \tau(res) \leq \tau(v) \\
\mathsf{put_field}^f_{\tau,\tau'} & : State_\tau \rightarrowtail (State_{\tau'} \to State_{\tau|_{-res}}) \\
& \quad \text{with } res \in \mathsf{dom}(\tau),\ \tau(res) \in Class_ref,\ f \in \mathsf{dom}(d(\tau(res)).\tau), \\
& \quad \tau' = \tau[t/res] \text{ with } t \leq d(\tau(res)).\tau(f) \\
=_\tau & : State_\tau \rightarrowtail (State_\tau \rightarrowtail State_{\tau[bool/res]}) \quad \text{with } res \in \mathsf{dom}(\tau) \\
\mathsf{scope}^{mr,v_1,\ldots,v_{\#p(mr).s-2}}_\tau & : State_\tau \rightarrowtail State_{p(mr).\tau|_{-n(mr)}} \\
& \quad \text{with } res \in \mathsf{dom}(\tau),\ \tau(res) \in Class_ref, \\
& \quad p(mr).s \setminus \{n(mr), this\} = \langle \iota_1, \ldots, \iota_n \rangle,\ \tau(res) \leq p(mr).\tau(this), \\
& \quad v_i \in \mathsf{dom}(\tau) \text{ and } \tau(v_i) \leq p(mr).\tau(\iota_i) \text{ for every } i = 1, \ldots, \#p(mr).s - 2 \\
\mathsf{unscope}^{mr}_\tau & : State_\tau \rightarrowtail (State_{p(mr).\tau|_{n(mr)}} \to State_{\tau[p(mr).\tau(n(mr))/res]}) \quad \text{with } res \not\in \mathsf{dom}(\tau) \\
\mathsf{restrict}^{vs}_\tau & : State_\tau \rightarrowtail State_{\tau|_{-vs}} \quad \text{with } vs \subseteq \mathsf{dom}(\tau) \\
\mathsf{expand}^{v:t}_\tau & : State_\tau \rightarrowtail State_{\tau[t/v]} \quad \text{with } v \not\in \mathsf{dom}(\tau),\ t \in Type \\
\mathsf{lookup}^{id,mr}_\tau & \subseteq \wp(State_\tau) \\
& \quad \text{with } res \in \mathsf{dom}(\tau),\ \tau(res) \in Class_ref,\ id \in \mathsf{dom}(d(\tau(res)).\nu),\ mr \in Method_ref \\
\mathsf{new}^{cr}_\tau & : State_\tau \rightarrowtail State_{\tau[cr/res]} \quad \text{with } res \not\in \mathsf{dom}(\tau),\ cr \in Class_ref \\
\mathsf{is_true}_\tau, \mathsf{is_false} & \subseteq \wp(State_\tau) \quad \text{with } \tau(res) = bool
\end{array}$$

Fig. 2. The signatures of the operations over the states.

Definition 3. *Let $\tau \in Typing$. Recall that $\leq$ is the subtype relation. We say that $\phi \in Frame_\tau$ is* τ-correct *in a memory μ, written $\phi \propto_\tau \mu$, if it binds the class variables in its domain to objects of classes compatible with τ. Namely, $\phi \propto_\tau \mu$ if and only if for every $v \in \mathsf{dom}(\phi)$ such that $\phi(v) \in Loc$ we have $\phi(v) \in \mathsf{dom}(\mu)$ and $(\mu(\phi(v))).\kappa \leq \tau(v)$. We define*

$$State_\tau = \left\{ \langle \phi, \mu \rangle \;\middle|\; \begin{array}{l} \phi \in Frame_\tau,\ \mu \in Memory,\ \phi \propto_\tau \mu \\ \text{and for every } \langle \kappa, \phi' \rangle \in \mathsf{codom}(\mu) \text{ we have } \phi' \propto_{d(\kappa).\tau} \mu \end{array} \right\} .$$

$$\mathsf{nop}_\tau(\langle\phi,\mu\rangle) = \langle\phi,\mu\rangle \qquad \mathsf{get\_int}^i_\tau(\langle\phi,\mu\rangle) = \langle\phi[i/res],\mu\rangle$$

$$\mathsf{get\_nil}^{cr}_\tau(\langle\phi,\mu\rangle) = \langle\phi[nil/res],\mu\rangle \qquad \mathsf{get\_bool}^b_\tau(\langle\phi,\mu\rangle) = \langle\phi[b/res],\mu\rangle$$

$$\mathsf{get\_var}^v_\tau(\langle\phi,\mu\rangle) = \langle\phi[\phi(v)/res],\mu\rangle \qquad \mathsf{put\_var}^v_\tau(\langle\phi,\mu\rangle) = \langle\phi[\phi(res)/v]|_{-res},\mu\rangle$$

$$\mathsf{get\_field}^f_\tau(\langle\phi',\mu\rangle) = \begin{cases} \langle\phi'[(\mu(\phi'(res)).\phi)(f)/res],\mu\rangle & \text{if } \phi'(res) \neq nil \\ undefined & \text{otherwise} \end{cases}$$

$$\mathsf{put\_field}^f_{\tau,\tau'}(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle) = \begin{cases} \langle\phi_2|_{-res}, \mu_2[\langle\mu_2(l).\kappa, \mu_2(l).\phi[\phi_2(res)/f]\rangle/l]\rangle \\ \quad \text{if } \mu_1 \lhd \mu_2 \text{ and } (l = \phi_1(res)) \neq nil \\ undefined \\ \quad \text{otherwise} \end{cases}$$

$$=_\tau(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle) = \begin{cases} \langle\phi_2[true/res],\mu_2\rangle \\ \quad \text{if } \mu_1 \lhd \mu_2 \text{ and } \phi_1(res) = \phi_2(res) \\ \langle\phi_2[false/res],\mu_2\rangle \\ \quad \text{if } \mu_1 \lhd \mu_2 \text{ and } \phi_1(res) \neq \phi_2(res) \\ undefined \quad \text{otherwise} \end{cases}$$

$$\mathsf{scope}^{mr,v_1,\ldots,v_n}_\tau(\langle\phi,\mu\rangle) = \langle[\iota_1 \mapsto \phi(v_1), \ldots, \iota_n \mapsto \phi(v_n), \mathtt{this} \mapsto \phi(res)],\mu\rangle$$
$$\text{where } \langle\iota_1,\ldots,\iota_n\rangle = p(mr).s \setminus \{n(mr), \mathtt{this}\}$$

$$\mathsf{unscope}^{mr}_\tau(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle) = \begin{cases} \langle\phi_1[\phi_2(n(mr))/res],\mu_2\rangle & \text{if } \mu_1 \lhd \mu_2 \\ undefined \quad \text{otherwise} \end{cases}$$

$$\mathsf{restrict}^{vs}_\tau(\langle\phi,\mu\rangle) = \langle\phi|_{-vs},\mu\rangle \qquad \mathsf{expand}^{v:t}_\tau(\langle\phi,\mu\rangle) = \langle\phi[\mathsf{init}(t)/v],\mu\rangle$$

$$\mathsf{lookup}^{id,mr}_\tau(\langle\phi,\mu\rangle) \text{ if and only if } \phi(res) \neq nil \text{ and } d(\mu(\phi(res)).\kappa).\nu(id) = mr$$

$$\mathsf{new}^{cr}_\tau(\langle\phi,\mu\rangle) = \langle\phi[l/res],\mu[\langle cr, \mathsf{init}(d(cr).\tau)\rangle/l]\rangle \qquad l \in Loc \text{ fresh}$$

$$\mathsf{is\_true}_\tau(\langle\phi,\mu\rangle) \text{ if and only if } \phi(res) = true \qquad \mathsf{is\_false}_\tau(\langle\phi,\mu\rangle) \text{ if and only if } \phi(res) = false,$$

where $\mathsf{init}(int) = 0$, $\mathsf{init}(bool) = false$, $\mathsf{init}(cr) = nil$ for $cr \in Class\_ref$ and $\mathsf{init}(\tau) = \lambda v \in \mathsf{dom}(\tau).\mathsf{init}(\tau(v))$ for $\tau \in Typing$.

Fig. 3. The operations over the states.

This set forms an algebraic structure with the operations whose signatures are given in Figure 2 and which are explicitly defined in Figure 3.

In these operations the variable *res* stands for the place where intermediate results are stored and retrieved. In a semantics for a stack-based language such as the Java bytecode, *res* would be the top of the operand stack. The binary operations of Figure 3 are undefined when the memory of the second argument is not an update of that of the first one, to guarantee that the $\propto_\tau$ relation between frame and memory of a state (Definition 3) is not broken by the operation.

The nop operation does nothing. The get_int operation loads an integer into *res*. Similarly for get_nil and get_bool. The get_var operation fetches the value of a variable v and loads it into *res*. The get_field operation fetches from *res* a non-*nil* reference to the object containing the field. The content of this field is loaded into *res*, whose type is updated with the declared type of the field. Note how this content is found. Since $\phi'(res)$ points to the object containing the field, $\mu(\phi'(res))$ is that object. Its fields are in $\mu(\phi'(res)).\phi$ and the field f is then $\mu(\phi'(res)).\phi(f)$. The put_var operation copies the content of *res* into v. The declared type of *res* must be a subtype of that of v. There is no resulting value. Thus, the variable *res* is removed from the typing of the result. In the put_field

operation the *res* variable of the first argument points to the target object, while that of the second argument contains the new value for the field, whose declared type must be a subtype of that of the field. By using two states instead of an object and a value, we deal with just one semantic domain. In Fig. 3 we see its implementation. Since $l = \phi_1(res)$ points to the target object, it must not be *nil* and $\mu_2(l)$ is that object (since $\mu_1 \lhd \mu_2$). We write $\phi_2(res)$ in the variable f of the frame $\mu_2(l).\phi$ of that object. For every binary operation of the language there is a corresponding binary operation on states, like the shown case of $=$.

Four operations (scope, unscope, restrict and expand) modify the structure of frames and states. The operations scope and unscope are used before and after a method call, respectively. The former creates a new frame in which the invoked method mr executes. Its typing is $p(mr).\tau|_{-n(mr)}$ since the variable $n(mr)$, i.e., the name of the method, is not among its inputs. Note that $p(mr).\tau|_{-n(mr)}$ contains exactly the parameters of the method and the implicit `this` parameter. In this operation *res* points to the object over which the method is called and becomes the new `this` variable, which justifies the subtype check in Fig. 2. The first argument of the unscope operation is the state before the method call. The second is the state at the end of the execution of the method. The frame of this second state contains just a variable with the name of the method, and holds its result like in Pascal procedures (in a complete operational semantics, the frames of these two states would be the top two elements of the stack). The operation restores the frame at the beginning of the execution of the method, by copying into *res* its result, and yields the memory at the end of its execution. The restrict operation removes some variables from a state and expand adds a new initialised variable to a state.

The lookup predicate for dynamic method lookup holds if by invoking the method *id* on the object referenced by *res* the method referenced by *mr* is selected. The object on which the method is invoked is $\mu(\phi(res))$, its class reference $\mu(\phi(res)).\kappa$ and the list of its methods $d(\mu(\phi(res)).\kappa).\nu$. If $d(\mu(\phi(res)).\kappa).\nu(id) = mr$ means that by calling *id* we select the method *mr*. The new operation creates a new initialised object and loads it into *res*. The is_true (is_false) predicate contains those states whose *res* variable contains *true* (*false*).

Example 2. In the situation of Ex. 1, consider how the following sequence of operations change the state. Reading downwards, we start from a state containing just the variable v_1, we introduce a new variable v_2, we create and store in *res* a newly initialised object, we read its `n` field into *res* and we store it into v_2.

Operation	State
	$\langle[v_1 \mapsto l_1], [l_1 \mapsto (\kappa_a, [\mathtt{n} \mapsto nil])]\rangle$ (initial state)
$\mathsf{expand}^{v_2:\kappa_a}_{[v_1 \mapsto \kappa_a]}$	$\langle[v_1 \mapsto l_1, v_2 \mapsto nil], [l_1 \mapsto (\kappa_a, [\mathtt{n} \mapsto nil])]\rangle$
$\mathsf{new}^{\kappa_b}_{[v_1,v_2 \mapsto \kappa_a]}$	$\langle[v_1 \mapsto l_1, v_2 \mapsto nil, res \mapsto l_2], [l_1 \mapsto (\kappa_a, [\mathtt{n} \mapsto nil]), l_2 \mapsto (\kappa_b, [\mathtt{n} \mapsto nil])]\rangle$
$\mathsf{get\_field}^{\mathtt{n}}_{[v_1,v_2 \mapsto \kappa_a, res \mapsto \kappa_b]}$	$\langle[v_1 \mapsto l_1, v_2 \mapsto nil, res \mapsto nil], [l_1 \mapsto (\kappa_a, [\mathtt{n} \mapsto nil]), l_2 \mapsto (\kappa_b, [\mathtt{n} \mapsto nil])]\rangle$
$\mathsf{put\_var}^{v_2}_{[v_1,v_2,res \mapsto \kappa_a]}$	$\langle[v_1 \mapsto l_1, v_2 \mapsto nil], [l_1 \mapsto (\kappa_a, [\mathtt{n} \mapsto nil]), l_2 \mapsto (\kappa_b, [\mathtt{n} \mapsto nil])]\rangle$

$$\mathsf{nop}_\tau^{rta}(s) = (\mathsf{get\_int}_\tau^i)^{rta}(s) = (\mathsf{get\_nil}_\tau^{cr})^{rta}(s) = \mathsf{is\_true}_\tau^{rta}(s) = s$$

$$(\mathsf{get\_bool}_\tau^b)^{rta}(s) = (\mathsf{get\_var}_\tau^v)^{rta}(s) = (\mathsf{put\_var}_\tau^v)^{rta}(s) = \mathsf{is\_false}_\tau^{rta}(s) = s$$

$$(\mathsf{get\_field}_\tau^f)^{rta}(s) = \begin{cases} \emptyset & \text{if } s \cap {\downarrow}\tau(res) = \emptyset \\ s & \text{otherwise} \end{cases}$$

$$(\mathsf{put\_field}_{\tau,\tau'}^f)^{rta}(s_1)(s_2) = \begin{cases} \emptyset & \text{if } s_1 \cap s_2 \cap {\downarrow}\tau(res) = \emptyset \\ s_2 & \text{otherwise} \end{cases}$$

$$=_\tau^{rta}(s_1)(s_2) = s_2 \qquad (\mathsf{scope}_\tau^{mr,v_1,\ldots,v_n})^{rta}(s) = s$$

$$(\mathsf{unscope}_\tau^{mr})^{rta}(s_1)(s_2) = s_2 \qquad (\mathsf{restrict}_\tau^{vs})^{rta}(s) = s$$

$$(\mathsf{expand}_\tau^{v:t})^{rta}(s) = s \qquad (\mathsf{new}_\tau^{cr})^{rta}(s) = s \cup \{cr\} \qquad \cup_\tau^{rta}(s_1)(s_2) = s_1 \cup s_2$$

$$(\mathsf{lookup}_\tau^{id,mr})^{rta}(s) = \begin{cases} s & \text{if there exists } \kappa \in s \cap {\downarrow}\tau(res) \\ & \text{such that } d(\kappa).\nu(id) = mr \\ \emptyset & \text{otherwise.} \end{cases}$$

Fig. 4. The instantiation over $State^{rta}$ of the operations of Figure 3 and of $\cup$.

4 Rapid Type Analysis (*rta*-Analysis)

Bacon and Sweeney defined [2] a simple and efficient *rapid type analysis* (*rta*) of little precision. They collect the set N of classes instantiated by a `new` command. The classes of an expression e are $({\downarrow}D(e)) \cap N$, where $D(e)$ is the declared type of e. In terms of abstract interpretation, this technique amounts to defining an abstraction which collects the classes of the objects in the memory of the states. A priori, abstract interpretation can distinguish this set of classes in different program points. Then it results in a more precise analysis than that in [2].

Definition 4. *For $\tau \in Typing$, the abstract domain is $State_\tau^{rta} = \wp(Class\_ref)$ with the concretisation map $\gamma^{rta} : State_\tau^{rta} \mapsto \wp(State_\tau)$ such that (here, $o \in \mathsf{codom}(\mu)$ means that o is an object in the memory μ) $\gamma^{rta}(s) = \{\langle \phi, \mu\rangle \in State_\tau \mid$ for every $\langle \kappa, \phi'\rangle \in \mathsf{codom}(\mu)$ we have $\kappa \in s\}$.*

Proposition 1. *For $\tau \in Typing$, the set $\gamma^{rta}(State_\tau^{rta})$ is a Moore family of $\wp(State_\tau)$. Hence $State_\tau^{rta}$ is an abstract domain whose induced abstraction map is $\alpha^{rta} : \wp(State_\tau) \mapsto State_\tau^{rta}$ such that*

$$\alpha^{rta}(S) = \{\kappa \in Class\_ref \mid \langle \phi, \mu\rangle \in S \text{ and } \langle \kappa, \phi'\rangle \in \mathsf{codom}(\mu)\} \ .$$

Proof (Sketch). The non-empty set $\gamma^{rta}(State_\tau^{rta})$ is a Moore family since it is $\cap$-closed. Indeed, it can be shown that for every $\{s_i\}_{i\in\mathbb{N}} \subseteq State_\tau^{rta}$ we have $\cap_{i\in\mathbb{N}}\gamma_\tau^{rta}(s_i) = \gamma_\tau^{rta}(\cap_{i\in\mathbb{N}}s_i)$. The α^{rta} map is derived [8,9] as $\alpha^{rta}(S) = \cap\{s \in State_\tau^{rta} \mid S \subseteq \gamma^{rta}(s)\} = \cap\{s \in \wp(Class\_ref) \mid \langle \phi, \mu\rangle \in S$ and $\langle \kappa, \phi'\rangle \in \mathsf{codom}(\mu)$ entails $\kappa \in s\} = \{\kappa \in Class\_ref \mid \langle \phi, \mu\rangle \in S$ and $\langle \kappa, \phi'\rangle \in \mathsf{codom}(\mu)\}$.

Proposition 2. *The optimal abstract counterparts of the powerset extension of the operations of Figure 3 and of* $\cup$ *are those given in Figure 4.*

The only operation in Figure 4 which enlarges the set of classes created during the execution of the program is new. The operations get_field and put_field check if a class compatible with the declared type of *res* has been instantiated, since otherwise *res* must be bound to *nil*, and the concrete operation fails. Similarly, lookup checks if it has been instantiated some class such that a call to the method *id* results in the execution of the method referenced by *mr*. Note how these operations allow the set of classes instantiated to shrink. This is a consequence of the use of abstract interpretation, which deals with different program points. It is a distinguishing feature w.r.t. the original technique in [2].

Example 3. We execute over $State_\tau^{rta}$ the same sequence of instructions of Ex. 2. The initial state is the abstraction (Prop. 1) of the initial concrete state. The final abstract state says that v_1 and v_2, both of declared type a, can be bound to objects of class a or b, which is a correct but rather imprecise approximation of the concrete final state of Example 2.

Operation	State
	$\{\kappa_a\}$ (initial state)
$(\mathsf{expand}^{v_2:\kappa_a}_{[v_1 \mapsto \kappa_a]})^{rta}$	$\{\kappa_a\}$
$(\mathsf{new}^{\kappa_b}_{[v_1,v_2 \mapsto \kappa_a]})^{rta}$	$\{\kappa_a, \kappa_b\}$
$(\mathsf{get\_field}^{\mathtt{n}}_{[v_1,v_2 \mapsto \kappa_a, res \mapsto \kappa_b]})^{rta}$	$\{\kappa_a, \kappa_b\}$
$(\mathsf{put\_var}^{v_2}_{[v_1,v_2,res \mapsto \kappa_a]})^{rta}$	$\{\kappa_a, \kappa_b\}$

5 Dataflow Analysis (*df*-Analysis)

In the dataflow analysis of [10] only program variables are analysed, while fields are approximated with the downwards closure $\downarrow(t)$ of their declared type t.

Compare this with Definition 2. An abstract value is a set of types which share a supertype. An abstract frame maps variables into abstract values consistent with the given typing. The special variable `this` cannot be unbound.

Definition 5. *Given* $\tau \in Typing$*, we define* $Value^{df} = \{S \in \wp(Type) \mid S \subseteq \downarrow t \text{ for some } t \in Type\}$ *and*

$$Frame_\tau^{df} = \left\{ \phi \;\middle|\; \begin{array}{l} \phi \in \mathsf{dom}(\tau) \mapsto Value^{df} \text{ and for every } v \in \mathsf{dom}(\tau) \\ \text{if } \tau(v) = int \text{ then } \phi(v) = \{int\} \\ \text{if } \tau(v) = bool \text{ then } \phi(v) = \{bool\} \\ \text{if } \tau(v) \in Class\_ref \text{ then } \phi(v) \subseteq \downarrow\tau(v) \\ \text{and if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ then } \phi(\mathtt{this}) \neq \emptyset \end{array} \right\}.$$

In $Frame_\tau^{df}$, integer and boolean variables are always mapped to $\{int\}$ and $\{bool\}$, respectively. We consider them just to simplify the presentation.

There is no abstract memory. Then abstract states are just abstract frames (compare with Definition 3). The special abstract state $\emptyset$ represents the empty set of concrete states and improves the precision of the analysis.

$$\mathsf{nop}_\tau^{df}(\phi) = \phi \qquad (\mathsf{get\_int}_\tau^{i})^{df}(\phi) = \phi[\{int\}/res]$$

$$(\mathsf{get\_nil}_\tau^{cr})^{df}(\phi) = \phi[\emptyset/res] \qquad (\mathsf{get\_bool}_\tau^{b})^{df}(\phi) = \phi[\{bool\}/res]$$

$$(\mathsf{get\_var}_\tau^{v})^{df}(\phi) = \phi[\phi(v)/res] \qquad (\mathsf{put\_var}_\tau^{v})^{df}(\phi) = \phi[\phi(res)/v]|_{-res}$$

$$(\mathsf{get\_field}_\tau^{f})^{df}(\phi) = \begin{cases} \emptyset & \text{if } \phi(res) = \emptyset \\ \phi[\downarrow(d(\tau(res)).\tau(f))/res] & \text{otherwise} \end{cases}$$

$$(\mathsf{put\_field}_{\tau,\tau'}^{f})^{df}(\phi_1)(\phi_2) = \begin{cases} \emptyset & \text{if } \phi_1(res) = \emptyset \\ \phi_2|_{-res} & \text{otherwise} \end{cases}$$

$$=_\tau^{df}(\phi_1)(\phi_2) = \phi_2[\{bool\}/res] \qquad \cup_\tau^{df}(\phi_1)(\phi_2) = \lambda v \in \mathsf{dom}(\tau).\phi_1(v) \cup \phi_2(v)$$

$$(\mathsf{scope}_\tau^{mr,v_1,\dots,v_n})^{df}(\phi) = [\iota_1 \mapsto \phi(v_1), \dots, \iota_n \mapsto \phi(v_n), \mathtt{this} \mapsto \phi(res)]$$
$$\text{where } \langle \iota_1, \dots, \iota_n \rangle = p(mr).s \setminus \{n(mr), \mathtt{this}\}$$

$$(\mathsf{unscope}_\tau^{mr})^{df}(\phi_1)(\phi_2) = \phi_1[\phi_2(n(mr))/res]$$

$$(\mathsf{restrict}_\tau^{vs})^{df}(\phi) = \phi|_{-vs} \qquad (\mathsf{expand}_\tau^{v:t})^{df}(\phi) = \phi[\mathsf{init}^{df}(t)/v]$$

$$(\mathsf{lookup}_\tau^{id,mr})^{df}(\phi) = \begin{cases} \phi[S/res] & \text{if } S = \{cr \in \phi(res) \mid d(cr).\nu(id) = mr\} \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$(\mathsf{new}_\tau^{cr})^{df}(\phi) = \phi[\{cr\}/res] \qquad \mathsf{is\_true}_\tau^{df}(\phi) = \mathsf{is\_false}_\tau^{df}(\phi) = \phi \ ,$$

where $\mathsf{init}^{df}(int) = \{int\}$, $\mathsf{init}^{df}(bool) = \{bool\}$, $\mathsf{init}^{df}(cr) = \emptyset$ for $cr \in Class\_ref$.

Fig. 5. The instantiation over $State^{df}$ of the operations of Figure 3 and of $\cup$.

Definition 6. *For $\tau \in Typing$, the abstract domain is $State_\tau^{df} = \{\emptyset\} \cup Frame_\tau^{df}$.*

The relation $\approx$ says when an abstract value (i.e., a set of types) approximates a concrete value. The set $\{int\}$ approximates the integers, the set $\{bool\}$ approximates the booleans and a set of classes S approximates *nil* and all locations containing an object of a class in S.

Definition 7. *Let $\mu \in Memory$. We define $\approx_\mu : Value^{df} \times Value$ as the minimal relation such that $\{int\} \approx_\mu i$, with $i \in Int$, $\{bool\} \approx_\mu b$, with $b \in Bool$, $S \approx_\mu nil$, with $S \neq \{int\}$ and $S \neq \{bool\}$, and $S \approx_\mu l$ with $l \in Loc$ and $\mu(l).\kappa \in S$. This relation is pointwise extended to $\approx_\mu : Frame_\tau^{df} \times Frame_\tau$, for $\tau \in Typing$.*

The concretisation of an abstract state ϕ is the set of concrete states $\langle \phi', \mu \rangle$ whose frame ϕ' binds every variable $v \in \mathsf{dom}(\phi)$ to a concrete value $\phi'(v)$ approximated ($\approx_\mu$) by the abstract value $\phi(v)$, i.e., $\phi(v) \approx_\mu \phi'(v)$.

Definition 8. *For $\tau \in Typing$, we define the concretisation map $\gamma^{df} : State_\tau^{df} \mapsto \wp(State_\tau)$ such that $\gamma^{df}(\emptyset) = \emptyset$ and $\gamma^{df}(\phi) = \{\langle \phi', \mu \rangle \in State_\tau \mid \phi \approx_\mu \phi'\}$ for every $\phi \in State_\tau^{df} \setminus \{\emptyset\}$.*

Proposition 3. *For $\tau \in Typing$, the set $\gamma^{df}(State_\tau^{df})$ is a Moore family of $\wp(State_\tau)$. Hence $State_\tau^{df}$ is an abstract domain whose induced abstraction map is $\alpha^{df} : \wp(State_\tau) \mapsto State_\tau^{df}$ such that $\alpha^{df}(\emptyset) = \emptyset$ and, for $S \neq \emptyset$, $\alpha^{df}(S) = a$ such that for $v \in \mathsf{dom}(\tau)$, $a(v) = \{int\}$ if $\tau(v) = int$, $a(v) = \{bool\}$ if $\tau(v) = bool$ and $a(v) = \{\mu(\phi(v)).\kappa \mid \langle \phi, \mu \rangle \in S$ and $\phi(v) \in Loc\}$ if $\tau(v) \in Class\_ref$. In this last case, α^{df} collects the classes of the objects bound to v in some concrete state.*

Proposition 4. *The optimal abstract counterparts of the powerset extension of the operations of Fig. 3 and of* $\cup$ *are those given in Fig. 5. All of them, except* $\cup_\tau^{df}$*, are strict on all their arguments. For instance,* $(\mathsf{unscope}_\tau^{mr})^{df}(\emptyset)(\phi_2) = (\mathsf{unscope}_\tau^{mr})^{df}(\phi_1)(\emptyset) = (\mathsf{unscope}_\tau^{mr})^{df}(\emptyset)(\emptyset) = \emptyset$*. For* $\cup_\tau^{df}$*, we define* $\cup_\tau^{df}(\emptyset)(\phi_2) = \phi_2$*,* $\cup_\tau^{df}(\phi_1)(\emptyset) = \phi_1$ *and* $\cup_\tau^{df}(\emptyset)(\emptyset) = \emptyset$*.*

Example 4. We execute over $State_\tau^{df}$ the same sequence of instructions of Ex. 2. The initial state is the abstraction (Prop. 3) of the initial concrete state.

Operation	State
	$[v_1 \mapsto \{\kappa_a\}]$ (initial state)
$(\mathsf{expand}^{v_2:\kappa_a}_{[v_1 \mapsto \kappa_a]})^{df}$	$[v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset]$
$(\mathsf{new}^{\kappa_b}_{[v_1,v_2 \mapsto \kappa_a]})^{df}$	$[v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset, res \mapsto \{\kappa_b\}]$
$(\mathsf{get\_field}^{\mathtt{n}}_{[v_1,v_2 \mapsto \kappa_a, res \mapsto \kappa_b]})^{df}$	$[v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset, res \mapsto \{\kappa_a, \kappa_b\}]$
$(\mathsf{put\_var}^{v_2}_{[v_1,v_2,res \mapsto \kappa_a]})^{df}$	$[v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \{\kappa_a, \kappa_b\}]$

The final abstract state says that v_1 can be bound to objects of class a and v_2 to objects of class a or b, which is a correct approximation of the concrete final state of Example 2. It is strictly more precise than that obtained in Example 3.

6 0-CFA (*ps*-Analysis)

In [11], Palsberg and Schwartzbach defined a class analysis as a constraint problem. Every variable and every field is given a set of classes. These sets are then related by constraints which model the dataflow of the program. This means that the information about program variables is that used in the case of the dataflow analysis of Section 5, but this analysis deals with fields too. However, all objects of the same class are identified. The analysis of a method is the same for every call point and hence this analysis is a 0-*CFA analysis* [13].

That analysis leads to an abstract domain made of an abstract frame, identical to those used in Section 5, and of an abstract memory. The abstract memory is an abstract frame for the fields of the classes. We can assume without any loss of generality that fields in different classes have different names.

The difference with the original technique in [11] is that abstract interpretation does not *a priori* identify different occurrences of the same variable, while in [11] the same variable has assigned the same set of classes in every program point. This, together with the formally proved optimality of our abstract operations (Proposition 6), suggests that our analysis should be more precise than that in [11], without using any technique like multiple variables for different uses of the same variable or method splitting [12].

Definition 9. *We define the typing* $\overline{\tau} = \cup_{\kappa \in Class\_ref} d(\kappa).\tau$ *of the fields of all classes. It makes sense since fields have different names. For* $\tau \in Typing$ *such that* $\mathsf{dom}(\tau) \subseteq \mathsf{dom}(\overline{\tau})$ *and* $\phi \in Frame_\tau$*, we define* $\overline{\phi} \in Frame_{\overline{\tau}}$ *as* $\overline{\phi}(v) = \phi(v)$ *if* $v \in \mathsf{dom}(\tau)$*, and* $\overline{\phi}(v) = \mathsf{init}(\overline{\tau}(v))$ *otherwise (*init *has been defined in Fig. 3).*

$$\mathsf{nop}^{ps}_{\tau}(\langle\phi,\mu\rangle) = \langle\phi,\mu\rangle \qquad (\mathsf{get\_int}^{i}_{\tau})^{ps}(\langle\phi,\mu\rangle) = \langle\phi[\{int\}/res],\mu\rangle$$

$$(\mathsf{get\_nil}^{cr}_{\tau})^{ps}(\langle\phi,\mu\rangle) = \langle\phi[\emptyset/res],\mu\rangle \qquad (\mathsf{get\_bool}^{b}_{\tau})^{ps}(\langle\phi,\mu\rangle) = \langle\phi[\{bool\}/res],\mu\rangle$$

$$(\mathsf{get\_var}^{v}_{\tau})^{ps}(\langle\phi,\mu\rangle) = \langle\phi[\phi(v)/res],\mu\rangle \qquad (\mathsf{put\_var}^{v}_{\tau})^{ps}(\langle\phi,\mu\rangle) = \langle\phi[\phi(res)/v]|_{-res},\mu\rangle$$

$$(\mathsf{get\_field}^{f}_{\tau})^{ps}(\langle\phi,\mu\rangle) = \begin{cases} \langle\emptyset,\mu\rangle & \text{if } \phi(res)=\emptyset \\ \langle\phi[\mu(f)/res],\mu\rangle & \text{otherwise} \end{cases}$$

$$(\mathsf{put\_field}^{f}_{\tau,\tau'})^{ps}(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle) = \begin{cases} \langle\emptyset,\mu_2\rangle \\ \quad \text{if } \phi_1(res)=\emptyset \\ \langle\phi_2|_{-res},\mu_2[\mu_2(f)\cup\phi_2(res)/f]\rangle \\ \quad \text{otherwise} \end{cases}$$

$$=^{ps}_{\tau}(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle) = \langle\phi_2[\{bool\}/res],\mu_2\rangle$$

$$(\mathsf{scope}^{mr,v_1,\dots,v_n}_{\tau})^{ps}(\langle\phi,\mu\rangle) = \langle[\iota_1 \mapsto \phi(v_1),\dots,\iota_n \mapsto \phi(v_n),\mathtt{this} \mapsto \phi(res)],\mu\rangle \quad \text{where } \langle\iota_1,\dots,\iota_n\rangle = p(mr).s \setminus \{n(mr),\mathtt{this}\}$$

$$(\mathsf{unscope}^{mr}_{\tau})^{ps}(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle) = \langle\phi_1[\phi_2(n(mr))/res],\mu_2\rangle$$

$$(\mathsf{restrict}^{vs}_{\tau})^{ps}(\langle\phi,\mu\rangle) = \langle\phi|_{-vs},\mu\rangle \qquad (\mathsf{expand}^{v:t}_{\tau})^{ps}(\langle\phi,\mu\rangle) = \langle\phi[\mathsf{init}^{df}(t)/v],\mu\rangle$$

$$(\mathsf{lookup}^{id,mr}_{\tau})^{ps}(\langle\phi,\mu\rangle) = \begin{cases} \langle\phi[S/res],\mu\rangle \\ \quad \text{if } S=\{cr\in\phi(res)\mid d(cr).\nu(id)=mr\}\neq\emptyset \\ \langle\emptyset,\mu\rangle \\ \quad \text{otherwise} \end{cases}$$

$$(\mathsf{new}^{cr}_{\tau})^{ps}(\langle\phi,\mu\rangle) = \langle\phi[\{cr\}/res],\mu\rangle \qquad \mathsf{is\_true}^{ps}_{\tau}(\langle\phi,\mu\rangle) = \mathsf{is\_false}^{ps}_{\tau}(\langle\phi,\mu\rangle) = \langle\phi,\mu\rangle$$

$$\cup^{ps}_{\tau}(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle) = \langle\cup^{df}_{\tau}(\phi_1)(\phi_2),\cup^{df}_{\overline{\tau}}(\mu_1)(\mu_2)\rangle\ ,$$

where init^{df} and $\cup^{df}$ are defined in Figure 5.

Fig. 6. The instantiation over $State^{ps}$ of the operations of Fig. 3 and of $\cup$.

Definition 10. *For $\tau \in Typing$, the abstract domain is $State^{ps}_{\tau} = (Frame^{ps}_{\tau} \times Memory^{ps})$, with $Frame^{ps}_{\tau}$=$State^{df}_{\tau}$ and $Memory^{ps}$=$Frame^{df}_{\overline{\tau}}$ (Defs. 5 and 6).*

Compare the following definition with Definition 8. The relation $\approx$ has been defined in Definition 7. An element $\langle\phi,\mu\rangle \in State^{ps}_{\tau}$ represents those concrete states whose frame is compatible with ϕ, i.e., their class variables are bound to objects of a class allowed by ϕ, and whose memory contains objects with an internal frame compatible with μ, i.e., their fields are bound to objects of a class allowed by μ. Since the frame of an object is for its instance variables only, we extend it to the whole set of fields before its comparison with μ.

Definition 11. *Given $\tau \in Typing$, we define the concretisation map $\gamma^{ps} : State^{ps}_{\tau} \mapsto \wp(State_{\tau})$ as*

$$\gamma^{ps}(\langle\phi,\mu\rangle) = \gamma^{df}(\phi) \cap \left\{ \langle\phi',\mu'\rangle \in State_{\tau} \;\middle|\; \begin{array}{l} \textit{for every } \langle\kappa,\phi''\rangle \in \mathsf{codom}(\mu') \\ \textit{we have } \mu \approx_{\mu'} \overline{\phi''} \end{array} \right\} .$$

The following abstraction map formalises the idea of the analysis. A set of concrete states S is abstracted through α^{df}, as in Section 5, but a second component provides more information about the fields. Namely, it is the abstraction through α^{df} of the states of the objects in memory.

Proposition 5. *For $\tau \in Typing$, the set $\gamma^{ps}(State^{ps}_\tau)$ is a Moore family of $\wp(State_\tau)$. Hence $State^{ps}_\tau$ is an abstract domain whose induced abstraction map is $\alpha^{ps} : \wp(State_\tau) \mapsto State^{ps}_\tau$ such that ($\mathsf{init}(\overline{\tau})$ avoids empty abstract memories)*

$$\alpha^{ps}(S) = \langle \alpha^{df}(S), \alpha^{df}(\{\mathsf{init}(\overline{\tau})\} \cup \{\langle \overline{\phi'}, \mu \rangle \mid \langle \phi, \mu \rangle \in S \text{ and } \langle \kappa, \phi' \rangle \in \mathsf{codom}(\mu)\}) \rangle.$$

Proposition 6. *The optimal abstract counterparts of the powerset extension of the operations of Fig. 3 and of $\cup$ are given in Fig. 6 for the case when the frame of their arguments is not $\emptyset$. Otherwise they yield $\langle \emptyset, \mu \rangle$ for an arbitrary $\mu \in Memory^{ps}$, except $\cup^{ps}$ which is such that $\cup^{ps}_\tau(\langle \emptyset, \mu \rangle)(e) = \cup^{ps}_\tau(e)(\langle \emptyset, \mu \rangle) = e$.*

In Figure 6, the frame component of the abstract states behaves like in Section 5, except in get_field. The memory component does not change if the memory of the concrete operations (Figure 3) does not change. The operations get_field and put_field allow a flow of information between abstract frames and abstract memories. Namely, get_field loads in the abstract frame the set of classes $\mu(f)$ of the objects stored in the field f. Conversely, put_field adds the classes contained in res, i.e., $\phi_2(res)$, to the set of classes already stored in the field, i.e., $\mu_2(f)$. In this way, we accumulate all classes stored in the field during the execution.

Example 5. We execute over $State^{ps}_\tau$ the same sequence of instructions of Ex. 2. The initial state is the abstraction (Prop. 5) of the initial concrete state.

Operation	State
	$\langle [v_1 \mapsto \{\kappa_a\}], [\mathtt{n} \mapsto \emptyset] \rangle$ (initial state)
$(\mathsf{expand}^{v_2:\kappa_a}_{[v_1 \mapsto \kappa_a]})^{ps}$	$\langle [v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset], [\mathtt{n} \mapsto \emptyset] \rangle$
$(\mathsf{new}^{\kappa_b}_{[v_1, v_2 \mapsto \kappa_a]})^{ps}$	$\langle [v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset res \mapsto \{\kappa_b\}], [\mathtt{n} \mapsto \emptyset] \rangle$
$(\mathsf{get\_field}^{\mathtt{n}}_{[v_1, v_2 \mapsto \kappa_a, res \mapsto \kappa_b]})^{ps}$	$\langle [v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset res \mapsto \emptyset], [\mathtt{n} \mapsto \emptyset] \rangle$
$(\mathsf{put\_var}^{v_2}_{[v_1, v_2, res \mapsto \kappa_a]})^{ps}$	$\langle [v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset], [\mathtt{n} \mapsto \emptyset] \rangle$

The final state says that v_1 can be bound to objects of class a, and that v_2 and the field n are bound to nil, the exact approximation of the final state of Ex. 2.

7 Comparison

We compare here the precision of the domains defined in the previous sections.

Proposition 7. *For $\tau \in Typing$, we have $\gamma^{df}(State^{df}_\tau) \subseteq \gamma^{ps}(State^{ps}_\tau)$ and the ps-analysis computes a more precise information than the df-analysis.*

Proof. Since $\gamma^{df}(\emptyset) = \emptyset = \gamma^{ps}(\langle \emptyset, \mu \rangle) \in \gamma^{ps}(State^{ps}_\tau)$ for $\mu \in Memory^{ps}$, it is enough to show that $\gamma^{df}(Frame^{df}_\tau) \subseteq \gamma^{ps}(State^{ps}_\tau)$. This will entail the desired result, since the abstract operations (Figs. 5 and 6) are optimal (§8 of [9]). The idea is that every $\phi \in Frame^{df}_\tau$ is the frame of a pair $\langle \phi, \mu \rangle \in State^{ps}_\tau$ where μ does not introduce any restriction. Let $\mu(v) = \{int\}$ if $\overline{\tau}(v) = int$, $\mu(v) = \{bool\}$ if $\overline{\tau}(v) = bool$ and $\mu(v) = \downarrow(\overline{\tau}(v))$ if $\overline{\tau}(v) \in Class\_ref$ for $v \in \mathsf{dom}(\overline{\tau})$. Then $\langle \phi', \mu' \rangle \in \gamma^{df}(\phi)$ iff $\phi \approx_{\mu'} \phi'$ iff ($\phi \approx_{\mu'} \phi'$ and for every $\langle \kappa, \phi'' \rangle \in \mathsf{codom}(\mu')$ we have $\mu \approx_{\mu'} \overline{\phi''}$), iff $\langle \phi', \mu' \rangle \in \gamma^{ps}(\langle \phi, \mu \rangle)$, i.e., $\gamma^{df}(\phi) = \gamma^{ps}(\langle \phi, \mu \rangle)$.

The *rta*- and the *df*-analysis are incomparable. In the final states of Exs. 3 and 4 the *df*- is more precise than the *rta*-analysis. But consider the initial states of those examples, i.e., the respective abstractions of the initial concrete state of Ex. 2. In Ex. 3 the possible classes for the field `n` are $\{\kappa_a\}$, while in Ex. 4 they are $\{\kappa_a, \kappa_b\}$, because that abstraction does not provide any information about the fields. Hence the *df*-analysis provides a coarser approximation of the set of classes for the field `n` than the *rta*-analysis.

We strongly believe that the *ps*-analysis is more precise than the *rta*-analysis. Indeed, the *ps*-analysis distinguishes between the classes stored in different variables, while the *rta*-analysis merges all classes in just one set. For instance, in the context of Ex. 1, the concrete states $\sigma_1 = \langle [v_1 \mapsto l, v_2 \mapsto nil], [l \mapsto \langle \kappa_a, [\mathtt{n} \mapsto nil] \rangle] \rangle$ and $\sigma_2 = \langle [v_1 \mapsto nil, v_2 \mapsto l], [l \mapsto \langle \kappa_a, [\mathtt{n} \mapsto nil] \rangle] \rangle$ are such that $\alpha^{rta}(\{\sigma_1\}) = \alpha^{rta}(\{\sigma_2\}) = \{\kappa_a\}$ while $\alpha^{ps}(\{\sigma_1\}) = \langle [v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset], [\mathtt{n} \mapsto \emptyset] \rangle$ and $\alpha^{ps}(\{\sigma_2\}) = \langle [v_1 \mapsto \emptyset\, v_2 \mapsto \{\kappa_a\}], [\mathtt{n} \mapsto \emptyset] \rangle$. In this case, the *ps*-analysis distinguishes σ_1 and σ_2 which do bind the variables to objects of different classes, while the *rta*-analysis does not. When trying to generalise this result into a formal proof, we are faced with the problem that the *rta*-analysis considers all objects in memory, while the *ps*-analysis only those reachable from the current frame or that of another object (Def. 11). For instance, in the context of Ex. 1, the concrete states $\sigma_3 = \langle [v \mapsto nil], [l \mapsto \langle \kappa_a, [\mathtt{n} \mapsto nil] \rangle] \rangle$ and $\sigma_4 = \langle [v \mapsto nil], [] \rangle$ are such that $\alpha^{ps}(\{\sigma_3\}) = \alpha^{ps}(\{\sigma_4\}) = \langle [v \mapsto \emptyset], [\mathtt{n} \mapsto \emptyset] \rangle$, while $\alpha^{rta}(\{\sigma_3\}) = \{\kappa_a\}$ and $\alpha^{rta}(\{\sigma_4\}) = \emptyset$. However, this ability to distinguish such states, which contain *the same* class information, is of no use for the class analysis. Indeed, objects in memory can affect an expression of the program only if they are reachable from some variable. A formal proof of this statement can be obtained, e.g., by quotienting [5] both domains w.r.t. class information.

8 Conclusions

Class analyses for object-oriented languages are many and varied. This paper shows that abstract interpretation leads to a formal framework for the development and comparison of such analyses. To demonstrate this, we have put three traditional techniques for class analysis inside that framework. This provides a systematic construction of the abstract operations of a particular analysis. Furthermore, it allows a formal comparison of the relative precision of the analyses. Due to space limitations, it has not been possible to show the details of how the set of operations can be used to give semantics to *e.g.*, Java or Java bytecode.

Several extensions to the present work are to be considered: There are more class analyses than the ones considered in this paper and the classification that we have initiated here should be carried further. Analyses can be combined so as to use the combination of two abstract domains in an analysis. Formally, this is characterised by the *reduced product* of two abstractions that defines a semi-lattice structure on the set of abstractions. An interesting problem is to construct a concrete representation of the reduced product of, say, the *rta*- and the *df*-analysis.

Since we have algorithms for the abstraction of a finite set of states (Props. 1, 3 and 5), and for the abstract operations (Props. 2, 4 and 6) and since our domains are finite (for a given $\tau \in Typing$), the framework provides a way of implementing the static class analyses. The practical aspects of such an implementation remain to be investigated.

References

1. O. Agesen. Constraint-Based Type Inference and Parametric Polymorphism. In B. Le Charlier, editor, *Proc. of the 1st Int. Static Analysis Symp.*, volume 864 of *Lecture Notes in Computer Science*, pages 78–100. Springer-Verlag, 1994.
2. D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proc. of OOPSLA'96*, volume 31(10) of *ACM SIGPLAN Notices*, pages 324–341, New York, 1996. ACM Press.
3. P. Bertelsen. Semantics of Java Byte Code. Technical report, Department of Information Technology, Technical University of Denmark, March 1997.
4. E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In L. Brim, J. Grunska, and J. Zlatusla, editors, *23rd Int. Symp. on Mathematical Foundations of Computer Science*. Springer LNCS vol. 1450, 1998.
5. A. Cortesi, G. Filé, and W. Winsborough. The Quotient of an Abstract Interpretation. *Theoretical Computer Science*, 202(1-2):163–192, 1998.
6. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. In S. Brookes and M. Mislove, editors, *13th Conf. on Math. Found. of Programming Semantics*, volume 6 of *Electronic Notes on Theoretical Computer Science*, Pittsburgh, PA, USA, March 1997. Elsevier Science Publishers. Available at `http://www.elsevier.nl/locate/entcs/volume6.html`.
7. P. Cousot. Types as Abstract Interpretations. In *24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 316–331. ACM Press, 1997.
8. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.
9. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *6th ACM Symp. on Principles of Programming Languages*, pages 269–282, 1979.
10. A. Diwan, J. E. B. Moss, and K. S. McKinley. Simple and Effective Analysis of Statically Typed Object-Oriented Programs. In *Proc. of OOPSLA'96*, volume 31(10) of *ACM SIGPLAN Notices*, pages 292–305, New York, 1996. ACM Press.
11. J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proc. of OOPSLA'91*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161. ACM Press, November 1991.
12. J. Plevyak and A. A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *Proc. of OOPSLA'94*, volume 29(10) of *ACM SIGPLAN Notices*, pages 324–340. ACM Press, October 1994.
13. O. Shivers. Control-Flow Analysis in Scheme. In *Proc. of the 1988 Conf. on Programming Languages Design and Implementation*, volume 23(7) of *ACM SIGPLAN Notices*, pages 164–174. ACM Press, July 1988.

On the Complexity of Parity Word Automata

Valerie King[1⋆], Orna Kupferman[2], and Moshe Y. Vardi[3⋆⋆]

[1] University of Victoria
Department of Computer Science, P.O. Box 3055, Victoria, BC, Canada V8W 3P6
Email: val@csr.csc.uvic.ca, URL: http://www.csr.uvic.ca/~val
[2] Hebrew University
School of Computer Science and Engineering, Jerusalem 91904, Israel
Email: orna@cs.huji.ac.il, URL: http://www.cs.huji.ac.il/~orna
[3] Rice University
Department of Computer Science, Houston, TX 77251-1892, U.S.A.
Email: vardi@cs.rice.edu, URL: http://www.cs.rice.edu/~vardi

Abstract. Different types of nondeterministic automata on infinite words differ in their succinctness and in the complexity for their nonemptiness problem. A simple translation of a parity automaton to an equivalent Büchi automaton is quadratic: a parity automaton with n states, m transitions, and index k may result in a Büchi automaton of size $O((n+m)k)$. The best known algorithm for the nonemptiness problem of parity automata goes through Büchi automata, leading to a complexity of $O((n+m)k)$. In this paper we show that while the translation of parity automata to Büchi automata cannot be improved, the special structure of the acceptance condition of parity automata can be used in order to solve the nonemptiness problem directly, with a dynamic graph algorithm of complexity $O((n+m)\log k)$.

1 Introduction

Today's rapid development of complex and safety-critical systems requires reliable verification methods. The automata-theoretic approach to system verification uses the theory of automata on infinite objects [Büc62,McN66,Rab69] as a unifying paradigm for the specification, verification, and synthesis of nonterminating systems. By translating specifications to automata, we reduce questions about systems and their specifications to questions about automata. More specifically, questions such as satisfiability of specifications and correctness of systems with respect to their specifications are reduced to questions such as nonemptiness and language containment [VW86,Kur94,VW94]. The automata-theoretic approach separates the logical and the algorithmic aspects of reasoning about systems. The translation of specifications to automata handles the logic and shifts all the algorithmic difficulties to automata-theoretic problems.

⋆ This work was done while the author was visiting the Hebrew University.
⋆⋆ Supported in part by NSF grants CCR-9700061 and CCR-9988322, and by a grant from the Intel Corporation.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 276–286, 2001.

Like automata on finite words, automata on infinite words either accept or reject an input word. Since a run on an infinite word does not have a final state, acceptance is determined with respect to the set of states visited infinitely often during the run. For example, in *Büchi* automata [Büc62], some of the states are designated as accepting states, and a run is accepting if it visits states from the accepting set infinitely often. In *parity* automata [Mos84,EJ91], the acceptance condition is a partition $\{F_1, F_2, \ldots, F_{2k}\}$ of the state space, and a run is accepting if the minimal index i for which the set F_i is visited infinitely often is even. In *Rabin* automata [Rab69], the acceptance condition is a set $\{\langle G_1, B_1\rangle, \ldots, \langle G_k, B_k\rangle\}$ of pairs of sets of states, and a run is accepting if there is a pair $\langle G_i, B_i\rangle$ for which the set G_i is visited visited infinitely often and the set B_i is visited only finitely often. The number k appearing in the acceptance condition of parity and Rabin automata is called the *index* of the automaton.

The type of the automaton influences its succinctness. For example, a simple transformation of a parity or a Rabin nondeterministic automaton with n states, m transitions, and index k to an equivalent nondeterministic Büchi automaton composes k copies of the automaton, and thus results in an automaton of size $O((n + m)k)$. The type of the automaton also influences the difficulty of answering questions about it. In particular, while the nonemptiness problem for nondeterministic Büchi automata can be easily reduced to the reachability problem (the automaton is nonempty iff there is an accepting state reachable from an initial state and from itself), and can therefore be solved in linear time or NLOGSPACE, there is no straightforward linear-time solution to the nonemptiness problem for parity and Rabin nondeterministic automata. An algorithm that first translates the parity or Rabin automaton to an equivalent Büchi automaton has time complexity $O((n + m)k)$, which is also the best known complexity for the problem. In particular, when $k = n$, the above implies that while the nonemptiness problem for Büchi word automata can be solved in linear time, the best known algorithm for parity automata is quadratic.

Parity automata are particularly useful in the context of verification. This follows from the fact that the parity acceptance condition can naturally recognize languages given by fixed-point expressions [Koz83,EJ91], and many properties of systems are naturally specified by means of fixed points. This is especially relevant for the branching case, where alternating parity tree automata are exactly as expressive as the μ-calculus [EJ91,JW95]. The parity acceptance condition can be viewed as a special case of the Rabin acceptance condition. Indeed, a parity condition $\{F_1, F_2, \ldots, F_{2k}\}$ is equivalent to a Rabin condition $\{\langle F_2, F_1\rangle, \langle F_4, F_1 \cup F_2 \cup F_3\rangle, \ldots, \langle F_{2k}, F_1 \cup \ldots \cup F_{2k-1}\rangle\}$. Some algorithms for parity automata use this equivalence and do not try to make use of the fact that the parity condition has much more structure in it. The question whether we can somehow exploit this structure is important and interesting, and is the key to some fundamental questions in automata and verification. In particular, while the nonemptiness problem for nondeterministic Rabin tree automata is

NP-complete [EJ88], we know that for parity automata the problem is in UP ∩ co-UP [Jur98][1], and its precise complexity is an open problem.

It is shown in [SN99] that the blow up in the translation of parity automata to Büchi automata cannot be avoided. For that, Seidl and Niwiński", show a family $\mathcal{L}_n$ of languages such that $\mathcal{L}_n$ can be recognized by a parity automaton with n states, $2n + 1$ transitions, and index n, yet the smallest Büchi automaton for $\mathcal{L}_n$ has $O(n^2)$ states and transitions. It follows that nonemptiness algorithms for parity automata that use a translation to Büchi automata cannot be improved.

In this paper we show that while the special structure of parity automata does not prevent them from defining rich languages succinctly, it is helpful when the nonemptiness problem is considered. We present an algorithm that circumvent the translation to Büchi automata: given a parity automaton with n states, m transitions, and index k, our algorithm solves the nonemptiness problem in time complexity $O((n + m) \log k)$. This improves the $O((n + m)k)$ known algorithm.

As detailed in Section 3, our algorithm is a variation of an algorithm for hierarchical clustering [Tar82], and is heavily based on the special structure of the acceptance condition of parity automata. In particular, it cannot be extended to Rabin automata. The algorithm handles a known-in-advance sequence of insertions of edges, thus it is an *off-line partially dynamic* algorithm.

While parity word automata do not have the same practical importance as parity tree automata, we believe that our results are interesting, from both a theoretical and practical points of view: this is not the first time that dynamic graph algorithms are used to improve the complexity of a nonemptiness problem of automata on infinite words. In [HT96], Henzinger and Telle developed an algorithm, called *lock-step search*, for the maintenance of the strongly connected components of a directed graph under edge deletion, and use the algorithm in order to improve the complexity of the nonemptiness problem for Streett automata. A dynamic algorithm was recently used in [BGS00] in order to find and analyze the strongly connected components of a graph of size n with $O(n \log n)$ symbolic steps, improving the quadratic complexity of this highly practical problem. So, we believe that dynamic graph algorithms can be used further to improve the complexity of problems from automata theory and verification. Creating a class of problems where dynamic graph algorithms have proven to be useful has clear practical importance. In particular, the same ideas used in our algorithm may be useful in the branching case.

2 Preliminaries

Given an alphabet Σ, an *infinite word over* Σ is an infinite sequence $w = \sigma_0 \cdot \sigma_1 \cdot \sigma_2 \cdots$ of letters in Σ. An *automaton over infinite words* (word automaton, for short) is $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$, where Σ is the input alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $Q_0 \subseteq Q$ is a set of initial states, and α is an acceptance condition (a condition that defines a subset of Q^ω; we define several acceptance conditions below). Intuitively, $\delta(q, \sigma)$ is the set of

[1] The class UP is a subset of NP, where each word accepted by the Turing machine has a unique accepting run.

states that $\mathcal{A}$ can move into when it is in state q and it reads the letter σ. Since $\mathcal{A}$ may have several initial states and since the transition function may specify many possible transitions for each state and letter, $\mathcal{A}$ may be *nondeterministic*.

Given an input infinite word $w = \sigma_0 \cdot \sigma_1 \cdot \sigma_2 \cdots \in \Sigma^\omega$, a *run* of $\mathcal{A}$ on w can be viewed as a function $r : \mathbb{N} \to Q$ where $r(0) \in Q_0$ (i.e., the run starts in one of the initial states) and for every $i \geq 0$, we have $r(i+1) \in \delta(r(i), \sigma_i)$ (i.e., the run obeys the transition function). Each run r induces a set $inf(r)$ of states that r visits *infinitely often*. Formally,

$$inf(r) = \{q \in Q \ : \ \text{for infinitely many } i \in \mathbb{N}, \ \text{we have } r(i) = q\}.$$

As Q is finite, it is guaranteed that $inf(r) \neq \emptyset$. The run r *accepts* w iff it satisfies the acceptance condition α. We consider here three acceptance conditions.

- A run r satisfies a *Büchi* acceptance condition $\alpha \subseteq Q$ if and only if $inf(r) \cap \alpha \neq \emptyset$.
- A run r satisfies a *parity* acceptance condition $\alpha = \{F_1, F_2, \ldots, F_{2k}\}$, where the F_i's are a partition of Q, iff the minimal index i for which $inf(r) \cap F_i \neq \emptyset$ is even. For $1 \leq i \leq 2k$, we use the notation $\tilde{F}_i = F_1 \cup F_2 \cup \ldots F_i$. Note that $\tilde{F}_1 \subseteq \tilde{F}_2 \subseteq \cdots \subseteq \tilde{F}_{2k} = Q$.
- A run r satisfies a *Rabin* acceptance condition $\alpha = \{\langle G_1, B_1\rangle, \ldots, \langle G_k, B_k\rangle\}$, where for $1 \leq i \leq k$, $G_i \subseteq Q$ and $B_i \subseteq Q$, if and only if there exists a pair $\langle G_i, B_i\rangle \in \alpha$ for which $inf(r) \cap G_i \neq \emptyset$ and $inf(r) \cap B_i = \emptyset$.

The number k appearing in a parity or Rabin condition is called the *index* of the automaton. Note that the acceptance condition of an (either a parity or Rabin) automaton with index k involves $2k$ sets. An automaton $\mathcal{A}$ accepts an input word w iff there exists a run r of $\mathcal{A}$ on w such that r accepts w. The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of infinite words that $\mathcal{A}$ accepts. Thus, each word automaton defines a subset of Σ^ω. We say that an automaton $\mathcal{A}$ is *nonempty* iff $\mathcal{L}(\mathcal{A}) \neq \emptyset$. For simplicity, we assume that all the states in the automaton are *reachable*; that is, for every state q, there is a word w and a run r of $\mathcal{A}$ on w such that r visits q. We also assume that no state is *redundant*; that is, for every state q, there is a word w and a run r of $\mathcal{A}$ on w such that r accepts w in a run that visits q. Note that we can omit states that are redundant or not reachable and get an equivalent smaller automaton.

A *labeled directed graph* $G = \langle D, V, E, l\rangle$ consists of a domain D, a set V of vertices, a set $E \subseteq V \times V$ of edges, and a labeling function $l : V \to D$ that maps each vertex to a value in D. We assume that $D = \{1, \ldots, n\}$ for some $n \geq 1$. A *path* in G is a sequence $v_1, v_2, \ldots, v_n$ of vertices such that $n \geq 1$ and $\langle v_i, v_{i+1}\rangle \in E$ for all $i \geq 1$. We say that vertex v is reachable from vertex v' iff there is a path $v_1, v_2, \ldots, v_n$ in G with $v' = v_1$ and $v = v_n$. A *cycle* is a path $v_1, v_2, \ldots, v_n$ with $n \geq 2$ and $v_1 = v_n$. A *strongly connected component* (SCC) of G is a set of states $C \subseteq V$ such that for all v and v' in C, the vertex v is reachable from v'. An SCC C is *maximal* (MSCC) if for all vertices $v \not\in C$, the set $C \cup \{v\}$ is no longer a SCC. An SCC is *nontrivial* if it contains a cycle. Note that a single vertex with a self loop is a nontrivial SCC. Each graph G has a unique partition into MSCC. By [Tar72], this partition can be found in time $O(|V| + |E|)$.

With each automaton $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$, we can associate a graph $G_{\mathcal{A}} = \langle D_\alpha, Q, E_\delta, l_\alpha \rangle$, where for $q, q' \in Q$, we have $E_\delta(q, q')$ if there is $\sigma \in \Sigma$ such that $q' \in \delta(q, \sigma)$. The labels of the graph encodes the acceptance condition α. For example, if $\mathcal{A}$ is a Büchi automaton, we can have $D_\alpha = \{1, 2\}$ and $l_\alpha(q) = 1$ iff $q \in \alpha$. Similarly, if $\mathcal{A}$ is a parity automaton with index k, we can have $D_\alpha = \{1, \ldots, 2k\}$ and $l_\alpha(q) = i$ for the i such that $q \in F_i$. When we refer to the number of states and edges of an automaton $\mathcal{A}$, we mean $|Q|$ and $|E_\delta|$.

Theorem 1. [CES86] *The nonemptiness problem for Büchi automata can be solved in linear time.*

Proof: It is easy to see that a Büchi automaton $\mathcal{A}$ is nonempty iff there is a state $q \in \alpha$ that is reachable in $\mathcal{A}$ from Q_0 and from itself. First, such a state q witnesses an accepting run of $\mathcal{A}$ that first leads from Q_0 to q and then visits q infinitely often. Also, if some word is accepted by $\mathcal{A}$, then there must be a run of $\mathcal{A}$ that visits infinitely often some state in α that is reachable from Q_0. Being visited more than once, this state is also reachable from itself. We can test for the existence of q as above by partitioning the graph $G_{\mathcal{A}}$ into MSCC's and looking for a component that is reachable from Q_0 and whose intersection with α is not empty. □

Note that by guessing a state q as in the proof of Theorem 1 and performing two reachability tests, the nonemptiness problem for Büchi automata can also be solved in NLOGSPACE, and is in fact NLOGSPACE-complete [VW94]. In this paper, however, we study the time complexity of the nonemptiness problem.

It is easy to see that a Büchi acceptance condition α is equivalent to a parity condition $\{\emptyset, \alpha\}$ and that a parity condition $\{F_1, F_2, \ldots, F_{2k}\}$ is equivalent to the Rabin condition $\{\langle F_2, \tilde{F}_1 \rangle, \ldots, \langle F_{2k}, \tilde{F}_{2k-1} \rangle\}$. It follows that one can easily translate a given Büchi automaton to an equivalent parity automaton, and can translate a given parity automaton to an equivalent Rabin automaton. It turns out that for nondeterministic automata, translations are possible also in the other directions. We describe here the translations of Rabin and parity nondeterministic automata to Büchi nondeterministic automata.

Theorem 2. [Cho74] *Given a Rabin automaton $\mathcal{A}$ with n states, m edges, and index k, there is a Büchi automaton $\mathcal{A}'$ with $O(nk)$ states and $O(mk)$ edges such that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$.*

Proof: Let $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$ with $\alpha = \{\langle G_1, B_1 \rangle, \ldots, \langle G_k, B_k \rangle\}$. For every $1 \leq i \leq k$, let $Q_i = (Q \setminus B_i) \times \{i\}$. We define the Büchi automaton $\mathcal{A}' = \langle \Sigma, Q', \delta', Q_0, \alpha' \rangle$, where

- $Q' = Q \cup \bigcup_{1 \leq i \leq k} Q_i$.
- For every $q \in Q$ and $\sigma \in \Sigma$, we have
  - $\delta'(q, \sigma) = \delta(q, \sigma) \cup \bigcup_{1 \leq i \leq k} ((\delta(q, \sigma) \setminus B_i) \times \{i\})$.
  - For every $1 \leq i \leq k$, we have $\delta'(\langle q, i \rangle, \sigma) = (\delta(q, \sigma) \setminus B_i) \times \{i\}$.
- $\alpha' = \bigcup_{1 \leq i \leq k} G_i \times \{i\}$.

Thus, $\mathcal{A}'$ consists of $k+1$ copies of $\mathcal{A}$. One copy ("the initial copy") is full and it contains all the states in Q. Then, k copies are partial: every such copy is associated with a pair $\langle G_i, B_i \rangle$, its states are labeled i, and it contains all the states in $Q \setminus B_i$. A run of $\mathcal{A}'$ starts at the initial copy. The run can nondeterministically choose between staying in the initial copy or moving to one of the other k copies. Once a run of $\mathcal{A}'$ moves to a copy associated with the i'th pair, it cannot visit states from B_i. Indeed, Q_i does not contain such states. The acceptance condition of $\mathcal{A}'$ guarantees that an accepting run eventually leaves the initial copy and moves to some Q_i, where it visits infinitely many states from G_i. □

Since a parity acceptance condition can be translated to a Rabin acceptance condition, Theorem 2 implies the following theorem.

Theorem 3. *Given a parity automaton $\mathcal{A}$ with n states, m edges, and index k, there is a Büchi automaton $\mathcal{A}'$ with $O(nk)$ states and $O(mk)$ edges such that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$.*

Theorem 1 together with Theorems 2 and 3 imply an $O((n+m)k)$ solution to the nonemptiness problem for Rabin and parity automata with n states, m edges, and index k. In this paper we show that while the blow up in the translation of parity and Rabin automata to Büchi automata cannot be avoided [SN99], one can solve the nonemptiness problem for a parity automaton with n states, m edges, and index k, in time $O((n+m)\log k)$.

3 An Efficient Solution to the Nonemptiness Problem

In this section we present an algorithm of running time $O((n+m)\log k)$ for solving the nonemptiness problem for a parity automaton with n states, m edges, and index k. The idea of the algorithm is as follows. Consider a parity automaton $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$, with $\alpha = \{F_1, \ldots, F_{2k}\}$. It is easy to see that the automaton $\mathcal{A}$ is nonempty if there is $1 \leq i \leq k$ such that the graph G_i obtained from $\mathcal{A}$ by omitting states in $F_1 \cup \ldots \cup F_{2i-1}$ contains a MSCC with a node in F_{2i}. Since the graph G_i is contained in the graph G_{i-1}, the search for a candidate i, if starts from $i = k$ and proceeds to $i = 1$, involves a sequence of increasing graphs. Proceeding from G_i to G_{i-1}, we need to calculate the MSCC's of G_{i-1}. The MSCC's of G_i *refine* these of G_{i-1}, in the sense that if two vertices belong to the same MSCC in G_i, they also belong to the same MSCC in G_{i-1}. Once we need to calculate the MSCC's of G_{i-1}, we have already calculated these of G_i. So, there is a hope that we can do better than calculating the MSCC's of G_{i-1} from scratch, which is indeed what our algorithm does.

Our algorithm is a variation of an algorithm for hierarchical clustering [Tar82]. Consider a directed graph with m weighted edges and n vertices. A *hierarchical decomposition* of the graph is a process in which the graph's edges are added one at a time in order of weight. Tarjan's algorithm constructs a decomposition tree whose leaves are the vertices of the graph and whose internal nodes are associated with MSCC's that are formed as the hierarchical-decomposition

process proceeds. Thus, the children of a node associated with a component C are the subcomponents that coalesce to form C. The running time of the algorithm is $O(m \log n)$. Our variation of the algorithm corresponds to the special case of the graphs G_i described above: it inserts edges in batches, rather than one at a time, and it simplifies steps and data structures that are not relevant to the nonemptiness check. In particular, once the algorithm detects a nontrivial MSCC in which the minimal label of all vertices is even, it terminates with a positive reply.

The formal description of the algorithm is below, given in terms of the labeled directed graph that corresponds to $\mathcal{A}$. Given a labeled directed graph $G = \langle D, V, E, l \rangle$ with $D \subseteq \{1, 2, 3, \ldots, 2k\}$, the *even-cycle problem* is to determine whether there is a cycle C in G such that $\min_{v \in C}\{l(v)\}$ is even. It is easy to see that the nonemptiness problem for a parity automaton $\mathcal{A}$ is equivalent to the even-cycle problem for $G_{\mathcal{A}}$. Indeed, an accepting run of $\mathcal{A}$ induces an even cycle, and vice versa. Also, recall that we assume that all the states in $\mathcal{A}$, and hence also in $G_{\mathcal{A}}$, are reachable; thus we do not have to worry about the cycle being reachable.

Theorem 4. *Let $G = \langle \{1, 2, 3, \ldots, 2k\}, V, E, l \rangle$ be a labeled directed graph with $|V| = n$ and $|E| = m$. The even-cycle problem for G can be solved in time $O((n + m) \log k)$.*

Proof: For $i \in \{1, \ldots, 2k\}$, let $V_i \subseteq V$ be the set of vertices v with $l(v) \geq i$, and let $E_i = E \cap (V_i \times V_i)$. Thus, E_i contains only edges whose both endpoints have labels greater than or equal to i. Then, let $G_i = \langle \{i, \ldots, 2k\}, V_i, E_i, l \rangle$ be G when restricted to vertices and edges in V_i and E_i.

We first claim that the answer to the even-cycle problem is positive iff there is some even $i \in \{1, \ldots, 2k\}$ and a nontrivial MSCC of G_i that contains a vertex labeled i. Indeed, since $\min_{v \in V_i}\{l(v)\} \geq i$, such an MSCC witnesses a cycle C in G for which $\min_{v \in C}\{l(v)\} = i$, which is even. Also, if there is a cycle C in G such that $\min_{v \in C}\{l(v)\}$ is some even i, this cycle belongs to a nontrivial MSCC of G_i. We call i a *witness* for G.

For all i and j with $i < j$, we have that $V_j \subseteq V_i$, thus the graph G_j is a subgraph of G_i. Consequently, the MSCC's of G_j *refine* these of G_i, in the sense that if two vertices belong to the same MSCC in G_j, they also belong to the same MSCC in G_i. This refinement is the key to our algorithm. Following the analysis above, the algorithm searches for $i \in \{1, \ldots, 2k\}$ such that i is even and there is a nontrivial MSCC of G_i that contains a vertex labeled i. Intuitively, when the algorithm examines a candidate i, it either terminates with "Yes", in case i is a witness, or examine further candidates $j \neq i$. We distinguish between two cases. If $j > i$, the examination of j proceeds with a subgraph of G_i, consisting of vertices and edges that belong to the nontrivial MSCC's of G_j. If $j < i$, the examination of j proceeds with a *compressed version* of G_j, in which vertices that belong to the same MSCC of G_i are represented by a single vertex. Consequently, the graphs we consider have fewer states and edges. More precisely, for all $j_1 > i$ and $j_2 < i$, the number of edges in the two graphs required for checking j_1 and j_2 is not greater than the number of edges in the graph required for checking i.

Formally, we solve the even-cycle problem by the recursive routine **solve**$(\langle D, V, E, l\rangle, i, j)$ described below. The routine gets as input a labeled directed graph and two indices i and j in D. The routine outputs "Yes" if there is a witness in $\{i, \ldots, j\}$ for the even-cycle problem for $\langle D, V, E, l\rangle$, and outputs "No" otherwise. To solve the even-cycle problem for a given graph G, we call **solve**$(G, 1, 2k)$. The idea behind **solve**(G, i, j) is as follows. Given G and $i \leq j$ (if $j < i$, no witness $i \leq l \leq j$ exists and the routine terminates with "No"), the routine first finds the MSCC's of G_{mid}, where mid is an index midway between i and j. If mid is even and there is a nontrivial MSCC of G_{mid} with a vertex labeled mid, the routine stops and returns "Yes" (mid is the witness). Otherwise, we call **solve**$(G_\perp, mid + 1, j)$ and **solve**$(G_\top, i, mid - 1)$, where the call for **solve**$(G_\perp, mid + 1, j)$ searches for a witness in $\{mid + 1, \ldots, j\}$ and $G_\perp$ is a subgraph of G_{mid}, and the call for **solve**$(G_\top, i, mid - 1)$ searches for a witness in $\{i, \ldots, mid - 1\}$ and $G_\top$ is a compressed version of G, in which a set of vertices that belong to the same MSCC of G_{mid} is represented by a single vertex.

procedure solve$(\langle D, V, E, l\rangle, i, j)$

1. If $j < i$, return "No" and STOP.
2. $mid := \lceil \frac{i+j}{2} \rceil$.
3. a) $V_{mid} := \{v \in V \mid l(v) \geq mid\}$.
   b) $E_{mid} := E \cap (V_{mid} \times V_{mid})$.
4. Find the MSCC's of $G_{mid} = \langle D, V, E_{mid}, l\rangle$.
   For each $v \in V$, let $C_{mid}(v)$ denote the MSCC containing v in G_{mid}, and let $\mathcal{C}_{mid}$ denote the set of all (trivial and nontrivial) MSCC's in G_{mid}. Note that each vertex $v \in V \setminus V_{mid}$ induces the trivial MSCC $\{v\} \in \mathcal{C}_{mid}$.
5. If mid is even and there is $v \in V_{mid}$ such that $l(v) = mid$ and $C_{mid}(v)$ is nontrivial, output "Yes" and STOP.
6. (*Search for a witness in* $\{mid + 1, \ldots, j\}$.)
   a) $V_\perp := \{v \in V_{mid} \mid l(v) \geq mid + 1$ and $C_{mid}(v)$ is nontrivial $\}$.
   b) $E_\perp := \{\langle u, v\rangle \in E_{mid} \cap (V_\perp \times V_\perp) \mid C_{mid}(u) = C_{mid}(v)\}$.
   c) **solve**$(\langle D, V_\perp, E_\perp, l\rangle, mid + 1, j)$.
7. (*Search for a witness in* $\{i, \ldots, mid - 1\}$.)
   a) $V_\top := \mathcal{C}_{mid}$.
   b) $E_\top := \{\langle C_{mid}(u), C_{mid}(v)\rangle \mid \langle u, v\rangle \in E$ and $(\langle u, v\rangle \notin E_{mid}$ or $C_{mid}(u) \neq C_{mid}(v))\}$.
   c) For $C \in \mathcal{C}_{mid}$, define $l_\top(C) = \min_{v \in C}\{l(v)\}$.
   d) **solve**$(\langle D, V_\top, E_\top, l_\top\rangle, i, mid - 1)$.

Let us explain Steps (6) and (7) in more detail. In Step (6), the graph $G_\perp$ removes an edge from G_{mid} if the edge has a vertex labeled mid or if it connects vertices of different MSCCs. Indeed, such edges cannot participate in an MSCC that witnesses a cycle C for which $\min_{v \in C}\{l(v)\}$ is in $mid + 1, \ldots, j$. In Step (7), the vertices of $G_\top$ are the MSCC's of G_{mid} and there is an edge between two MSCC's C_u and C_v if there are vertices $u \in C_u$ and $v \in C_v$ such that u or v are not in V_{mid} (in which case the corresponding MSCC is a singleton and is trivial) and $\langle u, v\rangle \in E$, or both u and v are in V_{mid} and they belong to different MSCC's

(in which case $C_u \neq C_v$). Indeed, searching for a witness in $i, \ldots, mid-1$, we can ignore the internal structure of MSCC's in $\mathcal{C}_{mid}$.

We now prove formally that for every graph G and indices i and j, the procedure **solve**(G, i, j) outputs "Yes" iff there is an even $i \leq w \leq j$ and there is a cycle C in G such that w is the minimal label in C. Since the initial call is to **solve**$(G, 1, 2k)$ and all the cycles in G are reachable, this implies the correctness of the algorithm. The proof proceeds by induction on $j - i$. When $j - i < 0$, thus $j < i$, no $i \leq w \leq j$ exists, and the procedure indeed outputs "No". For the induction step, assume that the correctness claim holds for $j - i \leq k$, and consider the case $j - i = k + 1$. Let $mid = \lceil \frac{i+j}{2} \rceil$. Clearly, there is an even $i \leq w \leq j$ and there is a cycle C in G such that w is the minimal label in C iff one of the following holds:

(a) mid is even and there is a cycle C in G such that mid is the minimal label in C,

(b) there is an even $mid + 1 \leq w \leq j$ and there is a cycle C in G such that w is the minimal label in C, or

(c) there is an even $i \leq w \leq mid - 1$ and there is a cycle C in G such that w is the minimal label in C.

We prove that **solve**(G, i, j) outputs "Yes" in Steps (5), (6), or (7), iff (a), (b), or (c) holds, respectively.

- Assume that (a) holds. Then, the cycle C exists in G_{mid}, the conditions in Step (5) are satisfied, and the algorithm outputs "Yes". Assume now that the algorithm outputs "Yes" is Step (5). Then, mid is even and there is a nontrivial MSCC in G_{mid} that contains a vertex labeled mid. By the definition of G_{mid}, this implies the existence of a cycle C in G such that mid is the minimal label in C. Hence, (a) holds.
- Assume that (b) holds. Then, the cycle C exists in $\langle D, V_\perp, E_\perp, l \rangle$. Indeed, since the minimal index in C is $mid+1 \leq w \leq j$, the vertices and edges that are removed from G do not participate in C. So, by the induction hypothesis, the algorithm outputs "Yes" in Step (6). Assume that the algorithm outputs "Yes" in Step (6). Then, by the induction hypothesis, there is an even $mid+1 \leq w \leq j$ and there is a cycle C in $\langle D, V_\perp, E_\perp, l \rangle$ such that w is the minimal label in C. Since the cycle C is also a cycle of G, it follows that (b) holds.
- Assume that (c) holds. Let $i \leq w \leq mid - 1$ and $C = v_1, v_2, \ldots, v_n$ be such that w is even and C is a circle in G for which w is the minimal label in C (that is, $v_1 = v_n$). Let $C' = C_{mid}(v_1), C_{mid}(v_2), \ldots, C_{mid}(v_n)$ be the sequence of MSCCs in $\mathcal{C}_{mid}$ induced by C. For all $1 \leq k \leq n-1$, we have that $C_{mid}(v_k) = C_{mid}(v_{k+1})$ or $E_\top(C_{mid}(v_k), C_{mid}(v_{k+1}))$. Hence, the sequence obtained from C' by omitting successive repetitions of the same MSCC in $\mathcal{C}_{mid}$ is a cycle $C_\top$ in $\langle D, V_\top, E_\top, l_\top \rangle$. Since the minimal label in C is w, then, by definition of $l_\top$, the same holds for $C_\top$. So, by the induction hypothesis, the algorithm outputs "Yes" in Step (7). Assume that the algorithm outputs "Yes" in Step (7). Then, by the induction hypothesis, there is an even $i \leq w \leq mid - 1$ and there is a cycle $C_\top$ in $\langle D, V_\top, E_\top, l_\top \rangle$ such that w is the minimal label in $C_\top$. By replacing each vertex in $C_\top$ (which is a MSCC in G) by the appropriate path in G, we can get from $C_\top$ the cycle required for (b) to hold.

We now analyze the time complexity of the algorithm. First, note that since all the states in the graph G are reachable, then all the intermediate graphs $G_\perp$ and $G_\top$ created by the procedure are such that $|E_\perp| \geq |V_\perp|$ and $|E_\top| \geq |V_\top|$. Also, note that for every graph G, each edge $\langle u, v\rangle \in E$ contributes an edge to at most one of $E_\perp$ and $E_\top$. That is, $|E_\perp| + |E_\top| \leq |E|$. Indeed, an edge $\langle u, v\rangle \in E$ is in $E_\perp$ only if $\langle u, v\rangle \in E_{mid}$ and $C_{mid}(u) = C_{mid}(v)$, and it contributes an edge to $E_\top$ only if it satisfies the complementary condition, namely $\langle u, v\rangle \notin E_{mid}$ or $C_{mid}(u) \neq C_{mid}(v)$.

Let $R(s, d)$ denote the cost of **solve**$(\langle D, V, E, l\rangle, i, j)$, where $s = |E|$ and $d = j - i + 1$ (i.e., d is the size of the set $\{i, \ldots, j\}$ within which a witness is searched). Since we can find the MSCC's of a graph in time $O(|V| + |E|)$ and $s = |E| \geq |V|$, the cost of steps (1-5) of the algorithm is $O(s)$, thus the cost of **solve**(G, i, j) is $O(s)$ plus the cost of the two recursive calls to **solve** in steps (6) and (7). Hence, we can describe $R(s, d)$, for all $s \geq 0$, by a recurrence as follows.

- $R(s, 0) = O(1)$.
- For all $d \geq 1$, we have $R(s, d) = O(s) + R(s_\perp, \lfloor \frac{d-1}{2} \rfloor) + R(s_\top, \lceil \frac{d-1}{2} \rceil)$, with $s_\perp + s_\top \leq s$.

It is easy to see that $R(s, 1) = R(s, 2) = O(s)$. We prove that for all $d > 2$, we have $R(s, d) = O(s \log d)$. In particular, the cost of **solve**$(\langle D, V, E, l\rangle, 1, 2k)$ is $O((|V| + |E|) \log k)$.

Let b be a constant such that for all s, we have $R(s, 2) \leq bs$. Note that the cost of steps (1-5) is then at most bs. We prove that there is $c \geq b$ such that for all $d \geq 2$, we have that $R(s, d) \leq cs \log_2 d$, thus $R(s, d) = O(s \log_2 d)$. The proof proceeds by an induction on d. Recall that $R(s, 2) \leq bs$, hence $R(s, 2) \leq cs \log_2 2$, and the induction claim holds for the base $d = 2$. Now, assume that for all $2 \leq d' \leq d$, we have that $R(s, d') \leq cs \log_2 d'$. By the recurrence above, $R(s, d + 1) \leq cs + R(s_\perp, \lfloor \frac{d}{2} \rfloor) + R(s_\top, \lceil \frac{d}{2} \rceil)$, for $s_\perp$ and $s_\top$ with $s_\perp + s_\top \leq s$. Then, by the induction hypothesis, $R(s, d+1) \leq cs + cs_\perp \log_2 \lfloor \frac{d}{2} \rfloor + cs_\top \log_2 \lceil \frac{d}{2} \rceil \leq cs(1 + \log_2 \lceil \frac{d}{2} \rceil) \leq cs \log_2(d + 1)$, and we are done.

□

References

[BGS00] R. Bloem, H.N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In *Formal Methods in Computer Aided Design*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[Cho74] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 328–337, White Plains, October 1988.

[EJ91] E.A. Emerson and C. Jutla. Tree automata, μ-calculus and determinacy. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 368–377, San Juan, October 1991.

[HT96] M. Henzinger and J.A. Telle. Faster algorithms for the nonemptiness of Streett automata and for communication protocol pruning. In *Proc. 5th Scandinavian Workshop on Algorithm Theory*, volume 1097 of *Lecture Notes in Computer Science*, pages 10–20. Springer-Verlag, 1996.

[Jur98] M. Jurdzinski. Deciding the winner in parity games is in up $\cap$ co-up. *Information Processing Letters*, 68(3):119–124, 1998.

[JW95] D. Janin and I. Walukiewicz. Automata for the modal μ-calculus and related results. In *Proc. 20th International Symp. on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, pages 552–562. Springer-Verlag, 1995.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Kur94] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[McN66] R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.

[Mos84] A.W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In *Computation Theory*, volume 208 of *Lecture Notes in Computer Science*, pages 157–168. Springer-Verlag, 1984.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

[SN99] H. Seidl and D. Niwiński. On distributive fixed-point expressions. *Theoretical Informatics and Applications*, 33(4–5):427–446, 1999.

[Tar72] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[Tar82] R.E. Tarjan. A hierarchical clustering algorithm using strong components. *Information Processing Letters*, 14:26–29, 1982.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

Foundations for a Graph–Based Approach to the Specification of Access Control Policies*

Manuel Koch[1], Luigi Vincenzo Mancini[2], and Francesco Parisi-Presicce[2,3]

[1] PSI AG, Berlin (DE)
[2] Univ. di Roma La Sapienza, Rome (IT)
[3] George Mason Univ., Fairfax VA (USA)
mkoch@psi.de lv.mancini@dsi.uniroma1.it parisi@dsi.uniroma1.it ; fparisi@ise.gmu.edu

Abstract. Graph Transformations provide a uniform and precise framework for the specification of access control policies allowing the detailed comparison of different policy models and the precise description of the evolution of a policy. Furthermore, the framework is used for an accurate analysis of the interaction between policies and of the behavior of their integration with respect to the problem of conflicting rules. The integration of policies is illustrated using the Discretionary Access Control and the Lattice Based Access Control policies.

1 Introduction

A considerable amount of work has been carried out recently on models and languages for Access Control. Access Control (AC) is concerned with determining the activities of legitimate users [SS94]. Usually AC is enforced by a reference monitor which mediates every attempted access by a *subject* (a program executing on behalf of a user) to *objects* in the system. In [KMPP00b,KMPP00a] we have proposed graph transformations as a uniform conceptual framework for the specification of access control policies. In this paper we discuss the formal properties of this framework and their applications to problems not addressed anywhere else in a formal way.

The three main AC policies commonly used in computer systems are discretionary policies [SS94], lattice-based policies (also called mandatory policies) [San93] and role-based policies [San98]. As illustrative examples, we use here the lattice-based access control (LBAC) and the access control list (ACL) that is an implementation of a discretionary policy. Role-based access control is not considered in this article, but is the main focus of [KMPP00b].

Lattice-based access control: Classic LBAC enforces unidirectional information flow in a lattice of security levels [1]. The diagram on the left-hand side of Fig. 1 shows a partial order security lattice.

* partially supported by the EC under TMR Network GETGRATS and under Esprit WG APPLIGRAPH, and by the Italian MURST.

[1] In [San93], security levels are called security labels. We use 'security level' here to avoid confusion with the notion of a label for a node or an edge in a graph.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 287–302, 2001.

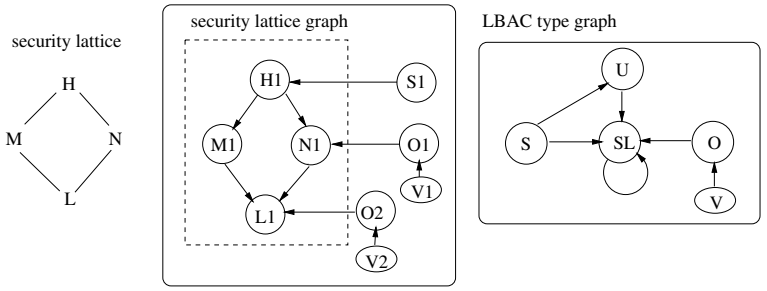


Fig. 1. A security lattice (left-hand side) and its presentation by a graph (middle). The type graph for the LBAC model (right-hand side).

The LBAC policy is expressed in terms of security levels attached to *subjects* and *objects*. A subject is a process in the system and each subject is associated to a single user, where one user may have several subjects concurrently running in the system. An object is a container of information, e.g. files or directories in an operating system. Usually the security levels on subjects and objects, once assigned, do not change. If $\lambda(x)$ denotes the security level of x (subject or object) then the specific LBAC rules for a lattice allow a subject S to read object O if $\lambda(S) \geq \lambda(O)$ and to write object O if $\lambda(S) = \lambda(O)$.

Access Control List: The ACL policy is an implementation of a discretionary AC policy. We consider an ACL policy similar, but simpler, to that one used in the UNIX operating system. Our model distinguishes only between the owner of an object and the rest of the world and for simplicity groups are not considered. The owner of the object has read, write and execution access and can change the access permissions of the object with respect to the world.

The different models for Access Control have been specified with expressive but ad hoc languages requiring ad hoc conversions to compare the relative strengths and weaknesses. Not much work [BdVS00] has been performed on the evolution of a policy, to construct in an incremental way complex policies, and on the integration of policies, to obtain the global policy of an organization by combining the security policies of different departments.

The main goal of this paper is to present some basic properties of a formal model for Access Control policies based on graphs and graph transformations and to address the problems of evolution and integration in a categorical setting. A system state is represented by a graph and graph transformation rules describe how a system state evolves. The specification ("framework") of an AC policy contains also declarative information ("invariants") on what a system graph must contain (positive) and what it cannot contain (negative). A crucial property of a framework is that it specifies a coherent policy (one without internal contradictions). As with any soft system, policies can evolve and can be combined: we have formalized here in a categorical setting the notions of evolution and of integration and their main properties. To help with the issue of conflicting rules in the integrated framework obtained by combining two distinct ones, the notion of metapolicy is proposed.

The paper is organized as follows: the next section reviews graph transformations and introduces the LBAC and the ACL policies; Sect. 3 defines the formal framework to specify AC policies and presents its main properties; Sect. 4 discusses the notion of integration of policies and Sect. 5 introduces metapolicies to resolve conflicts; the last section mentions related and future work.

2 Graph Transformations

This section introduces graph transformations [Roz97]. The LBAC model is used throughout the section to support the explanations by an example.

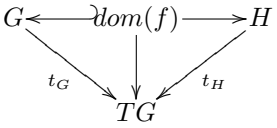
A *graph* $G = (G_V, G_E, s_G, t_G, l_G)$ consists of disjoint sets of nodes G_V and edges G_E, two total functions $s_G, t_G : G_E \to G_V$ mapping each edge to its source and target node, respectively, and a function $l_G : G_V \cup G_E \to L$ assigning to each node/edge a label. Labels are elements of a set $L = X \cup C$, where X is a set of *variables* and C is a set of *constants*. A binary relation $\prec \subseteq L \times L$ is defined on L as $\alpha \prec \beta$ if and only if $\alpha \in X$. A path between nodes a and b is indicated by an edge $a \overset{*}{\to} b$ and can be seen as an abbreviation for a set of paths each representing a possible sequence of edges between a and b.

A *total graph morphism* $f : G \to H$ between graphs $G = (G_V, G_E, s_G, t_G, l_G)$ and $H = (H_V, H_E, s_H, t_H, l_H)$ is a pair (f_V, f_E) of total mappings $f_V : G_V \to H_V$ and $f_E : G_E \to H_E$ that respect the graph structure, i.e. $f_V \circ s_G = s_H \circ f_E$ and $f_V \circ t_G = t_H \circ f_E$, as well as the label order, i.e. $l_G(v) \prec l_H(f_V(v))$ for each $v \in G_V$ and $l_G(e) \prec l_H(f_E(e))$ for each $e \in G_E$. A *partial graph morphism* $f : G \rightharpoonup H$ is a total graph morphism $\bar{f} : dom(f) \to H$ from a subgraph $dom(f) \subseteq G$ to H. Graphs and partial graph morphisms form a category $\mathbf{Graph^P}$. The subcategory of graphs and total graph morphisms is denoted by $\mathbf{Graph}$. The category $\mathbf{Graph^P}$ is in general not co-complete, but has pushouts for morphisms $f_p : G \rightharpoonup H$ and $f_c : G \rightharpoonup K$ where f_p is *label preserving*, i.e. $l_G(x) = l_H(f_p(x))$ for all $x \in G$ (node or edge) [PPEM87].

A *type graph* TG represents the type information in a graph transformation system [CELP96], specifying the node and edge types which may occur in the instance graphs modeling system states. For instance, the type graph in Fig. 1 on the right shows the possible types for the LBAC graph model. There is a type U for user nodes, a type S for subjects, a type O for objects, a type V for the value of objects and a type SL for the security levels. This type graph indicates also that there cannot be an edge from a node of type U to a node of type O.

A pair $\langle G, t_G \rangle$, where G is a graph and $t_G : G \to TG$ is a total graph morphism, is called a *graph typed over* TG. If the type graph is fixed, we denote the pair simply as G. The total graph morphism t_G is called *typing morphism* and is indicated in the examples by the symbols used for nodes and edges. In the middle of Fig. 1 a graph typed over the LBAC type graph is shown. The graph consists of a node with label $H1$ and type SL, a node with label $S1$ and type S, a node with label $O1$ and type O, etc. The security lattice on the left-hand side is modeled by a security lattice graph. From now on, the typing morphism maps a node with label Tx to the type T.

A *morphism* between typed graphs $\langle G, t_G \rangle$ and $\langle H, t_H \rangle$ is given by a partial graph morphism $f : G \hookleftarrow dom(f) \to H$ that preserves types, that is, the diagram



in **Graph** commutes. The morphism is total if the underlying graph morphism is total. Graphs typed over a fixed type graph TG and morphisms between them form a category **TG** [CELP96]. The existence of pushouts is inherited from the category $\mathbf{Graph^P}$.

A graph typed over a type graph TG can be *re-typed* over TG' if there is a total morphism $f : TG \to TG'$. The re-typing of a graph $\langle G, t_G \rangle$ typed over TG to a graph typed over TG' is a renaming of types in G. Re-typing from TG' to TG is a renaming of types and a forgetting of nodes and edges. Formally, the re-typing w.r.t. a morphism $f : TG \to TG'$ is specified by functors $F_f : \mathbf{TG} \to \mathbf{TG'}$ and $V_f : \mathbf{TG'} \to \mathbf{TG}$, called *forward typing* and *backward typing functor* [CELP96,GRPPS98].

A *graph rule* $p : r$, or just *rule*, is given by a rule name p, from a set $RNames$, and a label preserving *morphism* $r : L \rightharpoonup R$. The graph L, *left-hand side*, describes the elements a graph must contain for p to be applicable. The partial morphism is undefined on nodes/edges that are intended to be deleted, defined on nodes/edges that are intended to be preserved. Nodes and edges of R, *right-hand side*, without a pre-image are newly created.

Example 1 (Graph rules for the LBAC graph model). Figure 2 shows the rules for the LBAC policy. The labels for the nodes ($Ux, Sx, SLx, SLy, ...$) of the rules are variables taken from the set of variables in L. The rule `new object` creates a new object Ox connected to a node Vx (the initial value of the object). The object

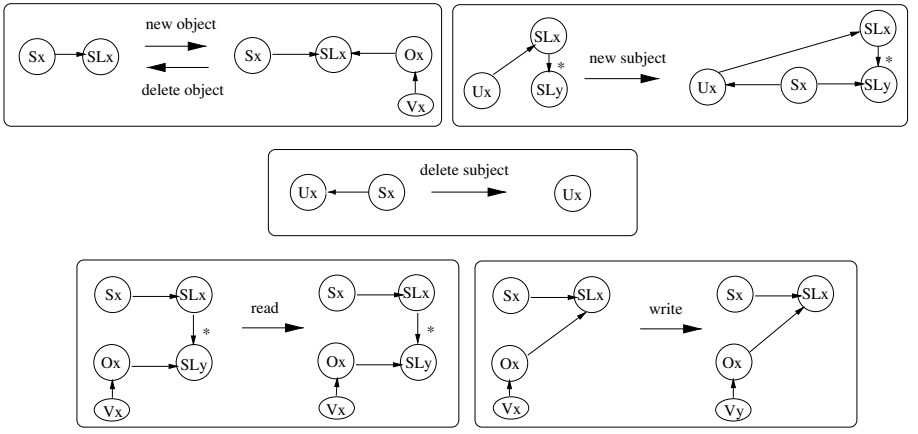
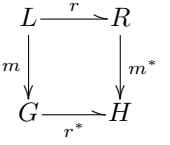


Fig. 2. Graph rules for the LBAC policy.

Ox gets the security level SLx. The variable SLx is generic: it is substituted by the actual security level of the subject when the rule is applied. The rule for `delete object` for the deletion of objects is represented by reversing the partial morphism of the rule `new object`. The rule `new subject` creates a subject Sx on behalf of a user Ux. The new subject is attached to a security level SLy that is lower in the hierarchy graph than the security level SLx of the user Ux. This requirement is specified by the path from SLx to SLy, since edges in the security lattice graph point from higher to lower levels.

For the application of rules we use the Single Pushout (SPO) approach to graph transformations [EHK+97]. Formally, the application of a graph rule $p : L \overset{r}{\rightharpoonup} R$ to a graph G is given by a total graph morphism $m : L \to G$, called *match* for p in G. The direct derivation $G \overset{p,m}{\Rightarrow} H$ from G to the derived graph H is given by the pushout of r and m in **TG** (see the diagram below). Note that the pushout exists, since the rule morphism is label preserving [PPEM87].



Example 2 (Application of a graph rule). In Fig. 3, the left-hand side L of the rule `new subject` occurs several times in G. In one possible match the node Ux in L is associated to the node U_2 in G and the nodes SLx and SLy to the specific security level H.

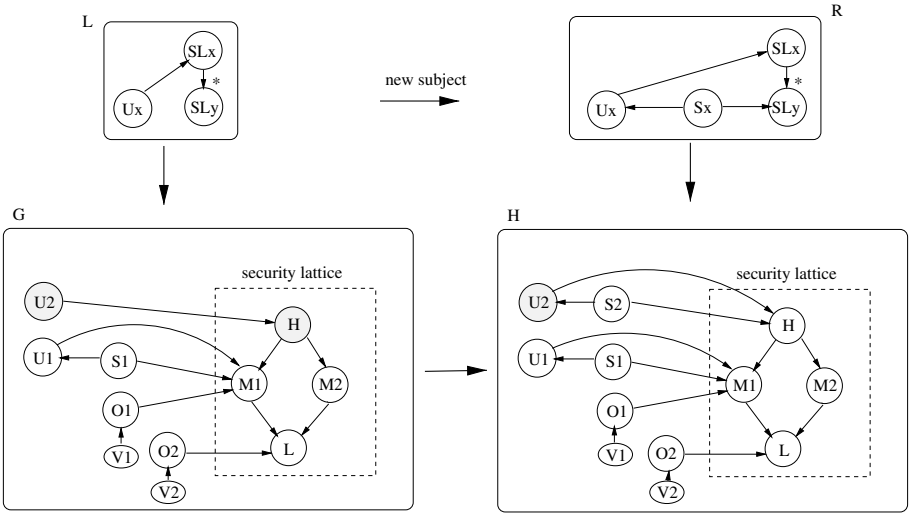


Fig. 3. Application of rule `new subject`.

For the specification of the ACL by graph transformations, *negative application conditions* for rules are needed. A negative application condition (NAC) for a rule $p : L \xrightarrow{r} R$ consists of a set $A(p)$ of pairs (L, X), where the graph L is a subgraph of X. The part $X \setminus L$ represents a structure that must not occur in a graph G for the rule to be applicable. In the figures, we depict (L, X) by the graph X, where the subgraph L is drawn with solid and $X \setminus L$ with dashed lines. A rule $p : L \xrightarrow{r} R$ with a NAC (L, X) is applicable to G if L occurs in G and it is not possible to extend L to X. Examples of rules with a NAC are the ACL rules `connect` and `give read` in Fig. 5.

Example 3 (Graph rules for the ACL). The type graph TG_{ACL} in Fig. 4 provides the node types U, O and P. Just as in the LBAC model, a node of type U represents a user and a node of type O an object. An edge between a user node U and an object node O specifies that U is the owner of the object O. A node

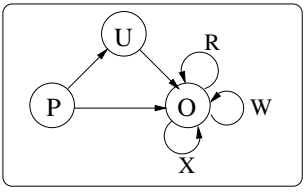


Fig. 4. The type graph for the ACL.

of type P represents a process. An edge of type R, W or X represents the read, write or execute permission of an object to the world. The owner of the object has always all the permissions for his/her objects and does not need the loops. Some of the ACL graph rules are shown in Fig. 5. The rule `new process` starts a new process on behalf of a user. To kill a process, the rule `remove process` deletes the process node and its connection to the user. The rule `create object` adds a new node Ox to the system, connecting it to the process node Px that

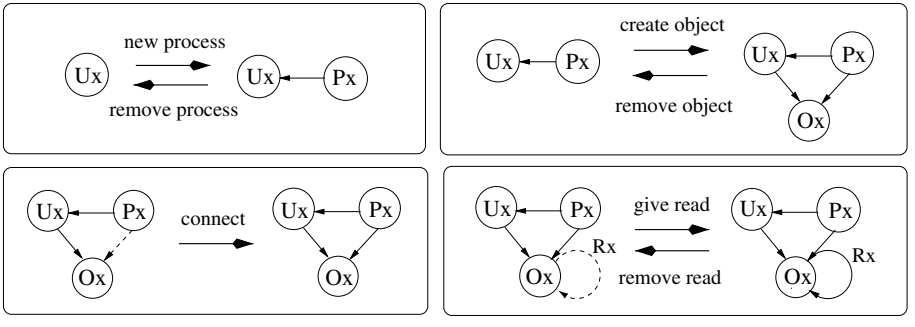


Fig. 5. Graph rules for the ACL model.

has created the object and to the user node Ux to which the process belongs. The rule `connect` connects a process of a user to an object of the user. The rule has a NAC (indicated by the dashed edge between Px and Ox on the left-hand side of the rule) that forbids the application of the rule to processes and objects of the user already connected. The rule `give read` gives the read permission provided that it has not been granted already. Other rules such as `give write` and `give execution` are not shown.

3 Security Policy Framework

This section introduces the framework for the specification of AC policies based on graph transformations. The framework is called *security policy framework* and consists of four components: The first component is a type graph that provides the type information of the AC policy. The second component is a set of graph rules specifying the policy rules that generate the graphs representing the states of the system accepted by the AC policy. For some AC policies, it is meaningful to restrict the set of system graphs constructed by the graph rules, since not all of them represent valid states. Therefore, a security policy framework contains also two sets of *constraints* that specify graphs that shall not be contained in any system graph (*negative constraints*) and graphs that must be explicitly constructed as parts of a system graph (*positive constraints*). In the actual implementation of an AC policy, the constraints are redundant since the only acceptable states are those explicitly built by the implemented rules. But when developing an AC policy through successive refinement steps, or when comparing different policies, or when trying to predict the behavior of the policy obtained by integrating two different ones, it is useful to have the additional information provided by the constraints. Furthermore, it is usually difficult to extract negative informations from "constructive" rules. Positive and negative constraints can be considered as formal documentation of the initial requirements and the development process of rules. Both positive and negative constraints are formally specified by morphisms. Only their semantics distinguishes them.

Definition 1 (Negative and positive constraints). *A* constraint *(positive or negative) is given by a total graph morphism $c : X \to Y$. A graph G* satisfies *a positive (negative) constraint c if for each total graph morphism $p : X \to G$ there exists (does not exist) a total graph morphism $q : Y \to G$ such that $X \xrightarrow{c} Y \xrightarrow{q} G = X \xrightarrow{p} G$.*

Example 4 (Constraints for LBAC and ACL). Figure 6 shows two positive constraints and two negative constraints for the LBAC model. The morphisms for the negative constraints are the identity on the graphs shown. The positive and the negative constraint on the left-hand side require that objects always have a security level (the positive constraint) and that there does not exist more than one security level for an object (negative constraint). The right-hand side of the figure specifies the same existence and uniqueness requirements for subjects.

The constraints for the ACL framework in Fig. 7 require that a process belong to a unique user (the positive and the first negative constraint), that an object

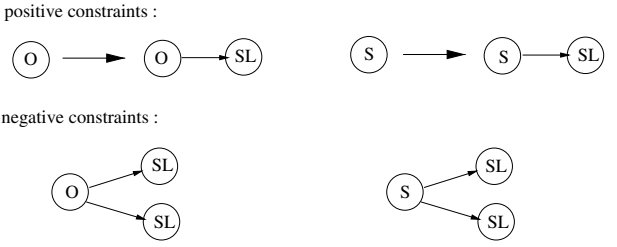


Fig. 6. Positive and negative constraints for LBAC.

does not belong to more than one user (the second negative one) and there is at most one permission loop with the same permission attached to the same object (the third negative one). Note that the third negative constraint represents three negative constraints, one for R, one for W and one for X.

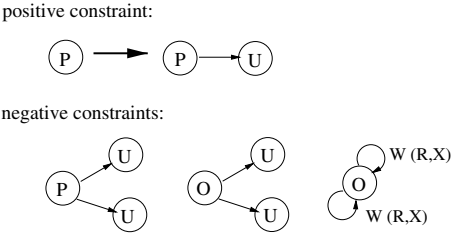


Fig. 7. Positive and negative constraints for the ACL.

Definition 2 (Security Policy Framework). *A* security policy framework, *or just* framework, *is a tuple* $SP = (TG, (P, r_P), Pos, Neg)$, *where* TG *is a type graph, the pair* (P, r_P) *consists of a set of rule names and a total mapping* $r_P : P \to |\mathbf{Rule}(TG)|$ *mapping each rule name to a rule* $L \overset{r}{\rightharpoonup} R$ *of* TG*–typed graphs,* Pos *is a set of positive and* Neg *is a set of negative constraints.*

The graphs that can be constructed by the rules of a framework represent the system states possible within the policy model. These graphs are called *system graphs* in the sequel.
A security policy framework is *positive* (resp. *negative*) *coherent* if all system graphs satisfy the constraints in Pos (resp. Neg). It is *coherent* if it is both positive and negative coherent.

The security policy framework for the LBAC policy consists of the type graph in Fig. 1 and the negative and positive constraints in Fig. 6. The rule names $p_1, ..., p_6$ are mapped to the rules in Example 1.

A *security policy framework morphism* $f : SP_1 \to SP_2$, or just *framework morphism*, relates security policy frameworks by a total graph morphism f_{TG} :

$TG_1 \to TG_2$ between the type graphs and a mapping $f_P : P_1 \to P_2$ between the sets of rule names. The mapping f_P must preserve the behavior of rules as in the sense that a rule name x can be mapped to a rule name $f_P(x)$ only if $f_P(x)$ does on the renamed types everything which x does and possibly more. The set of positive constraints in SP_2 can contain, in addition to Pos_1, new positive constraints and positive constraints of SP_1 extended w.r.t. new types. The set of negative constraints in SP_2 may contain additional negative constraints on new types, but must not impose new negative constraints on old types.

Definition 3 (Framework Morphism). *A* framework morphism *between security policy frameworks $SP_i = (TG_i, (P_i, r_{P_i}), Pos_i, Neg_i)$ for $i = 1, 2$ is a pair $f = (f_{TG}, f_P) : SP_1 \to SP_2$, where $f_{TG} : TG_1 \to TG_2$ is a total graph morphism and $f_P : P_1 \to P_2$ is a total mapping, so that $V_{f_{TG}}(r_{P_2}(f_P(p))) = r_{P_1}(p)$ for all $p \in P_1$, $Pos_1 \subseteq V_{f_{TG}}(Pos_2)$ and $V_{f_{TG}}(Neg_2) \subseteq Neg_1$.*

We provide now the categorical formalization by defining the category of security policy frameworks and framework morphisms.

Definition 4 (Category of Security Policy Frameworks). *The* category of security policy frameworks*, denoted by* **SP***, has as objects all security policy frameworks and as morphisms all framework morphisms. For each framework SP, $id_{SP} = (id_{TG}, id_P)$ is the identity and composition is defined componentwise.*

Proposition 1 (Initial Security Policy Framework). *The initial object in* **SP** *is given by the security policy framework $SP_I = (\emptyset, (\emptyset, r), \emptyset, \emptyset)$.*

Security policy frameworks can be glued together using the standard categorical constructions.

Proposition 2 (Pushouts). *The category* **SP** *has all pushouts.*

By combining the previous two, we obtain the main result of this section.

Theorem 1 (Colimits). *The category* **SP** *is finitely cocomplete.*

A security policy framework $SP = (TG, (P, r_P), Pos, Neg)$ can be changed by modifying its components, that is, the extension or reduction of the type graph, the addition/removal of a graph rule to/from P, the addition/removal of a positive constraint to/from Pos and the addition/removal of a negative constraint to/from Neg. A framework morphism $f : SP_1 \to SP_2$ describes the change of the framework SP_1 to the framework SP_2, but also from SP_2 to SP_1. We define an evolution as a sequence of framework morphisms in the category **SP** that can be travelled in both directions.

Definition 5 (evolution). *An* evolution *of a framework SP to a framework SP' is a sequence $e = (SP_0 SP_1 ... SP_{n-1} SP_n)$ of frameworks such that $SP_0 = SP$, $SP_n = SP'$ and, for each $i = 0, ..., n-1$, there is a framework morphism $m_i^f : SP_i \to SP_{i+1}$ or $m_i^b : SP_{i+1} \to SP_i$.*

The evolution of a security policy framework yields a new security policy framework that reflects the desired changes. The changes, however, do not ensure generally that the new security policy framework is coherent. From a semantical point of view, this problem can be solved by considering the full sub-category $\mathbf{SP}^c$ of $\mathbf{SP}$ that contains only coherent security policy frameworks. Evolution is possible only in this sub-category. From an operational point of view, we can solve the problem by using a mechanical construction originally introduced in [HW95]. The construction manipulates the rules of a framework by adding application conditions to ensure that the rules do not create graphs that do not satisfy the constraints. A methodology for generating a coherent security policy framework is presented in [KMPP00b,KMPP00a].

4 Integration of Security Policy Frameworks by Pushouts

Integration is concerned with the merging of AC policies. A merge is necessary on the syntactical level, i.e. a merge of the security policy frameworks, and on the semantical level, i.e. the merge of the system graphs representing the state at merge time. The merge on the semantical level is what distinguishes an evolution from an integration: evolution is expressed syntactically by considering only the security policy frameworks, integration includes semantical changes, too. The integration of two AC policies on the syntactical level is a pushout of the security policy frameworks in the category $\mathbf{SP}$[2]. Two security policy frameworks SP_1 and SP_2 are related by an auxiliary framework SP_0 that identifies the common parts (types and rules) in both frameworks; the actual integration is expressed by framework morphisms $f_1 : SP_0 \to SP_1$ and $f_2 : SP_0 \to SP_2$. The pushout of f_1 and f_2 in $\mathbf{SP}$ integrates the frameworks SP_1 and SP_2 in a new security policy framework SP called the *integrated framework*.

Throughout this section, the integration of the LBAC framework with the ACL framework (both introduced in Section 3) is used as an example.

Example 5 (Pushout integration of the ACL and LBAC frameworks). The type graph in the middle of Fig. 8 shows the types common to ACL and $LBAC$. The U and the O types are in common and the type P (processes in ACL) and the type S (subjects in LBAC) coincide. The edge between the U and the P (resp. S) node is a common part as well. The pushout of the two type graphs is the type graph at the bottom of Fig. 8. The pushout identifies the P and S node to a common type P. All rules are kept in the integrated security policy framework, where their graphs are now typed over the integrated type graph. The integrated policy framework contains the two positive constraints of the LBAC model (now typed over the integrated type graph) and the positive ACL constraint. The integrated framework has no negative constraints, since there are (after re-typing) no common negative constraints in ACL and $LBAC$.

An important integration aspect is the preservation of coherence: if the frameworks SP_1 and SP_2 are coherent, is SP? Generally, this is not the case. The

[2] The integration concepts of the paper can be easily generalized to an integration of several frameworks because of the existence of (finite) colimits in $\mathbf{SP}$.

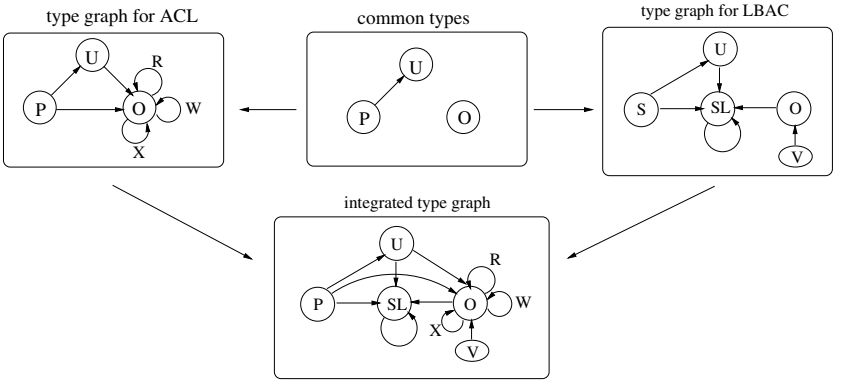


Fig. 8. Integrated type graph for the combined LBAC and ACL security model.

integrated framework for the ACL and the LBAC frameworks of the previous example contains a positive constraint that requires a security level for each node of type P (stemming from the identification of the LBAC type S and the ACL type P). This requirement can be destroyed by the ACL rule `new process`. Negative constraints, however, are preserved by the pushout.

Proposition 3 (Preservation of negative coherence). *Given framework morphisms $f_1 : SP_0 \to SP_1$ and $f_2 : SP_0 \to SP_2$ so that SP_1 and SP_2 are coherent, the pushout object $SP = (TG^{int}, (P^{int}, r_{P^{int}}), Pos^{int}, Neg^{int})$ of f_1 and f_2 in* **SP** *is coherent w.r.t. the set of negative constraints Neg^{int}.*

Coherence w.r.t. positive constraints is generally not preserved by the pushout construction as the counterexample above shows. The reason for incoherence w.r.t. positive constraints, however, can be reduced to the parts of positive constraints referring to common types. Coherence of positive constraints referring to types occuring only in SP_1 or only in SP_2 is preserved.

Proposition 4 (Preservation of positive coherence). *Given framework morphisms $f_1 : SP_0 \to SP_1$ and $f_2 : SP_0 \to SP_2$ with SP_1 and SP_2 coherent, the pushout $SP = (TG^{int}, (P^{int}, r_{P^{int}}), Pos^{int}, Neg^{int})$ of f_1 and f_2 in* **SP** *is incoherent if and only if SP is incoherent w.r.t. positive constraints of Pos^{int} containing types in TG_0.*

After merging the AC policies on the syntactical level to an integrated SP, the AC policies are merged on the semantical level by a pushout of the system graphs G_1 of SP_1 and G_2 of SP_2 representing the system states at merge time. The merge of the system graphs must yield a graph typed over the integrated type graph TG^{int} of SP.

Example 6 (Integration of LBAC and ACL System Graphs). Figure 9 shows a system graph for the ACL framework and the LBAC framework. The auxiliary graph G_0 contains the user $U1$ (may be working for both companies) and the objects $O1$ and $O2$ (may be files already shared before the merge). The integrated system graph at the bottom of Fig. 9 contains features of both frameworks.

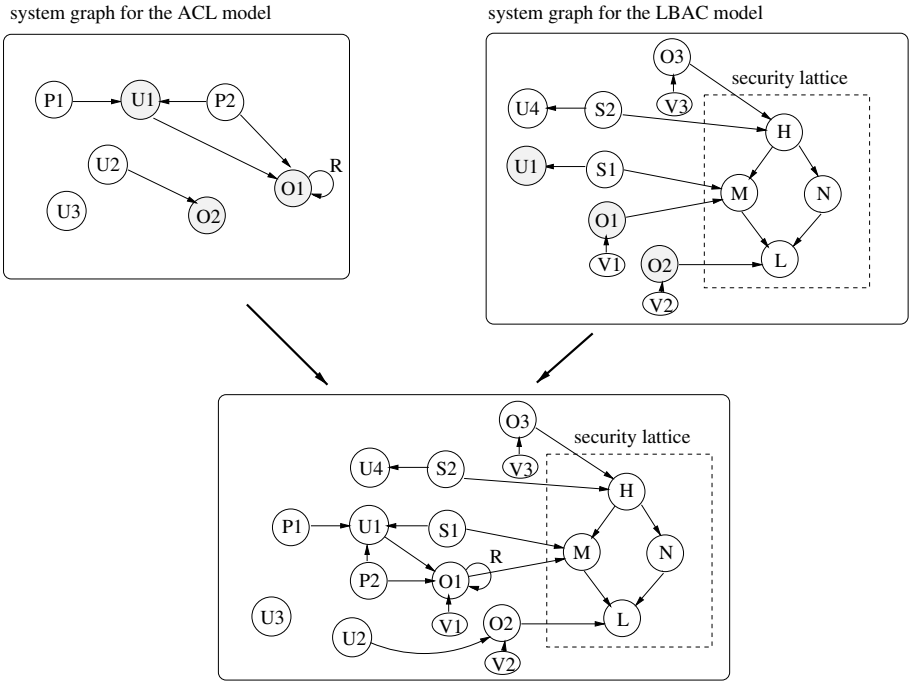


Fig. 9. Integrated system graph for the combined LBAC and ACL framework.

The integration of the system graphs does not ensure that the integrated system graph satisfies the constraints of the integrated framework SP. There are two possible solutions: first, the integrated system graph is adapted to satisfy the constraints. In the example, the integrated system graph in Fig. 9 could be modified by connecting objects and processes without security level with the security lattice graph. Second, the set of constraints can be changed by, for instance, removing the unsatisfied constraints. This is meaningful, if one policy is preferred and the constraint is from the other policy. In the example, the LBAC constraint for security levels could be removed if the ACL policy is preferred.

5 Solving Rule Conflicts by Meta Policies

In this section, we assume that two security policies are integrated by a pushout in **SP** for the frameworks and a pushout in $\mathbf{TG}^{int}$ for the system graphs. Even if the integrated system graph is modified to be coherent w.r.t. the constraints of the integrated framework, there may be rules in the integrated framework, coming from SP_1 and SP_2, respectively, that are applicable to the same part of the integrated system graph. If the rules specify two conflicting actions specific to the policy, we call such a situation a *rule conflict* and denote by $CR(SP)$ the set of conflicting rule pairs. For instance, the LBAC rule `new object` and the ACL rule `create object` are in conflict, since both are applicable to the user

$U1$ in the integrated system graph in Fig. 9. The rule `new object` would create an object with a security level, the rule `create object` an object without one. Which rule shall be applied in this case? Several strategies, called *meta policies*, are possible to resolve conflicts between rules. A meta policy is specified by mappings, on the rule names, that define a set of rules intended for addition or deletion from the integrated framework SP (the "gluing" of SP_1 and SP_2).

Definition 6 (meta policy). *A* meta policy $MP = (strategy_i)_{i \in I}$ *for* SP *is an* I*-indexed set of partial mappings* $strategy_i : D_i \to |\mathbf{Rules}|$ *from a set* $D_i \subseteq P_1 \cup P_2$ *to the set of rules* $|\mathbf{Rules}|$.

A mapping $strategy : D \to |\mathbf{Rules}|$ of a meta policy changes the framework SP with respect to the rules by deleting all rules from SP that occur in D and by adding the rules in the image $strategy(D)$. After a framework is modified for all mappings in a meta policy MP, it is called *closed under* MP.

Definition 7 (closure under meta policy). *Let* SP *be the integrated framework for* SP_1 *and* SP_2 *and* $MP = (strategy_i : D_i \to |\mathbf{Rules}|)_{i \in I}$ *a meta policy for* SP. *The* framework $SP^{MP} = (TG^{int}, (P^{MP}, r_{P^{MP}}), Pos^{int}, Neg^{int})$ closed under MP *consists of the set of rules names*

$$P^{MP} = P^{int} \setminus \bigcup_{i \in I} D_i \cup \bigcup_{i \in I} dom(strategy_i).$$

The mapping $r_{P^{MP}} : P^{MP} \to |\mathbf{Rules}|$ *is defined for* $p \in P^{int}$ *as* $r_{P^{MP}} = r_{P^{int}}(p)$ *and for* $p \not\in P^{int}$ *as* $r_{P^{MP}}(p) = strategy_i(p)$, *where* $p \in dom(strategy_i)$.

For the following examples we introduce the *weakrule* $p_2(p_1)$ of a rule p_2 w.r.t. a rule p_1. The weakrule is derived from rule p_2 by adding a negative application condition $WAP(p_2, p_1)$, that allows the application of the rule p_2 only if the rule p_1 is not applicable. The application condition $WAP(p_2, p_1)$ is constructed by extending the left-hand side L_2 of p_2 to $L_1 \cup L_2$.

Definition 8 (weak rule). *Given two rules* $p_1 : L_1 \xrightarrow{r_1} R_1$ *and* $p_2 : L_2 \xrightarrow{r_2} R_2$, *the* weak condition *for* p_2 *w.r.t.* p_1, *denoted by* $WAP(p_2, p_1)$, *is the pair* $(L_2, L_1 \cup L_2)$. *The* weakrule $p_2(p_1)$ *of* p_2 *w.r.t.* p_1 *is the rule* p_2 *extended by the negative application condition* $WAP(p_2, p_1)$.

Example 7 (weak rule). Figure 10 shows the example of the ACL rule `create object` and the LBAC rule `new object` (note that the rule LBAC rule is typed over the integrated type graph, with a Px node instead of a Sx node). The weak rule for `create object` w.r.t. `new object` has a NAC that forbids the application to a process with a security level. Therefore, the weak rule for `create object` is only applicable to processes coming from the ACL model and without a counterpart in the LBAC model. The weak rule for `new object` w.r.t. `create object` has a NAC that forbids the application if a user is connected to the process. Since each user is connected to a process, the rule is not applicable.

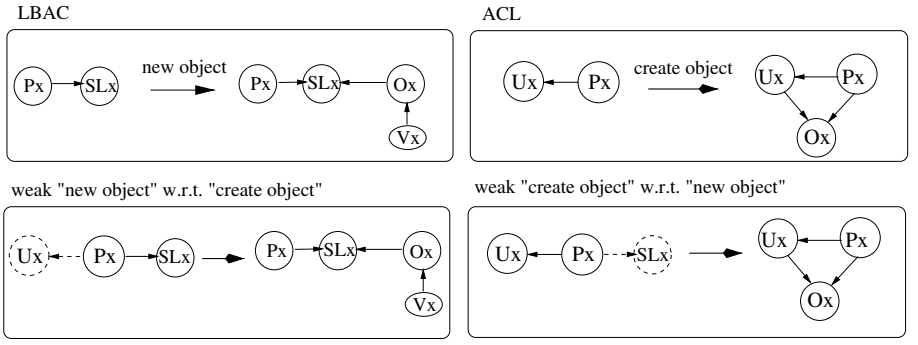


Fig. 10. The weak rule for `new object` and `create object`.

The weak rule ensures that the less important (minor) rule of SP_2 is applied only if the more important (major) rule of SP_1 is not applicable.

Lemma 1. *The weak rule $p_2(p_1)$ is applicable at a match $m : L_2 \to G$ if and only if the morphism m cannot be extended to a match for p_1.*

We introduce three possible meta policies, called *radical*, *weakRadical* and *static*. The meta policies *radical* and *weakRadical* choose a major SP_1 and a minor policy SP_2. The meta policy *radical* solves the problem of conflicting rules globally by selecting the rules of SP_1 and deleting the rules of SP_2. It consists of the mapping $killMinor : P_2 \to |\mathbf{Rules}|$ defined as identity for the rules of SP_2 not in a conflict with a rule in SP_1.

$$killMinor(p_2) = \begin{cases} \text{undefined} & \text{, if there is } (p_1, p_2) \in CR(SP) \\ p_2 & \text{, otherwise} \end{cases}$$

The effect of the meta policy *radical* is that SP_1 will "survive" during the subsequent evolution of the system, whereas the framework SP_2 "dies". The meta policy *weakRadical* keeps the conflicting rule of SP_2, extended by all NACs of the weak rule w.r.t. conflicting rules, and consists of the mapping $weakMinor : P_2 \to |\mathbf{Rules}|$ defined by

$$weakMinor(p_2) = p_2 \text{ with weak condition } WAP(p_1, p_2)\ \forall (p_1, p_2) \in CR(SP)$$

To non-conflict points, the weakened rule p_2 is still applicable so that the part of the system graph where the rule p_2 is applicable is only reduced.

Whereas the meta policies *radical* and *weakRadical* favor the major framwork SP_1, the choice in the meta policy *static* depends on the conflicting rule pair. The meta policy *static* consists of the two mappings $weakMinor : D_2 \to |\mathbf{Rules}|$ as above with a domain $D_2 \subseteq P_2$ and $weakMajor : D_1 \to |\mathbf{Rules}|$ with $D_1 \subseteq P_1$ similar to the mapping $weakMinor$, defined by

$$weakMajor(p_1) = p_1 \text{ with weak condition } WAP(p_2, p_1)\ \forall (p_1, p_2) \in CR(SP)$$

The meta policy *weakRadical* is a special case of the meta policy *static*, where $D_2 = P_2$ and $D_1 = \emptyset$.

Proposition 5 (conflictfreenes). *The closure of the framework SP under the metapolicies radical or weakRadical is conflictfree. If $D_1 \cap D_2 = \emptyset$ and $D_1 \cup D_2 = \{p, p' | (p, p') \in CR(SP)\}$, then the closure under static is conflictfree.*

6 Concluding Remarks

We have presented a formalism to specify AC policies. States are represented by graphs and their evolution by graph transformations. A policy is formalized by four components: a type graph, positive and negative constraints (a declarative way of describing what is wanted and what is forbidden) and a set of rules (an operational way of describing what can be constructed). The change over time of a policy can described in terms of sequences of framework morphisms, corresponding to step-by-step addition/deletion of rules and constraints.

We have also discussed the effect of integrating two policies using a pushout in the category of policy frameworks and framework morphisms. The problem of dealing with inconsistencies caused by conflicts between a rule of one policy and a constraint of the other policy has been addressed in part elsewhere [KMPP00b, KMPP00a], where it is also shown the adequacy of this framework to represent different Access Control policies.

Besides the new results in Sections 3 and 4, we have introduced the notion of metapolicy and shown that three natural ones transform a policy resulting from an integration into a conflict-free policy. The choice of the appropriate meta policy may depend on the specific application domain of the particular AC model. Among the problems still under investigation are the transition from a system using one policy to a system using another policy.

References

[BdVS00] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proc. of the 7th ACM Conference on Computer and Communication Security*. ACM, November 2000.

[CELP96] A. Corradini, H. Ehrig, M. Löwe, and J. Padberg. The category of typed graph grammars and their adjunction with categories of derivations. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science*, number 1073 in LNCS, pages 56–74. Springer, 1996.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations*, chapter Algebraic Approaches to Graph Transformation Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In Rozenberg [Roz97], 1997.

[GRPPS98] M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Spatial and Temporal Refinement of Typed Graph Transformation Systems. In *Proc. of MFCS'98*, number 1450 in LNCS, pages 553–561. Springer, 1998.

[HW95] R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars - a constructive approach. In *Proc. SEGRAGRA'95 Graph Rewriting and Computation*, number 2. Electronic Notes of TCS, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[KMPP00a] M. Koch, L. V. Mancini, and F. Parisi-Presicce. On the specification and evolution of access control policies. Technical Report SI-2000/05, Dip.Scienze dell'Informazione, Uni. Roma La Sapienza, May 2000.

[KMPP00b] M. Koch, L.V. Mancini, and F. Parisi-Presicce. A Formal Model for Role-Based Access Control using Graph Transformation. In F.Cuppens, Y.Deswarte, D.Gollmann, and M.Waidner, editors, *Proc. of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*, number 1895 in LNCS, pages 122–139. Springer, 2000.

[PPEM87] F. Parisi-Presicce, H. Ehrig, and U. Montanari. Graph Rewriting with unification and composition. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Int. Workshop on Graph Grammars and their Application to Computer Science*, number 291 in LNCS, pages 496–524. Springer, 1987.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations*. World Scientific, 1997.

[San93] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[San98] R. S. Sandhu. Role-Based Access Control. In *Advances in Computers*, volume 46. Academic Press, 1998.

[SS94] R.S. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Communication Magazine*, pages 40–48, 1994.

Categories of Processes Enriched in Final Coalgebras

Sava Krstić[1], John Launchbury[1], and Duško Pavlović[2]

[1] Oregon Graduate Institute {krstic,jl}@cse.ogi.edu
[2] Kestrel Institute dusko@kestrel.edu

Abstract. Simulations between processes can be understood in terms of coalgebra homomorphisms, with homomorphisms to the final coalgebra exactly identifying bisimilar processes. The elements of the final coalgebra are thus natural representatives of bisimilarity classes, and a denotational semantics of processes can be developed in a final-coalgebra-enriched category where arrows are processes, canonically represented.
In the present paper, we describe a general framework for building final-coalgebra-enriched categories. Every such category is constructed from a multivariant functor representing a notion of process, much like Moggi's categories of computations arising from monads as notions of computation. The "notion of process" functors are intended to capture different flavors of processes as dynamically extended computations. These functors may involve a computational (co)monad, so that a process category in many cases contains an associated computational category as a retract. We further discuss categories of resumptions and of hyperfunctions, which are the main examples of process categories. Very informally, the resumptions can be understood as computations extended in time, whereas hypercomputations are extended in space.

1 Introduction

A map $A \times X \longrightarrow B \times X$ can be construed as a function from A to B *extended in time*, represented by a set X of states. For each state in X and each input from A, such a function gives an output in B, and the next state in X. This is the usual representation of a *transducer*.

A map $A^X \longrightarrow B^X$, on the other hand, can be construed as a function from A to B *extended in space*, represented by a set X of storage cells. For each assignment of A-values to all X-cells, such a function gives an assignment of B-values to the same cells.

By transposition, a transducer $A \times X \longrightarrow B \times X$ can equivalently be viewed

- as a Kleisli morphism $A \longrightarrow (B \times X)^X$ for the monad $B \mapsto (B \times X)^X$, or
- as a coalgebra $X \longrightarrow (B \times X)^A$ for the functor $X \mapsto (B \times X)^A$.

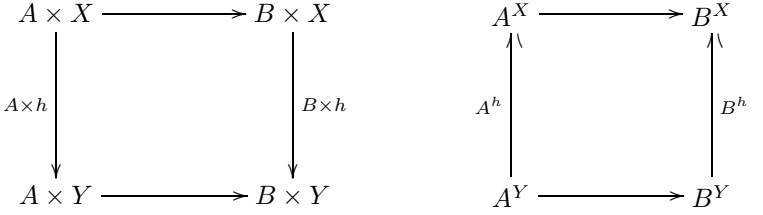
Similarly, a *function with storage* $A^X \longrightarrow B^X$ can be viewed

- as a Kleisli morphism $A^X \times X \longrightarrow B$ for the comonad $A \mapsto A^X \times X$, or

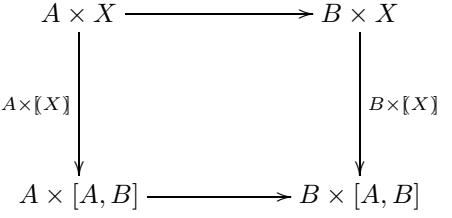
F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 303–317, 2001.

– as a coalgebra $X \longrightarrow B^{A^X}$ for the functor $X \mapsto B^{A^X}$.

In both cases, there is a double category [Gra74] displaying the structure and behaviour as the horizontal and the vertical arrows respectively. It has pairs $\langle A, X\rangle$ as objects, where the component A is to be thought of as a data type, and X as a state space. The horizontal arrows represent computations, captured as transducers (functions with storage), or by transposition as the Kleisli morphisms for the corresponding (co)monad. By the other transposition, they become coalgebras, and the vertical arrows arise as the coalgebra homomorphisms between them. They capture dynamics of the computations. The double morphisms are thus, respectively, the commutative squares of the form



This picture can be relativized, in favorable situations, over computational monads, or comonads. The problem with it, however, is a lot of redundancy, since arbitrary sets as state spaces provide a very loose representation of computational behavior. To pin down the computational semantics in the case of transducers, where many of them can be computationally equivalent, one usually seeks a way to construct *minimal representatives*. Recently, coalgebra has been proposed as a uniform framework for such constructions, where minimal representatives are obtained as elements of final coalgebras [Acz88,TR98,JR97]. Given a transducer $A \times X \longrightarrow B \times X$, there is a unique homomorphism $[\![X]\!]$ to the final coalgebra $[A, B]_R$ of the functor $RX = (B \times X)^A$



It turns out that two states $x \in X$ and $y \in Y$ (Y being the state space of another transducer $A \times Y \longrightarrow B \times Y$) lead to equivalent computations if and only if $[\![X]\!](x) = [\![Y]\!](y)$.

These remarks can be repeated for functions with storage: the stores $x \in X$ and $y \in Y$ can be reasonably assumed to be computationally indistinguishable if and only if their images in the final coalgebra $[A, B]_H$ of the functor $HX = B^{A^X}$ are equal.

For historical perspective, we note that methods of representing processes in categories by coalgebra-like structures go back at least to the systems theory work of Arbib and Manes in the late sixties [Arb66,AM74]. The idea that

minimal representatives of processes can be viewed and obtained as elements of final coalgebras has probably originated in Aczel's work [Acz88] on semantics of concurrency in terms of hypersets, which form final coalgebras of the powerset functor.

Elements of $[A, B]_R$ are known as *resumptions*. Composition of transducers induces composition of resumptions, and resumption domains are the arrow sets of a category. The observation that process categories are final coalgebra-enriched was developed in an unpublished joint work of Abramsky and the third author; some instances are given in [Abr96b]. On the other hand, *hyperfunction domains* $[A, B]_H$ also form a category, as recently discovered in [LKS00]. While double categories present both structural (Kleisli) and behavioral (coalgebra) aspect of transducers and functions with storage, by passing to final coalgebras we extract the canonical behaviors while preserving the structural aspect. The (one-dimensional) categories of resumptions and hyperfunctions provide a more convenient ambient than the double categories from which they are distilled.

Following the idea that minimal representatives of processses are elements of final coalgebras, in the next section we describe an abstract situation when final coalgebras for a family of functors yield a category, *viz.* its hom-sets. This is our general framework of the *coalgebra enriched categories*. It applies to the endofunctors $RX = (B \times X)^A$ and $HX = B^{A^X}$, and yields, respectively, the categories of resumptions and hyperfunctions; their morphisms can be thought of as (abstract, computational) functions extended in time, or space. The same construction applies, however, to a wide class of more general endofunctors, involving, for example, various notions of computation captured by computational monads and comonads [Mog91,BG92]. Notably, the functors

$$R_M XAB = (M(B \times X))^A \qquad \text{and} \qquad H_G XAB = B^{G(A^X)}$$

considered in Section 3, for suitable monads M and comonads G lead to coalgebra enriched categories of computations extended in time, or space.

In Section 4, we specialize to hyperfunctions and discuss the question of equivalence of coinductive definitions of hyperfunction operations with the recursive definitions given in the *Haskell* package of [LKS00]. This turned out to be a surprisingly difficult problem, so we limit ourselves to a brief discussion, leaving out the proofs.

2 Final coalgebra enrichment

We represent a notion of process as a functor, or type constructor, T, which for types X, A and B yields the type $TXAB$ of computations with inputs of type A and outputs of type B, parametrized by a *process type* X representing the process state. The process type can be understood as a space of states or storage cells, as described in the previous section, or, more generally, it can be any kind of space in which some computational, or even physical processes can be analyzed. Processes can then be presented as coalgebras

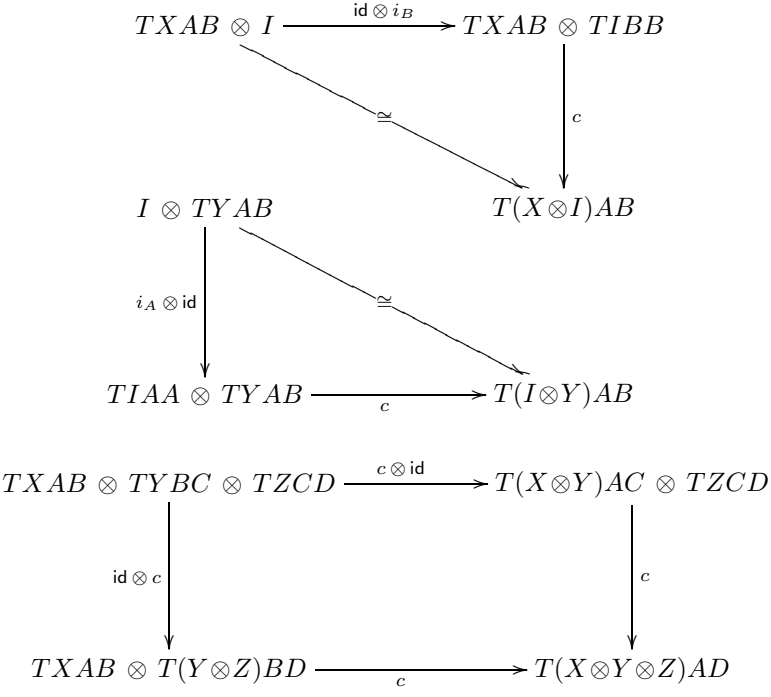
$$X \longrightarrow TXAB$$

assigning to each point of space a computation, involving the inputs from A, the outputs from B, as well as some further elements of X. For example, when X is the space of states, the computation yields the next states; when X is the set of storage cells, the computation may use and update their contents. Processes are thus viewed as computations extended through the process space X, which may be viewed as an abstraction of the temporal, or of the spatial dimension, in various ways.

Definition 1. *Let $\langle \mathcal{C}, \otimes, I \rangle$ be a monoidal category. A* notion of process *over $\mathcal{C}$ is a functor $T\colon \mathcal{C} \times \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \longrightarrow \mathcal{C}$ equipped with extranatural families of maps*

$$i_A : I \longrightarrow TIAA$$
$$c_{XYABC} : TXAB \otimes TYBC \longrightarrow T(X \otimes Y)AC$$

satisfying the conditions



(For unexplained terminology, the reader is referred to [Kel82].)

The idea behind the structure attached to each notion of process T is that

- for each type A, i_A denotes the identity process, which simply outputs its inputs, over the trivial process space I;
- for all types A, B and C, c_{XYABC} sequentially composes the processes from A to B over the space X with the processes from B to C over Y; it hides the intermediary data from B, and produces processes from A to C, over the product space $X \otimes Y$.

The imposed commutativity conditions pin down these intended meanings.

Examples of process notions are given by the functors R_M and H_G defined in the Introduction; see the next section.

As we already mentioned, the representation of processes as coalgebras is loose: many different coalgebras may represent behaviorally equivalent processes, indistinguishable in terms of their computational interpretation. This motivates a whole branch of research on the various notions of simulation, bisimulation, observational equivalence etc. The role of coalgebra in this realm is well known [TR98,Rut96]. The minimal representatives of coalgebras with respect to the canonical notions of behavior are usually captured as the elements of the final coalgebras.

So one naturally tries to use the final process coalgebras as the hom-sets of (canonically represented) processes. And indeed, the structure of the notion of process gives rise to the category structure upon such hom-sets.

Theorem 1. *If T is a notion of process over a category $\mathcal{C}$ and if final $T(-)AB$-coalgebras, denoted $[A,B]$, exist for every A,B, then there exists a $\mathcal{C}$-enriched category $\mathcal{P}$ whose objects are the same as the objects of $\mathcal{C}$ and whose hom-objects are the coalgebras $[A,B]$.*

The category $\mathcal{P} = \mathcal{P}_T$ is called the process category *induced by the notion of process T.*

Proof. We will use the notation $\xi = \xi_{AB}$ for the final coalgebras $[A,B] \longrightarrow T[A,B]AB$.

The map i_A is a $T(-)AA$-coalgebra, so we define the $\mathcal{P}$-identity on A as the anamorphism $u_A \colon I \longrightarrow [A,A]$ induced by i_A.

To define the $\mathcal{P}$-composition

$$k_{ABC} : [A,B] \otimes [B,C] \longrightarrow [A,C]$$

we use the composite

$$\begin{aligned} \kappa_{ABC} \;:\; [A,B] \otimes [B,C] &\xrightarrow{\xi \otimes \xi} T[A,B]AB \otimes T[B,C]BC \\ &\xrightarrow{c} T([A,B] \otimes [B,C])AC \end{aligned}$$

It is a $T(-)AC$-coalgebra on $[A,B] \otimes [B,C]$, and k_{ABC} is defined as the anamorphism induced by it.

To prove the associativity of k, it suffices to show that the maps occurring in the diagram

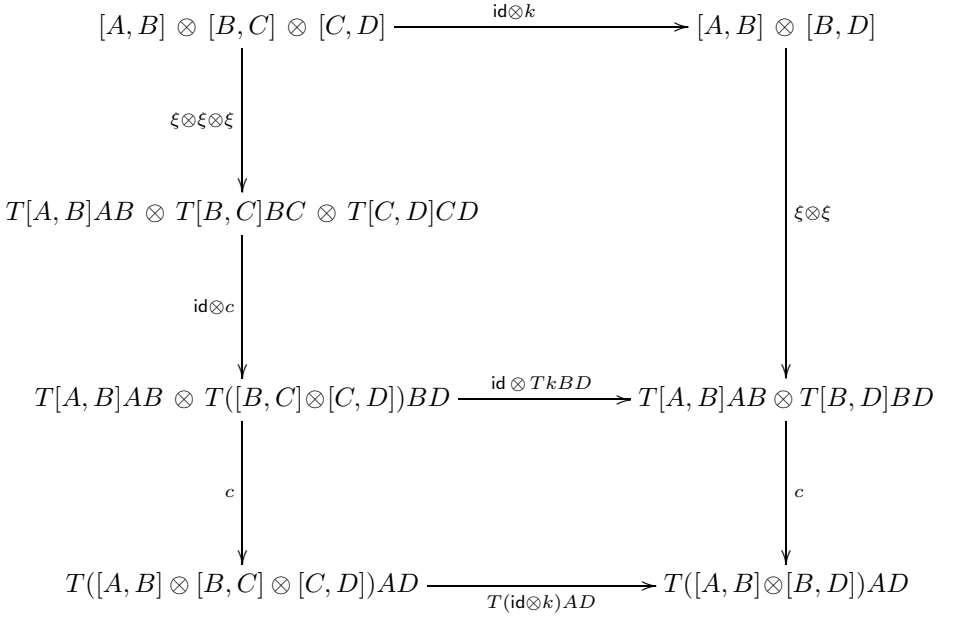
$$\begin{array}{ccc} [A,B] \otimes [B,C] \otimes [C,D] & \xrightarrow{\mathsf{id} \otimes k} & [A,B] \otimes [B,D] \\ \Big\downarrow{\scriptstyle k \otimes \mathsf{id}} & & \Big\downarrow{\scriptstyle k} \\ [A,C] \otimes [C,D] & \xrightarrow[k]{} & [A,D] \end{array}$$

are actually maps of $T(-)AD$-coalgebras. The diagram will then commute because $[A, D]$ is final. The two maps labelled k are k_{ABD} and k_{ACD}, and they are coalgebras by definition. Thus, we only need to define a $T(-)AD$-coalgebra κ_{ABCD} on $[A, B] \otimes [B, C] \otimes [C, D]$ and show that the maps $k_{ABC} \otimes \mathsf{id}$ and $\mathsf{id} \otimes k_{BCD}$ of the last diagram are coalgebra maps from κ_{ABCD} to κ_{ACD} and κ_{ABD} respectively.

We define κ_{ABCD} as the composite

$$\begin{aligned} \kappa_{ABCD} \;:\; & [A,B] \otimes [B,C] \otimes [C,D] \;\longrightarrow \\ & \xrightarrow{\xi \otimes \xi \otimes \xi} T[A,B]AB \otimes T[B,C]BC \otimes T[C,D]CD \\ & \xrightarrow{k'} T([A,B] \otimes [B,C] \otimes [C,D])AD \end{aligned}$$

where κ' is the diagonal of the rectangle expressing the associativity condition $c \circ (c \otimes \mathsf{id}) = c \circ (\mathsf{id} \otimes c)$ in Definition 1, with X, Y, Z instantiated with $[A, B]$, $[B, C]$, $[C, D]$ respectively. Thus, κ_{ABCD} is the left column map in the diagram



while κ_{ABD} is seen as the right column map. The diagram commutes because the two constituent diagrams commute: the top pentagon by definition of k_{ABD} and the bottom rectangle by the naturality of c.

An analogous diagram shows that $k \otimes \mathsf{id}$ is a coagebra map from κ_{ABCD} to κ_{ACD}, and this finishes the proof of associativity of the $\mathcal{P}$-composition k.

The unit axioms for u_A are checked in a similar fashion. Finally, the naturality of $\mathcal{P}$-composition and its units follows from extranaturality of families i_A and c_{XYABC}.

3 Examples: state and storage enrichment

Although the definition of the notion of process, and the general construction of Theorem 1 only require the monoidal structure on $\mathcal{C}$, our main motivating examples arise in the framework of *autonomous*, i.e. closed symmetric monoidal categories. In fact, the most interesting situations, some of which are referred to in the propositions below, are supported by the *cartesian* closed structure. The symbols $\multimap$ and $\Rightarrow$ will denote cotensor operations in autonomous and cartesian closed categories respectively.

3.1 Static process categories

For every monad M over an autonomous category $\mathcal{C}$, the Kleisli category $\mathcal{K}_M$ (the category of *M-computations* in the sense of Moggi [Mog91]) can be viewed as the process category $\mathcal{P}_T$ for the functor

$$T_M XAB = A \multimap MB$$

This is, of course, a degenerate process category: its hom-objects are the final coalgebras of functors constant in X. Indeed, if processes are construed as computations extended in time or space, then static processes boil down to mere computations, and the corresponding process categories $\mathcal{P}_{T_M}$ to the categories of computations $\mathcal{K}_M$.

More generally, there is a *static* process category corresponding to every functor $E\colon \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{C}$ that enriches $\mathcal{C}$ over itself; the requisite notion of process is $TXAB = EAB$.

Every process notion T has an associated static one, T_{st}, defined by $T_{st}XAB = TIAB$. Its final coalgebras are $[A,B]_{st} = TIAB$, and by precomposing the map $T\,!\,\mathsf{id}\,\mathsf{id}\colon T[A,B]AB \longrightarrow [A,B]_{st}$ with $\xi\colon [A,B] \longrightarrow T[A,B]AB$ we define

$$\mathsf{proj}\colon [A,B] \longrightarrow [A,B]_{st}$$

which, as one can easily check, is an identity-on-objects functor

$$\mathsf{proj}\colon \mathcal{P}_T \longrightarrow \mathcal{P}_{T_{st}}.$$

In the most interesting examples, the static process category is some category of computations. Since computations can be treated as (static) processes, proj will usually be a retraction.

3.2 Resumptions

Every strong monad M over an autonomous category $\mathcal{C}$ with enough final coalgebras defines a category $\mathcal{R}_M$ of *M-resumptions*. By definition, $\mathcal{R}_M$ is the proces category $\mathcal{P}_{R_M}$ induced by the notion of process consisting of the functor $R_M : \mathcal{C} \times \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \longrightarrow \mathcal{C}$, defined by

$$R_M XAB = A \multimap M(X \otimes B)$$

and the transformations

$$\begin{array}{lll} i: & I & \longrightarrow A \multimap M(I \otimes A) \\ c: & \left.\begin{array}{l} A \multimap M(X \otimes B) \otimes \\ B \multimap M(Y \otimes C) \end{array}\right\} & \longrightarrow A \multimap M(X \otimes Y \otimes C) \end{array}$$

obtained by transposing, respectively,

- the monad unit $I \otimes A \longrightarrow M(I \otimes A)$, and
- the composite

$$\begin{aligned} & A \otimes (A \multimap M(X \otimes B)) \otimes (B \multimap M(Y \otimes C)) \\ & \xrightarrow{(1)} M(X \otimes B) \otimes (B \multimap M(Y \otimes C)) \\ & \xrightarrow{(2)} M\,(X \otimes B \otimes (B \multimap M(Y \otimes C))) \\ & \xrightarrow{(3)} M\,(X \otimes M(Y \otimes C)) \\ & \xrightarrow{(4)} MM\,(X \otimes Y \otimes C) \\ & \xrightarrow{(5)} M\,(X \otimes Y \otimes C) \end{aligned}$$

  where maps (1) and (3) are derived from the evaluation $U \otimes (U \multimap V) \longrightarrow V$, (2) and (4) from the tensorial strength $U \otimes MV \longrightarrow M(U \otimes V)$, and (5) is a component of the monad cochain $MM \longrightarrow M$.

The fact that this structure satisfies the conditions of Definition 1 can be checked by a routine, though lengthy diagram chase.

When $\mathcal{C}$ is the category of sets and M is the power set functor, we obtain the classical example of resumptions: $[A, B]$ is the set of $A \times B$-labelled hypersets (synchronization trees) [Acz88], and $\mathcal{R}_M$ fully embeds in the category SProc of [Abr96a].

Proposition 1. *If $\mathcal{C}$ is* cartesian *closed and the category $\mathcal{R}_M$ of M-resumptions exists, then the category $\mathcal{K}_M$ of M-computations is a retract of $\mathcal{R}_M$.*

Proof. Since $R_M IAB = A \Rightarrow M(I \times B) \cong A \Rightarrow MB$, the static process category associated with $\mathcal{R}_M$ is indeed $\mathcal{K}_M$; we need maps

$$\mathsf{lift}\colon (A \Rightarrow MB) \longrightarrow [A, B]$$

that split $\mathsf{proj}\colon [A, B] \longrightarrow (A \Rightarrow MB)$.

Since $[A, B] = [A, B]_{R_M}$ is the final coalgebra for the functor $R_M(-)AB$, we can define lift as the anamorphism for the coalgebra on $A \Rightarrow MB$ obtained by transposing the composite

$$A \times (A \Rightarrow MB) \xrightarrow{\langle \epsilon, \pi' \rangle} MB \times (A \Rightarrow MB) \xrightarrow{\vartheta} M(B \times (A \Rightarrow MB))\,,$$

where ε is evaluation and ϑ is tensorial strength.

It remains to prove $\mathsf{proj} \circ \mathsf{lift} = \mathsf{id}$, but this follows from the definitions.

Viewing functions as degenerate computations, i.e. the base category $\mathcal{C}$ as the Kleisli category $\mathcal{K}_{\mathsf{Id}}$, we derive that $\mathcal{C}$ is a retract of $\mathcal{R} = \mathcal{R}_{\mathsf{Id}}$.

3.3 Hyperfunctions and hypercomputations

The *G-hypercomputations* are the arrows of the process category $\mathcal{H}_G$ induced by the notion of process consisting of the functor $H_G : \mathcal{C} \times \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \longrightarrow \mathcal{C}$ given on objects by

$$H_G XAB = G(X \multimap A) \multimap B$$

and the transformations

$$\begin{array}{lll} i: & I & \longrightarrow G(I \multimap A) \multimap A \\ c: & \left.\begin{array}{l} G(X \multimap A) \multimap B \otimes \\ G(Y \multimap B) \multimap C \end{array}\right\} & \longrightarrow G\,(X \otimes Y \multimap A) \multimap C \end{array}$$

where G is a comonad possessing a tensorial strength $A \otimes GB \longrightarrow G(A \otimes B)$ and a *cotensorial strength*

$$(GA \multimap B) \longrightarrow G(A \multimap B).$$

Space constraints do not allow us to elaborate on the non-standard notion of cotensorial strength; the coherence conditions that need to be imposed on it are analogous to the well-known ones for the tensorial strength. Computationally, tensorial strength allows variables to enter and exit computations, whereas the cotensorial strength ensures the same for the abstraction.

Of course, just like functions are degenerate computations relative to the identity monad, *hyperfunctions* are degenerate hypercomputations, obtained by omitting G in the above definition.

The transformations i and c are obtained by transposing

- the counit $G(I \multimap A) \longrightarrow (I \multimap A)$, and
- the composite

$$\begin{array}{l} G(X \otimes Y \multimap A) \otimes (G(X \multimap A) \multimap B) \otimes (G(Y \multimap B) \multimap C) \\ \xrightarrow{(1)} G\,(Y \multimap (X \multimap A)) \otimes (G(X \multimap A) \multimap B) \otimes (G(Y \multimap B) \multimap C) \\ \xrightarrow{(2)} G((Y \multimap (X \multimap A)) \otimes ((X \multimap A) \multimap B)) \otimes (G(Y \multimap B) \multimap C) \\ \xrightarrow{(3)} G(Y \multimap B) \otimes (G(Y \multimap B) \multimap C) \\ \xrightarrow{(4)} C \end{array}$$

  where cotensorial strength and currying are used for (1), tensorial strength for (2), and evaluation for (3) and (4).

Checking that the requirements of a process notion are satisfied is again routine, but this time fairly tedious.

Proposition 2. *If $\mathcal{C}$ is* cartesian *closed and the category $\mathcal{H}_G$ of G-hypercomputations exists, then the category $\mathcal{K}_G$ of G-computations is a retract of $\mathcal{H}_G$.*

Proof. The static process category associated with $\mathcal{H}_G$ is $\mathcal{K}_G$. The injection

$$\mathsf{lift}\colon (GA \Rightarrow B) \longrightarrow [A, B]$$

can be defined as the anamorphism for the coalgebra on $GA \Rightarrow B$, obtained by transposing the composite

$$\begin{aligned} & G((GA \Rightarrow B) \Rightarrow A) \times (GA \Rightarrow B) \\ & \xrightarrow{\langle \pi', \vartheta \rangle} (GA \Rightarrow B) \times G((GA \Rightarrow B) \times ((GA \Rightarrow B) \Rightarrow A)) \\ & \xrightarrow{\mathsf{id} \times G\epsilon} (GA \Rightarrow B) \times GA \\ & \xrightarrow{\epsilon'} B \end{aligned}$$

Again, $\mathsf{proj} \circ \mathsf{lift} = \mathsf{id}$ has a straightforward proof.

As a corollary (when $G = \mathsf{Id}$) we obtain that the base category $\mathcal{C}$ is a retract of the category $\mathcal{H}$ of hyperfunctions.

4 Hyperfunctions recursively

Hyperfunction types can be easily defined in a programming language which allows recursive definitions of data types (*Haskell*, for example). Any programming with them, however, would require using the basic operators like hyperfunction units, composition, and the lifting function. We have defined all of them as anamorphisms and it is not clear how such definitions can translate into a convenient programming language. The question is really not so much of translating, but of matching, for all basic hyperfunction operations were introduced in [LKS00] by means of recursive definitions, and we now have a problem of showing that each recursive definition is equivalent to the corresponding coalgebraic one.

Of course, the recursive definitions take place in a more specific setting provided by a category of domains. Without being too specific, let us assume that our base category is one of pointed domains, suitable for modeling recursive types by standard bilimit construction [AJ94]. The notation $[A, B]$ is now for the canonical solution of the equation $X \cong (X \Rightarrow A) \Rightarrow B$. By the Bekić Lemma, the pair of domains $[A, B]$ and $[B, A]$ must be the canonical solution to the system of equations

$$\begin{aligned} X &\cong Y \Rightarrow B \\ Y &\cong X \Rightarrow A. \end{aligned}$$

Consequently, there is a (canonical) isomorphism $[A, B] \longrightarrow ([B, A] \Rightarrow B)$ whose transpose

$$(\cdot) : [A, B] \times [B, A] \longrightarrow B$$

can be considered a *hyperfunction application operation*. Thus, any equation $u \cdot v = e$, where e is an expression of type B (assuming u and v are of types $[A, B]$ and $[B, A]$ respectively) defines a hyperfunction u. For example, for each b in B

there exists a "constant hyperfunction" k_b, defined by $k_b \cdot v = b$. More generally, for every function $f\colon A \longrightarrow B$ there exists a corresponding hyperfunction $\widetilde{f}$ recursively defined by

$$\widetilde{f} \cdot u = f(u \cdot \widetilde{f}). \tag{1}$$

Note that $\widetilde{f} \cdot k_a = f(a)$ so that $\widetilde{f}$ extends f in some sense, and the first question arises: *Does the equation (1) define the same function as* lift *of Proposition 2?*

Consider now the equation

$$\theta_A \cdot u = u \cdot \theta_A, \tag{2}$$

where both θ_A and u are in $[A, A]$. We can view the equation as a recursive definition of θ_A. It follows from (1) that $\widetilde{\mathsf{id}_A}$ satisfies it. But, are θ_A and $\widetilde{\mathsf{id}_A}$ equal? *Are they the same as the identity hyperfunction as defined in Section 3.3?*

Computing a little more with the use of (1),

$$\widetilde{f} \cdot \widetilde{g} = f(\widetilde{g} \cdot \widetilde{f}) = f(g(\widetilde{f} \cdot \widetilde{g}) = (f \circ g)(\widetilde{f} \cdot \widetilde{g}),$$

suggests that

$$\widetilde{f} \cdot \widetilde{g} = \mathsf{fix}(f \circ g),$$

but is it true? A special case is particularly interesting:

$$\mathsf{fix}(f) = \theta \cdot \widetilde{f}. \tag{3}$$

Does the least fixpoint operator coincide with the application of the unit hyperfunction?

Finally, let us look for a binary operation $\#\colon [B, C] \times [A, B] \longrightarrow [A, C]$ which behaves on the lifts of ordinary functions as the usual composition. In other words, we expect the equality $\widetilde{f} \,\#\, \widetilde{g} = \widetilde{f \circ g}$ to hold. The chain of equalities, some of them resting on the hypothetical equations above,

$$\begin{aligned}(\widetilde{f} \,\#\, \widetilde{g}) \cdot \widetilde{h} &= \widetilde{f \circ g} \cdot \widetilde{h} \\ &= \mathsf{fix}((f \circ g) \circ h) \\ &= \mathsf{fix}(f \circ (g \circ h)) \\ &= \widetilde{f} \cdot \widetilde{g \circ h} \\ &= \widetilde{f} \cdot (\widetilde{g} \,\#\, \widetilde{h})\end{aligned}$$

suggests the following recursive definition of $\#$:

$$(u \,\#\, v) \cdot w = u \cdot (v \,\#\, w). \tag{4}$$

Is this $\#$ associative? Do hyperfunctions θ behave like units for $\#$? Is $\#$ the same as the (reversed) hyperfunction composition, as defined in Section 3.3?

Theorems 2 and 3 below imply the answer yes to all our questions.

A hyperfunction package in *Haskell*

The above definitions can be coded in *Haskell*, with necessary modifications, as follows.

```
newtype H a b = Phi (H b a -> b)

(@) :: H a b -> H b a -> b
(Phi f) @ k = f k

konst :: a -> H b a
konst p = Phi (\k -> p)

(<<) :: (a -> b) -> H a b -> H a b
f << q = Phi (\k -> f (k @ q))

lift :: (a->b) -> H a b
lift f = f << lift f

proj :: H a b -> a -> b
proj f a = f @ (konst a)

self :: H a a
self = lift id

(#) :: H b c -> H a b -> H a c
f # g = Phi (\k -> f @ (g # k))

run :: H a a -> a
run f = f @ self
```

An explicit isomorphism, denoted `Phi`, is needed to identify the hyperfunction type `H a b` with the function type `H b a -> b`. Note that `@` is the application operator and `konst p` is a constant hyperfunction. The `(<<)` operator acts rather like a *cons* operator, taking a function element `f` and adding to the stack of functions `q` (if we can think of hyperfunctions as being stacks of functions, which is only partially true). `self` is the hyperfunction unit, and `run` is the application of it, so that `run (lift f)` is just `fix f`.

Hyperdomains

If the base category is cartesian closed with hyperfunction coalgebras, then there are natural maps

$$\phi_{AB} \colon ([B, A] \Rightarrow B) \longrightarrow [A, B]$$

defined as anamorphisms corresponding to the $T(-)AB$-coalgebras

$$(\xi_{BA}^{-1} \Rightarrow B) \colon ([B, A] \Rightarrow B) \;\longrightarrow\; ((([B, A] \Rightarrow B) \Rightarrow A) \Rightarrow B).$$

We will call our category a *category of hyperdomains* if ϕ_{AB} is an isomorphism for every A and B, and we will use the notation

$$p_{AB} \colon [B, A] \times [A, B] \longrightarrow B$$

for the transpose of the inverse of ϕ_{AB}. This is just the hyperfunction application operator, with a reversed order of arguments.

The application operator is all we need to give categorical formulation of the equations (1), (2) and (4) used in the domain setting to define the lifting function and the hyperfunction unit and composition. It turns out that no further assumptions are needed for proving that recursive definitions given by these equations are equivalent to the corresponding coalgebraic definitions.

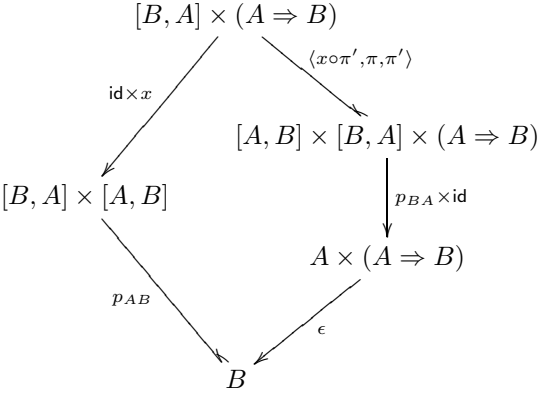


Diagram 1: Translation of
the equation $\widetilde{f} \cdot u = f(u \cdot \widetilde{f})$

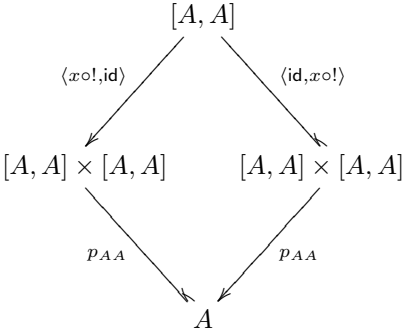


Diagram 2: Translation of
$\theta \cdot u = u \cdot \theta$.

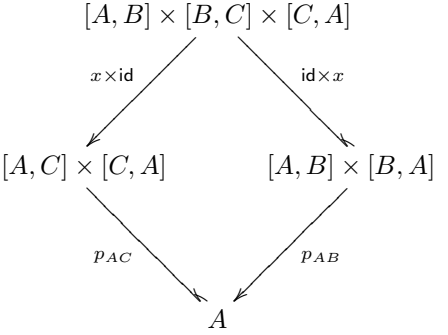


Diagram 3: Translation of
$(u \# v) \cdot w = u \cdot (v \# w)$.

The translations are given in Diagrams 1–3. Diagram 1 is an equation for $x \colon (A \Rightarrow B) \longrightarrow [A, B]$ and Diagram 2 is an equation for $x \colon I \longrightarrow [A, A]$. In

Diagram 3, however, the two occurrences of x refer to morphisms with different source and target. The unknown x here is a collection of functions $x_{ABC} \colon [A,B] \times [B,C] \longrightarrow [A,C]$, where A, B, C range over all objects. For each triple A, B, C, Diagram 3 expresses a relation between x_{ABC} and x_{BCA}. Due to the cyclic nature of the equation, this infinite system of equations partitions itself into systems of three equations in three unknowns x_{ABC}, x_{BCA} and x_{CAB}.

Theorem 2. *In any category of hyperdomains:*

(a) Diagram 1 commutes if and only if $x = \mathsf{lift}_{AB}$*;*
(b) Diagram 2 commutes if and only if $x = u_A$*;*
(c) Diagram 3 commutes (for every A, B, C*) if and only if* $x_{ABC} = k_{ABC}$ *(for every* A, B, C*).*

Turning to the remaining equation (3), we note first that the existence of fixpoints in a category of hyperdomains is granted by a result of Mulry [Mul99]: a fixpoint map $(A \Rightarrow A) \longrightarrow A$ exists whenever there is a retraction $X \longrightarrow (X \Rightarrow A)$, and we take $X = [A,A]$. This, of course, does not imply the equation (3), which we may rewrite as $\mathsf{fix} = Y$, where

$$Y_A \colon (A \Rightarrow A) \xrightarrow{\mathsf{lift}} [A,A] \xrightarrow{\langle \mathsf{id}, u_A \circ ! \rangle} [A,A] \times [A,A] \xrightarrow{p} A.$$

Theorem 3. *Over any category of hyperdomains, the operator* Y *is dinatural.*

For definition of dinaturality and the fact that all dinatual operators of type $(A \Rightarrow A) \longrightarrow A$ are necessarily fixpoint operators we refer to [BFSS90] and [SP00]. The equation (3) follows by combining Theorem 3 with a theorem of Simpson [Sim93] implying that the least fixpoint operator in standard categories of domains is the only dinatural fixpoint operator.

Another consequence of Theorem 3 is that the existence of hyperfunction coalgebras does not imply the isomorphisms $[A,B] \cong [B,A] \Rightarrow B$; a counterexample can be found using the PER model of polymorphism [Hyl88,LM91], as suggested by one of the referees.

5 Future work

We have initiated a study of the general structure of process categories. Under suitable conditions, there is premonoidal structure, others give fixpoints and recursion, and some induce the full framework for axiomatic domain theory. Finding such conditions precisely and determining their computational meaning seems to be a demanding but worthwhile task.

Special process categories deserve further study, starting with hyperfunctions. The recent example of hyperfunctions used as an instrumental datatype in a solution of a difficult programming problem [LKS00] suggests that the programming potential of process categories seems worth exploring as well.

References

[Abr96a] S. Abramsky. Interaction categories. In *Theory and Formal Methods '93*, Workshops in Computer Science, pages 57–70. Springer-Verlag, 1996.

[Abr96b] S. Abramsky. Retracing some paths in process algebra. In U. Montanari and V. Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 1–17, 1996.

[Acz88] P. Aczel. *Non-Well-Founded Sets*. CSLI Publications, 1988.

[AJ94] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, volume 3*. Clarendon Press, 1994.

[AM74] M. A. Arbib and E. G. Manes. Machines in a category: An expository introduction. *SIAM Review*, 16:163–192, 1974.

[Arb66] M. A. Arbib. A common framework for automata theory - a rapprochement. *Automatica*, 3:161–189, 1966.

[BFSS90] E. Bainbridge, P.J. Freyd, A. Scedrov, and P. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

[BG92] S. Brookes and S. Geva. Computational comonads and intensional semantics. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Categories in Computer Science*, pages 1–44. Cambridge University Press, 1992.

[Gra74] J. W. Gray. *Formal Category Theory: Adjointness for 2-categories*. Springer, 1974.

[Hyl88] M. Hyland. A small complete category. *Annals of Pure and Applied Logic*, 40:135–165, 1988.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

[Kel82] G. M. Kelly. *Basic Concepts of Enriched Category Theory*. Cambridge University Press, 1982.

[LKS00] J. Launchbury, S. Krstić, and T. E. Sauerwein. Zip fusion with hyperfunctions. Technical report, Oregon Graduate Institute, 2000. Preprint available on `http://www.cse.ogi.edu/~krstic`.

[LM91] G. Longo and E. Moggi. onstructive natural deduction and its 'omega-set' interpretation. *Mathematical Structures in Computer Science*, 1:215–254, 1991.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Mul99] P. S. Mulry. Categorical fixed-point semantics. *Theoretical Computer Science*, 118:301–314, 199.

[Rut96] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 1996.

[Sim93] A. Simpson. A characterization of the least-fixed-point operator by dinaturality. *Theoretical Computer Science*, 118:301–314, 1993.

[SP00] A. Simpson and G. Plotkin. Complete axioms for categorical fixed-point operators. In *15th Symposium on Logic in Computer Science (LICS 2000)*. IEEE Computer Society, 2000.

[TR98] D. Turi and J. J. M. M. Rutten. On the foundations of final coalgebra semantics: non-well-founded sets, partial orders, metric spaces. *Mathematical Structures in Computer Science*, 8:481–540, 1998.

Model Checking CTL$^+$ and FCTL Is Hard

François Laroussinie, Nicolas Markey, and Philippe Schnoebelen

Lab. Spécification & Vérification
ENS de Cachan & CNRS UMR 8643
61, av. Pdt. Wilson, 94235 Cachan Cedex France
email: {fl,markey,phs}@lsv.ens-cachan.fr

Abstract. Among the branching-time temporal logics used for the specification and verification of systems, CTL$^+$, FCTL and ECTL$^+$ are the most notable logics for which the precise computational complexity of model checking is not known. We answer this longstanding open problem and show that model checking these (and some related) logics is Δ_2^p-complete.

1 Introduction

Temporal Logic. Since [Pnu77], *temporal logic* is a widely used formalism for reasoning about reactive systems. Temporal logic allows *model checking*, i.e. the automatic verification that (a finite state model of) the system under study satisfies (the temporal formulae formalizing) its expected behavioral specifications. We refer to [Eme90,CGP99] for more motivations and background.

There exists a wide variety of different temporal logics, and it is still debated what should be the temporal logic of choice. However, it is fair to say that the three most popular temporal logics are PLTL, CTL and CTL*. PLTL is the linear-time logic built on U ("until") and X ("next") while CTL is the branching-time logic built on these same modalities (hence the notations PLTL = L(U,X) and CTL = B(U,X) in [Eme90]). CTL*, introduced in [EH86], was designed to be more expressive than both PLTL and CTL.

CTL *and fairness properties.* Several fragments of CTL* are defined and studied in [EH85,EH86,ES89,Eme90] and other papers, where their expressive powers are compared. Clearly, what CTL really lacks in practice is the ability to express fairness properties, and this is what motivates the introduction in [EH86] of ECTL [1], or B($\mathsf{U},\mathsf{X},\overset{\infty}{\mathsf{F}}$), an extension of CTL with the $\mathsf{E}\overset{\infty}{\mathsf{F}}$ modality for stating fairness conditions. ECTL sits between CTL and CTL* and, like CTL, it admits a polynomial-time model checking algorithm (while model checking CTL* is PSPACE-complete).

[1] For "Extended CTL". There are two standard ways of denoting the logics we consider in this paper: [ES89] and [EH83] use the names ECTL, ECTL$^+$, etc., while [EH86] and [Eme90] use the notation B(...). Here we use preferably the first series.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 318–331, 2001.

One thing ECTL lacks is the ability to *combine* fairness properties, and this is what motivated the introduction in [EH86] [2] of ECTL$^+$, where several temporal modalities can be combined in a boolean way (but not nested) under a path quantifier. Hence ECTL$^+$ allows stating $\mathsf{E}(\overset{\infty}{\mathsf{F}}A \wedge \overset{\infty}{\mathsf{F}}B)$, i.e. "there exists a path where A and B occur infinitely often", and $\mathsf{A}\big(\overset{\infty}{\mathsf{F}}A \;\Rightarrow\; B\mathsf{U}C\big)$, i.e. "all paths with infinitely many A satisfy $B\mathsf{U}C$". This makes ECTL$^+$ expressive enough in practical situations.

There exist other proposals aiming at extending CTL so that it can express fairness properties. These are FCTL, GFCTL (both from [EL87]), and CTL$^{\mathrm{F}}$ (from [CES86]), all of them logics where the fairness constraints are stated more or less outside of the temporal property itself (see section 2.4).

CTL *and* CTL$^+$. The idea of allowing boolean combinations of temporal modalities has also been applied to CTL (and other logics). In CTL$^+$ one can state $\mathsf{A}\big(\mathsf{G}C \wedge \mathsf{X}D \;\Rightarrow\; B\mathsf{U}C\big)$, i.e. "all paths with C everywhere and D in next state, satisfy $B\mathsf{U}C$".

A surprising result is that CTL$^+$ is not more expressive than CTL [EH85] while ECTL$^+$ is more expressive than ECTL [EH86] (see also [RS00]). However CTL$^+$ can be much more succinct than CTL, a fact that was conjectured since [EH85] but has only been proved recently [Wil99].

The complexity of model checking. That CTL$^+$ can be exponentially more succinct than CTL suggests that model checking can be harder for CTL$^+$ than for CTL. Indeed, while model checking CTL (or ECTL) is P-complete, model checking CTL$^+$ (or ECTL$^+$) is NP-hard and coNP-hard. This lower bound is a consequence of well-known results (from [ON80,SC85]) on the complexity of L(F). These same results entail that model checking CTL$^+$ can be done in P^{NP} (that is, in Δ_2^p, see section 3) as was observed in [CES86, Theo. 6.2]. Clearly, the same lower and upper bounds apply to FCTL and GFCTL.

Beyond these observations, nothing more is known about the complexity of model checking CTL$^+$, FCTL and ECTL$^+$, three notable branching-time logics for which the computational complexity of model checking has not been characterized precisely. Also note that [EL87, Coro. 4.8] incorrectly states that model-checking FCTL is NP-complete [3].

Our contribution. In this paper, we prove that model checking CTL$^+$, ECTL$^+$ and FCTL (and some related logics) is Δ_2^p-complete, thereby solving a long-standing open problem.

The result is surprising since Δ_2^p is a class for which very few complete problems are known. Indeed, in the polynomial-time hierarchy, the classes Σ_k^p or Π_k^p are more populated than the Δ_k^p. As far as we know, our result provides the first

[2] The logic called CTF in [EC80] is essentially ECTL$^+$.

[3] It seems that [EL87] implicitly assumes Turing or Cook reductions, instead of the usual many-one reductions. Turing reductions are too general for problems in NP and [EL87] does not prove membership in NP.

examples of Δ_2^p-complete problems from the field of temporal model checking and we believe it can be interesting outside of that field.

Plan of the paper. We first recall the necessary preliminary notions from temporal logic (section 2) and Δ_2^p-completeness (section 3). Sections 4 and 5 contain the main result, a reduction from SNSAT into model checking problems. Then section 6 shows that model checking for ECTL$^+$ is in Δ_2^p. Finally, a conclusion summarizes what has been proved.

2 Branching-Time Temporal Logic

2.1 Syntax

We write $\mathbb{N}$ for the set of natural numbers, and $AP = \{P_1, P_2, \dots\}$ for a countable set of *atomic propositions*.

The formulae of ECTL$^+$ are given by the following grammar:

$$\varphi, \psi \ ::= \ \mathsf{E}\varphi_p \mid \neg\varphi \mid \varphi \wedge \psi \mid P_1 \mid P_2 \mid \dots$$
$$\text{and} \ \ \varphi_p, \psi_p \ ::= \ \varphi \mathsf{U} \psi \mid \mathsf{X}\varphi \mid \overset{\infty}{\mathsf{F}}\, \varphi \mid \neg\varphi_p \mid \varphi_p \wedge \psi_p \mid P_1 \mid P_2 \mid \dots$$

where only *state formulae* (ranged over by $\varphi, \psi, \dots$) are considered as *bona fide* ECTL$^+$ formulae, while *path formulae* (ranged over by $\varphi_p, \psi_p, \dots$) only occur as subformulae.

We use the standard abbreviations $\top$, $\bot$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, as well as $\mathsf{A}\varphi_p$ (for $\neg\mathsf{E}\neg\varphi_p$), $\mathsf{F}\varphi$ (for $\top\mathsf{U}\varphi$), $\mathsf{G}\varphi$ (for $\neg\mathsf{F}\neg\varphi$) and $\overset{\infty}{\mathsf{G}}\, \varphi$ (for $\neg \overset{\infty}{\mathsf{F}}\, \neg\varphi$).

Remark 2.1. Classical definitions of CTL$^+$ and ECTL$^+$ do not allow atomic propositions $P_1, \dots$ as path formulae. We use such path formulae for clarity but will avoid them in the proof of our main hardness result. Hence all our results also hold with the restricted definition.

2.2 Semantics

ECTL$^+$ formulae are interpreted over states (also called nodes) in Kripke structures. Formally

Definition 2.2. *A Kripke structure (a "KS") is a tuple $S = \langle Q_S, q_0, R_S, l_S \rangle$ where $Q_S = \{q, \dots\}$ is a non-empty set of* nodes*, $R_S \subseteq Q_S \times Q_S$ is a total* transition relation*, and $l_S : Q_S \to 2^{AP}$ labels every node with the propositions it satisfies.*

We only consider finite KSs, i.e. KSs where Q_S and all $l_S(q)$ are finite. The size of a finite KS, written $|S|$, is defined as $|Q_S| + |R_S|$, i.e. the size of the underlying directed graph.

Below, we drop the "S" subscript in our notations whenever no ambiguity will arise. A *computation* (or a *path*) in a KS is an infinite sequence π of the

form $q_0 q_1 \ldots$ s.t. $(q_i, q_{i+1}) \in R$ for all $i \in \mathbb{N}$. For $i \in \mathbb{N}$, $\pi(i)$ denotes q_i, the i-th node of π. We write $\Pi(q)$ for the set of all computations starting from q. $\Pi(q)$ is never empty since R is total.

Fig. 1 defines when a node q (a path π) in some KS S, satisfies an ECTL$^+$ formula φ (resp. path formula φ_p), written $q \models_S \varphi$ (resp. $\pi \models_S \varphi_p$), by induction over the structure of the formulae. As usual, we write $S \models \varphi$ when $q_0 \models_S \varphi$.

$q \models \mathsf{E}\varphi_p$	iff there exists $\pi \in \Pi(q)$ s.t. $\pi \models \varphi_p$,
$q \models \neg\varphi$	iff $q \not\models \varphi$,
$q \models \varphi \wedge \psi$	iff $q \models \varphi$ and $q \models \psi$,
$q \models P_i$	iff $P_i \in l(q)$,
$\pi \models \varphi \mathsf{U} \psi$	iff there exists $i \geq 0$ s.t. $\pi(i) \models \psi$ and $\pi(j) \models \varphi$ for all $0 \leq j < i$,
$\pi \models \mathsf{X}\varphi$	iff $\pi(1) \models \varphi$,
$\pi \models \overset{\infty}{\mathsf{F}}\ \varphi$	iff for all $i \geq 0$ there is a $j > i$ s.t. $\pi(j) \models \varphi$,
$\pi \models \neg\varphi_p$	iff $\pi \not\models \varphi_p$,
$\pi \models \varphi_p \wedge \psi_p$	iff $\pi \models \varphi_p$ and $\pi \models \psi_p$,
$\pi \models P_i$	iff $P_i \in l(\pi(0))$.

Fig. 1. Semantics of ECTL$^+$

2.3 Fragments of ECTL$^+$

Several branching-time logics can be seen as fragments of ECTL$^+$:

- ECTL, denoted B(U, X, $\overset{\infty}{\mathsf{F}}$) in [Eme90], is the fragment of ECTL$^+$ where the path quantifiers E or A are immediately over a temporal modality U, X or $\overset{\infty}{\mathsf{F}}$ (no boolean combinator is allowed in between).
- CTL [CE81], or B(U, X), is the fragment of ECTL where $\overset{\infty}{\mathsf{F}}$ is not allowed.
- UB [BPM83], or B(X, F), is the fragment of CTL where U is only allowed in the weaker form of F.
- BTL [Lam80], or B(F), is the fragment of UB where X is not allowed.

All these logics can be extended so that boolean combinations of path formulae are allowed. [Eme90] denotes them by B($\ldots, \wedge, \neg$), so that ECTL$^+$ really is B(U, X, $\overset{\infty}{\mathsf{F}}$, $\wedge$, $\neg$). We let CTL$^+$, UB$^+$, BTL$^+$ denote the logics obtained by extending CTL, UB and BTL in the corresponding way. It is well known [EH85, EH86] that we have the following hierarchy:

$$\text{BTL} \begin{array}{c} < \\ \\ < \end{array} \begin{array}{c} \text{UB} \\ \\ \text{BTL}^+ \end{array} \begin{array}{c} < \\ \\ < \end{array} \text{UB}^+ \ < \ \text{CTL} \ = \ \text{CTL}^+ \ < \ \text{ECTL} \ < \text{ECTL}^+$$

where $L < L'$ means that L' is strictly more expressive than L, and $L = L'$ means that L and L' have the same expressive power.

2.4 CTL with Fairness

ECTL$^+$ is not the only logic where one can mix CTL formulae with fairness constraints, but other proposals can all be seen as fragments of ECTL$^+$:

- GFCTL [EL87] is CTL where every path quantifier is indexed with a fairness constraint. One write $\mathsf{E}_{\Phi}\varphi_p$ to state that there exists a fair path satisfying φ_p. The fairness constraint Φ can be any boolean combination of $\overset{\infty}{\mathsf{F}}\ \varphi_i$'s where the φ_i are state formulae. E.g. $\mathsf{E}_{(\overset{\infty}{\mathsf{F}} A \wedge \overset{\infty}{\mathsf{G}}\mathsf{EX}B)} C\mathsf{U}D$ is a GFCTL formula.
- FCTL [EL87] is GFCTL where the fairness constraint Φ is restricted to boolean combinations of $\overset{\infty}{\mathsf{F}}\ \pm A_i$ for atomic propositions A_is, and where Φ is the same for all occurrences of a path quantifier. Then it is more customary to see a FCTL formula as a pair (φ_s, Φ) of a CTL path formula and a global fairness constraint.
- CTL$^{\mathrm{F}}$ [CES86] is FCTL where the fairness constraint Φ is further restricted to a conjunctive $\bigwedge_i(\overset{\infty}{\mathsf{F}} \bigvee_j \pm A_{i,j})$.

2.5 Complexity of Model Checking

The *model checking problem* for a temporal logic L is to decide, given a KS S with distinguished node q_0, and a (state) formula $\varphi \in L$, whether $q_0 \models_S \varphi$. Model checking temporal logics has many practical applications [Eme90,McM93, CGP99] and it is important to be able to classify the most common temporal logics according to the computational complexity of their model checking problems.

Model checking for CTL and CTL* is known to be P-complete and PSPACE-complete respectively. Model checking for ECTL and CTL$^{\mathrm{F}}$ is P-complete too. For logics like CTL$^+$, FCTL, and ECTL$^+$, the exact complexity is not known. It has been observed [CES86, Theo. 6.2] that for CTL$^+$ the problem is NP-hard and coNP-hard and is in Δ_2^p (and thus believed to be easier than PSPACE-complete problems). The same applies to FCTL and GFCTL despite the wrong claim that model checking is NP-complete for FCTL [EL87, Coro. 4.8].

3 SNSAT and Δ_2^p-Complete Problems

Δ_2^p is the class P^{NP}, i.e. the class of problems solvable by a deterministic polynomial-time Turing machine querying an NP set oracle [Sto76]. This class is above NP and coNP in the polynomial-time hierarchy.

The class of problems complete for Δ_2^p does not contain many natural examples [Pap84,Kre88,Wag87]. In fact, in the polynomial-time hierarchy, it is easier to come up with problems complete for the Σ_k^p or Π_k^p levels than for the Δ_k^p levels.

In this paper we introduce SNSAT (for *sequentially nested* satisfiability), a logical problem with nested satisfiability questions, that is a convenient basis for our reducibility proof.

Definition 3.1. *An instance $\mathcal{I}$ of SNSAT is given by a set $X = \{x_1, \dots, x_n\}$ of boolean variables together with a list $\mathcal{L}$ of equivalences*

$$\begin{aligned} x_1 &:\Leftrightarrow \exists Z_1\ F_1(Z_1), \\ x_2 &:\Leftrightarrow \exists Z_2\ F_2(x_1, Z_2), \\ &\vdots \\ x_n &:\Leftrightarrow \exists Z_n\ F_n(x_1, \dots, x_{n-1}, Z_n), \end{aligned}$$

where, for $i = 1, \dots, n$, Z_i is a set $\{z_i^1, \dots, z_i^{p_i}\}$ of boolean variables, and F_i is a boolean formula with variables among $Z_i \cup \{x_1, \dots, x_{i-1}\}$.

Note that in $\mathcal{I}$ the sets X, Z_1, ..., and Z_n are pairwise disjoint. We write $Z = \{z_1, \dots, z_p\}$ for $Z_1 \cup \dots \cup Z_n$, and $Var = \{u, \dots\}$ for $X \cup Z$.

The equivalences $\mathcal{L}$ in $\mathcal{I}$ define a unique valuation $v_\mathcal{I}$ of the variables in X:

$$v_\mathcal{I}(x_i) = \top \overset{\text{def}}{\Leftrightarrow} F_i(v_\mathcal{I}(x_1), \dots, v_\mathcal{I}(x_{i-1}), Z_i) \text{ is satisfiable.} \tag{1}$$

Observe that there exists a simple algorithm in Δ_2^p that computes $v_\mathcal{I}$ one value at a time. When $v_\mathcal{I}$ is known over $\{x_1, \dots, x_{i-1}\}$, the value of $v_\mathcal{I}(x_i)$ is computed by solving a boolean satisfiability problem, "*is F_i satisfiable with the given values of $x_1, \dots, x_{i-1}$?*", for which a SAT oracle is sufficient.

The computational problem called SNSAT is, given an instance $\mathcal{I}$ as above, to decide whether $v_\mathcal{I}(x_n) = \top$ (in which case we say $\mathcal{I}$ is a positive instance).

Theorem 3.2. *SNSAT is Δ_2^p-complete.*

Proof. Membership in Δ_2^p has been explained above. Δ_2^p-hardness of SNSAT is shown incidentally in [Got95, proof of Theorem 3.4] where SNSAT is not identified as an interesting subproblem. (Alternatively, there are simple direct reductions from our SNSAT to the DSAT problem of [Pap84] and vice versa, but explaining DSAT requires a lot of notations.) □

The equivalences in $\mathcal{I}$ can be seen as a large satisfiability problem where we have to find correct values for the boolean variables in Z, aiming at satisfying the F_i's as much as possible, while respecting the values of the x_i's across equivalences. With this in mind, we say a valuation w of Var is:
safe: if, for all $i = 1, \dots, n$, $w(x_i)$ implies $F_i(v_\mathcal{I}(x_1), \dots, v_\mathcal{I}(x_{i-1}), w(Z_i))$,
correct: if, for all $i = 1, \dots, n$, $w(x_i) = F_i(v_\mathcal{I}(x_1), \dots, v_\mathcal{I}(x_{i-1}), w(Z_i))$,
admissible: if w is correct and coincide with $v_\mathcal{I}$ over X.
Thus a safe valuation only assigns positive values to some x_i if this is consistent with the values given to $x_1, \dots, x_{i-1}$ and the variables in Z_i. A correct valuation is safe and is also consistent for negative values assigned to some x_i. Still, there is no guarantee that the values of variables in Z are best possible. An arbitrary valuation over Z extends into a correct valuation in a unique way, and checking that a given w is correct can be done in polynomial-time.

An admissible valuation is just a valuation for Z that yields $v_\mathcal{I}$ for X. Hence it is optimal over Z. Clearly, admissible valuations exist for any SNSAT instance, positive or negative, but checking that a given w is admissible is Δ_2^p-complete.

4 Hardness of Model Checking CTL$^+$

In this section we show that there exists a logspace transformation from SNSAT into model checking for BTL$^+$. Aiming at improved clarity, we proceed in two steps: first we give a reduction of SNSAT to a model checking problem for CTL$^+$, then we adapt the construction and obtain a model checking problem for BTL$^+$.

From now on we assume that we are given an instance $\mathcal{I}$ of SNSAT with the notations of Def. 3.1. W.l.o.g. we assume that every F_i is a CNF, i.e. a conjunction of disjunctions of literals, and write F_i under the form $\bigwedge_l \bigvee_m \alpha_{i,l,m}$ where $\alpha_{i,l,m}$ is a literal $\pm u$ built with a variable u from $Z_i \cup \{x_1, \ldots, x_{i-1}\}$.

With $\mathcal{I}$ we associate a Kripke structure $S_\mathcal{I}$ and a CTL$^+$ formula $\Phi_\mathcal{I}$ s.t. $v_\mathcal{I}(x_n) = \top$ iff $S_\mathcal{I} \models \Phi_\mathcal{I}$ (see Coro 4.2). Figure 2 depicts $S_\mathcal{I}$.

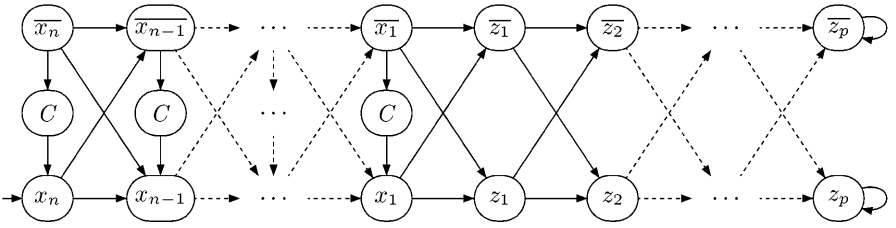


Fig. 2. Kripke structure $S_\mathcal{I}$ associated with SNSAT instance $\mathcal{I}$

As shown in Fig. 2, the nodes of $S_\mathcal{I}$ are of two kinds: (1) one node per literal u and $\overline{u}$ with $u \in \mathit{Var}$, and (2) one C-node between a $\overline{x_i}$-node and the corresponding x_i-node.

The nodes are labeled with propositions taken from $\{C\} \cup \{P_\alpha \mid \alpha \text{ a literal}\}$. The labeling is given by Fig. 2 where we shortly wrote α for P_α. Below we sometimes call α the literal-node labeled by P_α.

The transitions of $S_\mathcal{I}$ are of two kinds: (1) transitions from a literal $\pm u$ to a literal $\pm u'$ if u' immediately follows u in the left-to-rigth sequence x_n, x_{n-1}, …, x_2, x_1, z_1, z_2, …, z_p, (2) transitions from $\overline{x_i}$ to the ith C-node, and from there to x_i. Additionally, two self-loops on the $\pm z_p$-nodes ensure that the transition relation is total.

The structure of $S_\mathcal{I}$ is such that a path π from $\pm x_n$ that never visits a C-node visits exactly one literal for every $u \in \mathit{Var}$ so that there is a valuation w_π associated with π in the obvious way. Reciprocally, we can associate a path π_w with any valuation w in such a way that π_w starts from x_n or $\overline{x_n}$ (depending on $w(x_n)$) and never visits a C-node.

Furthermore, some properties of w can be stated as temporal properties of π_w: if $\pi \models \mathsf{G}\neg c$ then w_π is defined, and then w_π is safe iff $\pi \models \bigwedge_{i=1}^{n}\Big[\big(\mathsf{F}\, P_{x_i}\big) \Rightarrow \bigwedge_l \bigvee_m \mathsf{F}\, P_{\alpha_{i,l,m}}\Big]$.

We are now ready for the main technical difficulties: we define a sequence $\varphi_0, \varphi_1, \varphi_2, \ldots$ of CTL$^+$ formulae by $\varphi_0 \stackrel{\text{def}}{=} \top$ and, for $k > 0$,

$$\varphi_k \stackrel{\text{def}}{=} \mathsf{E}\left[\begin{array}{l} \mathsf{G}\Big[\big(P_{\overline{x_1}} \vee \cdots \vee P_{\overline{x_n}}\big) \Rightarrow \mathsf{EX}\big(C \wedge \mathsf{EX}(\neg\varphi_{k-1})\big)\Big] \\ \wedge\, \mathsf{G}\neg C \;\wedge\; \bigwedge_{i=1}^{n}\Big[\big(\mathsf{F}\, P_{x_i}\big) \Rightarrow \bigwedge_l \bigvee_m \mathsf{F}\, P_{\alpha_{i,l,m}}\Big] \end{array}\right].$$

Thus φ_k has the form $\mathsf{E}[\psi_{k-1} \wedge \mathsf{G}\neg C \wedge \rho]$ where ψ_{k-1} and ρ are complex path formulae, and where $\mathsf{G}\neg C \wedge \rho$ was used above to state that w_π is safe.

The next lemma states how φ_k is satisfied in nodes x_i and $\overline{x_i}$ of $S_\mathcal{I}$, justifying the whole construction:

Lemma 4.1 (Correctness of the reduction). *For $k \in \mathbb{N}$ and $r = 1, \ldots, n$:*
(a) if $k \geq 2r-1$ then $\big(v_\mathcal{I}(x_r) = \top$ iff $x_r \models \varphi_k\big)$,
(b) if $k \geq 2r$ then $\big(v_\mathcal{I}(x_r) = \bot$ iff $\overline{x_r} \models \varphi_k\big)$.

Proof. By induction on k. The case $k = 0$ holds vacuously. We now assume that $k > 0$ and that Lemma 4.1 holds for $k-1$.

i. We prove the "$\Rightarrow$" direction of both "iff"s:

Let w be an admissible valuation and π be the suffix of π_w that starts from x_r (or $\overline{x_r}$, depending on the value of $w(x_r)$). We claim that if $k \geq 2r-1$ (resp. $k \geq 2r$) then π is a witness for $x_r \models \varphi_k$ (resp. for $\overline{x_r} \models \varphi_k$). Clearly $\pi \models \mathsf{G}\neg C$ and $\pi \models \rho$ (because w is admissible) so that we only have to show $\pi \models \psi_{k-1}$, for which the $\overline{x_i}$ nodes must be checked. Now, whenever π visits a $\overline{x_i}$ for some $1 \leq i \leq r$, we have $v_\mathcal{I}(x_i) = \bot$ because w is admissible. We know $k \geq 2i$: if $i = r$ then we are proving the (b) part and $k \geq 2r$, and otherwise $i < r$. Hence $k-1 \geq 2i-1$ and the ind. hyp. entails $x_i \not\models \varphi_{k-1}$ so that $\overline{x_i} \models \mathsf{EX}(C \wedge \mathsf{EX}(\neg\varphi_{k-1}))$.

ii. We now prove the "$\Leftarrow$" direction of both "iff"s:

Assume $k \geq 2r-1$ and $x_r \models \varphi_k$ (or $k \geq 2r$ and $\overline{x_r} \models \varphi_k$). Thus there is a path π from x_r (resp. from $\overline{x_r}$) s.t. $\pi \models \psi_{k-1} \wedge \mathsf{G}\neg C \wedge \rho$. We claim that the valuation w_π induced by π is such that $w_\pi(x_i) = v_\mathcal{I}(x_i)$ for $i = 1, \ldots, r$, and prove this by induction on i. There are two cases:
(1) if $w_\pi(x_i) = \top$ then $\bigwedge_l \bigvee_m w(\alpha_{i,l,m}) = \top$ since $\pi \models \rho$, i.e. w is safe. Thus, by ind. hyp., $F_i(v_\mathcal{I}(x_1), \ldots, v_\mathcal{I}(x_{i-1}), w_\pi(Z_i)) = \top$ so that $v_\mathcal{I}(x_i) = \top$.
(2) if $w_\pi(x_i) = \bot$ then $\overline{x_i} \models \mathsf{EX}(C \wedge \mathsf{EX}(\neg\varphi_{k-1}))$ since $\pi \models \psi_{k-1}$ and thus $x_i \not\models \varphi_{k-1}$. Now if $i < r$, we have $k-1 \geq 2i-1$ and, by ind. hyp., $v_\mathcal{I}(x_i) = \bot$. If $i = r$ we must be in the case where $k \geq 2r$ and $\overline{x_r} \models \varphi_k$, so that $k-1 \geq 2i-1$ and again $v_\mathcal{I}(x_i) = \bot$ by ind. hyp. □

With Lemma 4.1, we get:

Corollary 4.2. *For any instance $\mathcal{I}$ of SNSAT, $v_{\mathcal{I}}(x_n) = \top$ iff $x_n \models_{S_{\mathcal{I}}} \varphi_{2n-1}$.*

The size of φ_{2n-1} is in $O(n \times |\mathcal{I}|)$. Since $S_{\mathcal{I}}$ and φ_{2n-1} can be built in logspace from $\mathcal{I}$, Coro. 4.2 effectively provides a transformation from SNSAT into model checking for CTL^+ (in fact, for UB^+), proving model checking for CTL^+ is Δ_2^p-hard.

The definition of the φ_k's uses EX and AX (in the ψ_{k-1} part) with the consequence that φ_k is a UB^+ and not a BTL^+ formula. However, a similar albeit clumsier construction can be given, proving the following

Theorem 4.3. *Model checking for BTL^+ is Δ_2^p-hard.*

Proof (Idea). We define a structure $S'_{\mathcal{I}}$ by modifying $S_{\mathcal{I}}$: Fig. 3 shows how so-called *stop nodes*, labeled with s, are inserted in $S_{\mathcal{I}}$, and how self-loops are added on the x_i-nodes.

We also modify the definition of φ_k by replacing $\mathsf{EX}(C \wedge \mathsf{EX}(\neg\varphi_{k-1}))$ in the ψ_{k-1} part with

$$\mathsf{E}\Big[\neg \mathsf{F} s \;\wedge\; \mathsf{F}\Big((P_{x_1} \vee \ldots \vee P_{x_n}) \;\wedge\; \neg\varphi_{k-1}\Big)\Big].$$

This gives BTL^+ formulae for which we can prove Lemma 4.1 adapted to $S'_{\mathcal{I}}$. □

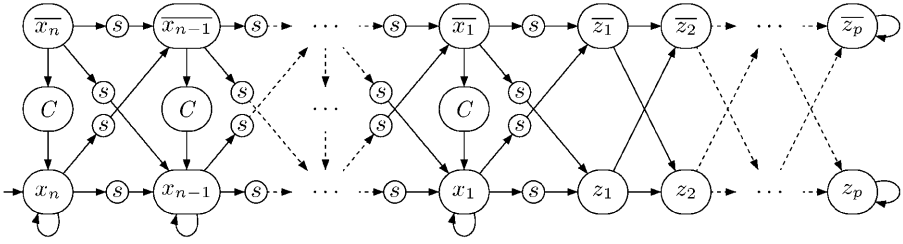


Fig. 3. $S'_{\mathcal{I}}$, a variant of $S_{\mathcal{I}}$ with *stop nodes*

5 Hardness of Model Checking FCTL

The ideas underlying the construction of $S_{\mathcal{I}}$ can be adapted in order to show Δ_2^p-hardness of model-checking for FCTL. Figure 4 describes $S''_{\mathcal{I}}$.

One sees that, because of the outermost loop, an infinite path π in $S''_{\mathcal{I}}$ may visit both u and $\overline{u}$ for any variable u (even if it never visits a C-node), so that

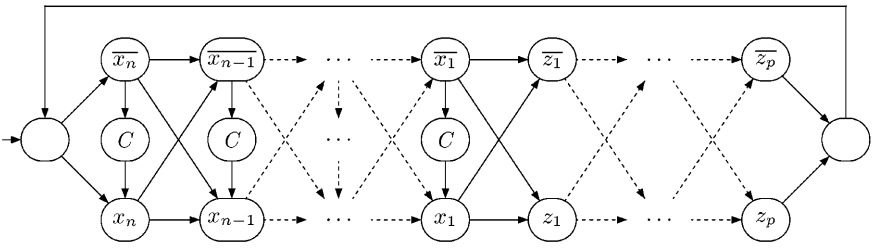


Fig. 4. $S''_{\mathcal{I}}$, a variant of $S_{\mathcal{I}}$ with outermost loop

there is no direct corresponding valuation w_π: we need more assumptions over paths.

Consider the following fairness constraint:

$$\Phi \stackrel{\text{def}}{=} \bigwedge_{u \in Var} \Big(\overset{\infty}{\mathsf{G}}\, \neg P_u \vee \overset{\infty}{\mathsf{G}}\, \neg P_{\overline{u}} \Big).$$

Now an infinite path π that verifies Φ defines a natural valuation: there exists a suffix π' of π s.t. for any $u \in Var$, π' never visits u (and then π' visits only $\overline{u}$) or π' never visits $\overline{u}$ (and then π' visits only u). Note that $\mathsf{G}\neg C$ holds for π'. Reciprocally, with any valuation w, we can associate a path π_w satisfying Φ in such a way that π_w visits infinitely often u or $\overline{u}$ depending on $w(u)$.

We now define FCTL formulae inspired by the φ_ks from the previous section. First we define ECTL$^+$ formulae ξ_k by $\xi_0 \stackrel{\text{def}}{=} \top$ and, for $k > 0$,

$$\xi_k \stackrel{\text{def}}{=} \mathsf{E} \left[\begin{array}{l} \mathsf{G}\Big[\big(P_{\overline{x_1}} \vee \cdots \vee P_{\overline{x_n}}\big) \;\Rightarrow\; \mathsf{EX}\big(C \wedge \mathsf{EX}(\neg \xi_{k-1})\big) \Big] \\ \wedge \bigwedge_{i=1}^{n} \Big[\big(\overset{\infty}{\mathsf{F}}\; P_{x_i} \big) \;\Rightarrow\; \bigwedge_l \bigvee_m \overset{\infty}{\mathsf{F}}\; P_{\alpha_{i,l,m}} \Big] \wedge \bigwedge_{u \in Var} \Big[\overset{\infty}{\mathsf{G}}\, \neg P_u \vee \overset{\infty}{\mathsf{G}}\, \neg P_{\overline{u}} \Big] \end{array} \right].$$

Note that ξ_k has the form $\mathsf{E}[\chi_{k-1} \wedge \rho' \wedge \Phi]$ where ρ' and Φ are fairness constraints which are used in any ξ_k. Clearly, in $S''_{\mathcal{I}}$ one can prove a variant of Lemma 4.1 for the ξ_ks, but the ξ_ks are not FCTL formulae.

Now observe that in $S''_{\mathcal{I}}$ the subformula $\mathsf{EX}(C \wedge \mathsf{EX}(\neg \xi_{k-1}))$ is equivalent to the following formula

$$\mathsf{E}\Big[\Phi \wedge \rho' \wedge \mathsf{X}\Big(C \wedge \mathsf{E}(\Phi \wedge \rho' \wedge \mathsf{X}(\neg \xi_{k-1})) \Big) \Big]$$

where we inserted the fairness constraint $\Phi \wedge \rho'$ under the two path quantifiers. The equivalence holds because, from any node in $S''_{\mathcal{I}}$, there exists an infinite fair path (it is sufficient to visit only $\overline{u}$ nodes).

We now have a variant of the ξ_ks where the same simple fairness constraint is used everywhere, that is, we have a FCTL formula! Formally, we define φ'_k by $\varphi'_0 \stackrel{\text{def}}{=} \top$ and $\varphi'_k \stackrel{\text{def}}{=} \mathsf{EG}\Big[\big(P_{\overline{x_1}} \vee \cdots \vee P_{\overline{x_n}}\big) \;\Rightarrow\; \mathsf{EX}\big(C \wedge \mathsf{EX}(\neg \varphi'_{k-1})\big) \Big]$ and

we couple this CTL formula with the fairness constraint $(\Phi \wedge \rho')$. Following the notations of section 2.4 we have:

$$\eta_k \stackrel{\text{def}}{=} \begin{cases} \varphi_s : \mathsf{EG}\Big[\big(P_{\overline{x_1}} \vee \cdots \vee P_{\overline{x_n}}\big) \;\Rightarrow\; \mathsf{EX}\big(C \wedge \mathsf{EX}(\neg\varphi'_{k-1})\big)\Big], \\ \Phi : \bigwedge\limits_{u \in Var} \Big[\overset{\infty}{\mathsf{G}}\, \neg P_u \vee \overset{\infty}{\mathsf{G}}\, \neg P_{\overline{u}}\Big] \wedge \bigwedge\limits_{i=1}^{n} \Big[\big(\overset{\infty}{\mathsf{F}}\; P_{x_i}\big) \;\Rightarrow\; \bigwedge\limits_{l} \bigvee\limits_{m} \overset{\infty}{\mathsf{F}}\; P_{\alpha_{i,l,m}}\Big]. \end{cases}$$

Now, Lemma 4.1 and corollary 4.2 can be reformulated for $S''_{\mathcal{I}}$ and η_k, proving the Δ_2^p-harness of FCTL model checking:

Theorem 5.1. *Model checking for* FCTL *is* Δ_2^p*-hard.*

6 Upper Bounds

In this section we show that model checking for ECTL$^+$ is in Δ_2^p. This is a slight extension of the corresponding result for CTL$^+$ (a result not widely known).

A path $\pi = q_0 q_1 \ldots$ (in some KS S) is *ultimately periodic* if there exist $m, k \in \mathbb{N}$ $(k > 0)$ s.t. $q_{i+k} = q_i$ for all $i \geq m$. Then π is written under the form $q_0 \ldots q_{m-1}(q_m \ldots q_{m+k-1})^\omega$ and we say π has size $m + k$.

A path π' is *extracted from* π if it has the form $\pi' = q_{i_0} \ldots q_{i_{p-1}}(q_{i_p} \ldots q_{i_s})^\omega$ where the sequence $i_0, i_1, \ldots$ is such that

$$0 \leq i_0 < i_1 < \ldots < i_{p-1} \leq m-1 < i_p < i_{p+1} < \ldots < i_s \leq m+k-1.$$

Let φ be an ECTL$^+$ formula of the form $\mathsf{E}\varphi_p$ where φ_p is *flat*, i.e. does not contain any path quantifier. The *principal subformulae* of φ_p are all subformulae of the form $\psi_1 \mathsf{U} \psi_2$ or $\overset{\infty}{\mathsf{F}}\, \psi$ or $\mathsf{X}\psi$, i.e. subformulae that have a modality at their root.

With $\pi = q_0 \ldots q_{m-1}(q_m \ldots q_{m+k-1})^\omega$ and φ_p we associate a set $w(\pi, \varphi_p) \subseteq \{0, 1, \ldots, m+k-1\}$ of *witness positions* along π: $w(\pi, \varphi_p)$ has one (or sometimes zero) position for every principal subformula of φ_p. Specifically:
- if $\mathsf{X}\psi$ is a principal subformula, then the witness position is 1,
- if $\overset{\infty}{\mathsf{F}}\, \psi$ is a principal subformula, then there is a witness position only if $\pi \models \overset{\infty}{\mathsf{F}}\, \psi$ and it is the first $i \geq m$ s.t. that $q_i \models \psi$,
- if $\psi_1 \mathsf{U} \psi_2$ is a principal subformula, then there are three cases: if $\pi \models \psi_1 \mathsf{U} \psi_2$, then the witness position is the first $i \geq 0$ s.t. $q_i \models \psi_2$, if $\pi \not\models \psi_1 \mathsf{U} \psi_2$ and $\pi \models \mathsf{F}\psi_2$, then it is the first $i \geq 0$ s.t. $q_i \models \neg(\psi_1 \wedge \psi_2)$, if $\pi \not\models \mathsf{F}\psi_2$, then there is no witness position for this subformula.

Lemma 6.1. *Assume* $\pi' = q_{i_0} q_{i_1} \ldots$ *is an ultimately periodic path extracted from* π*, with* $i_0 = 0$ *and such that* $w(\pi, \varphi_p) \subseteq \{i_0, i_1, \ldots, i_s\}$*. Then* $\pi' \models \varphi_p$ *iff* $\pi \models \varphi_p$*.*

Proof. By construction π' agrees with π on all principal subformulae, then on all subformulae, of φ_p. □

Lemma 6.2 (Small witnesses for ECTL$^+$). *Let S be a Kripke structure with n nodes, and $\mathsf{E}\varphi_p$ be a ECTL$^+$ formula where φ_p is flat. Then if $S \models \mathsf{E}\varphi_p$, there is a path $\pi \in \Pi(q_0)$ satisfying φ_p that is ultimately periodic and has size in $O(n \times |\varphi_p|)$.*

Proof. Assume $S \models \mathsf{E}\varphi_p$. Since φ_p is a PLTL formula, it is known (e.g. [SC85]) that there exists an ultimately periodic $\pi \in \Pi(q_0)$ s.t. $\pi \models \varphi_p$. Now we extract from π an ultimately periodic π' by keeping only positions in $w(\pi, \varphi_p)$ and the smallest number of intermediary positions that are required to ensure connectivity between the positions from $w(\pi, \varphi_p)$ (i.e. we want π' to be a path in S). Since $w(\pi, \varphi_p)$ has $O(|\varphi_p|)$ positions and since at most $n-1$ intermediary positions are required between any two positions in $w(\pi, \varphi_p)$, the size of π' is in $O(n \times |\varphi_p|)$. Finally, $\pi' \models \varphi_p$ by Lemma 6.1. □

The corollary is that there is an NP-algorithm for model checking ECTL$^+$ formulae of the form $\mathsf{E}\varphi_p$ with flat φ_p: one non-deterministically guesses an ultimately periodic π path of size $O(n \times |\varphi_p|)$ and then checks $\pi \models \varphi_p$ in time $O(n \times |\varphi_p|)$, e.g. seeing π as a deterministic Kripke structure on which φ_p can be read as a CTL formula.

Now, for model checking non-flat ECTL$^+$ formulae, we can use the general algorithm given in [EL87, Section 6] for branching-time logics of the form B(L(...)), i.e., logics obtained by adding path quantifiers to linear-time logics L(...). This algorithm is a simple polynomial-time procedure calling an oracle for model checking L(...). In the case of ECTL$^+$, we end with a P^{NP} algorithm, hence

Theorem 6.3. *Model checking for ECTL$^+$ is in Δ_2^p.*

7 Conclusions

Combining Theorems 4.3, 5.1 and 6.3, we obtain

Theorem 7.1. *The model checking problems for BTL$^+$, UB$^+$, CTL$^+$, FCTL, GFCTL and ECTL$^+$ are all Δ_2^p-complete.*

We also deduce

Theorem 7.2. *The model checking problem for BT* is Δ_2^p-complete.*

where BT* is the fragment of CTL* where F is the only allowed temporal modality (U and X are forbidden, G and $\overset{\infty}{\mathsf{F}}$ are allowed since they can be written with F).

Proof (of Theo. 7.2). Since BT* contains BTL$^+$, model checking BT* is Δ_2^p-hard. Since model checking flat $\mathsf{E}\varphi_p$ formulae is in NP when φ_p is in L(F) [SC85, DS98], a reasoning similar to the proof of Theo. 6.3 shows membership in Δ_2^p (already indicated in [CES86]). □

Acknowledgments. We thank A. Rabinovich for pointing to us that the complexity of model checking CTL^+ was not precisely characterized, and the anonymous referee who suggested that we look at FCTL.

References

[BPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Logics of Programs Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[DS98] S. Demri and Ph. Schnoebelen. The complexity of propositional linear temporal logics in simple cases (extended abstract). In *Proc. 15th Ann. Symp. Theoretical Aspects of Computer Science (STACS'98), Paris, France, Feb. 1998*, volume 1373 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 1998.

[EC80] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proc. 7th Coll. Automata, Languages and Programming (ICALP'80), Noordwijkerhout, NL, Jul. 1980*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980.

[EH83] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching versus linear time. In *Proc. 10th ACM Symp. Principles of Programming Languages (POPL'83), Austin, TX, USA, Jan. 1983*, pages 127–140, 1983.

[EH85] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1–24, 1985.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[EL87] E. A. Emerson and Chin-Laung Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B*, chapter 16, pages 995–1072. Elsevier Science, 1990.

[ES89] E. A. Emerson and J. Srinivasan. Branching time temporal logic. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Proc. REX School/Workshop, Noordwijkerhout, NL, May-June 1988*, volume 354 of *Lecture Notes in Computer Science*, pages 123–172. Springer, 1989.

[Got95] G. Gottlob. NP trees and Carnap's modal logic. *Journal of the ACM*, 42(2):421–457, 1995.

[Kre88] M. W. Krentel. The complexity of optimization problems. *Journal of Computer and System Sciences*, 36(3):490–509, 1988.

[Lam80] L. Lamport. "Sometimes" is sometimes "Not Never". In *Proc. 7th ACM Symp. Principles of Programming Languages (POPL'80), Las Vegas, NV, USA, Jan. 1980*, pages 174–185, 1980.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[ON80] H. Ono and A. Nakamura. On the size of refutation Kripke models for some linear modal and tense logics. *Studia Logica*, 39(4):325–333, 1980.

[Pap84] C. H. Papadimitriou. On the complexity of unique solutions. *Journal of the ACM*, 31(2):392–400, 1984.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science (FOCS'77), Providence, RI, USA, Oct.-Nov. 1977*, pages 46–57, 1977.

[RS00] A. Rabinovich and Ph. Schnoebelen. BTL_2 and expressive completeness for $ECTL^+$. Research Report LSV-00-8, Lab. Specification and Verification, ENS de Cachan, Cachan, France, October 2000. 21 pages.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[Sto76] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

[Wag87] K. W. Wagner. More complicated questions about maxima and minima, and some closures of NP. *Theoretical Computer Science*, 51(1–2):53–80, 1987.

[Wil99] T. Wilke. CTL$^+$ is exponentially more succint than CTL. In *Proc. 19th Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'99), Chennai, India, Dec. 1999*, volume 1738 of *Lecture Notes in Computer Science*, pages 110–121. Springer, 1999.

On Regular Message Sequence Chart Languages and Relationships to Mazurkiewicz Trace Theory

Rémi Morin*

Institut für Algebra, Technische Universität Dresden, D-01062 Dresden, Germany

Abstract. Hierarchical Message Sequence Charts are a well-established formalism to specify telecommunication protocols. In this model, numerous undecidability results were obtained recently through algebraic approaches or relationships to Mazurkiewicz trace theory. We show how to check whether a rational language of MSCs requires only channels of finite capacity. In that case, we also provide an upper bound for the size of the channels. This enables us to prove our main result: one can decide whether the iteration of a given regular language of MSCs is regular if, and only if, the Star Problem in trace monoids (over some restricted independence alphabets) is decidable too.

Message Sequence Charts (MSCs) are a popular model often used for the documentation of telecommunication protocols. They profit by a standartized visual and textual presentation (ITU-T recommendation Z.120 [20]) and are related to other formalisms such as sequence diagramms of UML [5] or message flow diagramms. An MSC gives a graphical description of the intended communications between processes. It abstracts away from the values of variables and the actual contents of messages. However, this formalism can be used at a very early stage of design to detect errors in the specification [18]. In this direction, several studies have already brought up methods and complexity results for the model checking of MSCs viewed as a specification language [14,26,27,1]. However, many undecidable problems arose by algebraic reductions to formal language theory [7] or relationships to Mazurkiewicz trace theory [28,16].

We are here interested in *regular* sets of MSCs, a notion recently introduced in [16]. These languages of MSCs are such the set of associated sequential executions can be described by a finite automaton; therefore model checking becomes decidable and particular complexity results could be obtained [1,28]. Moreover, regular languages of MSCs satisfy the *channel-bounded property*, that is, the number of messages stored in channels at any stage of any execution is bounded by a finite natural number. Consequently, as shown in [17,25], regular languages of MSCs admit a finite distributed abstract implementation in the form of message passing automata whose channels have a finite capacity; noteworthy, this result relies on asynchronous mappings [8] studied in order to associate a finite cellular asynchronous automaton to any recognizable subset of Mazurkiewicz traces. Another interesting characterization of regular MSC languages was established in [17]: they are precisely the languages that are definable in Monadic

* Supported by PROCOPE.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 332–346, 2001.

Second Order logic and which satisfy the channel-bounded property. Again, this result relies on technical results from trace theory.

It is the aim of this paper to contribute to this stream of relationships between traces and MSCs. These two formalisms consist of labelled partial orders (also known as pomsets [32]) provided with a concatenation that yields a monoidal structure. A *Hierarchical (or High-level) Message Sequence Chart (HMSC)* is then a description of a set of MSCs built from finite sets by use of union, concatenation (i.e. products), and iteration. Thus an HMSC is simply a rational expression in the monoid of MSCs. Moreover any HMSC can be flattened into a Message Sequence Graph (MSG), which is a kind of finite automaton labelled by MSCs. However, HMSCs are more succint descriptions than MSGs and they are also closer to the hierarchical specification of MSC languages. On the other hand, MSGs are often more convenient for the study of languages of MSCs [1, 28,16]. Since MSGs and HMSCs have the same expressive power, most results known for MSGs apply also to HMSCs. In [28], correction and consistency of a given HMSC are shown to be undecidable; more important here, it is shown in [16] that *one cannot decide whether a given HMSC describes a regular language of MSCs*. Both negative results rely actually on the undecidable Closure Problem in trace monoids [33]. A natural approach is now to restrict the class of HMSCs to be used, so that interesting properties of the associated languages are ensured or can effectively be checked.

In this direction, a particular class of HMSCs, called *locally synchronized* in [28] and *bounded* in [1] restricts iteration to sets of MSCs whose communication graphs are strongly connected. These sc-HMSC proved to be interesting since the associated languages are regular [28,1]. The converse was shown in [16]: any regular finitely generated language of MSCs can be described by an sc-HMSC. We shall see here how both relationships can be infered from Ochmański's theorem [31]. In our way, we show how one should adapt the definition of communication graph in order to deal with internal actions.

Our first result asserts that an HMSC satisfies the channel-bounded property if, and only if, iteration occurs only over sets of MSCs for which each connected component of the communication graph is strongly connected. Therefore *divergence of channels is easily decidable*. This evokes Ben-Abdallah & Leue static criterion [4] to check divergence freeness of HMSCs, although divergence freeness and channel-boundedness are in general distinct notions. Our proof also differs from [3,4] by providing a technically usefull upper bound for the size of the channels.

As mentionned above, regularity is however undecidable [16]. We consider in this paper a variation of this problem: we would like to check whether each subexpression of a given HMSC describes a regular language — and not only the whole HMSC itself. Since unions and products of regular languages are regular, we consider here the following problem for MSCs: *given a regular language of MSCs, decide whether its iteration is regular too.* Our second result asserts that *this problem is decidable if, and only if, the well-known Star Problem in trace monoids is decidable too* [12,21].

1 Basic Notions

Let $(\mathbb{M}, \cdot)$ be a monoid with unit 1. For any subsets $\mathcal{L}$ and $\mathcal{L}'$ of $\mathbb{M}$, the *product* of $\mathcal{L}$ by $\mathcal{L}'$ is $\mathcal{L} \cdot \mathcal{L}' = \{x \cdot x' \mid x \in \mathcal{L} \wedge x' \in \mathcal{L}'\}$. We let $\mathcal{L}^0 = \{1\}$ and for any $n \in \mathbb{N}$, $\mathcal{L}^{n+1} = \mathcal{L}^n \cdot \mathcal{L}$; then $\mathcal{L}^\star = \bigcup_{n\in\mathbb{N}} \mathcal{L}^n$ is the *iteration* of $\mathcal{L}$. A language $\mathcal{L} \subseteq \mathbb{M}$ is *finitely generated* if there is a finite subset $\mathcal{L}_0$ of $\mathbb{M}$ such that $\mathcal{L} \subseteq \mathcal{L}_0^\star$. A subset of $\mathbb{M}$ is *rational* if it can be obtained from the finite subsets of $\mathbb{M}$ by means of unions, products and iterations. Any rational language is finitely generated. A subset $\mathcal{L}$ of $\mathbb{M}$ is *recognizable* if there exists a finite monoid $\mathbb{M}'$ and a monoid morphism $\eta : \mathbb{M} \to \mathbb{M}'$ such that $\mathcal{L} = \eta^{-1} \circ \eta(\mathcal{L})$. Equivalently, $\mathcal{L}$ is recognizable if and only if there exists a finite $\mathbb{M}$-automaton recognizing $\mathcal{L}$ — because the collection of all sets $\mathcal{L}/x = \{y \in \mathbb{M} \mid x \cdot y \in \mathcal{L}\}$ is finite. Thus, the set of recognizable subsets of any monoid is closed under union, intersection and complement.

A *pomset* over an alphabet Σ is a triple $t = (E, \preccurlyeq, \xi)$ where $(E, \preccurlyeq)$ is a finite partial order and ξ is a mapping from E to Σ. We denote by $\mathbb{P}(\Sigma)$ the class of all pomsets over Σ. Let $t = (E, \preccurlyeq, \xi)$ be a pomset and $x, y \in E$. Then y *covers* x (denoted $x \multimap y$) if $x \prec y$ and $x \prec z \preccurlyeq y$ implies $y = z$. The elements x and y are *concurrent* or *incomparable* if $\neg(x \preccurlyeq y) \wedge \neg(y \preccurlyeq x)$. A pomset $t = (E, \preccurlyeq, \xi)$ is *without auto-concurrency* if $\xi(x) = \xi(y)$ implies ($x \preccurlyeq y$ or $y \preccurlyeq x$) for all $x, y \in E$. A pomset can be seen as an abstraction of an execution of a concurrent system [32]. In this view, the elements e of E are *events* and their label $\xi(e)$ describes the basic action of the system that is performed by the event. Furthermore, the order describes the causal dependence between the events. In particular, if two events are concurrent, they can be executed in any order or even in parallel. A pomset is without auto-concurrency if no action can be performed concurrently with itself. An *ideal* of a pomset $t = (E, \preccurlyeq, \xi)$ is a subset $H \subseteq E$ such that $x \in H \wedge y \preccurlyeq x \Rightarrow y \in H$. For all $z \in E$, we denote by $\downarrow_t z$ the ideal of events below z, i.e. $\downarrow_t z = \{y \in E \mid y \preccurlyeq z\}$. If H is a subset of E, we denote by $\#_t^a(H)$ the number of events $x \in H$ such that $\xi(x) = a$. (We will omit the subscript t when it is clear from the context.) An *order extension* of a pomset $t = (E, \preccurlyeq, \xi)$ is a pomset $t' = (E, \preccurlyeq', \xi)$ such that $\preccurlyeq \subseteq \preccurlyeq'$. A *linear extension* of t is an order extension that is linearly ordered. Linear extensions of a pomset $t = (E, \preccurlyeq, \xi)$ can naturally be regarded as words over Σ. By $\mathrm{LE}(t) \subseteq \Sigma^\star$, we denote the set of linear extensions of a pomset t over Σ. Clearly, two isomorphic pomsets admit the same linear extensions. Noteworthy the converse property holds for pomsets without auto-concurrency: two pomsets without auto-concurrency t and t' are isomorphic iff $\mathrm{LE}(t) = \mathrm{LE}(t')$. For any subclass $\mathcal{L}$ of $\mathbb{P}(\Sigma)$, $\mathrm{LE}(\mathcal{L})$ denotes $\bigcup_{t\in\mathcal{L}} \mathrm{LE}(t)$.

Mazurkiewicz Traces. Let us now recall some basic notions of trace theory [9]. The concurrency of a distributed system is often represented by an *independence relation* over the alphabet of actions Σ, that is a binary, symmetric and irreflexive relation $\| \subseteq \Sigma \times \Sigma$. The associated *trace equivalence* is the least congruence $\sim$ over $\Sigma^\star$ such that $\forall a, b \in \Sigma, a \| b \Rightarrow ab \sim ba$. A trace $[u]$ is the equivalence class of a word $u \in \Sigma^\star$. We denote by $\mathbb{M}(\Sigma, \|)$ the set of all traces w.r.t. $(\Sigma, \|)$. Traces

can easily be composed in the following way: $[u] \cdot [v] = [u.v]$. Then $\mathbb{M}(\Sigma, \|)$ appears as a monoid with the empty trace $[\varepsilon]$ as unit. A *trace language* is a subset $\mathcal{L} \subseteq \mathbb{M}(\Sigma, \|)$. It is easy to see that a trace language $\mathcal{L}$ is recognizable in $\mathbb{M}(\Sigma, \|)$ iff the set of associated linear extensions $\mathrm{LE}(\mathcal{L})$ is recognizable in the free monoid $\Sigma^\star$. Let $u \in \Sigma^\star$; then the trace $[u]$ is precisely the set of linear extensions $\mathrm{LE}(t)$ of a unique pomset $t = (E, \preccurlyeq, \xi)$ without auto-concurrency, that is, $[u] = \mathrm{LE}(t)$. Moreover t satisfies the following additional properties [24]:

MP_1: for all events $e_1, e_2 \in E$ with $\xi(e_1) \not| \xi(e_2)$, we have $e_1 \preccurlyeq e_2$ or $e_2 \preccurlyeq e_1$;

MP_2: for all events $e_1, e_2 \in E$ with $e_1 \prec\!\!\!\!- e_2$, we have $\xi(e_1) \not| \xi(e_2)$.

Conversely any pomset satisfying these two axioms is a pomset without auto-concurrency whose linear extensions form a trace of $\mathbb{M}(\Sigma, \|)$. Thus one usually identifies $\mathbb{M}(\Sigma, \|)$ with the class of pomsets satisfying MP_1 and MP_2 — up to isomorphisms. The product of traces can now be viewed as a concatenation of pomsets: let $t_1 = (E_1, \preccurlyeq_1, \xi_1)$ and $t_2 = (E_2, \preccurlyeq_2, \xi_2)$ be two traces over $(\Sigma, \|)$; the concatenation $t_1 \cdot t_2$ is the pomset $t = (E_1 \uplus E_2, \preccurlyeq, \xi_1 \cup \xi_2)$ where $\preccurlyeq$ is the transitive closure of $\preccurlyeq_1 \ \cup \ \preccurlyeq_2 \ \cup \ \{(e_1, e_2) \in E_1 \times E_2 \mid \xi_1(e_1) \not| \xi_2(e_2)\}$.

Basic Message Sequence Charts. MSCs are defined by several recommendations that indicate how one should represent them graphically [20]. More formally, they can be seen as particular labelled partial orders. Similar approaches can be traced to Lamport's diagrams [23] or Nielsen, Plotkin & Winskel's elementary event structures [29].

Let $\mathcal{I}$ be a finite set of processes, also called *instances*. For any instance $i \in \mathcal{I}$, Σ_i^{int} denotes a finite set of *internal actions*; the alphabet Σ_i is then the disjoint union of the set of *send actions* $\Sigma_i^! = \{i!j \mid j \in \mathcal{I} \setminus \{i\}\}$, the set of *receive actions* $\Sigma_i^? = \{i?j \mid j \in \mathcal{I} \setminus \{i\}\}$ and the set of internal actions Σ_i^{int}. We shall assume that the alphabets Σ_i are disjoint and we let $\Sigma_{\mathcal{I}} = \bigcup_{i \in \mathcal{I}} \Sigma_i$. Given an action $a \in \Sigma_{\mathcal{I}}$, we denote by $\mathrm{Ins}(a)$ the unique instance i such that $a \in \Sigma_i$, that is the particular instance on which each occurrence of action a occurs. Finally, for any pomset $(E, \preccurlyeq, \xi)$ over $\Sigma_{\mathcal{I}}$ we denote by $\mathrm{Ins}(e)$ the instance on which the event $e \in E$ occurs : $\mathrm{Ins}(e) = \mathrm{Ins}(\xi(e))$.

Definition 1.1. *A* basic message sequence chart (or basic MSC) *is a pomset* $M = (E, \preccurlyeq, \xi)$ *over* $\Sigma_{\mathcal{I}}$ *such that*

M_1: $\forall e, f \in E$: $\mathrm{Ins}(e) = \mathrm{Ins}(f) \Rightarrow (e \preccurlyeq f \vee f \preccurlyeq e)$

M_2: $\#^{i!j}(E) = \#^{j?i}(E)$ *for any distinct instances* i *and* j

M_3: $\big(\xi(e) = i!j \wedge \xi(f) = j?i \wedge \#^{i!j}(\downarrow e) = \#^{j?i}(\downarrow f)\big) \Rightarrow e \preccurlyeq f$

M_4: $[e \prec\!\!\!\!- f \wedge \mathrm{Ins}(e) \neq \mathrm{Ins}(f)]$
$\Rightarrow \big[\xi(e) = i!j \wedge \xi(f) = j?i \wedge \#^{i!j}(\downarrow e) = \#^{j?i}(\downarrow f)\big]$.

By M_1, events occurring on the same instance are linearly ordered : hence non-deterministic choice cannot be described within an MSC. Condition M_2 makes sure that there are as many send events from i to j than receive events from j to i; this expresses the reliability of the channels. Since the latter are assumed to be FIFO, the n-th message sent from i to j is received when the n-th event $j?i$ occurs; thus M_3 formalizes simply that the reception of any message will occur

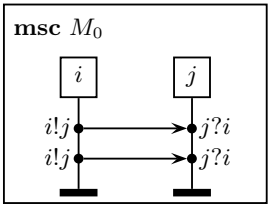


Fig. 1. A basic MSC

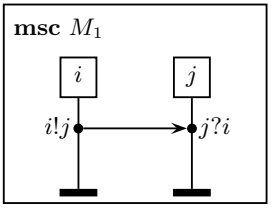


Fig. 2. $M_0 = M_1 \cdot M_1$

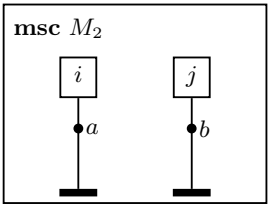


Fig. 3. Internal actions

only after the corresponding send event. Finally, by M_4, causality in M consists only in the linear dependency over each instance and the ordering of pairs of corresponding send and receive events.

Thus, basic MSCs are precisely the model studied in [1,14,7,16,17,25] although some authors forbid internal actions. Opposite to general sets of pomsets [22], basic MSCs satisfy a fundamental property known for Mazurkiewicz traces and also often considered in their generalization as P-traces [2,19].

LEMMA 1.2. *Let M and M' be two basic MSCs. If* $\mathrm{LE}(M) \cap \mathrm{LE}(M') \neq \emptyset$ *then M and M' are isomorphic.*

Examples of basic MSCs over two instances i and j are described informally in Figures 1 and 2. There, each arrow $i!j \to j?i$ represents a pair of send-receive related events (and as usual MSCs should be read from top to bottom). Thus the MSC M_0 has two linear extensions $i!j.i!j.j?i.j?i$ and $i!j.j?i.i!j.j?i$. Hence the second send event and the first receive event are concurrent (or independent). The MSC M_1 has only one linear extension $i!j.j?i$ and thus does not describe actually any concurrent behavior.

Hierarchical Message Sequence Charts. We denote by $\mathbb{bMSC}$ the set of (isomorphism classes) of basic MSCs. The *asynchronous concatenation* of two basic MSCs $M_1 = (E_1, \preccurlyeq_1, \xi_1)$ and $M_2 = (E_2, \preccurlyeq_2, \xi_2)$ is $M_1 \cdot M_2 = (E, \preccurlyeq, \xi)$ where $E = E_1 \uplus E_2$, $\xi = \xi_1 \cup \xi_2$ and the partial order $\preccurlyeq$ is the transitive closure of $\preccurlyeq_1 \cup \preccurlyeq_2 \cup \{(e_1, e_2) \in E_1 \times E_2 \mid \mathrm{Ins}(e_1) = \mathrm{Ins}(e_2)\}$. It is easy to check that the asynchronous concatenation of two basic MSCs is a basic MSC. With Lemma 1.2, this concatenation can be shown to be associative and admits the empty MSC $(\emptyset, \emptyset, \emptyset)$ as unit. Therefore we shall refer to $\mathbb{bMSC}$ as the *monoid of basic message sequence charts*. As observed in [7], $\mathbb{bMSC}$ can also be viewed as a sub-monoid of the product $\prod_{i \in \mathcal{I}} \Sigma_i^\star$ provided with the component-wise concatenation. Numerous indecidability results were derived from this simple reduction [7, Th. 5].

The monoidal structure of basic MSCs enables us to use hierarchical specifications for sets of MSCs by composing finite languages by unions, concatenations or iterations — as this is usually done when considering rational languages within a monoid. In that way, we obtain hierarchical message sequence charts.

Definition 1.3. *A* hierarchical message sequence chart (HMSC) *is a rational expression of* bMSC*, that is, an expression built from basic MSCs by use of union, product and iteration.*

Note that the language of basic MSCs associated to an HMSC is finitely generated w.r.t. bMSC. We follow here the approach adopted, e.g., in [1,7,16,17,25] where HMSCs are however often flattened into message sequence graphs.

2 Channel-Bounded vs. Regular Languages

In this section, we show how one can decide whether a given HMSC does not induce divergence into channels, i.e. it could be implemented with channels having a finite capacity. This interesting property is called *channel-boundedness*.

Channel-Boundedness, Regularity, and Mazurkiewicz Traces. In [4], Ben-Abdallah & Leue defined the process divergence of HMSC by the existence of an infinite sequential execution that induce unbounded numbers of messages in channels. More recently, a simpler notion of boundedness was considered in [16,17,25] for the study of regular languages.

Definition 2.1. *The* channel-width *of a basic MSC M is*

$$\max_{i,j\in\mathcal{I},i\neq j} \{\#^{i!j}(H) - \#^{j?i}(H) \mid H \text{ ideal of } M\}.$$

A language $\mathcal{L} \subseteq$ bMSC is channel-bounded *by an integer B if each basic MSC of $\mathcal{L}$ has a channel-width at most B.*

Recall now that graphical representations of MSCs should be read from top to bottom on each instance. Thus in the basic MSC M_3 of Fig. 4, the event labelled $k?j$ occurs before the event labelled $k!l$. Now the channel-width of M_3 of is 4. Removing the two events labelled $l!i$ and $i?l$ from M_3 would lead to a basic MSC with channel-width 5. Note also that the channel-width of $M_3 \cdot M_3$ is 5. Consider again the basic MSC M_1 of Fig. 2; the rational language $\{M_1\}^\star$ is not channel-bounded.

As explained in the introduction, a particularly interesting notion of regularity was introduced in [16] and related to MSO logic [17] and message passing automata [25].

Definition 2.2. *A language $\mathcal{L}$ of basic MSCs is* regular *if its set of linear extensions* $\mathrm{LE}(\mathcal{L}) = \bigcup_{M\in\mathcal{L}} \mathrm{LE}(M)$ *is recognizable in the free monoid $\Sigma_{\mathcal{I}}^\star$.*

We remark that regularity differs from recognizability. On one hand, any regular language is recognizable. But the converse fails: consider for instance again the MSC M_1 of Figure 2: the rational language $\{M_1\}^\star$ is recognizable in bMSC but not regular. Actually, as observed in [16, Prop. 2.1], any regular language is channel-bounded.

From a remark of [17,25] and Lemma 1.2, it follows that sets of basic MSCs can be seen as generalized trace languages [30] or CCI sets of P-traces [2,19]. Therefore, regular sets of MSCs can be represented by recognizable subsets of Mazurkiewicz traces by means of a relabeling [2,19]. But since MSCs are very particular P-traces this basic relationship can be established here as follows.

LEMMA 2.3 (D. Kuske). *Let B be a positive integer. We consider the finite alphabet $\Sigma = \Sigma_{\mathcal{I}} \times [0, B]$ and the independence relation $\| \subseteq \Sigma \times \Sigma$ such that*

$$(a, n) \,/\!\!|\, (a', n') \text{ if } \mathrm{Ins}(a) = \mathrm{Ins}(a') \text{ or } [\ \{a, a'\} = \{i!j, j?i\} \wedge n = n'\ ].$$

Let π_1 be the first projection from Σ to $\Sigma_{\mathcal{I}}$ which sends $(a, n) \in \Sigma$ to $a \in \Sigma_{\mathcal{I}}$. The map π_1 extends naturally to a map from the pomsets over Σ to the pomsets over $\Sigma_{\mathcal{I}}$ for which $(E, \preccurlyeq, \xi)$ is associated to $(E, \preccurlyeq, \pi_1 \circ \xi)$. Then for any basic MSC M with channel-width at most B, there exists a Mazurkiewicz trace $t \in \mathbb{M}(\Sigma, \|)$ such that $\pi_1(t) = M$.

Proof. Let $M = (e, \preccurlyeq, \xi)$ be an MSC with channel-width at most B. We easily check that the pomset $t = (E, \preccurlyeq, \xi_t)$ over Σ such that $\xi_t(e) = (\xi(e), \#_M^{\xi(e)}(\downarrow e) \bmod (B+1))$ is a trace over $(\Sigma, \|)$. ∎

By [17], any regular language of MSCs is MSO definable. Therefore we can use Büchi's Theorem for Mazurkiewicz traces [11,34] together with Lemma 2.3 to derive the following representation result.

COROLLARY 2.4. *Let $\mathcal{L}$ be a language of basic MSCs channel-bounded by B. With the notations of Lemma 2.3, let $\pi_1^{-1}(\mathcal{L})$ be the set of traces $t \in \mathbb{M}(\Sigma, \|)$ such that $\pi_1(t) \in \mathcal{L}$. If $\mathcal{L}$ is regular then $\pi_1^{-1}(\mathcal{L})$ is recognizable in $\mathbb{M}(\Sigma, \|)$ and $\pi_1 \circ \pi_1^{-1}(\mathcal{L}) = \mathcal{L}$.*

Thus any regular language of basic MSCs can be seen as a recognizable set of traces up to an adequate relabeling; moreover the latter depends on an upper bound for the channel-widths of the MSCs.

Easy Checking of the Channel-Boundedness Property. As established in [16, Th. 4.6], one cannot decide whether a rational language of bMSC is regular. In other words:

THEOREM 2.5. *[16] It is undecidable to check whether the language of basic MSCs associated to a given HMSC is regular.*

In this section, we shall cope with this negative result in two different ways. A first natural approach is to weaken the problem: since any regular language is channel-bounded, one could aim at checking only the channel-boundedness of the language associated to an HMSC, instead of its regularity. This will be achieved by Theorem 2.8 and Corollary 2.9 below. Another way to deal with Theorem 2.5 is to look for a subclass of HMSCs that describes only regular languages. This is achieved in particular by sc-HMSCs defined below (Def. 2.11 and Cor. 2.13).

In order to represent a channel-bounded language of MSCs by a trace language, Lemma 2.3 indicates that it suffices to compute an upper bound for the

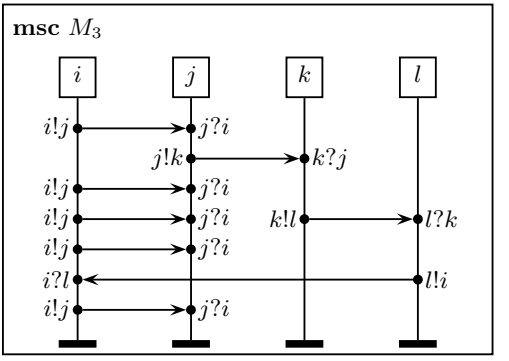


Fig. 4. A strongly connected MSC

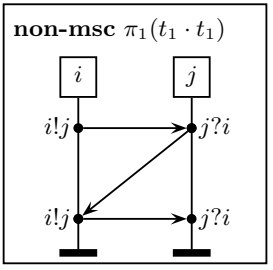


Fig. 5. $\pi_1(t_1 \cdot t_1) \neq M_0$

channel-width of the MSCs. First, if $\mathcal{L}_1$ and $\mathcal{L}_2$ are channel-bounded by B_1 and B_2 respectively then $\mathcal{L}_1 \cup \mathcal{L}_2$ is channel-bounded by $\max(B_1, B_2)$. For the product, we simply observe:

LEMMA 2.6. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages of basic MSCs channel-bounded by B_1 and B_2 respectively. Then $\mathcal{L}_1 \cdot \mathcal{L}_2$ is channel-bounded by $B_1 + B_2$.*

Thus, the main problem consists in deciding whether the iteration of a channel-bounded language is channel-bounded *and to compute a bound*. For later purposes, it is convenient to slightly extend now a useful notion related to MSCs.

DEFINITION 2.7. *The* communication graph *of a basic MSC $M = (E, \preccurlyeq, \xi)$ is the directed graph $(\mathcal{I}_M, {}^{\mathrm{I}}\!\!\rightarrow)$ where $\mathcal{I}_M$ is the set of* active instances *of M: $\mathcal{I}_M = \{i \in \mathcal{I} \mid \exists e \in E, \mathrm{Ins}(e) = i\}$, and such that $(i,j) \in {}^{\mathrm{I}}\!\!\rightarrow$ if there is $e \in E$ such that $\xi(e) = i!j$.*

Thus there is an edge from i to j if M specifies a communication from i to j. In [1,28,16], an *extended communication graph* $(\mathcal{I}, {}^{\mathrm{I}}\!\!\rightarrow)$ is considered without restriction to the active instances, but still with the same edges. Actually our slight variation is meant to cope with *internal actions* in Cor. 2.13.

The communication graph is the basis of a useful criterion to check whether the iteration of a language of basic MSCs is channel-bounded.

THEOREM 2.8. *Let $\mathcal{L} \subseteq \mathrm{bMSC}$. The following conditions are equivalent:*

(i) *$\mathcal{L}^\star$ is channel-bounded.*

(ii) *$\mathcal{L}$ is channel-bounded and the communication graph of each $M \in \mathcal{L}$ is* locally strongly connected *— i.e. each connected component is strongly connected.*

Moreover, if (ii) *holds and if $\mathcal{L}$ is channel-bounded by B then $\mathcal{L}^\star$ is channel-bounded by $2^{N^2}.(N+1).B$ where $N = \mathrm{Card}(\mathcal{I})$.*

Proof. First we prove (i) ⇒ (ii) by contradiction. We assume that $\mathcal{L}$ contains a basic MSC M for which at least one connected component of its communication graph is not strongly connected. This means that there are two distinct instances

i and j such that $i \mathrel{\shortmid\!\!\rightarrow} j$ and there is no path from j to i. We simply observe here that the channel-width of M^{B+2} is larger than $B+1$.

We prove now (ii) $\Rightarrow$ (i). Let i and j be two fixed distinct instances. We shall use several time the following observation:

Claim. *Let M_1,..., M_n be n basic MSCs of $\mathcal{L}$ and $M = M_1 \cdot ... \cdot M_n$. Let K be the set of integers $k \in [1,n]$ such that there is an edge $i \mathrel{\shortmid\!\!\rightarrow} j$ in the communication graph of M_k. Let $k_1 < k_2$ be two integers of $[1,n]$ and let $e \in E_{k_2}$ be such that $\xi(e) = i!j$. If $\forall f \in E, [f \in E_k \wedge f \preccurlyeq e \wedge \xi(f) = j?i] \Rightarrow k < k_1$ then* $\mathrm{Card}(K \cap [k_1, k_2]) \leqslant 2^{N.(N-1)}.(N+1)$. □

Let us prove this claim first, by contradiction. We assume that $\mathrm{Card}(K \cap [k_1, k_2]) > 2^{N.(N-1)}.(N+1)$. Since there are only $2^{N.(N-1)}$ distinct extended communication graphs, there are in the family $(M_k)_{k \in K \cap [k_1,k_2]}$ at least $N+1$ MSCs with the same communication graph G_0. The graph G_0 contains an edge (i,j). Since any connected component of G_0 is strongly connected, there is a path $j = i_1 \mathrel{\shortmid\!\!\rightarrow} i_2 \mathrel{\shortmid\!\!\rightarrow} ... \mathrel{\shortmid\!\!\rightarrow} i_r = i$ in G_0 with $r \leqslant \mathrm{Card}(\mathcal{I}) = N$. We denote by J the set of integers $k \in K \cap [k_1, k_2]$ such that the communication graph of M_k is G_0. Then $\mathrm{Card}(J) \geqslant N+1$. Let $j_1, j_2, ..., j_{\mathrm{Card}(J)}$ be an increasing enumeration of J. Let f be an event of M_{j_1} such that $\xi(f) = j?i$. Since $\mathrm{Card}(J)-1 \geqslant \mathrm{Card}(\mathcal{I}) \geqslant r$, there is an event g in $M_{\mathrm{Card}(J)-1}$ such that $\mathrm{Ins}(g) = i$ and $f \preccurlyeq g$. Now $g \preccurlyeq e$ hence $f \preccurlyeq e$. This contradicts $f \in E_{k_1}$.

We consider now some MSCs M_1,..., M_n in $\mathcal{L}$ and their concatenation $M = (E, \preccurlyeq, \xi) = M_1 \cdot ... \cdot M_n$. We let K be as above. Let $e_0 \in E$ be such that $\xi(e_0) = i!j$ and consider k_0 to be the integer of $[1,n] \cap K$ such that $e_0 \in E_{k_0}$. It is sufficient to show that $\#^{i!j}_M(\downarrow_M e_0) - \#^{j?i}_M(\downarrow_M e_0) \leqslant 2^{N^2}.(N+1).B$.

1. We assume first that $\mathrm{Card}(K \cap [1,k_0]) \leqslant 2^{N.(N-1)}.(N+1)$. We let $M' = M_1 \cdot ... \cdot M_{k_0}$. We observe that $\downarrow_M e_0 = \downarrow_{M'} e_0$. By Lemma 2.6, $\#^{i!j}_M(\downarrow_M e_0) - \#^{j?i}_M(\downarrow_M e_0) = \#^{i!j}_{M'}(\downarrow_{M'} e_0) - \#^{j?i}_{M'}(\downarrow_{M'} e_0) \leqslant 2^{N.(N-1)}.(N+1).B$.
2. We assume now that $\mathrm{Card}(K \cap [1,k_0]) > 2^{N.(N-1)}.(N+1)$. Then, according to the claim above, there are some events $f \in E$ such that $\xi(f) = j?i$ and $f \preccurlyeq e_0$. Among all these events below e_0 and labelled $j?i$, we consider f_1 to be the maximal one. Let k_1 be the integer such that $f_1 \in E_{k_1}$. Since M_{k_1} is basic, there is an event $e_1 \in E_{k_1}$ such that $\xi(e_1) = i!j$ and $\#^{i!j}_{M_{k_1}}(\downarrow e_1) = \#^{j?i}_{M_{k_1}}(\downarrow f_1)$. Therefore $\#^{i!j}_M(\downarrow e_1) = \#^{j?i}_M(\downarrow f_1)$. According to the claim above, $\mathrm{Card}(K \cap [k_1+1, k_0]) \leqslant 2^{N.(N-1)}.(N+1)$ — otherwise f_1 is not maximal. We consider $k' \in K \cap [k_1+1, k_0-1]$. Let $e \in E_k$ be such that $\xi(e) = i!j$. Then there is no event $f \in E_k$ such that $f \preccurlyeq_{M_k} e$ and $\xi(f) = j?i$ (otherwise $f \preccurlyeq e_0$ and f_1 is not maximal). Therefore $\#^{i!j}_{M_k}(E_k) \leqslant B$ because the channel-width of M_k is at most B. Similarly, $\mathrm{Card}\{e \in E_{k_1} \mid \xi(e) = i!j \wedge e_1 \prec e\} \leqslant B$ and $\#^{i!j}_{M_{k_0}}(\downarrow_{M_{k_0}} e_0) \leqslant B$. Consequently, $\#^{i!j}_M(\downarrow_M e_0) \leqslant (2 + 2^{N.(N-1)}.(N+1)).B + \#^{i!j}_M(\downarrow_M e_1)$. Hence $\#^{i!j}_M(\downarrow_M e_0) - \#^{j?i}_M(\downarrow_M e_0) \leqslant (2 + 2^{N.(N-1)}.(N+1)).B$.

■

Now, the product or union of two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ is channel-bounded if, and only if, $\mathcal{L}_1$ and $\mathcal{L}_2$ are channel-bounded. For the iteration, we observe that

one can inductively compute the set of all communication graphs of all the MSCs associated to a given HMSC. Thus channel-boundedness is easily decidable:

COROLLARY 2.9. *An HMSC is channel-bounded iff iteration occurs only over sets of MSCs whose communication graphs are locally strongly connected.*

As a consequence, a *rational* language of MSCs is channel-bounded iff it is divergence-free in the sense of [4].

First Application to Regular Languages. We are here interested in a subclass of HMSCs that describes only regular languages. Again, union and product do not raise problems at all:

LEMMA 2.10. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two regular languages of basic MSCs. Then $\mathcal{L}_1 \cup \mathcal{L}_2$ and $\mathcal{L}_1 \cdot \mathcal{L}_2$ are regular too.*

Now, a rather simple way to ensure the regularity of languages associated to hierarchical message sequence charts is to restrict to sc-HMSCs. This restriction is actually a reformulation of a condition of "local synchronization" or "boundedness" introduced in the framework of message sequence graphs [28,1].

DEFINITION 2.11. *A hierarchical MSC is an* sc-HMSC *if iteration occurs only over sets of MSCs whose communication graphs are strongly connected.*

We show here that this restriction corresponds precisely to an approach previously followed by Ochmański in the framework of Mazurkiewicz traces [31]. Recall that a trace $t \in \mathbb{M}(\Sigma, \|)$ is *connected* if the restriction of the dependence graph $(\Sigma, \not\|)$ to the subset of actions appearing in t is connected. Then a subset of $\mathbb{M}(\Sigma, \|)$ is *c-rational* if it can be obtained from finite subsets by means of unions, products and iterations over subsets of connected traces.

THEOREM 2.12. *[31] A trace language is recognizable iff it is c-rational.*

The next result was originally shown in [28,1] and [16] however under the assumption that there is no internal action. The (restricted) communication graph of Def. 2.7 enables us to extend this relationship in the more general present setting. We also show how it can be inferred from Theorem 2.12.

COROLLARY 2.13. *Let $\mathcal{L}$ be a* finitely generated *language of basic MSCs. Then $\mathcal{L}$ is regular if, and only if, it is the language of an sc-HMSC.*

Proof. We consider first a regular, finitely generated language $\mathcal{L}$ of basic MSCs. We consider Σ to be a finite family of basic MSCs such that $\mathcal{L} \subseteq \Sigma^\star \subseteq \mathrm{bMSC}$. We may assume that the communication graph of each MSC $M \in \Sigma$ is connected. Since $\mathcal{L}$ is recognizable, there is a deterministic finite full bMSC-automaton $\mathcal{A} = (Q, \{\imath\}, \longrightarrow, F)$ that recognizes $\mathcal{L}$. Let $\mathcal{A}_0 = (Q, \{\imath\}, \longrightarrow_0, F)$ be the automaton over the alphabet Σ such that $\xrightarrow{M}_0 = \xrightarrow{M} \cap (Q \times \Sigma \times Q)$. We consider the independence relation over Σ such that $M \| M'$ if for all events e of M and for all events e' of M', $\mathrm{Ins}(e) \neq \mathrm{Ins}(e')$. Then, if $q \xrightarrow{M}_0 q_1 \xrightarrow{M'}_0 q_2$ and $M \| M'$

then there is a state q_3 such that $q \xrightarrow{M'}_0 q_3 \xrightarrow{M} q_2$ because $M \cdot M' = M' \cdot M$ and $\mathcal{A}$ is deterministic. The language $\mathcal{L}_0$ recognized by $\mathcal{A}_0$ is recognizable in the free monoid $\Sigma^\star$; it is also closed for the commutation of independent MSCs. Therefore $\mathcal{L}_0$ can be identified to a recognizable trace language of $\mathbb{M}(\Sigma, \|)$. By Th. 2.12, we can consider a c-rational expression h that describes $\mathcal{L}_0 \in \mathbb{M}(\Sigma, \|)$. We can see also h as a rational expression over bMSC — that is, as an HMSC — that describes actually $\mathcal{L}$. Recall now that the communication graph of each MSC $M \in \Sigma$ is connected; moreover the star operation is taken in h only over sets of connected traces of $\mathbb{M}(\Sigma, \|)$. Therefore the star operation is taken in h over sets of MSCs which are connected. Now we know that $\mathcal{L}$ is regular, hence channel-bounded. Therefore Th. 2.8 ensures that the star operation in h is only taken over sets of MSCs which are *strongly* connected, i.e. h is an sc-HMSC.

For the converse, Lemma 2.10 shows it is sufficient to consider the iteration of a regular language $\mathcal{L}_0$ of basic MSCs. Then $\mathcal{L}_0$ is channel-bounded by some integer B_0. By Th. 2.8, the language $\mathcal{L}_0^\star$ is channel-bounded by $B = 2^{N^2}.(N+1).B_0$ where $N = \mathrm{Card}(\mathcal{I})$. We use here Lemma 2.3 with $\Sigma = \Sigma_{\mathcal{I}} \times [0, B]$. Then $\pi_1^{-1}(\mathcal{L}_0)$ is recognizable in $\mathbb{M}(\Sigma, \|)$ (Corollary 2.4). Moreover $\pi_1^{-1}(\mathcal{L}_0)$ is connected because each MSC of $\mathcal{L}_0$ consists of strongly connected MSCs. Therefore $\mathcal{L}^\dagger = \pi_1^{-1}(\mathcal{L}_0)^\star$ is recognizable too (Th. 2.12). Consider now the language $\mathcal{L}_B \subseteq \mathbb{M}(\Sigma, \|)$ that consists of the traces t such that $\pi_1(t)$ is a basic MSC of channel-width at most B. We can show that $\mathcal{L}_B$ is definable in MSO logic, hence it is recognizable in $\mathbb{M}(\Sigma, \|)$ [34]. Then $\mathcal{L}^\dagger \cap \mathcal{L}_B$ is recognizable in $\mathbb{M}(\Sigma, \|)$. Consequently, the set of linear extensions $L^\dagger = \mathrm{LE}(\mathcal{L}^\dagger \cap \mathcal{L}_B)$ is recognizable in $\Sigma^\star$ and its image through $\pi_1 : \Sigma^\star \to \Sigma_{\mathcal{I}}^\star$ is a recognizable language of $\Sigma_{\mathcal{I}}^\star$. To conclude we can show that $\mathcal{L}^\dagger \cap \mathcal{L}_B = \pi_1^{-1}(\mathcal{L}_0^\star)$ hence $\pi_1(L^\dagger) = \mathrm{LE}(\mathcal{L}_0^\star)$. ∎

Note finally that the restriction of communication graphs to active instances makes sense: with M_2 of Fig. 3, $\{M_2\}^\star$ is obviously not regular.

3 Connecting Two Star Problems

The subclass of sc-HMSCs describes precisely all regular languages of MSCs (Corollary 2.13). However allowing iteration over sets of strongly connected MSCs only can be considered to be too restrictive. We investigate now how we could weaken this restriction while keeping the same expressive power. By Lemma 2.10, it suffices to forbid the iteration of a regular language whenever the resulting language is not regular. However, this might lead to an intractable criterion. Indeed, we shall prove here the following result.

THEOREM 3.1. *Consider the two following problems:*

Pb_1*: Given a finite independence alphabet* $(\Sigma, \|)$ *and a recognizable language* $\mathcal{L}$ *of* $\mathbb{M}(\Sigma, \|)$*, decide whether* $\mathcal{L}^\star$ *is recognizable in* $\mathbb{M}(\Sigma, \|)$*.*

Pb_2*: Given a finite set of instances* $\mathcal{I}$ *and a regular language* $\mathcal{L}$ *of* bMSC *over* $\mathcal{I}$*, decide whether* $\mathcal{L}^\star$ *is regular.*

Then Pb_1 *is decidable if, and only if,* Pb_2 *is decidable.*

Recall that Pb_1 is known as "the Star Problem in trace monoids," and it is still an open question to know whether it is decidable [21]. The proof of Theorem 3.1 proceeds from Propositions 3.2 and 3.6 below.

It is well-known that some classical telecommunication protocols — such as the alternating bit protocol — cannot be described by HMSCs because they are not *finitely generated* languages. For this reason, compositional MSCs are introduced in [13]: they enable to describe any regular language of MSCs by a rational expression. That is why, we stress that *an important aspect of Theorem 3.1 is that we do not restrict to finitely generated languages in the statement of* Pb_2. In fact, Propositions 3.2 and 3.6 also show that Theorem 3.1 still holds if we restrict to finitely generated languages. First, the "if" part of Theorem 3.1.

PROPOSITION 3.2. *Consider the following variation of* Pb_2.

Pb'_2*: Given a finite set of instances* $\mathcal{I}$ *and a regular* finitely generated *language* $\mathcal{L}$ *of* bMSC *over* $\mathcal{I}$*, decide whether* $\mathcal{L}^\star$ *is regular.*

If Pb'_2 *is decidable then the Star Problem* Pb_1 *of Theorem 3.1 is decidable too.*

Proof. Let $(\Sigma, \|)$ be a finite independence alphabet. There exists a finite set of instances $\mathcal{I}$ and a family of basic MSCs $(M_a)_{a\in\Sigma}$ with the following properties:

- $a\|b \Rightarrow M_a \cdot M_b = M_b \cdot M_a$;
- the morphism $\psi : \mathbb{M}(\Sigma, \|) \to \text{bMSC}$ such that $\psi(a) = M_a$ is *one-to-one.*
- for any recognizable language $\mathcal{L} \in \mathbb{M}(\Sigma, \|)$, given a finite automaton over Σ that recognizes $\text{LE}(\mathcal{L})$, one can effectively build a finite automaton over $\Sigma_\mathcal{I}$ that recognizes $\text{LE}(\psi(\mathcal{L}))$.

Note here that ψ is well-defined because $a\|b \Rightarrow M_a \cdot M_b = M_b \cdot M_a$. Moreover $\psi(\mathbb{M}(\Sigma, \|))$ is finitely generated. (See [15] for an example of such a family that was used to prove Th. 2.5).

We show that we can decide whether $\mathcal{L}_0^\star$ is recognizable in $\mathbb{M}(\Sigma, \|)$ when $\mathcal{L}_0$ is a recognizable language of $\mathbb{M}(\Sigma, \|)$ given by a finite automaton over Σ that recognizes $\text{LE}(\mathcal{L}_0)$. We can effectively construct an automaton $\mathcal{A}'$ that recognizes $\text{LE}(\psi(\mathcal{L}_0))$. Then $\psi(\mathcal{L}_0)$ is finitely generated. Thus, we need only to show that $\mathcal{L}_0^\star$ is recognizable if and only if $\psi(\mathcal{L}_0)^\star$ is regular. Assume first that $\mathcal{L}_0^\star$ is recognizable. Let $\mathcal{A}_0$ be a finite automaton over Σ that recognizes $\text{LE}(\mathcal{L}_0^\star)$. Then there is a finite automaton $\mathcal{A}'$ over $\Sigma_\mathcal{I}$ that recognizes $\text{LE}(\psi(\mathcal{L}_0^\star))$. Since ψ is a monoid morphism, $\psi(\mathcal{L}_0^\star) = \psi(\mathcal{L}_0)^\star$ hence $\psi(\mathcal{L}_0)^\star$ is regular. Conversely, assume that $\psi(\mathcal{L}_0)^\star$ is regular. Then $\mathcal{L}_0^\star = \psi^{-1}(\psi(\mathcal{L}_0)^\star)$ because ψ is one-to-one and $\psi(\mathcal{L}_0^\star) = \psi(\mathcal{L}_0)^\star$. Since $\psi(\mathcal{L}_0)^\star$ is regular, it is recognizable in bMSC hence $\mathcal{L}_0^\star$ is recognizable in $\mathbb{M}(\Sigma, \|)$. ∎

The other direction of Theorem 3.1 turns out to be more difficult and is, in our opinion, the most interesting part of this paper. The reason is that we cannot simply use Kuske's relabeling technique (Lemma 2.3) because the mapping π_1 does not preserves products: consider for instance MSC M_1 of Fig. 2 and let $t_1 \in \mathbb{M}(\Sigma, \|)$ be such that $\pi_1(t_1) = M_1$. Then t_1 is simply a word $(i!j, n).(j?i, n)$ for some $n \in \mathbb{N}$. We observe here that $\pi_1(t_1 \cdot t_1) \neq M_1 \cdot M_1$ (cf. Fig. 1 and 5).

To cope with this algebraic flaw, we shall adapt the representation technique as follows.

Definition 3.3. *Let B be a positive integer and let $(\Sigma, \|)$ be the corresponding independence alphabet defined in Lemma 2.3. We denote by $\rho : \mathbb{P}(\Sigma) \to \mathbb{P}(\Sigma_{\mathcal{I}})$ the function from the pomsets over Σ to the pomsets over $\Sigma_{\mathcal{I}}$ such that $t = (E, \preccurlyeq, \xi)$ maps to $(E, \preccurlyeq^{\dagger}, \pi_1 \circ \xi)$ where $\preccurlyeq^{\dagger}$ is the transitive closure of*
$\{(e, f) \in E^2 \mid \xi(e) \not\| \xi(f) \wedge e \preccurlyeq f \wedge \forall i, j \in \mathcal{I}, (\pi_1(\xi(e)) \neq j?i \vee \pi_1(\xi(f)) \neq i!j)\}$
For any language $\mathcal{L} \in \mathrm{bMSC}$, we denote by $\rho^{-1}(\mathcal{L})$ the set of all traces $t \in \mathbb{M}(\Sigma, \|)$ such that $\rho(t) \in \mathcal{L}$.

We first observe that Lemma 2.3 yields

Corollary 3.4. *For any basic MSC M with channel-width at most B, there exists a Mazurkiewicz trace $t \in \mathbb{M}(\Sigma, \|)$ such that $\rho(t) = M$.*

Now the map $\rho : \mathbb{P}(\Sigma) \to \mathbb{P}(\Sigma_{\mathcal{I}})$ satisfies two crucial properties that are not fulfilled by $\pi_1 : \mathbb{P}(\Sigma) \to \mathbb{P}(\Sigma_{\mathcal{I}})$. First, $\rho^{-1}(\mathrm{bMSC})$ is a sub-monoid of $\mathbb{M}(\Sigma, \|)$ and $\rho : \rho^{-1}(\mathrm{bMSC}) \to \mathrm{bMSC}$ is a monoid morphism. Second $\rho^{-1}(\mathcal{L}^{\star}) = \rho^{-1}(\mathcal{L})^{\star}$ for any language $\mathcal{L} \subseteq \mathrm{bMSC}$. For this, it suffices to check the following property.

Lemma 3.5. *With the notations of Def. 3.3, let M_1 and M_2 be two basic MSCs and let t be a trace over $\mathbb{M}(\Sigma, \|)$ such that $\rho(t) = M_1 \cdot M_2$. Then there are two traces t_1 and t_2 such that $\rho(t_1) = M_1$, $\rho(t_2) = M_2$ and $t = t_1 \cdot t_2$.*

Proof. We consider $M_1 = (E_1, \preccurlyeq_1, \xi_1)$ and $M_2 = (E_2, \preccurlyeq_2, \xi_2)$ two basic MSCs. We may assume here that $E_1 \cap E_2 = \emptyset$. Let $t = (E, \preccurlyeq, \xi_t)$ be a trace over $\mathbb{M}(\Sigma, \|)$ such that $\rho(t) = M_1 \cdot M_2$. Then $E = E_1 \cup E_2$ and E_1 is an ideal of $\rho(t)$. Actually, the proof follows from the key observation that E_1 is an ideal of t as well. We proceed by contradiction. We can show that there are $e \in E_2$ and $f \in E_1$ such that $e \prec\!\!\!\!- _t f$. Since t is a trace, $\xi(e) \not\| \xi(f)$. But $\neg(e \preccurlyeq_{\rho(t)} f)$ since E_1 is an ideal of $\rho(t)$. Therefore, $\xi_2(e) = j?i$ and $\xi_1(f) = i!j$. Now M_1 is a basic MSC so there exists an event $e_0 \in E_1$ such that $\xi(e_0) = j?i$ and $\#^{j?i}_{M_1}(\downarrow e_0) = \#^{i!j}_{M_1}(\downarrow f)$. This implies $\#^{j?i}_{M_1 \cdot M_2}(\downarrow e_0) = \#^{i!j}_{M_1 \cdot M_2}(\downarrow f)$. Hence $f \preccurlyeq_{\rho(t)} e_0$ because $M_1 \cdot M_2$ is also a basic MSC. But $e_0 \preccurlyeq_{\rho(t)} e$ because $e \in E_2$, $e_0 \in E_1$ and $\mathrm{Ins}(e_0) = \mathrm{Ins}(e)$. Therefore $f \preccurlyeq_{\rho(t)} e$. This contradicts $e \prec\!\!\!\!- _t f$. ∎

Finally, similarly to π_1, we observe that $\rho^{-1}(\mathrm{bMSC})$ is definable in MSO logic. From this we can adapt Corollary 2.4 to prove a third technical remark: for any regular language $\mathcal{L} \subseteq \mathrm{bMSC}$, $\rho^{-1}(\mathcal{L})$ is recognizable in $\mathbb{M}(\Sigma, \|)$.

Proposition 3.6. *Consider the following variation of* Pb_1.
Pb_1'*: Given a finite independence alphabet $(\Sigma, \|)$ such that each action appears in at most two maximal cliques of the dependence graph $(\Sigma, \not\|)$ and a recognizable language $\mathcal{L}$ of $\mathbb{M}(\Sigma, \|)$, decide whether $\mathcal{L}^{\star}$ is recognizable.*
If Pb_1' *is decidable then* Pb_2 *of Theorem 3.1 is decidable too.*

Proof. Let $\mathcal{L}$ be a regular language of bMSC described by a finite automaton over $\Sigma_{\mathcal{I}}$ that recognizes $\mathrm{LE}(\mathcal{L})$. Let B be the number of states of $\mathcal{A}$. Then $\mathcal{L}$ is channel-bounded by B. Clearly, we can decide from $\mathcal{A}$ whether each connected component in the communication graph of all MSCs of $\mathcal{L}$ is strongly connected.

If this is not the case, then $\mathcal{L}^\star$ is not channel-bounded (Th. 2.8) hence not regular. Therefore, we can assume now that this is the case. Then, again by Th. 2.8, $\mathcal{L}^\star$ is channel-bounded by $B' = 2^{N^2}.(N+1).B$. We use now the notations of Def. 3.3 with $\Sigma = \Sigma_{\mathcal{I}} \times [0, B']$. Since $\mathcal{L}$ is regular, $\mathcal{L}_0 = \rho^{-1}(\mathcal{L})$ is recognizable in $\mathbb{M}(\Sigma, \|)$; moreover *an automaton recognizing* $\mathrm{LE}(\mathcal{L}_0)$ *can effectively be computed from* $\mathcal{A}$. To conclude this proof, we show that $\mathcal{L}_0^\star$ is recognizable in $\mathbb{M}(\Sigma, \|)$ if, and only if, $\mathcal{L}^\star$ is regular in bMSC. By the second observation, we have $\rho^{-1}(\mathcal{L}^\star) = \rho^{-1}(\mathcal{L})^\star = \mathcal{L}_0^\star$. Assume first that $\mathcal{L}^\star$ is regular; then (third observation) $\rho^{-1}(\mathcal{L}^\star)$ is recognizable in $\mathbb{M}(\Sigma, \|)$. Conversely, assume that $\mathcal{L}_0^\star$ is recognizable in $\mathbb{M}(\Sigma, \|)$. Then $\mathrm{LE}(\mathcal{L}_0^\star)$ is recognizable in $\Sigma^\star$ hence definable in MSO logic. We consider now lexicographic normal forms of pomsets over $\Sigma_{\mathcal{I}}$ with the condition that $j?i < i!j$ for any two distinct instances i and j. Since ρ is a morphism, $\rho(\mathcal{L}_0^\star) \subseteq \mathcal{L}^\star$, hence $\rho(\mathcal{L}_0^\star) = \mathcal{L}^\star$. Therefore $\mathcal{L}^\star$ is the set of basic MSCs M whose lexicographic normal forms belong to $\pi_1(\mathrm{LE}(\mathcal{L}_0^\star))$. Thus $\mathcal{L}^\star$ is MSO definable [10,11]. Since it is also channel-bounded, it is regular [17]. ∎

Discussion. Another corollary of Prop. 3.2 and 3.6 is the following reduction of the Star Problem:

Corollary 3.7. *The Star Problem is decidable for all independence alphabets (*Pb_1 *of Theorem 3.1) if, and only if, it is decidable for all independence alphabets such that each action appears in at most 2 maximal cliques (*Pb_1' *of Prop. 3.6).*

To our knowledge, this reduction does not follow from known results. It remains however unclear to us whether this reduction could be useful to provide a new approach for an answer to this difficult question.

Acknowledgments. Many thanks to Dietrich Kuske for numerous motivating discussions on this subject and several suggestions to simplify the proofs and improve the presentation of the results.

References

1. Alur R. and Yannakakis M.: *Model Checking of Message Sequence Charts.* CONCUR'99, LNCS **1664** (1999) 114–129
2. Arnold A.: *An extension of the notion of traces and asynchronous automata.* Theoretical Informatics and Applications **25** (1991) 355–393
3. Ben-Abdallah H. and Leue S.: *Syntactic Analysis of Message Sequence Chart Specifications.* Technical report 96-12 (University of Waterloo, Canada, 1996)
4. Ben-Abdallah H. and Leue S.: *Syntactic Detection of Process Divergence and Non-local Choice in Message Sequence Charts.* TACAS'97, LNCS **1217** (1997) 259–274
5. Booch G., Jacobson I. and Rumbough J.: *Unified Modelling Language User Guide.* (Addison-Wesley, 1997)
6. Büchi J.R.: *Weak second-order arithmetic and finite automata.* Z. Math. Logik Grundlagen Math. **6** (1960) 66–92
7. Caillaud B., Darondeau Ph., Hélouët L. and Lesventes G.: *HMSCs as partial specifications... with PNs as completions.* Proc. of MOVEP'2k, Nantes (2000) 87–103
8. Cori R., Métivier Y. and Zielonka W.: *Asynchronous mappings and asynchronous cellular automata.* Information and Computation **106** (1993) 159–202

9. Diekert V. and Rozenberg G.: *The Book of Traces.* (World Scientific, 1995)
10. Droste M. and Kuske D.: *Logical definability of recognizable and aperiodic languages in concurrency monoids.* LNCS **1092** (1996) 233–251
11. Ebinger W. and Muscholl A.: *Logical definability on infinite traces.* Theoretical Comp. Science **154** (1996) 67–84
12. Gastin P., Ochmański E., Petit A. and Rozoy, B.: *On the decidability of the star problem.* Information Processing Letters **44** (1992) 65–71
13. Gunter E.L., Muscholl A. and Peled D.: *Compositional Message Sequence Charts.* TACAS 2001, LNCS (2001) – To appear.
14. Hélouët L., Jard C. and Caillaud B.: *An effective equivalence for sets of scenarios represented by HMSCs.* Technical report, PI-1205 (IRISA, Rennes, 1998)
15. Henriksen J.G, Mukund M., Narayan Kumar, K. and Thiagarajan P.S.: *Towards a theory of regular MSC languages.* Technical report (BRICS RS-99-52, 1999)
16. Henriksen J.G., Mukund M., Narayan Kumar K. and Thiagarajan P.S.: *On message sequence graphs and finitely generated regular MSC language.* LNCS **1853** (2000) 675–686
17. Henriksen J.G., Mukund M., Narayan Kumar K. and Thiagarajan P.S.: *Regular collections of message sequence charts.* MFCS 2000, LNCS **1893** (2000) 405–414
18. Holzmann G.J.: *Early Fault Detection.* TACAS'96, LNCS **1055** (1996) 1–13
19. Husson J.-Fr. and Morin R.: *On Recognizable Stable Trace Languages.* FoSSaCS 2000, LNCS **1784** (2000) 177–191
20. ITU-TS: *Recommendation Z.120: Message Sequence Charts.* (Geneva, 1996)
21. Kirsten D. and Richomme G.: *Decidability Equivalence Between the Star Problem and the Finite Power Problem in Trace Monoids.* Technical Report ISSN 1430-211X, TUD/FI99/03 (Dresden University of Technology, 1999)
22. Kuske D. and Morin R.: *Pomsets for Local Trace Languages: Recognizability, Logic and Petri Nets.* CONCUR 2000, LNCS **1877** (2000) 426–441
23. Lamport L.: *Time, Clocks and the Ordering of Events in a Distributed System.* Comm. of the ACM, vol. 21, N **27** (1978) – ACM
24. Mazurkiewicz A.: *Concurrent program schemes and their interpretations.* Aarhus University Publication (DAIMI PB-78, 1977)
25. Mukund M., Narayan Kumar K. and Sohoni M.: *Synthesizing distributed finite-state systems from MSCs.* CONCUR 2000, LNCS **1877** (2000) 521–535
26. Muscholl A., Peled D. and Su Z.: *Deciding Properties for Message Sequence Charts.* FoSSaCS'98, LNCS **1378** (1998) 226–242
27. Muscholl A.: *Matching Specifications for Message Sequence Charts.* FoSSaCS'99, LNCS **1578** (1999) 273–287
28. Muscholl A. and Peled D.: *Message sequence graphs and decision problems on Mazurkiewicz traces.* Proc. of MFCS'99, LNCS **1672** (1999) 81–91
29. Nielsen M., Plotkin G. and Winskel G.: *Petri nets, events structures and domains, part 1.* Relationships between Models of Concurrency, TCS **13** (1981) 85–108
30. Nielsen M., Sassone V. and Winskel G.: *Relationships between Models of Concurrency.* Rex'93: A decade of concurrency, LNCS **803** (1994) 425–475
31. Ochmański E.: *Regular behaviour of concurrent systems.* Bulletin of the EATCS **27** (Oct. 1985) 56–67
32. Pratt V.: *Modelling concurrency with partial orders.* Int. J. of Parallel Programming **15** (1986) 33–71
33. Sakarovitch J.: *The "last" decision problem for rational trace languages.* Proc. LATIN'92, LNCS **583** (1992) 460–473
34. Thomas W.: *On logical definability of trace languages.* Technical University of Munich, report TUM-I9002 (1990) 172–182

Verified Bytecode Verifiers

Tobias Nipkow

Fakultät für Informatik, Technische Universität München
http://www.in.tum.de/~nipkow/

Abstract. Using the theorem prover Isabelle/HOL we have formalized and proved correct an executable bytecode verifier in the style of Kildall's algorithm for a significant subset of the Java Virtual Machine. First an abstract framework for proving correctness of data flow based type inference algorithms for assembly languages is formalized. It is shown that under certain conditions Kildall's algorithm yields a correct bytecode verifier. Then the framework is instantiated with a model of the JVM.

1 Introduction

Over the past few years there has been considerable interest in formal models for Java and the JVM, and in particular its bytecode verifier (*BCV*). So far most of the work has concentrated on abstract models of particularly tricky aspects of the JVM, in particular the idiosyncratic notion of "subroutines". This paper complements those studies by focussing on machine-checked proofs of executable models. Its distinctive features are:

- The first machine-checked proof of correctness of a BCV implementation for a nontrivial subset of the JVM.
- The BCV is an almost directly executable functional program (which can be generated automatically from its Isabelle/HOL formalization). The few non-executable constructs (some choice functions) are easily implemented.
- The work is modular: the BCV (and its correctness proof!) becomes a simple instance of a general framework for data flow analysis.
- Almost all details of the model are presented. The complete formalization, including proofs, is available via the author's home page.

Thus the novelty is not in the actual mathematics but in setting up a framework that matches the needs of the JVM, instantiating it with a description of the JVM, and doing this in complete detail, down to an executable program, and including all the proofs. Moreover this is not an isolated development but is based firmly on existing formalizations of Java and the JVM in Isabelle/HOL [11,12, 13]. From them it also inherits the absence of subroutines, object initialization, and exception handling. Although our only application is the JVM, our modular framework is in principle applicable to other "typed assembly languages" as well; hence the plural in the title.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 347–363, 2001.

What does the BCV do and what does it guarantee? The literature already contains a number of (partial) answers to this question. We follow the type system approach of Stata, Abadi and others [16,4,5,14,7]. The type systems check a program w.r.t. additional type annotations that provide the missing typing of storage locations for each instruction. These type systems are then shown to guarantee type soundness, i.e. absence of type errors during execution (which was verified formally by Pusch [13] for a subset of the system by Qian [14]). Only Qian [15] proves the correctness of an algorithm for turning his type checking rules into a data flow analyzer. However, his algorithm is still quite abstract.

Closely related is the work by Goldberg [6] who rephrases and generalizes the overly concrete description of the BCV given in [9] as an instance of a generic data flow framework. Work towards a verified implementation in the SPECWARE system is sketched by Coglio *et al.* [3]. Although we share the lattice-theoretic foundations with this work and it appears to consider roughly the same instruction set, it is otherwise quite different: whereas we solve the data flow problem directly, they generate constraints to be solved separately, which is not described. Furthermore, they state desired properties axiomatically but do not prove them. Casset and Lanet [2] also sketch work using the B method to model the JVM. However, their subset of the JVM is extremely simple (it does not even include classes), and it remains unclear what exactly they have proved. Based on the work of Freund and Mitchell [4] Bertot [1] has recently used the Coq system to prove the correctness of a bytecode verifier that handles object initialization (but again without classes).

The rest of the paper is structured as follows. The basic datatypes and semilattices required for data flow analysis are introduced in §2. Our abstract framework relating type systems, data flow analysis, and bytecode verification is set up in §3. It is proved that under certain conditions bytecode verification determines welltypedness. In §4, Kildall's algorithm, an iterative data flow analysis, is shown to be a bytecode verifier. Finally, in §5, the results of the previous sections are used to derive a bytecode verifier for a subset of the JVM.

2 Types, Orders, and Semilattices

This section introduces the basic mathematical concepts and their formalization in Isabelle/HOL. Note that HOL distinguishes types and sets: types are part of the meta-language and of limited expressiveness, whereas sets are part of the object language and very expressive.

2.1 Basic Types

Isabelle's type system is similar to ML's. There are the basic types *bool*, *nat*, and *int*, and the polymorphic types α *set* and α *list*, and a conversion function set from lists to sets. The "cons" operator on lists is the infix #, concatenation the infix @. The length of a list is denoted by size. The i-th element (starting with 0!) of list xs is denoted by $xs!i$. Overwriting the i-th element of a list xs with a

new value x is written $xs[i := x]$. Recursive datatypes are introduced with the **datatype** keyword. The remainder of this section introduces the HOL-formalization of the basic lattice-theoretic concepts required for data flow analysis and its application to the JVM.

2.2 Partial Orders

Partial orders are formalized as binary predicates. Based on the type synonym $\alpha\ ord = \alpha \to \alpha \to bool$ and the notations $x \leq_r y\ =\ r\ x\ y$ and $x <_r y\ =\ (x \leq_r y \wedge x \neq y)$ we say that $r :: \alpha\ ord$ is a **partial order** iff the predicate $\mathsf{order} :: \alpha\ ord \to bool$ holds for r:

$$\begin{aligned}\mathsf{order}\ r = {} & (\forall x.\ x \leq_r x) \wedge (\forall xy.\ x \leq_r y \wedge y \leq_r x \longrightarrow x = y) \wedge \\ & (\forall xyz.\ x \leq_r y \wedge y \leq_r z \longrightarrow x \leq_r z)\end{aligned}$$

We say that r satisfies the **ascending chain condition** if there is no infinite ascending chain $x_0 <_r x_1 <_r \cdots$ and call T a **top element** if $x \leq_r T$ for all x.

2.3 Semilattices

Based on the type synonyms $\alpha\ binop = \alpha \to \alpha \to \alpha$ and $\alpha\ sl = \alpha\ set \times \alpha\ ord \times \alpha\ binop$ and the supremum notation $x +_f y\ =\ f\ x\ y$ we say that $(A, r, f) :: \alpha\ sl$ is a **semilattice** iff the predicate $\mathsf{semilat} :: \alpha\ sl \to bool$ holds:

$$\begin{aligned}\mathsf{semilat}(A, r, f) = {} & \mathsf{order}\ r\ \wedge\ \mathsf{closed}\ A\ f\ \wedge\ (\forall xy \in A.\ x \leq_r x +_f y)\ \wedge \\ & (\forall xy \in A.\ y \leq_r x +_f y)\ \wedge\ (\forall xyz \in A.\ x \leq_r z \wedge y \leq_r z \longrightarrow x +_f y \leq_r z)\end{aligned}$$

where $\mathsf{closed}\ A\ f\ =\ \forall xy \in A.\ x +_f y \in A$

Usually data flow analysis is phrased in terms of infimum semilattices. We have chosen a supremum semilattice because it fits better with our intended application, where the ordering is the subtype relation and the join of two types is the least common supertype (if it exists).

We will now look at a few datatypes and the corresponding semilattices which are required for the construction of the JVM bytecode verifier. In order to avoid name clashes, Isabelle provides separate names spaces for each *theory*, where a theory is like a module in a programming language. Qualified names are of the form $\mathsf{Theoryname.localname}$.

2.4 The Error Type and *err*-Semilattices

Theory Err introduces an error element to model the situation where the supremum of two elements does not exist. We introduce both a datatype and an equivalent construction on sets:

$$\textbf{datatype}\ \alpha\ err = \mathsf{Err} \mid \mathsf{OK}\ \alpha \qquad \mathsf{err}\ A = \{\mathsf{Err}\} \cup \{\mathsf{OK}\ a \mid a \in A\}$$

Orderings r on α can be lifted to $\alpha\ err$ by making Err the top element:

$$\begin{array}{lll} \mathsf{le}\ r\ (\mathsf{OK}\ x)\ (\mathsf{OK}\ y) & = & x \le_r y \\ \mathsf{le}\ r\ \_\ \mathsf{Err} & = & \mathsf{True} \\ \mathsf{le}\ r\ \mathsf{Err}\ (\mathsf{OK}\ y) & = & \mathsf{False} \end{array}$$

We proved that le preserves the ascending chain condition.

The following lifting functional is frequently useful:

$$\begin{array}{lll} \mathsf{lift2}\ f\ (\mathsf{OK}\ x)\ (\mathsf{OK}\ y) & = & f\ x\ y \\ \mathsf{lift2}\ f\ \_\ \_ & = & \mathsf{Err} \end{array}$$

This brings us to the genuinely new notion of an *err*-semilattice. It is a variation of a semilattice with top element. Because the behaviour of the ordering and the supremum on the top element are fixed, it suffices to say how they behave on non-top elements. Thus we can represent a semilattice with top element Err compactly by a triple of type *esl*:

$$\alpha\ ebinop = \alpha \to \alpha \to \alpha\ err \qquad \alpha\ esl = \alpha\ set \times \alpha\ ord \times \alpha\ ebinop$$

Conversion between the types *sl* and *esl* is easy:

$$\begin{array}{ll} \mathsf{esl} :: \alpha\ sl \to \alpha\ esl & \mathsf{sl} :: \alpha\ esl \to \alpha\ err\ sl \\ \mathsf{esl}(A, r, f) = (A, r, \lambda xy.\ \mathsf{OK}(f\ x\ y)) & \mathsf{sl}(A, r, f) = (\mathsf{err}\ A, \mathsf{le}\ r, \mathsf{lift2}\ f) \end{array}$$

Now we define $L :: \alpha\ esl$ to be an ***err*-semilattice** iff $\mathsf{sl}\ L$ is a semilattice. It follows easily that $\mathsf{esl}\ L$ is an *err*-semilattice if L is a semilattice. The supremum operation of $\mathsf{sl}(\mathsf{esl}\ L)$ is useful on its own:

$$\mathsf{sup}\ f = \mathsf{lift2}(\lambda xy.\ \mathsf{OK}(x +_f y))$$

In a strongly typed environment like HOL we found *err*-semilattices easier to work with than semilattices with top element.

2.5 The Option Type

Theory Opt introduces type *option* and set opt as duals to type *err* and set err,

$$\textbf{datatype}\ \alpha\ option = \mathsf{None} \mid \mathsf{Some}\ \alpha \qquad \mathsf{opt}\ A = \{\mathsf{None}\} \cup \{\mathsf{Some}\ a \mid a \in A\}$$

an ordering that makes None the bottom element, and a corresponding supremum operation:

$$\begin{array}{llll} \mathsf{le}\ r\ (\mathsf{Some}\ x)\ (\mathsf{Some}\ y) & = x \le_r y & \mathsf{sup}\ f\ (\mathsf{Some}\ x)\ (\mathsf{Some}\ y) & = \mathsf{Some}(f\ x\ y) \\ \mathsf{le}\ r\ \mathsf{None}\ \_ & = \mathsf{True} & \mathsf{sup}\ f\ \mathsf{None}\ z & = z \\ \mathsf{le}\ r\ (\mathsf{Some}\ x)\ \mathsf{None} & = \mathsf{False} & \mathsf{sup}\ f\ z\ \mathsf{None} & = z \end{array}$$

We proved that function $\mathsf{sl}(A, r, f) = (\mathsf{opt}\ A, \mathsf{le}\ r, \mathsf{sup}\ f)$ maps semilattices to semilattices and that le preserves the ascending chain condition.

2.6 Products

Theory Product provides what is known as the *coalesced* product, where the top elements of both components are identified. In terms of *err*-semilattices, this is

$$\begin{array}{l}\mathsf{esl} :: \alpha\ esl \to \beta\ esl \to (\alpha \times \beta)\ esl\\ \mathsf{esl}\ (A, r_A, f_A)\ (B, r_B, f_B) = (A \times B, \mathsf{le}\ r_A\ r_B, \mathsf{sup}\ f_A\ f_B)\end{array}$$

$$\begin{array}{l}\mathsf{le} :: \alpha\ ord \to \beta\ ord \to (\alpha \times \beta)\ ord\\ \mathsf{le}\ r_A\ r_B = \lambda(a_1, b_1)(a_2, b_2).\ a_1 \leq_{r_A} a_2 \wedge b_1 \leq_{r_B} b_2\end{array}$$

$$\begin{array}{l}\mathsf{sup} :: \alpha\ ebinop \to \beta\ ebinop \to (\alpha \times \beta) ebinop\\ \mathsf{sup}\ f\ g = \lambda(a_1, b_1)(a_2, b_2).\ \mathsf{Err.sup}\ (\lambda xy.(x, y))\ (a_1 +_f a_2)\ (b_1 +_g b_2)\end{array}$$

Note that we use $\times$ both on the type and set level.

We have shown that if both L_1 and L_2 are *err*-semilattices, so is $\mathsf{esl}\ L_1\ L_2$, and that if both r_A and r_B satisfy the ascending chain condition, so does $\mathsf{le}\ r_A\ r_B$.

2.7 Lists of Fixed Length

Theory Listn provides the concept of lists of a given length over a given set. In HOL, this is formalized as a set rather than a type:

$$\mathsf{list}\ n\ A = \{xs \mid \mathsf{size}\ xs = n \wedge \mathsf{set}\ xs \subseteq A\}$$

This set can be turned into a semilattice in a componentwise manner, essentially viewing it as an n-fold cartesian product:

$$\begin{array}{ll}\mathsf{sl} :: nat \to \alpha\ sl \to \alpha\ list\ sl & \mathsf{le} :: \alpha\ ord \to \alpha\ list\ ord\\ \mathsf{sl}\ n\ (A, r, f) = (\mathsf{list}\ n\ A, \mathsf{le}\ r, \mathsf{map2}\ f) & \mathsf{le}\ r = \mathsf{list\_all2}\ (\lambda xy.\ x \leq_r y)\end{array}$$

where $\mathsf{map2} :: (\alpha \to \beta \to \gamma) \to \alpha\ list \to \beta\ list \to \gamma\ list$ and $\mathsf{list\_all2} :: (\alpha \to \beta \to bool) \to \alpha\ list \to \beta\ list \to bool$ are the obvious functions. We introduce the notation $xs \leq_{[r]} ys = xs \leq_{(\mathsf{le}\ r)} ys$. We have shown (by induction on n) that if L is a semilattice, so is $\mathsf{sl}\ n\ L$, and that if r is a partial order and satisfies the ascending chain condition, so does $\mathsf{le}\ r$.

In case we want to combine lists of different lengths, or if the supremum on the elements of the list may return Err, we use the following function:

$$\begin{array}{l}\mathsf{sup} :: (\alpha \to \beta \to \gamma\ err) \to \alpha\ list \to \beta\ list \to \gamma\ list\ err\\ \mathsf{sup}\ f\ xs\ ys = \texttt{if}\ \mathsf{size}\ xs = \mathsf{size}\ ys\ \texttt{then}\ \mathsf{coalesce}(\mathsf{map2}\ f\ xs\ ys)\ \texttt{else}\ \mathsf{Err}\\ \mathsf{coalesce}\ [] = \mathsf{OK}\ []\\ \mathsf{coalesce}\ (e\#es) = \mathsf{Err.sup}\ (\lambda x\ xs.\ x\#xs)\ e\ (\mathsf{coalesce}\ es)\end{array}$$

This corresponds to the coalesced product. Below we also need the structure of all lists up to a specific length:

$$\begin{array}{l}\mathsf{upto\_esl} :: nat \to \alpha\ esl \to \alpha\ list\ esl\\ \mathsf{upto\_esl}\ n\ (A, r, f) = (\bigcup_{i \leq n} \mathsf{list}\ i\ A, \mathsf{le}\ r, \mathsf{sup}\ f)\end{array}$$

We have shown that if L is an *err*-semilattice, so is $\mathsf{upto\_esl}\ n\ L$.

3 Relating Type Checking and Data Flow Analysis

The purpose of this section is to set up an abstract framework for relating type checking and data flow analysis of machine code, i.e. lists of instructions. We assume that instructions may be typed (e.g. distinguishing integer from floating point addition) but storage locations are not necessarily typed and may also change their type during execution. Thus it is necessary to infer the type of each storage location at each instruction to see if the instruction will manipulate values of the required type. To keep things abstract, we do not fix the type system or the machine architecture. We simply assume that our model contains a type of **states** that characterizes the state of the machine. This state type is a parameter of our setup and will be represented by the type variable σ. Note that σ is intended not to represent values but their abstraction, types. For example, in a register machine σ would be a list of types, one for each register (roughly speaking). We can now define a **program type**, i.e. the type of a program, simply as a list of state types: each element in the list characterizes the state of the machine before execution of the corresponding instruction.

In order to lessen the confusion between types in the programming language under consideration and types in our modeling language, the latter are sometimes referred to as **HOL types**. For example, σ is a HOL type that represents part of the programming language type system.

In this abstract setting, we do not yet have to talk about the instruction sequences themselves. They will be hidden inside functions that characterize their behaviour. These functions form the parameters of our model, namely the type system and the data flow analyzer. In the Isabelle formalization, these functions are parameters of everything. In this article, we pretend they are global constants, thus increasing readability.

3.1 Welltyped Instructions

The type system is characterized by a function and an order with top element:

wti :: $\sigma\ list \to nat \to bool$ stands for *w*ell *t*yped *i*nstruction: wti $ss\ p$ is true iff instruction number p is welltyped w.r.t. the program type ss. Thus wti characterizes welltypedness of the instructions of a particular program, which remains implicit. When instantiating the framework, wti will be instantiated by the partial application of some type system to a program.

r :: $(\sigma \times \sigma)set$, an ordering relation on σ representing the subtype relation lifted to program states. Thus our framework can deal with programming languages with subtypes.

$\top$ :: σ is the top element w.r.t. r and should be thought of as the inconsistent state, indicating a type error.

The program embodied by wti is welltyped w.r.t. some program type ss iff all instructions are welltyped with non-$\top$ states:

$$\mathsf{welltyping}\ ss = \forall p < \mathsf{size}(ss).\ \mathsf{wti}\ ss\ p \wedge ss!p \neq \top$$

3.2 Abstract Semantics

The semantics of a program is characterized by the functions

step :: $nat \to \sigma \to \sigma$ is the abstract execution function: $\mathsf{step}\ p\ s$ is the result of executing instruction p starting in state s. In the literature $\mathsf{step}\ p$ is called the *transfer function* or *flow function* associated with instruction p.

succs :: $nat \to nat\ list$ computes the possible successor instructions: $\mathsf{succs}\ p = [q_1, \ldots, q_k]$ means that execution of instruction p may transfer control to any of the instructions $q_1, \ldots, q_k$. We use lists instead of sets for reasons of executability.

We say that succs is **bounded by** n if for all $p < n$ the elements of $\mathsf{succs}\ p$ are less than n, i.e. control never leaves the list of instructions below n.

3.3 A Specification of Data Flow Analysis and Bytecode Verification

Data flow analysis is concerned with solving data flow equations, i.e. systems of equations involving the flow functions over a semilattice. In our case step is the flow function and σ the semilattice. Instead of an explicit formalization of the data flow equation it suffices to consider certain prefixed points. To that end we define what it means that a program type ss is **stable at** p and **stable**:

$$\begin{aligned} \mathsf{stable}\ ss\ p &= \forall q \in \mathsf{set}(\mathsf{succs}\ p).\ \mathsf{step}\ p\ (ss!p) \leq_{\mathsf{r}} ss!q \\ \mathsf{stables}\ ss &= \forall p < \mathsf{size}(ss).\ \mathsf{stable}\ ss\ p \end{aligned}$$

We call a function $dfa :: \sigma\ list \to \sigma\ list$ a **data flow analyzer** (w.r.t. $n :: nat$ and $A :: \sigma\ set$) iff for all $ss \in \mathsf{list}\ n\ A$

1. dfa preserves A: $dfa\ ss \in \mathsf{list}\ n\ A$,
2. dfa produces stable program types: $\mathsf{stables}\ (dfa\ ss)$
3. dfa is increasing: $ss \leq_{[\mathsf{r}]} dfa\ ss$
4. dfa is bounded by stable program types:

$$\forall ts \in \mathsf{list}\ n\ A.\ ss \leq_{[\mathsf{r}]} ts \wedge \mathsf{stables}\ ts \longrightarrow dfa\ ss \leq_{[\mathsf{r}]} ts$$

In case you are not initimately familiar with data flow analysis: the correctness of this specification is not an issue because it is merely a stepping stone that does not occur in our main result.

We need to introduce the subset A of σ to make distinctions beyond HOL's type system: for example, when representing a list of registers, σ is likely to be a HOL list type; but the fact that in any particular program the number of registers is fixed cannot be expressed as a HOL type, because it requires dependent types to formalize lists of a fixed length. We use sets to express such fine grained distinctions.

A bytecode verifier is defined as a function between program types, where the absence of $\top$ in the result indicates welltypedness. Formally, a function $bcv :: \sigma\ list \to \sigma\ list$ is a **bytecode verifier** (w.r.t. $n :: nat$ and $A :: \sigma\ set$) iff

$$\forall ss \in \mathsf{list}\ n\ A.\ \top \notin \mathsf{set}(bcv\ ss)\ =\ (\exists ts.\ ss \leq_{[\mathsf{r}]} ts \wedge \mathsf{welltyping}\ ts)$$

3.4 Relating Data Flow Analysis and Bytecode Verification

Since the data flow analyzer is specified in terms of stability and the bytecode verifier in terms of welltyping, the two notions need to be related. Naively one may assume that the stable program types are exactly the welltypings. This is almost true, but for lists containing $\top$: they may be stable but will not be welltypings. We say that **wti and stable agree for $\top$-free program types** (w.r.t. $n :: nat$ and $A :: \sigma\ set$) iff for all $ss \in \mathsf{list}\ n\ A$ and $p < n$

$$\top \notin \mathsf{set}(ss) \longrightarrow (\mathsf{wti}\ ss\ p = \mathsf{stable}\ ss\ p)$$

Note that because stable is defined in terms of step, r and succs, this property relates wti to those three constants.

Theorem 1. *Let r be a partial order with top element $\top$. If wti and stable agree for $\top$-free program types and dfa is a data flow analyzer then dfa is a bytecode verifier (all w.r.t. some n and A).*

This theorem follows easily from the definitions. The hard part is to discharge the premise that *dfa* is a data flow analyzer, i.e. to provide a correct implementation. This is the topic of §4.

Note that instead of having both wti and step as parameters of the framework and requiring that they agree, one could define wti in terms of stable. We have not done so because much of the work on Java's bytecode verifier is based on type systems rather than flow functions. In particular, in §5.5 we build on the verification by Pusch which is phrased in terms of a type system.

4 Kildall's Algorithm

This section defines and verifies a functional version of Kildall's algorithm [8,10], a standard data flow analysis tool. In fact, the description of bytecode verification in the official JVM specification [9, pages 129–130] is essentially Kildall's algorithm. We define the algorithm in the context of semilattice $(\mathsf{A}, \mathsf{f}, \mathsf{r})$ and the functions step and succs. Its core is the iteration on a state type ss and a **worklist** $w :: nat\ set$ that holds the set of all indices of ss whose contents has changed and still needs to be propagated. Once w becomes empty, ss is returned:

$$\begin{array}{l}\mathsf{iter}(ss, w) = \texttt{if}\ \ w = \{\}\ \texttt{then}\ ss\ \texttt{else}\\ \qquad\qquad\quad \texttt{let}\ \ p = \varepsilon p.\ p \in w;\ \ t = \mathsf{step}\ p\ (ss!p)\\ \qquad\qquad\quad \texttt{in}\ \mathsf{iter}(\mathsf{propa}\ (\mathsf{succs}\ p)\ t\ ss\ (w - \{p\}))\end{array}$$

The choice of which position to consider next is made by Hilbert's ε-operator: $\varepsilon x.P(x)$ is some arbitrary but fixed x such that $P(x)$ holds; if there is no such x, then the value of $\varepsilon x.P(x)$ is arbitrary but still defined. Since the choice in iter is guarded by $w \neq \{\}$, we know that $p \in w$. An implementation is free to chose whichever element it wants.

The body of the iteration is hidden in function propa which propagates the new value at $ss!p$ to all its successors, thus reflecting the execution of a single instruction p. This means, the result of $\mathsf{step}\ p\ (ss!p)$ has to be "merged" [9, page 130] with the state $ss!q$ of all successor instructions q of p. "Merging" means the computation of the supremum w.r.t. $(\mathsf{A}, \mathsf{f}, \mathsf{r})$. The worklist is updated in case this merging results in a change.

$$
\begin{array}{lll}
\mathsf{propa}\ []\ & t\ ss\ w = (ss, w) & \\
\mathsf{propa}\ (q\#qs)\ & t\ ss\ w = \mathtt{let}\ \ u = t +_{\mathsf{f}} ss!q & \\
& \qquad\qquad\quad w' = \mathtt{if}\ \ u = ss!q\ \mathtt{then}\ \ w\ \mathtt{else}\ \ \{q\} \cup w & \\
& \qquad\quad \mathtt{in}\ \ \mathsf{propa}\ qs\ t\ (ss[q := u])\ w' &
\end{array}
$$

Kildall's algorithm is simply a call to iter where the worklist is initialized with the set of indices that need propagating:

$$\mathsf{kildall}\ ss = \mathsf{iter}(ss, \{p.\ p < \mathsf{size}\ ss \wedge \exists q \in \mathsf{set}(\mathsf{succs}\ p).\mathsf{step}\ p\ (ss!p) +_{\mathsf{f}} ss!q \neq ss!q\})$$

Function iter is partial in general. Thus its above definition is illegal in HOL, a logic of total functions. The actual definition of iter in HOL requires that the ordering r of our semilattice satisfies the ascending chain condition. For reasons discussed already towards the end of §3.3 we also require that the functions step and succs do not lead outside the semilattice carrier A:

1. succs is bounded by $\mathsf{size}\ ss$
2. **step preserves A**: $\forall s \in \mathsf{A}, p < n.\ \mathsf{step}\ p\ s \in \mathsf{A}$

Finally, ss and w must also be initialized appropriately: $\mathsf{set}\ ss \subseteq \mathsf{A}$ and w **is bounded by size** ss: $\forall p \in w.\ p < \mathsf{size}\ ss$.

4.1 Kildall's Algorithm Is a Bytecode Verifier

We will now sketch the proof that, under suitable assumptions, Kildall's algorithm is indeed a bytecode verifier. For that purpose we introduce some more terminology. We call step **monotone** (w.r.t. A and n) iff for all $s \in \mathsf{A}$, $p < n$, $s \leq_{\mathsf{r}} t$ implies $\mathsf{step}\ p\ s \leq_{\mathsf{r}} \mathsf{step}\ p\ t$. The main theorem states that iter satisfies the properties of a data flow analyzer as defined in §3.3:

Theorem 2. *If r satisfies the ascending chain condition, step preserves A and is monotone w.r.t. A and n, succs and w are bounded by n, $ss \in \mathsf{list}\ n\ \mathsf{A}$, and ss is stable at $p < n$ for all $p \notin w$, then*

$$
\begin{array}{l}
\mathsf{iter}(ss, w) \in \mathsf{list}\ n\ \mathsf{A}\ \wedge\ \mathsf{stables}\ (\mathsf{iter}(ss, w))\ \wedge\ ss \leq_{[\mathsf{r}]} \mathsf{iter}(ss, w)\ \wedge \\
\forall ts \in \mathsf{list}\ n\ \mathsf{A}.\ ss \leq_{[\mathsf{r}]} ts \wedge \mathsf{stables}\ ts \longrightarrow \mathsf{iter}(ss, w) \leq_{[\mathsf{r}]} ts
\end{array}
$$

The following two corollaries follow easily from the definition of a data flow analyzer and $\mathsf{kildall}$ and from Theorem 1:

Corollary 3. *If r satisfies the ascending chain condition, step preserves A and is monotone w.r.t. A and n, and succs is bounded by n, then $\mathsf{kildall}$ is a data flow analyzer.*

Corollary 4. *If* r *satisfies the ascending chain condition,* $\top$ *is its top element,* step *preserves* A *and is monotone w.r.t.* A *and* n, succs *is bounded by* n, *and* wti *and* stable *agree for* $\top$*-free program types, then* kildall *is a bytecode verifier.*

The proof of Theorem 2 is by induction along the recursion of iter. In an imperative language, iter would be a while-loop and the proof would proceed by the following invariant:

$$ss \in \mathsf{list}\ n\ \mathsf{A}\ \wedge\ (\forall p < n.\ p \notin w \longrightarrow \mathsf{stable}\ ss\ p)\ \wedge\ cs \leq_{[\mathsf{r}]} ss\ \wedge$$
$$\forall ts \in \mathsf{list}\ n\ \mathsf{A}.\ cs \leq_{[\mathsf{r}]} ts \wedge \mathsf{stables}\ ts \longrightarrow ss \leq_{[\mathsf{r}]} ts$$

where cs is a logical variable that refers to the initial value of ss. Of course, the proof of Theorem 2 follows exactly the same line. As a first step we show that propa can be split into two independent computations: if w is bounded by $\mathsf{size}\ ss$ then

$$\mathsf{propa}\ qs\ t\ ss\ w = (\mathsf{merges}\ t\ qs\ ss, \{q.\ q \in \mathsf{set}\ qs \wedge t +_{\mathsf{f}} ss!q \neq ss!q\} \cup w)$$

where merges is defined recursively:

$$\begin{array}{ll} \mathsf{merges}\ t\ []\ & ss = ss \\ \mathsf{merges}\ t\ (p\#ps)\ ss & = \mathsf{merges}\ t\ ps\ (ss[p := t +_{\mathsf{f}} ss!p]) \end{array}$$

Thus the verification can be phrased in terms of the simpler merges instead of propa. Preservation of the invariant can be shown with a few suitable lemmas about merges. When $w = \{\}$ it is easy to see and prove that the invariant implies the corresponding conditions in Theorem 2.

5 Application to JVM

In this section we apply the generic framework to a subset of the JVM. We do not consider the details of how information, e.g. the subclass relationship, is encoded in the compiled class files. Instead, we represent this information abstractly, e.g. as a binary relation between class names. To minimize explicit parameterization, such class file derived information is dealt with via implicit parameters, just as explained in the beginning of §3.

5.1 Types

Theory JType describes the types of our JVM. The machine only supports the void type, integers, null references and class types (based on a type *cname* of class names):

$$\textbf{datatype}\ ty = \mathsf{Void} \mid \mathsf{Integer} \mid \mathsf{NullT} \mid \mathsf{Class}\ cname$$

Each class file records the direct superclass. In our formalization this gives rise to an implicit parameter S of HOL type $(cname \times cname)set$, where $(D, C) \in \mathsf{S}$ means that D is a direct subclass of C. Subclasses induce a subtype relation:

$$\begin{aligned}
&\mathsf{subtype}\ \tau_1\ \tau_2 &&= (\tau_1 = \tau_2\ \vee\ \tau_1 = \mathsf{NullT} \wedge \mathsf{is_Class}\ \tau_2\ \vee \\
&&&\quad \exists C\ D.\ \tau_1 = \mathsf{Class}\ C \wedge \tau_2 = \mathsf{Class}\ D \wedge (C, D) \in \mathsf{S}^*) \\
&\mathsf{is_Class}\ (\mathsf{Class}\ C) &&= \mathsf{True} \\
&\mathsf{is_Class}\ _ &&= \mathsf{False} \\
&\mathsf{is_ref}\ \tau &&= (\tau = \mathsf{NullT} \vee \mathsf{is_Class}\ \tau)
\end{aligned}$$

Corresponding to it we have a supremum operation on types:

$$\begin{aligned}
&\mathsf{sup}\ \mathsf{NullT}\ &&(\mathsf{Class} D) &&= \mathsf{OK}(\mathsf{Class}\ D) \\
&\mathsf{sup}\ (\mathsf{Class}\ C)\ &&\mathsf{NullT}\ &&= \mathsf{OK}(\mathsf{Class}\ C) \\
&\mathsf{sup}\ (\mathsf{Class}\ C)\ &&(\mathsf{Class}\ D) &&= \mathsf{OK}(\mathsf{Class}(\mathsf{some_lub}\ C\ D)) \\
&\mathsf{sup}\ \tau_1\ &&\tau_2\ &&= \texttt{if}\ \tau_1 = \tau_2\ \texttt{then}\ \mathsf{OK}\ \tau_1\ \texttt{else}\ \mathsf{Err}
\end{aligned}$$

The auxiliary function some_lub used in the computation of the supremum of two classes is defined non-constructively (as some least upper bound, using Hilbert's ε-operator). Of course we also prove that (under suitable conditions) least upper bounds are uniquely determined and exist. Thus our work is independent of the particular algorithm used for this calculation.

The type *cname* is assumed to have a distinguished element Object. Predicate is_type :: *ty* $\to$ *bool* is true for all basic types and for all class types below Class Object (w.r.t. the given subclass hierarchy S). As abbreviations we introduce types $= \{\tau \mid \mathsf{is\_type}\ \tau\}$ and $\tau_1 \sqsubseteq \tau_2 = \tau_1 \leq_{\mathsf{subtype}} \tau_2$.

Theorem 5. *The triple* esl $= (types, subtype, sup)$ *is an err-semilattice provided* S *is* ***univalent*** *(each subclass has at most one direct superclass, i.e.* S *represents a single inheritance hierarchy) and* S *is acyclic.*

Univalence and acyclicity together imply that S is a set of trees, and is_type focusses on the subtree below Object.

Because any infinite subtype chain would induce an infinite subclass chain we also obtain

Lemma 6. *If* S^{-1} *is wellfounded (there is no infinite ascending subclass chain* $(C_i, C_{i+1}) \in S$*) then* subtype *satisfies the ascending chain condition.*

Note that by incorporating a fixed class hierarchy into our model we assume that all required classes have been loaded, i.e. we model an eager class loader. Although Sun's JVM is a bit lazier, the JVM specification [9, p. 127] does permit eager loading.

5.2 Instructions

For our subset of the JVM we have selected the following representative instruction set

$$\begin{aligned}
\textbf{datatype}\ \mathit{instr} = {} & \mathsf{Load}\ \mathit{nat} \mid \mathsf{Store}\ \mathit{nat} \mid \mathsf{AConst_Null} \mid \mathsf{IConst}\ \mathit{int} \mid \mathsf{IAdd} \\
\mid {} & \mathsf{Getfield}\ \mathit{ty}\ \mathit{cname} \mid \mathsf{Putfield}\ \mathit{ty}\ \mathit{cname} \mid \mathsf{New}\ \mathit{cname} \\
\mid {} & \mathsf{Invoke}\ \mathit{cname}\ \mathit{mname}\ \mathit{ty}\ \mathit{ty} \mid \mathsf{CmpEq}\ \mathit{nat} \mid \mathsf{Return}
\end{aligned}$$

where *mname* is the type of method names. The main difference to Sun's JVM is that Load, Store, CmpEq, and Return are polymorphic (the real JVM has one such instruction for each base type), Invoke only supports methods with a single parameter (the first *ty* argument), Getfield and Putfield carry the field type but not its name, and CmpEq uses absolute instead of relative addressing.

The JVM is a stack machine where each activation record consists of a stack for expression evaluation and a list of local variables (which we call **registers**). The abstract semantics, which operates on the type level, records the type of each stack element and each register. At a specific program point, a register may hold either of two incompatible types, e.g. an integer or a reference, depending on the computation path that lead to that point. This facilitates reuse of registers and is modeled by the HOL type *ty err*, where OK τ represents type τ itself and Err represents the unusable/inconsistent type. In contrast, the stack should only hold values that can actually be used. Thus the configurations of our abstract JVM are pairs of an expression stack and a list of registers:

$$config = ty\ list \times ty\ err\ list$$

The execution of a single instruction is modeled by function exec :: *instr* $\rightarrow$ *config* $\rightarrow$ *config err* where the result Err indicates a run time error, either because of stack over or underflow or because of type mismatch. In Sun's JVM the maximal stack size is recorded in the class file as the code attribute `max_stack` for each method. To simplify the presentation, we concentrate on the verification of a single method and introduce two further implicit parameters maxs :: *nat* and rT :: *ty*, the method's maximal stack size and its return type. The definition of exec below is quite straightforward.

Note that access to a register beyond the size of *reg* is not checked during execution because it is prevented easily by a single pass over the instruction sequence (see wfi below). A dynamic check would merely add clutter.

We now start to instantiate the parameters of our abstract framework in §3: $\sigma = config\ err\ option$ where None indicates a program point that has not been reached yet, Some(OK c) is a normal configuration c and Some Err an error. The introduction of the extra *option* layer means that our data flow analyzer will also determine reachability of instructions and enforce type correctness for reachable instructions only. Function step is merely a lifted version of exec

$$\begin{array}{l}\mathsf{step}\ p = \mathsf{option\_map}\ (\mathsf{lift}\ \mathsf{exec}\ (\mathsf{bs}!p))\\ \mathsf{lift}\ f\ e = \mathtt{case}\ \ e\ \ \mathtt{of}\ \mathsf{Err} \Rightarrow \mathsf{Err} \mid \mathsf{OK}\ x \Rightarrow f\ x\\ \mathsf{option\_map}\ f\ \mathsf{None} = \mathsf{None}\\ \mathsf{option\_map}\ f\ (\mathsf{Some}\ x) = \mathsf{Some}(f\ x)\end{array}$$

where bs is yet another implicit parameter, the list of instructions of the method under examination. As dictated by the general framework in §3.2, we also need a successor function. Its definition is straightforward:

$$\mathsf{succs}\ p = \mathtt{case}\ \ \mathsf{bs}!p\ \ \mathtt{of}\ \mathsf{Return} \Rightarrow [p] \mid \mathsf{CmpEq}\ q \Rightarrow [p{+}1, q] \mid \_ \Rightarrow [p{+}1]$$

```
exec instr (st, reg) = case instr of
  Load n ⇒ if size st < maxs then case reg!n of Err ⇒ Err | OK τ ⇒ OK(τ#st, reg)
           else Err
| Store n ⇒ (case st of [] ⇒ Err | τ#st₁ ⇒ OK(st₁, reg[n := OK τ]))
| AConst_Null ⇒ if size st < maxs then OK(NullT#st, reg) else Err
| IConst i ⇒ if size st < maxs then OK(Integer#st, reg) else Err
| IAdd ⇒ (case st of Integer#Integer#st₂ ⇒ OK(Integer#st₂, reg) | _ ⇒ Err)
| Getfield τ_f C ⇒ (case st of [] ⇒ Err
                    | τ_o#st₁ ⇒ if τ_o ⊑ Class C then OK(τ_f#st₁, reg) else Err)
| Putfield τ_f C ⇒ (case st of τ_v#τ_o#st₂ ⇒ if τ_v ⊑ τ_f ∧ τ_o ⊑ Class C
                                              then OK(st₂, reg) else Err
                   | _ ⇒ Err)
| New C ⇒ if size st < maxs then OK((Class C)#st, reg) else Err
| Invoke C m τ_p τ_r ⇒ (case st of τ_a#τ_o#st₂ ⇒ if τ_o ⊑ Class C ∧ τ_a ⊑ τ_p
                                                 then OK(τ_r#st₂, reg) else Err
                       | _ ⇒ Err)
| CmpEq q ⇒ (case st of τ₁#τ₂#st₂ ⇒ if τ₁ = τ₂ ∨ is_ref τ₁ ∧ is_ref τ₂
                                     then OK(st₂, reg) else Err
              | _ ⇒ Err)
| Return ⇒ (case st of [] ⇒ Err | τ#_ ⇒ if τ ⊑ rT then OK(st, reg) else Err)
```

Modelling the Return instruction by a self loop may seem odd, but it needs a successor instruction because otherwise data flow analysis would never execute it. And a loop is simpler than introducing a fictitious successor.

The main omissions in our model of the JVM are exceptions, object initialization, and `jsr/ret` instructions because our work builds on the type safety proof by Pusch (see §5.5) which does not cover these features either. Including object initialization is straightforward, but exceptions require a modification of our abstract framework: flow functions (step) need to be associated with edges rather than nodes of the program graph because exceptional behaviour can differ from normal behaviour. The difficulty of including `jsr/ret` is unclear.

5.3 Type System

We start with those context conditions that can be checked separately for each instruction. This check uses a **method dictionary** of HOL type

$$mdict = cname \to (mname \times ty \to ty\ option)$$

A method dictionary is a functional abstraction of the information about the types of all methods defined in the current collection of class files. It maps a class name C, method name m, and a type τ to the result type of the method that is selected when m is invoked with an object of class C and a parameter of type τ (remember that Invoke only supports single argument methods). In case there is no applicable method, None is returned. The argument type is necessary

because Java allows overloading of method names, which are disambiguated via their argument types. The formal details need not concern us here and can be found elsewhere [11].

The wellformedness of an instruction is checked in the context of two further implicit parameters: a method dictionary md :: *mdict* and a limit maxr :: *nat* for the maximal register index in the current method (the `max_locals` code attribute in Sun's JVM).

$$\begin{array}{ll}
\mathsf{wfi}(\mathsf{Load}\ n) & = n < \mathsf{maxr} \\
\mathsf{wfi}(\mathsf{Store}\ n) & = n < \mathsf{maxr} \\
\mathsf{wfi}(\mathsf{Getfield}\ \tau\ C) & = \mathsf{is_type}\ \tau \wedge \mathsf{is_type}\ (\mathsf{Class}\ C) \\
\mathsf{wfi}(\mathsf{Putfield}\ \tau\ C) & = \mathsf{is_type}\ \tau \wedge \mathsf{is_type}\ (\mathsf{Class}\ C) \\
\mathsf{wfi}(\mathsf{New}\ C) & = \mathsf{is_type}\ (\mathsf{Class}\ C) \\
\mathsf{wfi}(\mathsf{Invoke}\ C\ m\ \tau_p\ \tau_r) & = \mathsf{md}\ C\ (m, \tau_p) = \mathsf{Some}(\tau_r) \\
\mathsf{wfi}(_) & = \mathsf{True}
\end{array}$$

The instruction sequence bs is **wellformed** iff all of its elements are: $\forall instr \in \mathsf{set\ bs}.\ \mathsf{wfi}\ instr$.

The method dictionary also comes with a wellformedness condition reflecting the type soundness requirement that a method redefined in a subclass must have a more specialized result type: md is **wellformed** iff

$$\begin{array}{l}
\mathsf{md}\ C\ mT = \mathsf{Some}\ \tau \longrightarrow \\
\mathsf{is_type}\ \tau \wedge (\forall D.\ (D, C) \in \mathsf{S}^* \longrightarrow (\exists \tau'.\ \mathsf{md}\ D\ mT = \mathsf{Some}\ \tau' \wedge \tau' \sqsubseteq \tau))
\end{array}$$

Note that the official JVM specification does not mention this wellformedness property but imposes a stronger one implicitly: one method can override another only if their result types are identical, a restriction we have relaxed.

Now we come to the core of the type system, namely the semilattice structure on σ and the predicate wti. Turning σ into a semilattice is easy, because all of its constituent types are (*err*-)semilattices. The expression stacks form an *err*-semilattice because the supremum of stacks of different size is Err; the list of registers forms a semilattice because the number of registers is fixed:

$$\begin{array}{ll}
\mathsf{stk_esl} :: \mathit{ty\ list\ esl} & \mathsf{reg_sl} :: \mathit{ty\ err\ list\ sl} \\
\mathsf{stk_esl} = \mathsf{upto_esl}\ \mathsf{maxs}\ \mathsf{JType.esl} & \mathsf{reg_sl} = \mathsf{Listn.sl}\ \mathsf{maxr}\ (\mathsf{Err.sl}\ \mathsf{JType.esl})
\end{array}$$

See Theorem 5 for JType.esl. Since any error on the stack must be propagated, the stack and registers are combined in a coalesced product via Product.esl and then embedded into *err* and *option* to form the final semilattice sl :: σ *sl*:

$$\mathsf{sl} = \mathsf{Opt.sl}(\mathsf{Err.sl}(\mathsf{Product.esl}\ \mathsf{stk\_esl}\ (\mathsf{Err.esl}\ \mathsf{reg\_sl}))) \tag{1}$$

As a shorthand, the three components of sl are named states (the carrier set), le (the ordering), and sup (the supremum). Furthermore we introduce an infix abbreviation for the ordering le: $s \ll t\ =\ s \leq_{\mathsf{le}} t$

Combining the theorems about the various (*err*-)semilattice constructions involved in the definition of sl (starting from Theorem 5), it is easy to prove

Corollary 7. *If* S *is univalent and acyclic then* sl *is a semilattice.*

It is trivial to show that $\top = \mathsf{Some}\ \mathsf{Err}$ is the top element of this semilattice.

Below you find the definition of wti, the only remaining parameter of our abstract framework. If you expect type systems to come as a set of inference rules, note that each case in the definition of wti could be turned into an equivalent inference rule for a judgment $ss, p \vdash instr$.

$$
\begin{array}{l}
\mathsf{wti}\ ss\ p = \mathtt{case}\ ss!p\ \mathtt{of}\ \mathsf{None} \Rightarrow \mathsf{True} \mid \mathsf{Some}\ e \Rightarrow \mathtt{case}\ e\ \mathtt{of}\ \mathsf{Err} \Rightarrow \mathsf{False} \\
\mid \mathsf{OK}(st, reg) \Rightarrow \mathtt{case}\ \mathsf{bs}!p\ \mathtt{of} \\
\quad \mathsf{Load}\ n \Rightarrow \mathsf{size}\ st < \mathsf{maxs} \wedge \exists \tau.\ reg!n = \mathsf{OK}\ \tau \wedge \mathsf{Some}(\mathsf{OK}(\tau \# st, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{Store}\ n \Rightarrow \exists \tau\ st_1.\ st = \tau \# st_1 \wedge \mathsf{Some}(\mathsf{OK}(st_1, reg[n := \mathsf{OK}\ \tau])) \ll ss!(p{+}1) \\
\mid \mathsf{AConst\_Null} \Rightarrow \mathsf{size}\ st < \mathsf{maxs} \wedge \mathsf{Some}(\mathsf{OK}(\mathsf{NullT} \# st, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{IConst}\ i \Rightarrow \mathsf{size}\ st < \mathsf{maxs} \wedge \mathsf{Some}(\mathsf{OK}(\mathsf{Integer} \# st, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{IAdd} \Rightarrow \exists st_2.\ st = \mathsf{Integer} \# \mathsf{Integer} \# st_2 \wedge \mathsf{Some}(\mathsf{OK}(\mathsf{Integer} \# st_2, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{Getfield}\ \tau_f\ C \Rightarrow \exists \tau\ st_1.\ st = \tau \# st_1 \wedge \tau \sqsubseteq \mathsf{Class}\ C \wedge \\
\qquad \mathsf{Some}(\mathsf{OK}(\tau_f \# st_1, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{Putfield}\ \tau_f\ C \Rightarrow \exists \tau_v\ \tau_o\ st_2.\ st = \tau_v \# \tau_o \# st_2 \wedge \tau_v \sqsubseteq \tau_f \wedge \tau_o \sqsubseteq \mathsf{Class}\ C \wedge \\
\qquad \mathsf{Some}(\mathsf{OK}(st_2, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{New} C \Rightarrow \mathsf{size}\ st < \mathsf{maxs} \wedge \mathsf{Some}(\mathsf{OK}((\mathsf{Class}\ C) \# st, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{Invoke}\ C\ m\ \tau_p\ \tau_r \Rightarrow \exists \tau_a\ \tau_o\ st_2.\ st = \tau_a \# \tau_o \# st_2 \wedge \tau_o \sqsubseteq \mathsf{Class}\ C \wedge \tau_a \sqsubseteq \tau_p \wedge \\
\qquad \mathsf{Some}(\mathsf{OK}(\tau_r \# st_2, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{CmpEq}\ q \Rightarrow \exists \tau_1\ \tau_2\ st_2.\ st = \tau_1 \# \tau_2 \# st_2 \wedge (\tau_1 = \tau_2 \vee \mathsf{is\_ref}\ \tau_1 \wedge \mathsf{is\_ref}\ \tau_2) \wedge \\
\qquad \mathsf{Some}(\mathsf{OK}(st_2, reg)) \ll ss!(p{+}1) \wedge \mathsf{Some}(\mathsf{OK}(st_2, reg)) \ll ss!q \\
\mid \mathsf{Return} \Rightarrow \exists \tau\ st_1.\ st = \tau \# st_1 \wedge \tau \sqsubseteq \mathsf{rT}
\end{array}
$$

5.4 A Verified Bytecode Verifier

We can now instantiate Kildall's algorithm with its remaining (implicit) parameter, namely the semilattice. Clearly $(\mathsf{A}, \mathsf{f}, \mathsf{r})$ must be sl as defined in (1) and hence $\mathsf{A} = \mathsf{states}$, $\mathsf{f} = \mathsf{sup}$ and $\mathsf{r} = \mathsf{le}$. We call the resulting specialized algorithm $\mathsf{kiljvm} :: \sigma\ list \to \sigma\ list$. Before we can use Corollary 4 to prove that kiljvm is a bytecode verifier, we need three properties of step:

Lemma 8. *If* bs *and* md *are wellformed,* step *preserves* states *and is monotone. If* succs *is bounded by* size bs *then* wti *and* stable *agree for* $\top$*-free program types.*

Each property is proved by a case distinction over the different instructions.

Corollary 9. *If* S *is univalent and* S^{-1} *wellfounded (see Lemma 6),* bs *and* md *are wellformed, and* succs *is bounded by* size bs*, then* kiljvm *is a bytecode verifier.*

It follows directly from Corollary 4: some of its preconditions are covered by Lemma 8; the ascending chain property of le follows from Lemma 6 because the semilattice constructions involved preserve the property. This corollary clearly shows that the verification of class files can be split up into separate tasks: a first phase where static wellformedness conditions of class files are checked (the class hierarchy, the method result types, and the individual instructions), and the actual data flow analysis. The same separation is also present in the official JVM specification.

5.5 The Global Picture

So far we have only considered an abstraction of the JVM that works on types rather than values. It remains to show that this abstraction is faithful, i.e. that welltypedness on this abstract level actually guarantees the absence of type errors in the concrete JVM operating on values. To this end we connect our work with that of Pusch [13] who gave a specification of a bytecode verifier and showed that it implies type soundness of the concrete JVM. Roughly speaking, she proved that during execution of welltyped code, all values conform to the types given in the welltyping. That is, she defined a type *jvmstate* of concrete machine states (which include the pc), a function $\mathsf{exec}_{val} :: \mathit{jvmstate} \to \mathit{jvmstate\ option}$ for executing a single instruction, and a conformance relation $\mathsf{corrstate}\ s\ ss$ between a concrete machine state s and a program type ss (all in the context of a set of class files). And she proved

$$\mathsf{welltyping}\ ss \wedge \mathsf{corrstate}\ s\ ss \wedge \mathsf{exec}_{val}\ s = \mathsf{Some}\ t \longrightarrow \mathsf{corrstate}\ t\ ss$$

i.e. a welltyping guarantees type safe execution. By our definition of "bytecode verifier" in terms of welltyping (§3.3) and the fact that kiljvm is a bytecode verifier (Corollary 9) it follows that if $\top \notin \mathsf{kiljvm}\ ss_0$ then execution is type safe from any concrete state s conforming to ss_0 (because corrstate is monotone: $\mathsf{corrstate}\ s\ ss_0$ and $ss_0 \leq_{[\mathsf{le}]} ss$ imply $\mathsf{corrstate}\ s\ ss$).

The program type ss_0 that iteration starts from is initialized as follows. When checking a method in class C with parameter types $\tau_1, \ldots, \tau_n$ we set

$$ss_0 = [\mathsf{Some}(\mathsf{OK}(\mathit{init})), \mathsf{None}, \ldots, \mathsf{None}]$$
$$\mathit{init} = ([], [\mathsf{Some}(\mathsf{Class}\ C), p_1, \ldots, p_n, \mathsf{None}, \ldots, \mathsf{None}])$$

such that $\mathsf{size}\ ss_0 = \mathsf{size}\ \mathsf{bs}$; *init* models the state upon entry into the method: the stack is empty, the first register contains the `this`-pointer, the next n registers contain the parameters $p_i = \mathsf{Some}(\tau_i)$, and the remaining registers, which hold the actual local variables, are uninitialized, i.e. do not contain a usable value.

6 Conclusion

By verifying an executable BCV, this work closes a significant gap in the effort to provide a machine-checked formalization of the Java/JVM architecture. Despite its relative compactness (500 lines of specifications and programs, and 2000 lines of proofs), the amount of work to construct such a detailed model should not be underestimated. But when it comes to security, there is no substitute for complete formality. And since both the theory and the tools are available, we intend to verify implementations of more realistic BCVs (incl. object initialization and exceptions) in the not too distant future.

Acknowledgments. I thank Zhenyu Qian, Gilad Bracha, Gerwin Klein and David von Oheimb for many helpful discussions and for reading drafts of this paper.

References

1. Y. Bertot. A Coq formalization of a type checker for object initialization in the Java Virtula Machine. Technical Report RR-4047, INRIA, Nov. 2000.
2. L. Casset and J. L. Lanet. A formal specification of the Java bytecode semantics using the B method. In *ECOOP'99 Workshop Formal Techniques for Java Programs*, 1999.
3. A. Coglio, A. Goldberg, and Z. Qian. Toward a provably-correct implementation of the JVM bytecode verifier. In *Proc. DARPA Information Survivability Conference and Exposition (DISCEX'00), Vol. 2*, pages 403–410. IEEE Computer Society Press, 2000.
4. S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, 1998.
5. S. N. Freund and J. C. Mitchell. A formal framework for the java bytecode language and verifier. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, 1999.
6. A. Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conf. Computer and Communications Security*, 1998.
7. M. Hagiya and A. Tozawa. On a new method for dataflow analysis of Java virtual machine subroutines. In G. Levi, editor, *Static Analysis (SAS'98)*, volume 1503 of *Lect. Notes in Comp. Sci.*, pages 17–32. Springer-Verlag, 1998.
8. G. A. Kildall. A unified approach to global program optimization. In *Proc. ACM Symp. Principles of Programming Languages*, pages 194–206, 1973.
9. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.
10. S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.
11. T. Nipkow and D. v. Oheimb. Java$_{light}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170, 1998.
12. T. Nipkow, D. v. Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, pages 117–144. IOS Press, 2000.
13. C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lect. Notes in Comp. Sci.*, pages 89–103. Springer-Verlag, 1999.
14. Z. Qian. A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 271–311. Springer-Verlag, 1999.
15. Z. Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Trans. Programming Languages and Systems*, ?:?–?, 200? Accepted for publication.
16. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 149–161. ACM Press, 1998.

Higher-Order Abstract Syntax with Induction in Isabelle/HOL: Formalizing the π-Calculus and Mechanizing the Theory of Contexts

Christine Röckl[1], Daniel Hirschkoff[2], and Stefan Berghofer[1]

[1] Fakultät für Informatik, Technische Universität München, D-80290 München
Email: {roeckl,berghofe}@in.tum.de

[2] LIP – École Normale Supérieure de Lyon, 46, allée d'Italie, F-69364 Lyon Cedex 7
Email: Daniel.Hirschkoff@ens-lyon.fr

Abstract. Higher-order abstract syntax is a natural way to formalize programming languages with binders, like the π-calculus, because α-conversion, instantiations and capture avoidance are delegated to the meta-level of the provers, making tedious substitutions superfluous. However, such formalizations usually lack structural induction, which makes syntax-analysis impossible. Moreover, when applied in logical frameworks with object-logics, like Isabelle/HOL or standard extensions of Coq, *exotic* terms can be defined, for which important syntactic properties become invalid.
The paper presents a formalization of the π-calculus in Isabelle/HOL, using *well-formedness* predicates which both eliminate exotic terms and yield structural induction. These induction-principles are then used to derive the *Theory of Contexts* fully within the mechanization.

1 Motivation

The π-calculus was introduced to model and analyse mobile systems [18,17]. In it, communication channels and messages belong to the same sort, called *names*. This simplicity gives the π-calculus the power to encode the λ-calculus [16], as well as higher-order object-oriented and imperative languages [27,26]. Communications are synchronous, that is, a sender $\bar{\mathbf{a}}\mathbf{b}.P$ transmits a message $\mathbf{b}$ to a recipient $\mathbf{a}\mathbf{x}.Q$, in a transition $\bar{\mathbf{a}}\mathbf{b}.P \,|\, \mathbf{a}\mathbf{x}.Q \xrightarrow{\tau} P \,|\, Q\{\mathbf{b}/\mathbf{x}\}$. Usually, a substitution is applied to describe that $\mathbf{b}$ replaces $\mathbf{x}$ in Q. This can be tedious for processes with binders, like $Q = (\nu\mathbf{b})Q'$, where a further substitution is necessary to avoid name-capture: $Q\{\mathbf{b}/\mathbf{x}\} =_\alpha (\nu\mathbf{b}')Q'\{\mathbf{b}'/\mathbf{b}, \mathbf{b}/\mathbf{x}\}$. Higher-order abstract syntax builds upon a functional view, considering binders as abstractions with respect to an underlying λ-calculus to which the replacement of names and capture-avoidance are delegated. The above transition could thus be rewritten as $\bar{\mathbf{a}}\mathbf{b}.P \,|\, \mathbf{a}x.f_Q(x) \xrightarrow{\tau} P \,|\, f_Q(\mathbf{b})$, where the function $f_Q(x) = (\nu b)f_{Q'}(b, x)$ corresponds to the process Q, and the process $f_Q(\mathbf{b})$ represents $Q\{\mathbf{b}/\mathbf{x}\}$.

Proofs in the π-calculus, and in particular bisimulation proofs, tend to be very large and tedious, hence machine-assistance is necessary to prevent errors.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 364–378, 2001.

The work at hand is part of a larger project to provide a platform for machine-assisted reasoning in and about the π-calculus. We have chosen Isabelle/HOL [22,20], as it is generic and offers a large range of powerful proof-techniques.

Formalizing the π-calculus. General-purpose theorem provers distinguish two levels of reasoning. Upon a *meta-logic* provided by the implementors, users can create *object-logics*, in which they define new data-structures and derive proofs. Programming-languages or calculi can be formalized, either fully within the object-level using a first-order syntax (*deep embedding*), or by exploiting the functional mechanisms of the meta-level (*shallow embedding*).

• Following the classical way, the syntax of the π-calculus is described in terms of a recursive datatype $P ::= 0 \mid \bar{\mathbf{a}}\mathbf{b}.P \mid \mathbf{ab}.P \mid \ldots$, and substitution functions are introduced explicitely by the user. These deep embeddings of first-order syntax allow the user to make full use of structural induction, which is a vital proof-tool in syntax analysis. Several formalizations of this kind have been studied in various theorem provers [14,1,9,12,8]. They give evidence that proofs about π-calculus processes in deep embeddings are generally hard, and that it would be tedious to try to tackle larger proofs. The reason is that the π-calculus is particularly characterized by its binders, *input* and *restriction*; hence, a lot of effort goes into intricate reasoning about substitutions.

• In contrast, higher-order abstract syntax used in a shallow embedding, builds on a recursive datatype of the form $P ::= 0 \mid \bar{\mathbf{a}}\mathbf{b}.P \mid \mathbf{a}x.f_P(x) \mid \ldots$, where f_P is a function mapping names to π-calculus processes. Here, the capture-avoiding replacement of names are dealt with automatically by the meta-level of the theorem-prover, freeing the user from a tedious implementation and application of substitutions. Shallow embeddings of the π-calculus have been studied in Coq and λProlog [15,11,3]. Unfortunately, higher-order datatypes are not recursive in a strict sense, due to the functions in the continuations of binders. As a consequence, plain structural induction does not work, making syntax-analysis impossible. Even worse, in logical frameworks with object-level constructors, so-called *exotic* terms can be derived. Consider, for example,

$$\begin{array}{lcl} f_E & \stackrel{def}{=} & \lambda(x : \mathit{names}).\ \ \text{if}\ \ x = \mathbf{a}\ \ \text{then}\ \ 0\ \ \text{else}\ \ \mathbf{a}y.0, \\ f_W & \stackrel{def}{=} & \lambda(x : \mathit{names}).\ \ \mathbf{a}y.0. \end{array}$$

The term f_E is *exotic*, because it is built from an object-level conditional (not from a π-calculus conditional, which we represent in terms of matching and mismatching), and does not correspond to any process in the usual syntax of the π-calculus, whereas f_W can be considered as *valid*, or *well-formed*.

• A third variation uses higher-order abstract syntax, but within a deep embedding, using a first-order formalization of a λ-calculus as a (pseudo) "meta-level" for embedding the π-calculus thereupon. As a consequence, substitutions do not have to be defined for the (large) language, but only for the (smaller) λ-calculus, while still reasoning entirely on the object-level. For formalizations of the π-calculus, this approach has been followed in [7,6]. Up to this point, however, these frameworks have not been tested in larger-scale syntax-analyses.

The theory of contexts. The *theory of contexts* for the π-calculus consists of three syntactic properties that are essential for a semantic analysis of processes in a shallow embedding. It was introduced in the shape of axioms by Honsell, Miculan, and Scagnetto to reason about strong transitions of π-calculus processes in a in Coq formalization, with justification on paper [11,10]. Our own experience gives evidence that the three properties are sufficient for weak transitions as well[1]. One of the properties, which Honsell et al. call *extensionality of contexts*, deserves further mention, as it does not hold in the presence of exotic terms: *Two process abstractions are equal, if they are equal for a fresh name.* Consider f_E and f_W from above, and some $\mathbf{b} \neq \mathbf{a}$. Then $f_E(\mathbf{b}) = \mathbf{a}y.0 = f_W(\mathbf{b})$, because the conditional in f_E evaluates to the negative argument. Yet, still $f_E \neq f_W$, because $f_E(\mathbf{a}) \neq f_W(\mathbf{a})$. See also [10] for a discussion.

Outlook of the paper. In this paper, we present a shallow embedding of the π-calculus in Isabelle/HOL using inductive *well-formedness predicates* which rule out exotic terms and, simultaneously, allow us to perform structural induction on π-calculus processes. Our work was inspired by a similar approach for shallow embeddings of the λ-calculus in Coq, by Despeyroux, Felty, and Hirschowitz [5,4]. As a result, we are able to derive the theory of contexts fully mechanically within Isabelle/HOL. The resulting formalization thus provides a generic framework for the semantic analysis of the π-calculus (for instance, transitions, bisimulations), as well as of concurrent and mobile systems modelled within the π-calculus.

The paper is organized as follows: In Section 2, we give some background of Isabelle/HOL. In Section 3, we introduce the π-calculus, and describe how it is formalized in our framework. In Section 4, we derive the theory of contexts. In Section 5, we discuss some questions related to our results.

2 Isabelle/HOL

We use the general-purpose theorem-prover Isabelle [22], implementing higher-order intuitionistic logic on its meta-level, and formalize the π-calculus in its instantiation HOL for higher-order logic [20]. Proofs in Isabelle are based on unification, and are usually conducted in a backward-resolution style: the user formulates the goal he/she intends to prove, and then—in interaction with Isabelle—continuously reduces it to simpler subgoals until all of the subgoals have been accepted by the tool. Upon this, the goal can be stored in the theorem-database of Isabelle/HOL to be applicable in further proofs. The prover offers various tactics, most of them applying to single subgoals. The basic resolution tactic `resolve_tac`, for instance, allows the user to instantiate a theorem from Isabelle's database so that its conclusion can be applied to transform a current subgoal into instantiations of its premises. Besides these *classical tactics*, Isabelle offers *simplification tactics* based on algebraic transformations. Powerful *automatic tactics* apply the basic tactics to prove given subgoals according to

[1] Technically, properties of weak transitions are usually derived from corresponding properties of strong transitions by induction on the number of silent steps.

different heuristics. These heuristics have in common that a provable goal is always transformed into a set of provable subgoals; rules that might yield unprovable subgoals are only applied if they succeed in terminating the proof of a subgoal. In Isabelle/HOL, the user can define, for instance, recursive datatypes and inductive sets. Isabelle then automatically computes rules for induction and case-injection. It should be noted that all these techniques have been fully formalized and verified on the object-level, that is, they are a conservative generic extension of Isabelle/HOL [2,21]. A recent extension of Isabelle/HOL allows function types in datatype definitions to contain strictly positive occurrences of the type being defined [2]. This allows for formalizations of programming languages in higher-order abstact syntax, like the one we develop in Section 3 of this paper. Isabelle/HOL implements an extensional equality, $=$, which relates functions if they are equal for all arguments. We employ this equivalence as syntactic equivalence of π-calculus processes.

3 Formalizing Processes

The π-calculus is a value-passing calculus, and was introduced to reason about mobile systems [17,18]. In the π-calculus, *names* are used both for the communication channels and the values sent along them, allowing processes to emit previously private names and create new communication links with the recipients. The π-calculus is particularly characterized by its binding operators *input*, $\mathbf{a}y.P$, and *restriction*, $(\nu x)P$. The former implements the functional aspects of the calculus—apply a process abstraction to a received *name*—whereas the latter characterizes its imperative aspects—create a fresh location, that is, a fresh name. In this section, we present a shallow embedding of the π-calculus, and present inductive well-formedness predicates that simultaneously rule out exotic terms and provide structural induction. We use the same datatype as [3,11], so that our results are comparable to these formalizations.

Names. In semantic analysis, processes are often instantiated with *fresh* names; hence, the type of names has to be at least countably infinite. Also the theory of contexts hinges on fresh names. We do not commit ourselves to a specific type but use an axiomatic type-class *inf_class* comprising all types $\mathcal{T}$ for which there exists an injection from $\mathbb{N}$ into $\mathcal{T}$. We neither require nor forbid the existence of a surjection, see also our discussion in Section 5. We use $\mathbf{a}, \mathbf{b}, \ldots$ to range over names, and f_a and ff_a to denote *names-abstractions*, that is, functions mapping one, respectively two, names to names. In order to make names and meta-variables distinguishable, we use bold face letters for the former, as above, and italics, that is, $x, y, \ldots$, for the latter.

Processes. Processes in the π-calculus are built from *inaction* and the basic mechanisms for the exchange and creation of names, *input*, *output*, and *restriction*, by applying constructors for *choice* (or, *summation*), *parallel composition*, *matching*, *mismatching*, and *replication*. In a shallow embedding, we formalize

Table 1. Computing the free names *fn* and depth of binders *db* of a process.

$$
\begin{array}{rcl@{\qquad}rcl}
fn(0) & = & \emptyset & db(0, \mathbf{c}) & = & 0 \\
fn(\tau.P) & = & fn(P) & db(\tau.P, \mathbf{c}) & = & db(P, \mathbf{c}) \\
fn(\bar{\mathbf{a}}\mathbf{b}.P) & = & \{\mathbf{a}, \mathbf{b}\} \cup fn(P) & db(\bar{\mathbf{a}}\mathbf{b}.P, \mathbf{c}) & = & db(P, \mathbf{c}) \\
fn(\mathbf{a}x.f_P(x)) & = & \{\mathbf{a}\} \cup fna(f_P) & db(\mathbf{a}x.f_P(x), \mathbf{c}) & = & 1 + dba(f_P, \mathbf{c}) \\
fn((\nu\, x)f_P(x)) & = & fna(f_P) & db((\nu\, x)f_P(x), \mathbf{c}) & = & 1 + dba(f_P, \mathbf{c}) \\
fn(P + Q) & = & fn(P) \cup fn(Q) & db(P + Q, \mathbf{c}) & = & max(db(P, \mathbf{c}), db(Q, \mathbf{c})) \\
fn(P \parallel Q) & = & fn(P) \cup fn(Q) & db(P \parallel Q, \mathbf{c}) & = & max(db(P, \mathbf{c}), db(Q, \mathbf{c})) \\
fn([\mathbf{a} = \mathbf{b}]P) & = & \{\mathbf{a}, \mathbf{b}\} \cup fn(P) & db([\mathbf{a} = \mathbf{b}]P, \mathbf{c}) & = & db(P, \mathbf{c}) \\
fn([\mathbf{a} \neq \mathbf{b}]P) & = & \{\mathbf{a}, \mathbf{b}\} \cup fn(P) & db([\mathbf{a} \neq \mathbf{b}]P, \mathbf{c}) & = & db(P, \mathbf{c}) \\
fn(!P) & = & fn(P) & db(!P, \mathbf{c}) & = & db(P, \mathbf{c})
\end{array}
$$

$$
fna(f_P) \stackrel{def}{=} \{\, \mathbf{a} \mid \forall \mathbf{b}.\, \mathbf{a} \in fn(f_P(\mathbf{b})) \,\} \qquad dba(f_P, \mathbf{c}) \stackrel{def}{=} db(f_P(\mathbf{c}), \mathbf{c})
$$

$$
fnaa(ff_P) \stackrel{def}{=} \{\, \mathbf{a} \mid \forall \mathbf{b}.\, \mathbf{a} \in fna(\lambda x.\, ff_P(\mathbf{b}, x)) \,\}
$$

input and restriction by means of *process-abstractions* f_P, that is, functions from names to processes. This can be implemented directly in Isabelle/HOL, because in the type of the declaration, processes only occur in a positive position.

$$
\begin{array}{rcll}
P & ::= & 0 & \textit{Inaction} \\
 & | & \tau.P & \textit{Silent Prefix} \\
 & | & \bar{\mathbf{a}}\mathbf{b}.P & \textit{Output Prefix} \\
 & | & \mathbf{a}x.f_P(x) & \textit{Input Prefix} \\
 & | & (\nu\, x)f_P(x) & \textit{Restriction} \\
 & | & P + P & \textit{Choice (Summation)} \\
 & | & P \parallel P & \textit{Parallel Composition} \\
 & | & [\mathbf{a} = \mathbf{b}]P & \textit{Matching} \\
 & | & [\mathbf{a} \neq \mathbf{b}]P & \textit{Mismatching} \\
 & | & !P & \textit{Replication}
\end{array}
$$

It is obvious that this datatype definition is not recursive in a strict sense, due to the use of process abstractions f_P as continuations of input and restriction. Therefore, induction and case injection are not applicable. Further, it is possible to derive exotic terms in Isabelle/HOL, like f_E from the motivation. We use $P, Q, \ldots$ to range over processes, and f_P and ff_P for process abstractions.

Free and Fresh Names. Names which are not in the scope of a binder are called *free*, whereas names in the scope of a binder are called *bound*. In higher-order abstract syntax, it is neither necessary nor possible to compute the bound names of a process, because they are represented by meta-variables of the theorem-prover. Free names are represented by object-variables, and we compute them with a primitively recursive function *fn*, see Table 1. Note that for exotic process

Table 2. Well-formed Processes.

$$\frac{}{\mathbf{wfp}\,(0)}\ \mathbf{W}_0 \qquad \frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,(\tau.P)}\ \mathbf{W}_1 \qquad \frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,(\bar{\mathbf{a}}\mathbf{b}.P)}\ \mathbf{W}_2 \qquad \frac{\mathbf{wfpa}\,(f_P)}{\mathbf{wfp}\,(\mathbf{a}y.f_P(y))}\ \mathbf{W}_3$$

$$\frac{\mathbf{wfpa}\,(f_P)}{\mathbf{wfp}\,((\nu\, y)f_P(y))}\ \mathbf{W}_4 \qquad \frac{\mathbf{wfp}\,(P) \quad \mathbf{wfp}\,(Q)}{\mathbf{wfp}\,(P+Q)}\ \mathbf{W}_5 \qquad \frac{\mathbf{wfp}\,(P) \quad \mathbf{wfp}\,(Q)}{\mathbf{wfp}\,(P \parallel Q)}\ \mathbf{W}_6$$

$$\frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,([\mathbf{a}=\mathbf{b}]P)}\ \mathbf{W}_7 \qquad \frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,([\mathbf{a}\neq\mathbf{b}]P)}\ \mathbf{W}_8 \qquad \frac{\mathbf{wfp}\,(P)}{\mathbf{wfp}\,(!P)}\ \mathbf{W}_9$$

Table 3. Well-formed Process-Abstractions.

$$\frac{}{\mathbf{wfpa}\,(\lambda x.\,0)}\ \mathbf{W}_0^a \qquad \frac{\mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,\tau.f_P(x))}\ \mathbf{W}_1^a \qquad \frac{\mathbf{wfna}\,(f_a) \quad \mathbf{wfna}\,(f_b) \quad \mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,\overline{f_a(x)}f_b(x).f_P(x))}\ \mathbf{W}_2^a$$

$$\frac{\mathbf{wfna}\,(f_a) \quad \forall\, b.\ \mathbf{wfpa}\,(\lambda x.\,ff_P(b,x)) \quad \forall\, b.\ \mathbf{wfpa}\,(\lambda x.\,ff_P(x,b))}{\mathbf{wfpa}\,(\lambda x.\,f_a(x)y.ff_P(y,x))}\ \mathbf{W}_3^a$$

$$\frac{\forall\, b.\ \mathbf{wfpa}\,(\lambda x.\,ff_P(b,x)) \quad \forall\, b.\ \mathbf{wfpa}\,(\lambda x.\,ff_P(x,b))}{\mathbf{wfpa}\,(\lambda x.\,(\nu\, y)ff_P(y,x))}\ \mathbf{W}_4^a$$

$$\frac{\mathbf{wfpa}\,(f_P) \quad \mathbf{wfpa}\,(f_Q)}{\mathbf{wfpa}\,(\lambda x.\,f_P(x)+f_Q(x))}\ \mathbf{W}_5^a \qquad \frac{\mathbf{wfpa}\,(f_P) \quad \mathbf{wfpa}\,(f_Q)}{\mathbf{wfpa}\,(\lambda x.\,f_P(x) \parallel f_Q(x))}\ \mathbf{W}_6^a$$

$$\frac{\mathbf{wfna}\,(f_a) \quad \mathbf{wfna}\,(f_b) \quad \mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,[f_a(x)=f_b(x)].f_P(x))}\ \mathbf{W}_7^a \qquad \frac{\mathbf{wfna}\,(f_a) \quad \mathbf{wfna}\,(f_b) \quad \mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,[f_a(x)\neq f_b(x)].f_P(x))}\ \mathbf{W}_8^a$$

$$\frac{\mathbf{wfpa}\,(f_P)}{\mathbf{wfpa}\,(\lambda x.\,!f_P(x))}\ \mathbf{W}_9^a$$

terms like f_E from Section 1, *fn* and *fna* need not necessarily compute the free names as one might expect; for f_E, for instance, *fna* computes the empty set. For all *well-formed* processes, however, *fn* and *fna* yield the expected results. A name is *fresh* in a process or process abstraction if it is not among its free names. This can be formalized in terms of $\mathbf{fresh}\,(a, P)$ iff $a \not\in fn\,(P)$, and $\mathbf{fresha}\,(a, f_P)$ iff $a \not\in fna\,(f_P)$ and $\mathbf{freshaa}\,(a, ff_P)$ iff $a \not\in fnaa\,(ff_P)$, respectively.

Well-formedness. We introduce *well-formedness* predicates with which we simultaneously eliminate exotic processes like f_E from Section 1, and obtain structural induction. The predicates are defined inductively, and concern three levels of reasoning: **wfp** defines the set of well-formed processes, see Table 2 for the introduction rules, **wfpa** yields the set of well-formed process-abstractions, see Table 3, and **wfna** and **wfnaa** describe the well-formed names-abstractions, see Table 4. Rules W_3, W_4, W_3^a, and W_4^a, concerning the binders, are of particular interest. For a restricted or input process to be well-formed according to **wfp**, the continuation f_P has to be well-formed according to **wfpa**. With f_P possibly containing inputs and/or restrictions itself, this argument could have to be continued ad infinitum. However, a second-order predicate suffices to rule

Table 4. Well-formed Names-Abstractions.

$$\frac{}{\mathbf{wfna}\,(\lambda x.\, x)}\ \mathbf{W}_1^n \qquad \frac{}{\mathbf{wfna}\,(\lambda x.\, \mathbf{a})}\ \mathbf{W}_2^n$$

$$\frac{}{\mathbf{wfnaa}\,(\lambda(x,y).\, x)}\ \mathbf{W}_3^n \qquad \frac{}{\mathbf{wfnaa}\,(\lambda(x,y).\, y)}\ \mathbf{W}_4^n \qquad \frac{}{\mathbf{wfnaa}\,(\lambda(x,y).\, \mathbf{a})}\ \mathbf{W}_5^n$$

out at least those exotic terms that might render syntactic properties of the original language incorrect in the encoding, see Section 5 for a discussion. The process abstraction f_E from the introduction, for instance, is ruled out as exotic by **wfpa**. We are thus able to derive in Section 4 the validity of the theory of contexts for the set of well-formed processes and abstractions.

Counting Binders. In the proof in Section 4.4, we use coercion from higher-order syntax to first-order syntax by instantiating meta-variables with fresh names. In order to provide a sufficient amount of fresh names, we statically compute the *depth of binders* with a primitively recursive function, *db*; for a formal definition, see Table 1. The function computes the maximal number of binders along each path in the process-tree, instantiating process-abstractions with an auxiliary name **c**. Like *fn* above, *db* only yields sensible results for well-formed processes.

4 Deriving Syntactic Properties in Isabelle/HOL

We now turn to a formal derivation of the *theory of contexts* [11] for well-formed processes. It consists of three general syntactic properties of languages with binders, which can be described intuitively as follows:

(Mon) Monotonicity: *If a name* **a** *is fresh in an instantiated process-abstraction* $f_P(\mathbf{b})$*, it is fresh in* f_P *already.*

(Ext) Extensionality: *Two process-abstractions* f_P *and* f_Q *are equal, if they are equal for a fresh name* **a***.*

(Exp) β-Expansion: *Every process* P *can be abstracted over an arbitrary name* **a***, yielding a suitable process-abstraction.*

A formal description is depicted in Table 5, for well formed processes and process abstractions. Recall from Section 1 that extensionality only holds for well-formed process-abstractions. Also in the third law (Exp), describing β-expansion, we only consider well-formed processes and process-abstractions. The reason is that, although β-expansion holds for arbitrary processes and abstractions over them, we want to strengthen it as much as possible, so that it can be used together with (Ext) in the semantic analysis of processes.

Table 5. Formalizations of *monotonicity*, *extensionality*, and β-expansion.

$$\frac{\mathbf{fresh}\,(\mathbf{a},\ f_P(\mathbf{b}))}{\mathbf{fresha}\,(\mathbf{a},\ f_P)}\ (\textsc{Mon}) \qquad \frac{\mathbf{fresha}\,(\mathbf{a},\ \lambda x.\ ff_P(\mathbf{b},x))}{\mathbf{freshaa}\,(\mathbf{a},\ ff_P)}\ (\textsc{Mona})$$

$$\frac{\mathbf{wfpa}\,(f_P) \quad \mathbf{wfpa}\,(f_Q) \quad \mathbf{fresha}\,(\mathbf{a},f_P) \quad \mathbf{fresha}\,(\mathbf{a},f_Q) \quad f_P(\mathbf{a}) = f_Q(\mathbf{a})}{f_P = f_Q}\ (\textsc{Ext})$$

$$\frac{\mathbf{wfp}\,(P)}{\exists\, f_P.\ \mathbf{wfpa}\,(f_P)\ \wedge\ \mathbf{fresha}\,(\mathbf{a},f_P)\ \wedge\ P = f_P(\mathbf{a})}\ (\textsc{Exp})$$

4.1 Free and Fresh Names

In the proofs of (Ext) and (Exp), we rely on the fact that there exist at least countably infinitely many names—see Section 3—so we can always find a fresh name with which to instantiate a process abstraction. Laws (f1)–(f7) formalize these basic properties; their proofs in Isabelle/HOL are standard, and yield scripts of a few lines only.

$$\text{(f1)}\ \exists\,\mathbf{b}.\ \mathbf{a} \neq \mathbf{b} \qquad \text{(f2)}\ \frac{\mathit{finite}\,(A)}{\exists\,\mathbf{b}.\ \mathbf{b} \notin A}$$

$$\text{(f3)}\ \mathit{finite}\,(\mathit{fn}\,(P)) \qquad \text{(f4)}\ \mathit{finite}\,(\mathit{fna}\,(f_P)) \qquad \text{(f5)}\ \mathit{finite}\,(\mathit{fnaa}\,(ff_P))$$

$$\text{(f6)}\ \frac{\mathbf{wfpa}\,(f_P) \quad \mathbf{fresha}\,(\mathbf{a},f_P) \quad \mathbf{c} \neq \mathbf{a}}{\mathbf{fresh}\,(\mathbf{a}, f_P(\mathbf{c}))}$$

$$\text{(f7)}\ \frac{\forall \mathbf{b}.\ \mathbf{wfpa}\,(\lambda x.\ ff_P\,(\mathbf{b},x)) \quad \forall \mathbf{b}.\ \mathbf{wfpa}\,(\lambda x.\ ff_P\,(x,\mathbf{b})) \quad \mathbf{freshaa}\,(\mathbf{a},ff_P) \quad \mathbf{c} \neq \mathbf{a}}{\mathbf{fresha}\,(\mathbf{a}, \lambda x.\ ff_P(\mathbf{c},x))}$$

Laws (f6) and (f7) express that a name **a** which is fresh for a well-formed process-abstraction, is necessarily fresh for every instantiation except **a**. (f6) is proved by induction over **wfpa**, and all cases are proved automatically by Isabelle; (f7) can then be derived as a corollary, by a single call to an automatic tactic.

4.2 Monotonicity

The monotonicity law, see (Mon) in Table 5, is implicitely encoded in our formalization. That is, a name **a** is only free in a process-abstraction f_P according to *fnaa*, if it is free in every instantiation; hence for **a** to be fresh in f_P, it suffices to present a single name **b** as a witness for which **a** is fresh in $f_P(\mathbf{b})$. The proof in Isabelle requires one call to a standard automatic tactic. Monotonicity can be derived similarly for **freshaa**, see (Mona) in Table 5.

4.3 Extensionality

Two process-abstractions should be equal if they are equal for a single fresh name. This variation of extensionality, where usually a universal quantification is used, is natural in the absence of exotic terms, yet does not hold in their presence, as the counter-example in Section 1 shows.

We prove (Ext) by induction over one of the two involved well-formed processes, f_P, using case-injection for the other, f_Q. Eight out of the ten cases resulting from the induction are purely technical. The two intricate cases are those concerning input and restriction, because they involve process-abstractions taking two names as arguments. For them, induction yields the following subgoal:

$$\begin{array}{l}\forall \mathbf{b}, f_Q, \mathbf{a}.\ \ \mathbf{wfpa}\,(f_Q) \ \wedge\ \mathbf{fresha}\,(\mathbf{a}, \lambda x.\, ff_P\,(\mathbf{b}, x)) \ \wedge\ \mathbf{fresha}\,(\mathbf{a}, f_Q) \ \wedge \\ \qquad ff_P\,(\mathbf{b}, \mathbf{a}) = f_Q\,(\mathbf{a}) \ \longrightarrow\ \lambda x.\, ff_P(b, x) = \lambda x.\, f_Q(x) \\ \forall \mathbf{b}, f_Q, \mathbf{a}.\ \ \mathbf{wfpa}\,(f_Q) \ \wedge\ \mathbf{fresha}\,(\mathbf{a}, \lambda x.\, ff_P\,(x, \mathbf{b})) \ \wedge\ \mathbf{fresha}\,(\mathbf{a}, f_Q) \ \wedge \\ \qquad ff_P\,(\mathbf{a}, \mathbf{b}) = f_Q\,(\mathbf{a}) \ \longrightarrow\ \lambda x.\, ff_P(x, \mathbf{b}) = \lambda x.\, f_Q(x)\end{array}$$

$$\begin{array}{ll}\forall \mathbf{b}.\ \mathbf{wfpa}\,(\lambda x.\, ff_P\,(\mathbf{b}, x)) & \forall \mathbf{b}.\ \mathbf{wfpa}\,(\lambda x.\, ff_P\,(x, \mathbf{b})) \\ \forall \mathbf{b}.\ \mathbf{wfpa}\,(\lambda x.\, ff_Q\,(\mathbf{b}, x)) & \forall \mathbf{b}.\ \mathbf{wfpa}\,(\lambda x.\, ff_Q\,(x, \mathbf{b}))\end{array}$$

$$\frac{\mathbf{freshaa}\,(\mathbf{a}, ff_P) \qquad \mathbf{freshaa}\,(\mathbf{a}, ff_Q) \qquad \lambda x.\, ff_P\,(x, \mathbf{a}) \ =\ \lambda x.\, ff_Q\,(x, \mathbf{a})}{\lambda x.\, ff_P\,(x, \mathbf{c}) \ =\ ff_Q\,(x, \mathbf{c})}$$

The first two premises are the induction-hypotheses corresponding to instantiations of the first (respectively second) parameter of ff_P. We use both of them by subsequently instantiating the first arguments of ff_P and ff_Q and then the second. Laws (f5) and (f2) from Section 4.1 allow us to choose a name $\mathbf{d}$ which does not occur in $\{\mathbf{a}, \mathbf{c}\} \cup \mathit{fnaa}\,(ff_P) \cup \mathit{fnaa}\,(ff_Q)$. Instantiating the first components of ff_P and ff_Q in the first induction hypothesis, we obtain,

$$\begin{array}{l}\mathbf{wfpa}\,(\lambda x.\, ff_Q(\mathbf{d}, x)) \ \wedge\ \mathbf{fresha}\,(\mathbf{a}, \lambda x.\, ff_P\,(\mathbf{d}, x)) \ \wedge\ \mathbf{fresha}\,(\mathbf{a}, \lambda x.\, ff_Q(\mathbf{d}, x)) \ \wedge \\ ff_P\,(\mathbf{d}, \mathbf{a}) = ff_Q\,(\mathbf{d}, \mathbf{a}) \ \longrightarrow\ \lambda x.\, ff_P\,(\mathbf{d}, x) = \lambda x.\, ff_Q(\mathbf{d}, x).\end{array}$$

As all the hypotheses for the implication can be established directly from the premises, or from (f7) and the fact that $\mathbf{d} \neq \mathbf{a}$, this implication can be resolved into a new premise of the form $\lambda x.\, ff_P\,(\mathbf{d}, x) \ =\ \lambda x.\, ff_Q(\mathbf{d}, x)$. Similarly, by instantiating the second arguments of ff_P and ff_Q with $\mathbf{c}$ in the second induction hypothesis, we obtain,

$$\begin{array}{l}\mathbf{wfpa}\,(\lambda x.\, ff_Q(x, \mathbf{c})) \ \wedge\ \mathbf{fresha}\,(\mathbf{d}, \lambda x.\, ff_P\,(x, \mathbf{c})) \ \wedge\ \mathbf{fresha}\,(\mathbf{d}, \lambda x.\, ff_Q(x, \mathbf{c})) \ \wedge \\ ff_P\,(\mathbf{d}, \mathbf{c}) = ff_Q\,(\mathbf{d}, \mathbf{c}) \ \longrightarrow\ \lambda x.\, ff_P\,(x, \mathbf{c}) = \lambda x.\, ff_Q(x, \mathbf{c}).\end{array}$$

The conditions of the implications can be derived like in the above case, this time employing that $\mathbf{c} \neq \mathbf{d}$, yielding the conclusion $\lambda x.\, ff_P\,(x, \mathbf{c}) = \lambda x.\, ff_Q(x, \mathbf{c})$.

In all of the proofs, we have used standard Isabelle proof-techniques. Altogether, the proofs of the theorems leading to the extensionality result, contain a bit less than 200 lines of proof-script code. Note that it was not obvious at the beginning that our well-formedness predicate would suffice to prove (Ext), as it does not rule out all exotic terms. From the fact that we have been able to

Table 6. Abstracting over a name in a process.

$$
\begin{array}{rcl}
[\![\mathbf{a}, []]\!] & = & \lambda x.\, x \\
[\![\mathbf{a}, (\mathbf{b}, f_a)xs]\!] & = & \mathit{if}\ \mathbf{a} = \mathbf{b}\ \mathit{then}\ f_a\ \mathit{else}\ [\![\mathbf{a}, xs]\!] \\
 & & \\
[\![0,\, xs,\, ys]\!] & = & \lambda x.\, 0 \\
[\![\tau.P,\, xs,\, ys]\!] & = & \lambda x.\, \tau.[\![P,\, xs,\, ys]\!] \\
[\![\bar{\mathbf{a}}\mathbf{b}.P,\, xs,\, ys]\!] & = & \lambda x.\, \overline{[\![a,\, xs]\!](x)}[\![\mathbf{b},\, xs]\!](x).[\![P,\, xs,\, ys]\!](x) \\
[\![\mathbf{a}y.f_P(y),\, xs,\, ys]\!] & = & \lambda x.\, [\![\mathbf{a},\, xs]\!](x)y.\, [\![f_P(\mathit{fst}(ys)),\, (\mathit{fst}(ys),\, (\lambda x.\, y))xs,\, \mathit{tl}(ys)]\!](x) \\
[\![(\nu\, y)f_P(y),\, xs,\, ys]\!] & = & \lambda x.\, (\nu\, y)\, [\![f_P(\mathit{fst}(ys)),\, (\mathit{fst}(ys),\, (\lambda x.\, y))xs,\, \mathit{tl}(ys)]\!](x) \\
[\![P + Q,\, xs,\, ys]\!] & = & \lambda x.\, [\![P,\, xs,\, ys]\!](x) + [\![Q,\, xs,\, ys]\!](x) \\
[\![P \parallel Q,\, xs,\, ys]\!] & = & \lambda x.\, [\![P,\, xs,\, ys]\!](x) \parallel [\![Q,\, xs,\, ys]\!](x) \\
[\![[\mathbf{a} = \mathbf{b}]P,\, xs,\, ys]\!] & = & \lambda x.\, [[\![\mathbf{a},\, xs]\!](x) = [\![\mathbf{b},\, xs]\!](x)][\![P,\, xs,\, ys]\!](x) \\
[\![[\mathbf{a} \neq \mathbf{b}]P,\, xs,\, ys]\!] & = & \lambda x.\, [[\![\mathbf{a},\, xs]\!](x) \neq [\![\mathbf{b},\, xs]\!](x)][\![P,\, xs,\, ys]\!](x) \\
[\![!P,\, xs,\, ys]\!] & = & \lambda x.\, ![\![P,\, xs,\, ys]\!](x)
\end{array}
$$

prove *ext*, we can infer that every remaining exotic term is extensionally equal (in the universally quantified sense) to a term which directly corresponds to a process in the π-calculus.

Extensionality for process abstractions taking two names as arguments can be derived from (Ext) if the process abstractions are well-formed for all instantiations of their first and second arguments. In the proof, a fresh name is chosen, and (Ext) is instantiated twice, once with that new fresh name, and a second time with the fresh name from the premise; that is, the argument from the proof of (Ext) is replayed, in an Isabelle proof-script of about 20 lines of code.

4.4 Beta Expansion

Though seeming fully natural, β-expansion (Exp) has turned out to be the trickiest law to prove. The reason for this is two-fold: (1) Unlike in the proof of (Ext), we cannot directly apply induction, due to the existential quantification in the conclusion. Instead, we encode a primitively recursive translation-function $[\![\_]\!]$ abstracting over a name in a well-formed process. (2) This function has to compare all names with the name to be abstracted over, which works well for object-variables, but in a naive implementation, could accidentally replace every meta-variable with a conditional. As a result, every well-formed process with binders would be transformed into an exotic process-abstraction. For example, an abstraction $\mathbf{a}y.0$ over $\mathbf{a}$ would result in $\lambda x.\, x(\mathbf{if}\ \mathbf{a} = y\ \mathbf{then}\ x\ \mathbf{else}\ y).0$.

The transformation. We therefore propose a function coercing from higher-order to first-order syntax and back. The two lists, xs and ys, in $[\![P, xs, ys]\!]$ are computed prior to the transformation. List xs is the *transformation list* telling for

every free name in P the names-abstraction it shall be mapped to during the transformation; except for the name to be abstracted over, it associates a constant function $\lambda x.\,\mathbf{a}$ with every free name $\mathbf{a}$ in P. List ys contains as many fresh names as are necessary to instantiate every meta variable in P; we compute it with the help of $db(P, \mathbf{c})$ (see Table 1) for some arbitrary name $\mathbf{c}$, and law (f2). The transformation intuitively proceeds as follows (refer to Table 6 for its formalization): every name that is encountered is mapped to the names-abstraction denoted in the transformation list xs. Only the name that is to be abstracted over does not occur in xs, hence it is transformed into $\lambda x.x$. Whenever the transformation comes across a binder, that is, input or restriction, it instantiates the continuation with the first fresh name from ys, that is, $fst(ys)$, and adds a pair $(fst(ys), (\lambda x.\,y))$ to xs, where y is the meta-variable given by the binder. When the transformation later encounters the instantiated (object-level) name, it thus abstracts over it again. This methodology—that is, first instantiating and later restoring meta-variables in a process abstraction—prevents meta-variables from being compared with the object-variable to be abstracted over.

Well-formedness. We call an abstraction over a transformation list well-formed if it only applies well-formed names-abstractions (see Table 4 for a definition):

$$\frac{}{\mathbf{wftrl}\,(\lambda x.\,[])}\;\mathbf{W}_1^t \qquad \frac{\mathbf{wfnaa}\,(ff_a) \quad \mathbf{wftrl}\,(f_{xs})}{\mathbf{wftrl}\,(\lambda x.\,(\mathbf{a}, ff_a(x))f_{xs}(x))}\;\mathbf{W}_2^t$$

The following two theorems prove that the transformation described above produces well-formed process abstractions when applied to well-formed processes:

$$\frac{\mathbf{wfpa}\,(f_P) \quad \mathbf{wftrl}\,(f_{xs})}{\begin{array}{c}\mathbf{wfpa}\,([\![f_P(\mathbf{c}), f_{xs}(\mathbf{d}), ys]\!]) \wedge \\ \mathbf{wfpa}\,(\lambda x.\,[\![f_P(\mathbf{c}), f_{xs}(x), ys]\!](\mathbf{b}))\end{array}} \qquad \frac{\mathbf{wfp}\,(P) \quad \forall(\mathbf{a}, f_a) \in xs.\,\mathbf{wfna}\,(f_a)}{\mathbf{wfpa}\,([\![P, xs, ys]\!])}$$

The proofs of the two theorems are tedious but purely technical inductions. The main difficulty was to formulate a suitable notion of abstraction over transformation-lists (see above), and the first of the two theorems. Note that the second theorem can only be proved as a consequence of the first.

Freshness. In order to prove that the transformation really eliminates the intended name $\mathbf{a}$, we choose a name $\mathbf{b} \neq \mathbf{a}$, and derive by two technical inductions,

$$\frac{\mathbf{wfpa}\,(f_P) \qquad \forall(\mathbf{d}, f_d) \in xs.\;\; \mathbf{a} \neq f_d(\mathbf{b}) \qquad \mathbf{a} \neq \mathbf{b}}{\mathbf{fresh}\,(\mathbf{a}, [\![f_P(\mathbf{c}), xs, ys]\!](\mathbf{b}))}$$

$$\frac{\mathbf{wfp}\,(P) \qquad \forall(\mathbf{d}, f_d) \in xs.\;\; \mathbf{a} \neq f_d(\mathbf{b}) \qquad \mathbf{a} \neq \mathbf{b}}{\mathbf{fresh}\,(\mathbf{a}, [\![P, xs, ys]\!](\mathbf{b}))}$$

Again the proof of the second theorem is based on that of the first. In the proofs, we make extensive use of law (f6) from Section 4.1.

Equality. It remains to show, again by induction, that a reinstantiation of a transformation yields the original process. The proofs make use of the monotonicity and extensionality theorems proved in Sections 4.2 and 4.3, as well as of the well-formedness and freshness results from the previous two sections. Therefore, we have to guarantee, by using *db*, that *ys* contains at least as many names as there are nested binders in a process. We use a predicate **nodups**, to ensure that *ys* does not contain duplicates. The function *fst* maps pairs to their first item; when applied to a list $(a_1, b_1) \ldots (a_n, b_n)$ it returns $a_1 \ldots a_n$.

$$\frac{\mathbf{wfpa}\,(f_P) \quad \forall(\mathbf{b}, f_b) \in xs.\ f_b = \lambda x.\,\mathbf{b} \quad dba\,(f_P, \mathbf{c}) \leq |ys| \quad fna\,(f_P) \subseteq \{\mathbf{a}\} \cup fst\,(xs) \quad \mathbf{a} \not\in fst\,(xs) \quad \mathbf{d} \in fst\,(xs) \quad \mathbf{nodups}\,(ys) \quad ys \cap (\{\mathbf{a}\} \cup fst\,(xs)) = \emptyset}{[\![f_P(\mathbf{d}), xs, ys]\!](\mathbf{a}) = f_P(\mathbf{d})}$$

$$\frac{\mathbf{wfp}\,(P) \quad \forall(\mathbf{b}, f_b) \in xs.\ f_b = \lambda x.\,\mathbf{b} \quad db\,(P, \mathbf{c}) \leq |ys| \quad fn\,(P) \subseteq \{\mathbf{a}\} \cup fst\,(xs) \quad \mathbf{a} \not\in fst\,(xs) \quad \mathbf{nodups}\,(ys) \quad ys \cap (\{\mathbf{a}\} \cup fst\,(xs)) = \emptyset}{[\![P, xs, ys]\!](\mathbf{a}) = P}$$

The proofs are tedious but purely technical. Whenever a process abstraction is encountered, the first name in *ys* is used as a fresh name, and (Ext) is applied.

The mechanization of the proofs of β-expansion in Isabelle/HOL consist of about 350 lines of proof-script code.

4.5 General Evaluation and Further Work

In this section, we have formally derived the theory of contexts for a shallow embedding of the π-calculus in Isabelle/HOL, using structural induction as provided by a well-formedness predicate for processes and process-abstractions. A similar theory of contexts can be applied to every languages with binders for which free names play a role in the semantic analysis.

Applying (Ext) from the theory of contexts, we have further derived adequacy of the encoding in Isabelle/HOL [25] with respect to a first-order formalization of the π-calculus. An interesting fact is that it would have been hardly possible to derive adequacy without the previous establishment of the theory of contexts.

All proof-scripts referred to in this section, including adequacy, are available at `http://www7.in.tum.de/~roeckl/PI/syntax.shtml`.

5 Discussion

Why well-formed processes? Like for the replacement of names, there are two principal ways of ruling out exotic terms. [15,11] rely on λProlog and Coq to produce the set of well-formed processes. They hence delegate well-formedness to the meta-level of the provers. In our formalization, we do not rely on Isabelle to rule out exotic terms, but do this explicitely on the object-level by means

of inductive predicates. As a consequence, we can mimic structural induction—which is generally non-existent in shallow embeddings—by rule-induction over the well-formedness predicates. This gives us a powerful tool for syntax analysis, which the formalizations in [15,11] do not possess. As a result, we can reason both *within* and *about* the π-calculus. Further, we are not restricted to specialized inductive frameworks for making (Ext) valid, but could adapt our framework to any general-purpose theorem-prover allowing for shallow embeddings.

What about other provers than Isabelle/HOL? In principle, our mechanization can be replayed on any theorem prover offering shallow embeddings, recursive functions, and inductive sets. There is no need that the framework rules out exotic terms automatically. Adapting our proofs to Coq would require modifying the notion of equality, by adding an extensionality axiom, and stating decidability of equality on names: both properties come indeed for free in Isabelle/HOL. In principle, one could then—and would have to, in that case (See Section 1 and [10])—add well-formedness predicates to the formalization presented in [11] and formally derive (Mon), (Ext), and (Exp).

In logical frameworks like λProlog [19] and Elf [23], encodings naturally exploit higher-order abstract syntax, and exotic terms are excluded automatically by the meta-logic. On the other hand, these frameworks do not offer adequate induction principles, hence syntactic properties often cannot be derived within the encoding. Recent work attemps to bridge this gap: the theorem-prover Twelf [24] implements a meta-logic based on Elf which offers a form of automated induction. Similarly, McDowell and Miller propose $FO\lambda^{\Delta\mathbb{N}}$ [13], a meta-logic where induction over natural numbers can be used to reason on a subset of simply typed λ-calculus. While it may be possible to adapt the results presented in this paper to such frameworks, it remains an open question how much support these systems can offer in semantic proofs about transition-systems and bisimulations.

How many names do we need? Any type with at least countably infinitely many elements fits our formalization. The reason why there cannot be less names is that the proofs of extensionality and β-expansion are based on the creation of fresh names for processes or process-abstractions. The situation is less simple in the work of Honsell et al. [11], where the meta-level of Coq is used as an inductive logical framework, fully stripped of an object-logic. Still, to guarantee for the absence of exotic terms, it is necessary that the ability to compare names is only defined in `Prop` and not in `Set`. This rules out in particular any inductive type for names.

What about justifying the theory of contexts? This work is not a mechanized justification of the work of Honsell, Miculan, and Scagnetto [11]. To do this, we would have to encode the meta-level of Coq, that is, the Calculus of Constructions, in a prover, and then employ, for instance, a category-theoretical argument, which seems quite illusive. Our work should rather be seen as a formal justification of the theory of contexts within a shallow embedding of the π-calculus. As such, our work can be related to that of Gordon and Melham

[7]. There, an axiomatization of α-conversion in HOL is proposed, which serves as a framework for the derivation of syntax definitions, as well as principles for substitution and induction.

Is the theory of contexts really necessary? The three properties presented in the theory of contexts, and formally justified in this paper, are essential for the semantic analysis of π-calculus processes. The reason is that in transition-systems and bisimulation-proofs, both free and bound names play a role. Recently, Despeyroux has proposed a formalized strong transition-system for a fragment of the π-calculus within a shallow embedding, which reduces the number of instantiations by modelling derivatives in terms of functions [3]. More precisely, she presents a shallow embedding of Milner's transition system for the π-calculus based on abstractions and concretions [16]. It will have to be investigated how this formalism can be extended to the full π-calculus, and whether a theory of contexts is also necessary to reason about semantics or not.

Is HOAS worth it? So far, work on higher-order abstract syntax for the π-calculus has focused on introducing the main concepts [15,11] or presenting fundamental applications [11,3]. The work at hand belongs to the first category, in that it provides a framework for reasoning within the π-calculus in a shallow embedding. It should not be regarded as a case-study for the application of HOAS to the π-calculus; we rather hope it might contribute to a language-independent method of syntax analysis in shallow embeddings, in terms of the general concepts and proofs described in the previous sections. The practical aspects of applying HOAS to the π-calculus will have to be further examined by large-scale case-studies based on frameworks like the ones presented in [11] and in this paper.

Acknowledgements. We thank Joëlle Despeyroux, Gilles Dowek, Javier Esparza, René Lalement, Tobias Nipkow, and Peter Rossmanith, for helpful comments and discussions. This work has been supported by the PROCOPE project 9723064, "Verification Techniques for Higher Order Imperative Concurrent Languages".

References

1. O. Aït-Mohamed. *Pi-Calculus Theory in HOL.* PhD thesis, Henry Poincarré University, Nancy, 1996.
2. S. Berghofer and M. Wenzel. Inductive datatypes in HOL — lessons learned in Formal-Logic Engineering. In *Proc. TPHOL'99*, volume 1690 of *LNCS*, pages 19–36, 1999.
3. J. Despeyroux. A higher-order specification of the π-calculus. In *Proc. TCS'00*, LNCS. Springer, 2000. To appear.
4. J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In *Proc. TLCA'95*, volume 902 of *LNCS*, pages 124–138. Springer, 1995.
5. J. Despeyroux and A. Hirschowitz. Higher-order abstract syntax with induction in Coq. In *Proc. LPAR'94*, volume 822 of *LNCS*, pages 159–173. Springer, 1994.

6. S. Gay. A framework for the formalisation of pi-calculus type-systems in Isabelle/HOL. Technical report, University of Glasgow, 2000.
7. A. Gordon and T. Melham. Five axioms of alpha-conversion. In *Proc. TPHOL'96*, volume 1125 of *LNCS*, pages 173–190. Springer, 1996.
8. L. Henry-Gréard. Proof of the subject reduction property for a pi-calculus in Coq. Technical Report RR-3698, INRIA, 1999.
9. D. Hirschkoff. A full formalisation of π-calculus theory in the calculus of constructions. In *Proc. TPHOL'97*, volume 1275 of *LNCS*, pages 153–169. Springer, 1997.
10. M. Hofmann. Semantical analysis of higher-order abstract syntax. In *Proc. LICS'99*, volume 158, pages 204–213. IEEE, 1999.
11. F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.
12. B. Mammass. *Méthodes et Outils pour les Preuve Compositionnelles de Systèmes Paralleèles (in french)*. PhD thesis, Pierre et Marie Curie University, Paris, 1999.
13. R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *Transactions on Computational Logic*, 2000. to appear.
14. T. Melham. A mechanized theory of the π-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1995.
15. D. Miller. Specification of the pi-calculus. available at `http://www.cse.psu.edu/~dale/lProlog/examples/pi-calculus/toc.html`.
16. R. Milner. Functions as processes. *Journal of Math. Struct. in Computer Science*, 17:119–141, 1992.
17. R. Milner. *Communicating and Mobile Processes*. Cambridge University Press, 1999.
18. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
19. G. Nadathur and D. Miller. An overview of λprolog. In M. Press, editor, *Proc. LPC'98*, pages 810–827, 1998.
20. L. C. Paulson. Isabelle's object-logics. Technical Report 286, University of Cambridge, Computer Laboratory, 1993.
21. L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In *Procs CADE'94*, volume 814 of *LNAI*, pages 148–161. Springer, 1994.
22. L. C. Paulson, editor. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, 1994.
23. F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proc. LICS'89*, pages 313–321. IEEE, 1989.
24. F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Proc. CAD'99*, volume 1632 of *LNAI*, pages 202–206. Springer, 1999.
25. C. Röckl. *On the Mechanized Validation of Infinite-State and Parameterized Reactive and Mobile Systems*. PhD thesis, Technische Universität München, 2001. Submitted.
26. C. Röckl and D. Sangiorgi. A π-calculus process semantics of concurrent idealised ALGOL. In *Proc. FOSSACS'99*, volume 1578 of *LNCS*, pages 306–321. Springer, 1999.
27. D. Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.

Decidability of Weak Bisimilarity for a Subset of Basic Parallel Processes

Colin Stirling

Division of Informatics
University of Edinburgh
email: cps@dcs.ed.ac.uk

1 Introduction

In the past decade there has been a variety of results showing decidability of bisimulation equivalence between infinite state systems. The initial result, due to Baeten, Bergstra and Klop [1], proved decidability for normed BPA processes, described using irredundant context-free grammars. This was extended to all BPA processes and then to pushdown automata [5,16,14]. Decidability of bisimilarity was also shown for Basic Parallel (BP) processes, a restricted subset of Petri nets, [4]. For full Petri nets Jančar proved that bisimulation equivalence is undecidable [11].

An open question is the dividing line between decidability and undecidability of bisimilarity in the case of "sequential" systems. For instance, is bisimulation equivalence decidable for the general class of prefix-recognisable transition graphs introduced by Caucal [2]? A poignant problem is that these graphs exhibit infinite branching. Families of infinite state systems for which bisimilarity is known to be decidable are finitely branching. For each label a and for each configuration the set of its a-successors is finite and easily computable. Therefore if two systems are not bisimulation equivalent then there is a least approximant $n > 0$ such that they are not equivalent at level n, and for each n the equivalence at level n is decidable. But if processes are infinite branching then inequivalence may be manifested at higher ordinals, and therefore a new technique is required to establish semidecidability of inequivalence.

Instead of examining richer families of infinite state systems one can look at the problem of deciding weak bisimulation equivalence for restricted classes. Weak bisimilarity abstracts from silent activity, with the consequence that BPA and BP processes are infinitely branching. Weak bisimulation inequivalence is then generally not finitely approximable.

In this paper we examine the decision problem of weak bisimilarity for normed BP processes. Esparza [6] observes that weak bisimilarity is semidecidable, because a positive witness is semilinear. Decidability was proved for a restricted subclass, the totally normed processes, by Hirshfeld [7]. And Jančar, Kučera and Mayr show decidability of weak bisimilarity between general (PA) processes which includes BP processes and finite state processes [12]. However in both these cases inequivalence is finitely approximable. In this paper we prove decidability

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 379–393, 2001.

of weak bisimilarity for a subset of normed BP processes for which inequivalence need not be finitely approximable. Underpinning this result is a finite symbolic characterisation of the infinite branching of normed BP processes. Indeed we believe that the technique will establish decidability of weak bisimilarity for all normed BP processes, but the combinatorics become awesome.

In section 2 we define normed Basic Parallel processes and weak bisimulation equivalence. Section 3 is devoted to the finite characterisation of the infinite transition relations. Then in section 4 we utilise the characterisation to prove the decidability result using the tableau method. Proofs of two crucial lemmas are given in section 5.

2 Normed Basic Parallel Processes

Ingredients of Basic Parallel (BP) processes are a finite set $\Gamma = \{X_1, \ldots, X_n\}$ of atoms, a finite set $\mathsf{A} = \{a_1, \ldots, a_k\}$ of actions and a finite set T of basic transitions, each of the form $X \stackrel{a}{\longrightarrow} \alpha$ where X is an atom, $a \in \mathsf{A} \cup \{\tau\}$ and α is a multiset of atoms whose size is at most 2. A BP process, or configuration, is a parallel composition of atoms. We let α, β, ... range over such processes. A process therefore has the form $X_1^{k_1} \ldots X_n^{k_n}$, which is the parallel composition of k_1 copies of X_1, ... and k_n copies of X_n where each $k_i \geq 0$. We let ϵ be the empty composition, where each $k_i = 0$. If α and β are two processes then $\alpha\beta$ is their multiset union (and we often write $X\alpha$ or αX as an abbreviation for the multiset union of $\{X\}$ and α). The behaviour of a BP process is determined by the following extension rule: if $X \stackrel{a}{\longrightarrow} \alpha \in \mathsf{T}$ then $X\beta \stackrel{a}{\longrightarrow} \alpha\beta$. The silent action $\tau \not\in \mathsf{A}$ is included as a possible action. We assume the usual expansion of the transition relation to words, $\alpha \stackrel{w}{\longrightarrow} \beta$ where $w \in (\mathsf{A} \cup \{\tau\})^*$.

Example 1 The atoms Γ are $\{A, Y, Z\}$ and A is the singleton set $\{a\}$. The basic transitions are $A \stackrel{a}{\longrightarrow} \epsilon$, $Y \stackrel{a}{\longrightarrow} A$, $Z \stackrel{a}{\longrightarrow} A$, $Y \stackrel{\tau}{\longrightarrow} YA$, $Z \stackrel{a}{\longrightarrow} Z$, $Z \stackrel{\tau}{\longrightarrow} ZA$. ZA^3 has the following transitions, $ZA^3 \stackrel{a}{\longrightarrow} A^4$, $ZA^3 \stackrel{a}{\longrightarrow} ZA^2$, $ZA^3 \stackrel{a}{\longrightarrow} ZA^3$ and $ZA^3 \stackrel{\tau}{\longrightarrow} ZA^4$. □

Example 2 The atoms are $\{C, D, U, V\}$, $\mathsf{A} = \{c, d\}$ and the basic transitions are $U \stackrel{\tau}{\longrightarrow} UD$, $U \stackrel{c}{\longrightarrow} \epsilon$, $U \stackrel{c}{\longrightarrow} C$, $U \stackrel{c}{\longrightarrow} U$, $V \stackrel{\tau}{\longrightarrow} VD$, $V \stackrel{c}{\longrightarrow} \epsilon$, $V \stackrel{c}{\longrightarrow} C$, $C \stackrel{c}{\longrightarrow} C$, $C \stackrel{c}{\longrightarrow} \epsilon$, $D \stackrel{d}{\longrightarrow} \epsilon$, $D \stackrel{\tau}{\longrightarrow} \epsilon$. For each $n \geq 0$ there is the extended transition $U \stackrel{\tau^n c}{\longrightarrow} UD^n$. □

BP processes are communication free Petri nets, where the places are the atoms, and transition $X \stackrel{a}{\longrightarrow} \alpha$ is a firing rule. A configuration $X_1^{k_1} \ldots X_n^{k_n}$ represents the marking when there are k_i tokens on place X_i. They are communication free because each transition requires just one place to fire. As is usual in process calculi when there are silent transitions, the weak transition relations $\stackrel{\epsilon}{\Longrightarrow}$ and $\stackrel{a}{\Longrightarrow}$ for $a \in \mathsf{A}$ are defined as follows.

$$\alpha \stackrel{\epsilon}{\Longrightarrow} \beta \text{ iff } \exists n \geq 0.\ \alpha \stackrel{\tau^n}{\longrightarrow} \beta \qquad \alpha \stackrel{a}{\Longrightarrow} \beta \text{ iff } \exists \alpha_1, \beta_1.\ \alpha \stackrel{\epsilon}{\Longrightarrow} \alpha_1 \stackrel{a}{\longrightarrow} \beta_1 \stackrel{\epsilon}{\Longrightarrow} \beta$$

There can be infinite branching with respect to these transition relations, as illustrated by Examples 1 and 2, $Y \stackrel{\epsilon}{\Longrightarrow} YA^n$ and $V \stackrel{c}{\Longrightarrow} CD^n$ for all $n \geq 0$.

An atom X is normed if there is a word $w \in \mathsf{A}^*$ such that $X \stackrel{w}{\Longrightarrow} \epsilon$. A BP definition is normed if all its atoms are normed. Both Examples 1 and 2 are normed. The norm of atom X, written $\mathrm{N}(X)$, is the length of a shortest word w such that $X \stackrel{w}{\Longrightarrow} \epsilon$. $\mathrm{N}(Y) = 2$ and $\mathrm{N}(D) = 0$ where Y and D are from the examples above. Norm extends to configurations α, written $\mathrm{N}(\alpha)$, which is the length of a shortest word w such that $\alpha \stackrel{w}{\Longrightarrow} \epsilon$. If a BP definition is normed then so is any process $\alpha = X_1^{k_1} \ldots X_n^{k_n}$ and $\mathrm{N}(\alpha)$ is $\sum_{1 \leq i \leq n}(k_i \times \mathrm{N}(X_i))$. A subset of normed BP processes is the totally normed processes, as introduced by Hüttel and examined by Hirshfeld [9,7]. A BP process definition is totally normed if all its atoms are normed and have norm greater than 0. Example 1 is totally normed but Example 2 is not because $\mathrm{N}(D) = 0$.

Our interest is with deciding when two normed BP processes are weak bisimulation equivalent. There is more than one way to define this equivalence. First we start with the natural (and "symmetric") version.

Definition 1 A binary relation B between BP processes is a weak bisimulation relation provided that whenever $\alpha B \beta$ and $a \in (\mathsf{A} \cup \{\epsilon\})$

$$\text{if } \alpha \stackrel{a}{\Longrightarrow} \alpha' \text{ then there is a } \beta' \text{ such that } \beta \stackrel{a}{\Longrightarrow} \beta' \text{ and } \alpha' B \beta'$$
$$\text{if } \beta \stackrel{a}{\Longrightarrow} \beta' \text{ then there is an } \alpha' \text{ such that } \alpha \stackrel{a}{\Longrightarrow} \alpha' \text{ and } \alpha' B \beta'$$

Two processes α and β are weakly bisimilar, written $\alpha \approx \beta$, if there is a weak bisimulation relation B such that $\alpha B \beta$. Important properties of weak equivalence are that it is a congruence for BP processes, see [7] for instance, and that it preserves norm.

Fact 1 *If $\alpha \approx \beta$ then $\alpha\delta \approx \beta\delta$ and $\mathrm{N}(\alpha) = \mathrm{N}(\beta)$.*

An alternative (and equivalent) basis for the definition of weak bisimilarity is as follows, where if $a \in A$ then $\hat{a}$ is a and if $a = \tau$ then $\hat{a}$ is ϵ, [13].

Definition 2 A binary relation B between BP processes is a wb relation provided that whenever $\alpha B \beta$ and $a \in (\mathsf{A} \cup \{\tau\})$

$$\text{if } \alpha \stackrel{a}{\longrightarrow} \alpha' \text{ then there is a } \beta' \text{ such that } \beta \stackrel{\hat{a}}{\Longrightarrow} \beta' \text{ and } \alpha' B \beta'$$
$$\text{if } \beta \stackrel{a}{\longrightarrow} \beta' \text{ then there is an } \alpha' \text{ such that } \alpha \stackrel{\hat{a}}{\Longrightarrow} \alpha' \text{ and } \alpha' B \beta'$$

Fact 2 *B is a wb relation iff B is a weak bisimulation relation.*

To establish that $\alpha \approx \beta$ it therefore suffices to exhibit a binary relation containing α and β and prove that it is a wb relation. In general such a relation will be infinite. The relation $\{(YA^n, ZA^n), (A^n, A^n) : n \geq 0\}$ over processes of Example 1 is a wb relation, which proves that $Y \approx Z$.

The "symmetric" definition of equivalence supports weak bisimulation approximants, $\approx_o$ for any ordinal o, which are themselves equivalence relations.

Definition 3 The relations $\approx_o$ for ordinals o are defined inductively as follows, where we assume that l is a limit ordinal (such as ω).

$\alpha \approx_0 \beta$
$\alpha \approx_{o+1} \beta$ iff for $a \in (\mathsf{A} \cup \{\epsilon\})$
if $\alpha \stackrel{a}{\Longrightarrow} \alpha'$ then $\exists \beta'.\ \beta \stackrel{a}{\Longrightarrow} \beta'$ and $\alpha' \approx_o \beta'$
if $\beta \stackrel{a}{\Longrightarrow} \beta'$ then $\exists \alpha'.\ \alpha \stackrel{a}{\Longrightarrow} \alpha'$ and $\alpha' \approx_o \beta'$
$\alpha \approx_l \beta$ iff $\forall o < l.\ \alpha \approx_o \beta$

Fact 3 *$\alpha \approx \beta$ iff for all ordinals o. $\alpha \approx_o \beta$.*

Example 2 illustrates the need for ordinals beyond ω. Although $U \not\approx V$ for any $n \geq 0$, $U \approx_n V$. The inequivalence is due to the transition $U \stackrel{c}{\Longrightarrow} U$. Process V does not have a similar transition. However for any $n \geq 0$, $V \stackrel{c}{\Longrightarrow} CD^n$ and $U \approx_n CD^n$ but $U \not\approx_{n+1} CD^n$ because of the transition $U \stackrel{d}{\Longrightarrow} UD^n$. Therefore it follows that $U \not\approx_{\omega+1} V$. It is conjectured by Stříbrná [15] that one only needs ordinals which are less than $\omega \times 2$ to establish inequivalence between all BP processes, including the unnormed. She proves this in the special case when there are no atoms of norm 0 and A is a singleton set. And in the case of totally normed processes she shows that the closure ordinal is ω: if $\alpha \not\approx \beta$ then for some $n \geq 0$, $\alpha \not\approx_n \beta$.

Esparza observes that a wb relation which witnesses the equivalence $\alpha \approx \beta$ is semilinear [6], which establishes semidecidability of weak bisimilarity for all BP processes (and therefore decidability for totally normed processes). But the problem is establishing semidecidability of inequivalence.

A new approach to deciding weak equivalence is now developed. First we finitely characterise the infinite branching of a normed BP process, and then we use the characterisation to show that equivalence and inequivalence can be captured by examining only boundedly many transitions. However we are only able to prove decidability for a subset of normed BP processes which includes the totally normed processes. The family also includes Example 2 where inequivalence is not finitely approximable. The subset is given by a technical restriction, whose notation is now developed.

3 Stratification and Generators

In this section we symbolically characterise the weak transition relations of normed BP processes. Assume a fixed normed BP process definition with atoms Γ, action set A and transitions T. The initial step is to stratify the basic transitions in T, by including a numerical index on the transition relation which represents the change in norm produced by the transition. If $X \stackrel{a}{\longrightarrow} \alpha \in \mathsf{T}$ then we re-write it as $X \stackrel{a}{\longrightarrow}_n \alpha$ where $n = \mathrm{N}(\alpha) - \mathrm{N}(X)$. The index n is bounded, $-1 \leq n \leq 2\mathrm{M}$, where M is the maximum norm of any atom in Γ. Either a transition is norm reducing, but then by at most 1, or it is nondecreasing and because $|\alpha| \leq 2$ the increase in norm is at most 2M. An important, but simple, observation is that for a stratified τ-transition, $X \stackrel{\tau}{\longrightarrow}_n \alpha$, the index n must be nondecreasing, $n \geq 0$. A selection of stratified transitions from Examples 1 and 2 of the previous section is $A \stackrel{a}{\longrightarrow}_{-1} \epsilon$, $Z \stackrel{a}{\longrightarrow}_0 Z$, $Y \stackrel{\tau}{\longrightarrow}_1 YA$, $U \stackrel{c}{\longrightarrow}_{-1} \epsilon$, $D \stackrel{\tau}{\longrightarrow}_0 \epsilon$, $V \stackrel{\tau}{\longrightarrow}_0 VD$.

The definition of stratification is extended to the weak transition relations as follows.

$$\begin{array}{ll}
\alpha \stackrel{\epsilon}{\Longrightarrow}_0 \beta & \text{iff } \exists m > 0.\, \exists \alpha_1, \ldots, \alpha_m.\, \alpha = \alpha_1 \stackrel{\tau}{\longrightarrow}_0 \ldots \stackrel{\tau}{\longrightarrow}_0 \alpha_m = \beta \\
\alpha \stackrel{\epsilon}{\Longrightarrow}_{n+1} \beta & \text{iff } \exists \alpha', \beta'.\, \alpha \stackrel{\epsilon}{\Longrightarrow}_j \alpha' \stackrel{\tau}{\longrightarrow}_{k+1} \beta' \stackrel{\epsilon}{\Longrightarrow}_l \beta \quad \text{where } n = j + k + l \\
\alpha \stackrel{a}{\Longrightarrow}_n \beta & \text{iff } \exists \alpha', \beta'.\, \alpha \stackrel{\epsilon}{\Longrightarrow}_j \alpha' \stackrel{a}{\longrightarrow}_k \beta' \stackrel{\epsilon}{\Longrightarrow}_l \beta \quad \text{where } n = j + k + l
\end{array}$$

For instance, $U \stackrel{\epsilon}{\Longrightarrow}_0 UD^{64}$ and $U \stackrel{c}{\Longrightarrow}_{-1} D^{80}$ are stratified weak transitions arising from Example 2 of the previous section.

Weak bisimulation equivalence can be redefined using the stratified weak transition relations. Assume that K is the largest increase in norm of the BP process definition, $\max\{n : X \stackrel{a}{\longrightarrow}_n \alpha \in \mathsf{T}\}$. For instance, K = 1 for Example 1 of the previous section.

Definition 1 A relation B between normed BP processes is a stratified weak bisimulation relation provided that whenever $\alpha B \beta$ and $n \leq \mathrm{K}$ and $a \in (\mathsf{A} \cup \{\epsilon\})$

$$\begin{array}{l}
\text{if } \alpha \stackrel{a}{\Longrightarrow}_n \alpha' \text{ then there is a } \beta' \text{ such that } \beta \stackrel{a}{\Longrightarrow}_n \beta' \text{ and } \alpha' B \beta' \\
\text{if } \beta \stackrel{a}{\Longrightarrow}_n \beta' \text{ then there is an } \alpha' \text{ such that } \alpha \stackrel{a}{\Longrightarrow}_n \alpha' \text{ and } \alpha' B \beta'
\end{array}$$

Proposition 1 *B is a stratified weak bisimulation relation iff B is a wb relation.*

Hence $\alpha \approx \beta$ iff there is a stratified weak bisimulation relation B which contains the pair α and β. In the next section we shall also define associated stratified weak approximants.

The crucial feature of totally normed processes is that they are finitely branching with respect to the stratified weak transition relations.

Fact 1 *If α is totally normed then for all a and n $\{\delta : \alpha \stackrel{a}{\Longrightarrow}_n \delta\}$ is finite.*

This is not generally the case for normed BP processes. For instance $V \stackrel{c}{\Longrightarrow}_{-1} D^n$ for all $n \geq 0$. The crucial component of infinite branching is the relation $\stackrel{\epsilon}{\Longrightarrow}_0$, to which we now direct our analysis. The following result is useful.

Proposition 2 *If $\alpha \stackrel{\epsilon}{\Longrightarrow}_0 \beta$ and $\beta \stackrel{\epsilon}{\Longrightarrow}_0 \delta$ and $\alpha \approx \delta$ then $\alpha \approx \beta$.*

Consequently if $X \stackrel{\epsilon}{\Longrightarrow}_0 Y$ and $Y \stackrel{\epsilon}{\Longrightarrow}_0 X$ then $X \approx Y$. In this circumstance, if $X \neq Y$ then we say that atom Y is redundant because of X. A BP definition can therefore be replaced with an equivalent definition which does not contain redundant atoms. If $Y \in \Gamma$ is redundant because of $X \in \Gamma$ then we change Γ to $\Gamma - \{Y\}$ and we replace all transitions $Y \stackrel{a}{\longrightarrow} \alpha \in \mathsf{T}$ with $X \stackrel{a}{\longrightarrow} \alpha$ and all transitions $Z \stackrel{a}{\longrightarrow} Y\alpha \in \mathsf{T}$ with $Z \stackrel{a}{\longrightarrow} X\alpha$. It is clear that this transformation of a BP definition preserves weak bisimulation equivalence. We therefore assume that atoms of a BP definition adhere to the following condition: (1) if $X \neq Y$ and $X \stackrel{\epsilon}{\Longrightarrow}_0 Y$ then $\mathrm{not}(Y \stackrel{\epsilon}{\Longrightarrow}_0 X)$.

The reason that the transition relation $\stackrel{\epsilon}{\Longrightarrow}_0$ can be infinite branching is because atoms can "generate" other atoms. If $X \stackrel{\epsilon}{\Longrightarrow}_0 XA$ then we say that X generates A. And for each atom X, the set of atoms generated by X, written

$G(X)$, is $\{A : X \stackrel{\epsilon}{\Longrightarrow}_0 XA\}$. In Example 2 of the previous section, $G(D) = \emptyset$ and $G(U) = G(V) = \{D\}$.

Proposition 3

1. *If* $A \in G(X)$ *then* $\mathrm{N}(A) = 0$
2. *If* $A \in G(X)$ *and* $A \stackrel{\epsilon}{\Longrightarrow}_0 B$ *then* $B \in G(X)$
3. *If* $A \in G(B)$ *and* $B \in G(X)$ *then* $A \in G(X)$
4. *If* $\alpha \in G(X)^*$ *then* $X \stackrel{\epsilon}{\Longrightarrow}_0 X\alpha$
5. *If* $\alpha \in G(X)^*$ *then* $X \approx X\alpha$

If $A \in G(X)$ then $A^{n+1}X\alpha \approx X\alpha$. Hence any configuration α can be reduced to an equivalent minimal normal form $\mathrm{nf}(\alpha)$.

Definition 2 If $\alpha = X_1^{k_1} \ldots X_n^{k_n}$ then $\mathrm{nf}(\alpha) = X_1^{l_1} \ldots X_n^{l_n}$ where

1. if $j \neq i$ and $X_j \in G(X_i)$ and $k_i > 0$ then $l_j = 0$,
2. if $X_i \in G(X_i)$ and $k_i > 0$ and $\forall j \neq i.\ k_j = 0$ or $X_i \not\in G(X_j)$ then $l_i = 1$.
3. if $k_j = 0$ or for all i such that $k_i \neq 0$, $X_j \not\in G(X_i)$ then $l_j = k_j$.

Proposition 4

1. *If* $X_1^{l_1} \ldots X_n^{l_n} = \mathrm{nf}(\alpha)$ *and* $X_1^{j_1} \ldots X_n^{j_n} = \mathrm{nf}(\alpha)$ *then* $l_i = j_i$ *for each* i
2. $\alpha \approx \mathrm{nf}(\alpha)$

Assume a BP process definition which obeys condition (1) and let Γ^0 be the set of generable atoms, $\{A \in \Gamma : \exists X.\ A \in G(X)\}$. An "extended" configuration either has the form β where $\beta = \mathrm{nf}(\beta)$, or has the form $\beta A_1^* \ldots A_k^*$ where $\beta = \mathrm{nf}(\beta)$ and each $A_i \in \Gamma^0$ and $A_i \not\in G(X)$ for any $X \in \beta$, and $A_i \not\in \beta$.

Theorem 1 *For any configuration* α *and* $a \in (\mathsf{A} \cup \{\epsilon\})$ *and* n *there is a finite set of extended configurations* $\mathsf{E}(\alpha, a, n)$ *such that*

1. *if* $\alpha \stackrel{a}{\Longrightarrow}_n \delta$ *then either* $\mathrm{nf}(\delta) \in \mathsf{E}(\alpha, a, n)$ *or* $\delta = \delta_1 A_1^{l_1} \ldots A_k^{l_k}$ *and each* $l_i \geq 0$ *and* $\beta = \mathrm{nf}(\delta_1)$ *and* $\beta A_1^* \ldots A_k^* \in \mathsf{E}(\alpha, a, n)$,
2. *if* $\beta \in \mathsf{E}(\alpha, a, n)$ *then* $\alpha \stackrel{a}{\Longrightarrow}_n \beta$,
3. *if* $\beta A_1^* \ldots A_k^* \in \mathsf{E}(\alpha, a, n)$ *then* $\forall l_1 \geq 0. \ldots \forall l_k \geq 0.\ \ \alpha \stackrel{a}{\Longrightarrow}_n \beta A_1^{l_1} \ldots A_k^{l_k}$.

Proof: Assume configuration α and assume $a \in \mathsf{A} \cup \{\epsilon\}$ and $n \geq -1$. Any transition $\alpha \stackrel{a}{\Longrightarrow}_n \delta$ can be decomposed as follows $\alpha \stackrel{\epsilon}{\Longrightarrow}_j \alpha_1 \stackrel{a}{\longrightarrow}_k \delta' \stackrel{\epsilon}{\Longrightarrow}_l \delta$ where $j + k + l = n$. Clearly for the set $\{\delta : \alpha \stackrel{a}{\Longrightarrow}_n \delta\}$ there are only finitely many different indices j, k and l which can be involved in a decomposition (because $k \leq \mathrm{K}$ and both j and l are at least 0). In turn a transition $\lambda \stackrel{\epsilon}{\Longrightarrow}_m \lambda'$ can also be decomposed. If $m = 0$ then $\lambda \stackrel{\tau}{\longrightarrow}_0 \ldots \stackrel{\tau}{\longrightarrow}_0 \lambda'$ and if $m > 0$ then $\lambda \stackrel{\epsilon}{\Longrightarrow}_0 \lambda_1 \stackrel{\tau}{\longrightarrow}_k \lambda_1' \stackrel{\epsilon}{\Longrightarrow}_{m-k} \lambda'$ where $k > 0$. Hence any transition $\stackrel{a}{\Longrightarrow}_n$ is built from only finitely many compositions of transitions $\stackrel{\epsilon}{\Longrightarrow}_0$, $\stackrel{a}{\longrightarrow}_k$ and $\stackrel{\tau}{\longrightarrow}_m$ where $m > 0$. And for each λ the sets $\{\lambda' : \lambda \stackrel{a}{\longrightarrow}_k \lambda'\}$ and $\{\lambda' : \lambda \stackrel{\tau}{\longrightarrow}_m \lambda'\}$ are finite and bounded, from the BP process definition. Hence the important transitions

involved in a decomposition are the $\stackrel{\epsilon}{\Longrightarrow}_0$ transitions, which we now concentrate on. A transition of the form $X \stackrel{\tau}{\longrightarrow}_0 XA$ is a generating transition, and a transition $X \stackrel{\tau}{\longrightarrow}_0 X$ is useless. Consider any configuration β_0 and derivation $d = \beta_0 \stackrel{\tau}{\longrightarrow}_0 \beta_1 \stackrel{\tau}{\longrightarrow}_0 \ldots \stackrel{\tau}{\longrightarrow}_0 \beta_n$ such that no transition in the derivation is either a generating transition or a useless transition. For a fixed β_0 there are only finitely many such derivations, and therefore only finitely many configurations appearing in any such derivation, $\{\beta_0, \ldots, \beta_m\}$. This follows from condition (1) earlier: if $X \stackrel{\tau^+}{\longrightarrow}_0 Y\alpha$ and $Y \stackrel{\tau^+}{\longrightarrow}_0 X\delta$ and $X \neq Y$ then Y is redundant because of X (as $\mathrm{N}(\alpha) = 0 = \mathrm{N}(\delta)$). In fact a crude upper bound on the number of such final configurations[1] is $|\beta_0| \times 2^{|\Gamma|}$. For each derivation d of β_i let $d(\beta_i) \subseteq \Gamma$ be the subset of atoms which occur anywhere within the derivation, and let $G(d(\beta_i))$ be the set $\bigcup\{G(X) : X \in d(\beta_i)\}$. There are only finitely many different such sets associated with each β_i. For each such subset we introduce a preliminary extended configuration as follows. First if $G(d(\beta_i)) = \emptyset$ for some derivation d then one preliminary configuration is β_i. Next if $G(d(\beta_i)) = \{A_1, \ldots, A_k\}$ then another preliminary configuration is $\beta_i' A_1^* \ldots A_k^*$ where β_i' is the result of removing all occurrences of A_j from β_i. There are only finitely many such preliminary extended configurations associated with each β_i. Preliminary extended configurations are preliminary because they may not yet be in normal form. However it is easy to see that if $\beta_0 \stackrel{\epsilon}{\Longrightarrow}_0 \delta$ then either $\delta = \beta_i$ or $\delta = \beta_i' A_1^{l_1} \ldots A_k^{l_k}$ for some $l_1 \geq 0$, …, $l_k \geq 0$. Moreover if $\beta_i' A_1^* \ldots A_k^*$ is a preliminary extended configuration then by Proposition 3.4 for all $l_1 \geq 0$, …, $l_k \geq 0$, $\beta_0 \stackrel{\epsilon}{\Longrightarrow}_0 \beta_i' A_1^{l_1} \ldots A_k^{l_k}$. We now complete the argument of the theorem. First if $a = \epsilon$ and $n = 0$ and $\alpha = \beta_0$ then we merely tidy the preliminary extended configurations. If β_i is such a configuration then we let $\beta_i'' = \mathrm{nf}(\beta_i)$, and if $\beta_i' A_1^* \ldots A_k^*$ is a configuration then we let $\beta_i'' = \mathrm{nf}(\beta_i')$ and we remove each A_j^* such that $A_j \in G(X)$ when $X \in \beta_i''$. By the reasoning above the resulting finite set of extended configurations, $\mathsf{E}(\alpha, \epsilon, 0)$, obey the theorem. Otherwise $a \neq \epsilon$ or $n \neq 0$. Assume the finite set of preliminary extended forms associated with $\{\alpha_1 : \alpha \stackrel{\epsilon}{\Longrightarrow}_0 \alpha_1\}$. Consider the possible transitions $\stackrel{a}{\longrightarrow}_k$ from a preliminary extended form where $a \neq \tau$ or $k \neq 0$. There are two cases. First if the form is β_i then the required finite set is $\{\beta_{ij} : \beta_i \stackrel{a}{\longrightarrow}_k \beta_{ij}\}$. Second is that the preliminary form is $\beta_i' A_1^* \ldots A_k^*$. We now take the following finite set

$$\{\beta_{ij} A_1^* \ldots A_k^* : \beta_i' \stackrel{a}{\longrightarrow}_k \beta_{ij}\} \cup \{\beta_i' \delta A_1^* \ldots A_k^* : A_j \stackrel{a}{\longrightarrow}_k \delta\}$$

and then tidy their elements by removing any occurrences of A_j from β_{ij} and δ. The result is a finite set of preliminary extended forms, with the crucial property that if $\alpha \stackrel{\epsilon}{\Longrightarrow}_0 \alpha_1 \stackrel{a}{\longrightarrow}_k \alpha_1'$ then either α_1' is a preliminary form or $\alpha_1' = \beta_{ij}' A_1^{l_1} \ldots A_k^{l_k}$ for some $l_1 \geq 0$, …, $l_k \geq 0$ and $\beta_{ij}' A_1^* \ldots A_k^*$ is a preliminary form and for each such form and $l_1 \geq 0$, …, $l_k \geq 0$ there is an α_1 such that $\alpha \stackrel{\epsilon}{\Longrightarrow}_0 \alpha_1 \stackrel{a}{\longrightarrow}_k \beta_{ij}' A_1^{l_1} \ldots A_k^{l_k}$. The argument is now repeated, so that there is a finite set of preliminary extended forms which characterise the set $\{\alpha_2 : \alpha \stackrel{\epsilon}{\Longrightarrow}_0 \alpha_1 \stackrel{a}{\longrightarrow}_k \alpha_1' \stackrel{\epsilon}{\Longrightarrow}_2\}$, and so on. Because there can be only finitely many different

[1] The size of a configuration δ, $|\delta|$, is its number of occurrences of atoms.

indices involved in a decomposition of the transition $\stackrel{a}{\Longrightarrow}_n$ it follows that there is a finite set of preliminary extended forms which characterise $\{\delta : \alpha \stackrel{a}{\Longrightarrow}_n \delta\}$. This finite set is then tidied into a set of extended forms, $\mathsf{E}(\alpha, a, n)$, as described earlier. □

Theorem 1 offers a finite symbolic characterisation of the infinite branching of normed BP processes. Moreover its proof shows how a finite set of extended configurations $\mathsf{E}(\alpha, a, n)$ which characterises $\{\delta : \alpha \stackrel{a}{\Longrightarrow}_n \delta\}$ is computed.

Example 1 Consider $\mathsf{E}(U, c, 0)$ where U is from Example 2 of the previous section. There is only one decomposition of the transition $\stackrel{c}{\Longrightarrow}_0$ in this example, $U \stackrel{\epsilon}{\Longrightarrow}_0 \alpha_1 \stackrel{c}{\longrightarrow}_0 \alpha_1' \stackrel{\epsilon}{\Longrightarrow}_0 \delta$. First consider the preliminary extended configurations for $\{\alpha_1 : U \stackrel{\epsilon}{\Longrightarrow}_0 \alpha_1\}$. This consists of the singleton set $\{UD^*\}$. Next we examine the $\stackrel{c}{\longrightarrow}_0$ transitions, and there are two possibilities $U \stackrel{c}{\longrightarrow}_0 U$ and $U \stackrel{c}{\longrightarrow}_0 C$. Hence $\{UD^*, CD^*\}$ contains the preliminary extended configurations for $\{\alpha_1' : \alpha \stackrel{\epsilon}{\Longrightarrow}_0 \alpha_1 \stackrel{c}{\longrightarrow}_0 \alpha_1'\}$. This is the same set of preliminary extended configurations for $\{\delta\,;\, \alpha \stackrel{c}{\Longrightarrow}_0 \delta\}$. Now we tidy this set. Because $D \in G(U)$ the resulting set $\mathsf{E}(U, c, 0)$ is $\{U, CD^*\}$. We show that this set obeys the three conditions of Theorem 1. Suppose $U \stackrel{c}{\Longrightarrow}_0 \delta$. There are two cases. First $U \stackrel{c}{\Longrightarrow}_0 UD^n$ and because $D \in G(U)$ it follows that $U = \mathrm{nf}(UD^n)$ and $U \in \mathsf{E}(U, c, 0)$. Second is that $U \stackrel{c}{\Longrightarrow}_0 CD^n$ and $UD^* \in \mathsf{E}(U, c, 0)$. This establishes condition 1 of Theorem 1. For 2 note that $U \stackrel{c}{\Longrightarrow}_0 U$. And for condition 3, $U \stackrel{c}{\Longrightarrow}_0 CD^n$ for all $n \geq 0$. In contrast the set $\mathsf{E}(V, c, 0)$, where V is also from Example 2 of the previous section, is the singleton set $\{CD^*\}$. □

4 The Decidability Result

Given a normed BP process definition it is easy to find its redundant atoms, and to remove them. The sets $G(X)$ for each atom X is easily computable. Moreover for each configuration α, a and $n \leq \mathrm{K}$, one can compute a finite set of extended configurations $\mathsf{E}(\alpha, a, n)$ which characterises $\{\delta : \alpha \stackrel{a}{\Longrightarrow}_n \delta\}$, using Theorem 1 of the previous section. The main problem with deciding whether or not $\alpha \approx \beta$ is their infinite branching. The technique for overcoming this is to use the finite characterisation to show that we only need to examine boundedly many transitions of α and β. However we are only able to show this for a subset of normed BP processes which includes Examples 1 and 2 of section 2.

We restrict to the subset of normed BP process definitions which obey the following condition.

$$\text{If } G(X) \neq \emptyset \text{ and } X \stackrel{\tau}{\longrightarrow}_0 \alpha \text{ then } \alpha = X\alpha'$$

Effectively this imposes the constraint that generators are "pure", if X is a generator then any transition $X \stackrel{\tau}{\longrightarrow}_0 \alpha$ is a generating transition or is useless ($X \stackrel{\tau}{\longrightarrow}_0 X$). Both Examples 1 and 2 of section 2 obey this condition. The next result relies on this constraint.

Proposition 1

1. *If* $G(X) \neq \emptyset$ *and* $X\alpha \stackrel{\epsilon}{\Longrightarrow}_0 \beta$ *then* $X \in \beta$.
2. *For each* α *the set* $\{\mathrm{nf}(\delta) : \alpha \stackrel{\epsilon}{\Longrightarrow}_0 \delta\}$ *is finite.*

The restriction on BP processes does not imply that the sets $\{\delta : \alpha \stackrel{\epsilon}{\Longrightarrow}_0 \delta\}$ and $\{\mathrm{nf}(\delta) : \alpha \stackrel{a}{\Longrightarrow}_n \delta$ and $a \neq \epsilon$ or $n > 0\}$ are finite[2].

Assume that $\mathsf{E}(\alpha, a, n)$ is the finite set of extended configurations given by Theorem 1 of the previous section, which characterises $\{\delta : \alpha \stackrel{a}{\Longrightarrow}_n \delta\}$. Elements of this set are either finite, of the form β, or infinite, of the form $\beta A_1^* \ldots A_k^*$. It follows from the restriction on BP processes that if β is a finite element and $\beta \stackrel{\epsilon}{\Longrightarrow}_0 \beta'$ then $\mathrm{nf}(\beta')$ is also a finite element because if β contains a generator X then β' also contains X. Furthermore if $\beta \stackrel{\epsilon}{\Longrightarrow}_0 \beta'$ and $\mathrm{nf}(\beta') \neq \beta$ and $\beta' \stackrel{\epsilon}{\Longrightarrow} \beta''$ then $\mathrm{nf}(\beta'') \neq \beta$ (because otherwise β' would contain a redundant atom). Therefore the following holds because of the restriction on processes.

Fact 1 *If* β_0 *is a finite element of* $\mathsf{E}(\alpha, a, n)$ *whose size* $|\mathsf{E}(\alpha, a, n)| = m$ *and* $\beta_0 \stackrel{\epsilon}{\Longrightarrow}_0 \beta_1 \stackrel{\epsilon}{\Longrightarrow}_0 \ldots \stackrel{\epsilon}{\Longrightarrow}_0 \beta_m$ *then for some* $i < m$, $\mathrm{nf}(\beta_i) = \mathrm{nf}(\beta_{i+1})$.

There is a similar property in the case of infinite elements. Assume that $\delta = \beta A_1^{n_1} \ldots A_k^{n_k}$ is an instance of an infinite element $\beta A_1^* \ldots A_k^* \in \mathsf{E}(\alpha, a, n)$ and $\delta \stackrel{\epsilon}{\Longrightarrow}_0 \delta'$. Then $\delta' \approx \lambda A_1^{l_1} \ldots A_k^{l_k}$ where each $l_i \leq n_i$ and either $\mathrm{nf}(\lambda) = \beta$ or $\mathrm{nf}(\lambda) \neq \beta$ and $\beta \stackrel{\epsilon}{\Longrightarrow}_0 \lambda$. Moreover in the case that $\mathrm{nf}(\lambda) \neq \beta$ and $\delta' \stackrel{\epsilon}{\Longrightarrow}_0 \lambda' A_1^{l'_1} \ldots A_k^{l'_k}$ then $\mathrm{nf}(\lambda') \neq \beta$.

Fact 2 *If* $\delta_0 = \beta A_1^{n_1} \ldots A_k^{n_k}$ *and* $\beta A_1^* \ldots A_k^* \in \mathsf{E}(\alpha, a, n)$ *and* $|\mathsf{E}(\alpha, a, n)| = m$ *and* $\delta_0 \stackrel{\epsilon}{\Longrightarrow}_0 \delta_1 \stackrel{\epsilon}{\Longrightarrow}_0 \ldots \stackrel{\epsilon}{\Longrightarrow}_0 \delta_m$ *then for some* $i < m$, $\delta_i \approx \lambda_i A_1^{l_1} \ldots A_k^{l_k}$ *and* $\delta_{i+1} \approx \lambda'_i A_1^{l'_1} \ldots A_k^{l'_k}$ *and* $\mathrm{nf}(\lambda_i) = \mathrm{nf}(\lambda'_i)$ *and* $l'_i \leq l_i$.

Stratified weak bisimulation approximants, $\approx'_o$, are now defined as follows.

Definition 1 The relations $\approx'_o$ for ordinals o are defined inductively as follows, where we assume that l is a limit ordinal (such as ω).

$\alpha \approx'_0 \beta$
$\alpha \approx'_{o+1} \beta$ iff
  if $\alpha \stackrel{\epsilon}{\Longrightarrow}_0 \alpha'$ then $\exists \beta'.\ \beta \stackrel{\epsilon}{\Longrightarrow}_0 \beta'$ and $\alpha' \approx'_{o+1} \beta'$
  if $\beta \stackrel{\epsilon}{\Longrightarrow}_0 \beta'$ then $\exists \alpha'.\ \alpha \stackrel{\epsilon}{\Longrightarrow}_0 \alpha'$ and $\alpha' \approx'_{o+1} \beta'$
and for ($a \in \mathsf{A}$ and $n \leq K$) or ($a = \epsilon$ and $1 \leq n \leq$ K)
  if $\alpha \stackrel{a}{\Longrightarrow}_n \alpha'$ then $\exists \beta'.\ \beta \stackrel{a}{\Longrightarrow}_n \beta'$ and $\alpha' \approx'_o \beta'$
  if $\beta \stackrel{a}{\Longrightarrow}_n \beta'$ then $\exists \alpha'.\ \alpha \stackrel{a}{\Longrightarrow}_n \alpha'$ and $\alpha' \approx'_o \beta'$
$\alpha \approx'_l \beta$ iff $\forall o < l.\ \alpha \approx'_o \beta$

[2] It is possible to define hierarchies of BP processes according to finiteness of these sets. At the lowest level are the totally normed processes, where for any α, a and n the set $\{\delta : \alpha \stackrel{a}{\Longrightarrow}_n \delta\}$ is finite.

The definition of $\approx'_o$ is unusual because we distinguish between $\stackrel{\epsilon}{\Longrightarrow}_0$ transitions and the remaining transitions $\stackrel{a}{\Longrightarrow}_n$ for reasons that will become clearer in the decision procedure. Because of Proposition 1 the definition is well-defined, as there can only be finitely many "different" elements in the set $\{\delta : \alpha \stackrel{\epsilon}{\Longrightarrow}_0 \delta\}$. For each ordinal o, the relation $\approx'_o$ is an equivalence relation. However for particular ordinals o the relation $\approx'_o$ may differ from $\approx_o$ as defined in section 2.

Proposition 2

1. $\alpha \approx \beta$ *iff for all ordinals* o. $\alpha \approx'_o \beta$.
2. *For all ordinals* o, $\alpha \approx'_o \mathrm{nf}(\alpha)$.
3. *If* $\alpha \approx'_o \beta$ *then* $\alpha\delta \approx'_o \beta\delta$.

The procedure for deciding whether $\alpha \approx \beta$ is given by a tableau proof system, which is goal directed. One starts with the initial goal $\mathrm{nf}(\alpha) = \mathrm{nf}(\beta)$, to be understood as "is $\alpha \approx \beta$?", and then one reduces it to subgoals using a small number of rules. Goal reduction continues until one reaches either obviously true goals (such as $\delta = \delta$) or obviously false subgoals (such as $\gamma = \delta$ and one of these processes has an $\stackrel{a}{\Longrightarrow}_n$ transition which the other does not have). Such a procedure was used for deciding strong bisimilarity between arbitrary BP processes [4], and we will make use of this decidability proof.

Goals have the form $\alpha = \gamma$ where $\alpha = \mathrm{nf}(\alpha)$ and $\gamma = \mathrm{nf}(\gamma)$. There are four reduction rules, reducing goals to subgoals. Let $a_n(\delta) = \{\mathrm{nf}(\delta') : \delta \stackrel{a}{\Longrightarrow}_n \delta'\}$ where $n \leq \mathrm{K}$. The first tableau rule is for $\stackrel{\epsilon}{\Longrightarrow}_0$ transitions, and by Proposition 1.2 the set $\epsilon_0(\delta)$ is a finite set.

$$\frac{\alpha = \gamma}{\alpha_1 = \gamma_1 \quad \ldots \quad \alpha_l = \gamma_l}\ \mathrm{C}$$

where C is the following condition

$$(\forall \alpha' \in \epsilon_0(\alpha).\, \exists i.\ \alpha' = \alpha_i) \wedge (\forall i.\ \alpha_i \in \epsilon_0(\alpha)) \quad \wedge$$
$$(\forall \gamma' \in \epsilon_0(\gamma).\, \exists i.\ \gamma' = \gamma_i) \wedge (\forall i.\ \gamma_i \in \epsilon_0(\gamma))$$

The rule is sound, if all the subgoals are true then so is the goal. A finer account shows soundness with respect to approximants, if $\alpha_i \approx'_o \gamma_i$ for all subgoals then $\alpha \approx'_o \gamma$. The rule is also complete in the sense that if the goal $\alpha = \gamma$ is true then there is an application of the rule such that all subgoals $\alpha_i = \gamma_i$ are also true.

Next we want a similar rule which covers the remaining transitions $\stackrel{a}{\Longrightarrow}_n$, when $n \leq \mathrm{K}$ and either $a \neq \epsilon$ or $n > 0$. A similar rule would reduce a goal $\alpha = \gamma$ to only a finite set of subgoals. However a set $a_n(\delta)$ may be infinite. Therefore we need a mechanism which shows that we only need to consider bounded finite subsets of elements of $a_n(\alpha)$ and $a_n(\gamma)$ for each $\stackrel{a}{\Longrightarrow}_n$. Lemmas 1 and 2 below establish this. These results are quite involved, and so we delay their proofs until the next section where they are presented in full. They constitute the heart of the decidability result.

Although the set $a_n(\delta)$ may be infinite, the set $\mathsf{E}(\delta, a, n)$ is not only finite but also bounded. We show that we need only examine "small" elements of $a_n(\delta)$.

The first lemma covers the case when the goal $\alpha = \gamma$ is not true. Without loss of generality assume $\alpha \not\approx_{o+1} \gamma$ and $\alpha \stackrel{a}{\Longrightarrow}_n \alpha'$ and for all γ' such that $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$, $\alpha' \not\approx_o \gamma'$. Lemma 1, the bounded inequivalence lemma, shows that there is a small $\alpha' \in a_n(\alpha)$ with this property. A small element of $a_n(\alpha)$ is either a finite element of $\mathsf{E}(\alpha, a, n)$ or is an element $\beta A_1^{l_1} \ldots A_k^{l_k}$ where each $l_i \leq |\mathsf{E}(\gamma, a, n)| + 1$ and $\beta A_1^* \ldots A_k^* \in \mathsf{E}(\alpha, a, n)$.

Lemma 1 *Let $\alpha \not\approx_{o+1} \gamma$ and assume that $\alpha \stackrel{a}{\Longrightarrow}_n \alpha'$ and for all γ' such that $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$, $\alpha' \not\approx_o \gamma'$. Then either*

1. *$\mathrm{nf}(\alpha')$ is a finite element in $\mathsf{E}(\alpha, a, n)$ or*
2. *$\alpha' = \delta A_1^{n_1} \ldots A_k^{n_k}$ and $\beta A_1^* \ldots A_k^* \in \mathsf{E}(\alpha, a, n)$ and $\mathrm{nf}(\delta) = \beta$ and there exists $l_1, \ldots, l_k$ such that each $l_i \leq |\mathsf{E}(\gamma, a, n)| + 1$ and for all γ' such that $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$, $\beta A_1^{l_1} \ldots A_k^{l_k} \not\approx_o \gamma'$.*

Lemma 2, the bounded equivalence lemma, covers the case when $\alpha = \gamma$ is true. It shows that small elements of $a_n(\alpha)$ can be matched with elements of $a_n(\gamma)$ which have bounded size.

Lemma 2 *Assume $\alpha \approx \gamma$. Then for each a and n*

1. *if $\alpha \stackrel{a}{\Longrightarrow}_n \alpha'$ and $\mathrm{nf}(\alpha') \in \mathsf{E}(\alpha, a, n)$ then*
   a) *either $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$ and $\mathrm{nf}(\gamma') \in \mathsf{E}(\gamma, a, n)$ and $\mathrm{nf}(\alpha') \approx \mathrm{nf}(\gamma')$*
   b) *or $\gamma \stackrel{a}{\Longrightarrow}_n \lambda B_1^{s_1} \ldots B_m^{s_m}$ and $\lambda' B_1^* \ldots B_m^* \in \mathsf{E}(\gamma, a, n)$ and $\mathrm{nf}(\lambda) = \lambda'$ and each $s_i \leq |\mathsf{E}(\alpha, a, n)|$ and $\mathrm{nf}(\alpha') \approx \lambda' B_1^{s_1} \ldots B_m^{s_m}$,*
2. *if $\alpha \stackrel{a}{\Longrightarrow}_n \delta A_1^{n_1} \ldots A_k^{n_k}$ and $\beta A_1^* \ldots A_k^* \in \mathsf{E}(\alpha, a, n)$ and $\beta = \mathrm{nf}(\delta)$ and each $n_i \leq |\mathsf{E}(\gamma, a, n)| + 1$ then*
   a) *either $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$ and $\mathrm{nf}(\gamma') \in \mathsf{E}(\gamma, a, n)$ and $\beta A_1^{n_1} \ldots A_k^{n_k} \approx \mathrm{nf}(\gamma')$*
   b) *or $\gamma \stackrel{a}{\Longrightarrow}_n \lambda B_1^{s_1} \ldots B_m^{s_m}$ and $\lambda' B_1^* \ldots B_m^* \in \mathsf{E}(\gamma, a, n)$ and $\mathrm{nf}(\lambda) = \lambda'$ and each $s_i \leq \sum n_i + |\mathsf{E}(\alpha, a, n)|$ and $\beta A_1^{n_1} \ldots A_k^{n_k} \approx \lambda' B_1^{s_1} \ldots B_m^{s_m}$.*

Consequently the second tableau proof rule has a similar form to the $\stackrel{\epsilon}{\Longrightarrow}_0$ transition rule except that the condition C is that for all a_n, $n \leq \mathrm{K}$ and $a \neq \epsilon$ or $n > 0$,

$\forall \,\mathrm{small}\, \alpha' \in a_n(\alpha). \exists i. \alpha = \alpha_i \ \wedge \ \forall i. \exists a_n. \alpha_i \in a_n(\alpha)$ and α_i has bounded size $\wedge$
$\forall \,\mathrm{small}\, \gamma' \in a_n(\gamma). \exists i. \gamma' = \gamma_i \ \wedge \ \forall i. \exists a_n. \gamma_i \in a_n(\gamma)$ and γ_i has bounded size

where the precise notion of bounded size is given in Lemma 2.2 part (b). Lemma 1 guarantees that the rule is sound, if the goal $\alpha = \gamma$ is false and so $\alpha \not\approx_{o+1} \gamma$ then for at least one of the subgoals $\alpha_i = \gamma_i$ it is the case that $\alpha_i \not\approx_o \gamma_i$ (where the approximant index decreases). Lemma 2 justifies completeness, if the goal $\alpha = \gamma$ is true then there is an application of the rule such that all subgoals $\alpha_i = \gamma_i$ are also true.

The final two rules SUB(L) and SUB(R) are taken from the tableau decision procedure for strong bisimulation equivalence for arbitrary BP processes [4]. We assume a fixed linear ordering $<$ on the atoms Γ which is defined so that if atom $Y \in G(X)$ and $Y \neq X$ then $X < Y$. The ordering $<$ is extended to

configurations, as the lexicographical ordering. Assume $X_1 < X_2 < \ldots < X_n$. Consequently

$$X_1^{k_1} \ldots X_n^{k_n} < X_1^{l_1} \ldots X_n^{l_n} \text{ iff there is an } i \geq 1 \text{ such that } k_i < l_i \text{ and for all } j < i.\ k_j = l_j.$$

Clearly, $\mathrm{nf}(\alpha) \leq \alpha$ and if $\alpha < \gamma$ then $\mathrm{nf}(\alpha\beta) < \mathrm{nf}(\gamma\beta)$.

The SUB rules are given below, where SUB(L) is the left rule and SUB(R) is the right rule.

$$\frac{\begin{array}{l}\alpha = \gamma \\ \vdots \quad \gamma < \alpha \text{ and at least one} \\ \vdots \quad \text{application of } \stackrel{a}{\Longrightarrow}_n \\ \alpha\delta = \lambda\end{array}}{\mathrm{nf}(\gamma\delta) = \lambda} \qquad \frac{\begin{array}{l}\gamma = \alpha \\ \vdots \quad \gamma < \alpha \text{ and at least one} \\ \vdots \quad \text{application of } \stackrel{a}{\Longrightarrow}_n \\ \lambda = \alpha\delta\end{array}}{\lambda = \mathrm{nf}(\gamma\delta)}$$

We explain the rule SUB(L). If the current goal is $\alpha\delta = \lambda$ and in the proof tree above the goal on the path to the root there is the goal $\alpha = \gamma$ and $\gamma < \alpha$ and there is at least one application of the rule $\stackrel{a}{\Longrightarrow}_n$ where $a \neq \epsilon$ or $n > 0$ along this path then we reduce the goal to the subgoal $\mathrm{nf}(\gamma\delta) = \lambda$. Note that this has the effect of reducing the size of the left configuration, as $\mathrm{nf}(\gamma\delta) < \alpha\delta$. The SUB rules are sound and complete.

Fact 3

1. *If* $\alpha\delta \not\approx'_o \lambda$ *and* $\alpha \approx'_o \gamma$ *then* $\mathrm{nf}(\gamma\delta) \not\approx'_o \lambda$
2. *If* $\alpha \approx \gamma$ *and* $\alpha\delta \approx \lambda$ *then* $\mathrm{nf}(\gamma\delta) \approx \lambda$

One builds a proof tree starting from an initial goal $\alpha = \gamma$ and repeatedly applying the tableau proof rules as follows. First one applies the $\stackrel{\epsilon}{\Longrightarrow}_0$ rule and then one applies the $\stackrel{a}{\Longrightarrow}_n$ rule to all the resulting subgoals. Call this a simple block. And then one repeatedly applies the SUB rules to the subgoals of a block, until they no longer apply. At which point the whole process is repeated. One applies the $\stackrel{\epsilon}{\Longrightarrow}_0$ rule to all subgoals and so on.

There is also the important notion of when a goal is a final goal. Final goals are either successful or unsuccessful. A successful final goal has the form $\alpha = \alpha$ and an unsuccessful final goal has the form $\delta = \lambda$ and for some a and n either $a_n(\delta) = \emptyset$ and $a_n(\lambda) \neq \emptyset$ or $a_n(\delta) \neq \emptyset$ and $a_n(\lambda) = \emptyset$. Clearly a successful final goal is true and an unsuccessful final goal is false.

A successful tableau proof for $\alpha = \gamma$ is a finite proof tree whose root is $\alpha = \gamma$ and all of whose inner subgoals are the result of an application of one of the rules, and all of whose leaves are successful final goals. The following results establish decidability of $\approx$ between restricted BP processes. The proofs of these results are minor variants of proofs in [3,4].

Theorem 1 *Every tableau for* $\alpha = \gamma$ *is finite and there is only a finite number of tableau for* $\alpha = \gamma$.

Theorem 2 *$\alpha \approx \gamma$ iff there is a successful tableau for $\alpha = \gamma$.*

Therefore the main result follows that $\approx$ is decidable between restricted BP processes. We now examine how this result shows equivalence and inequivalence for the examples from section 2.

Example 1 We show that $Y \approx Z$ where these atoms are from Example 1 of section 2. Assume $Y < Z$. We build a tableau with root $Y = Z$. First we apply the $\stackrel{\epsilon}{\Longrightarrow}$ rule which results in the same goal $Y = Z$. Then we apply the $\stackrel{a}{\Longrightarrow}_n$ rule. In this example K = 1 and so we need to consider the four transitions $\stackrel{\epsilon}{\Longrightarrow}_1$, $\stackrel{a}{\Longrightarrow}_{-1}$, $\stackrel{a}{\Longrightarrow}_0$ and $\stackrel{a}{\Longrightarrow}_1$.

$$\epsilon_0(Y) = \{YA\} \quad a_{-1}(Y) = \{A\} \quad a_0(Y) = \{A, Y\} \quad a_1(Y) = \{YA, AA\}$$
$$\epsilon_0(Z) = \{ZA\} \quad a_{-1}(Z) = \{A\} \quad a_0(Z) = \{A, Z\} \quad a_1(Z) = \{ZA, AA\}$$

So the goal $Y = Z$ reduces to the following subgoals, (1)$YA = ZA$, (2)$A = A$, (3)$Y = Z$ and (4)$AA = AA$. Goals (2) and (4) are successful leaves. In the cases of (1) and (3) because $Y < Z$ and $Y = Z$ appears on their paths to the root, and in both cases there is at least one application of the rule $\stackrel{a}{\Longrightarrow}_n$, we can apply SUB(R) to yield (1′) $YA = YA$ and (3′) $Y = Y$, which completes the successful tableau. □

Example 2 We show that $U \not\approx V$ where U, V are from Example 2 of section 2 even though $U \approx'_n V$ for all $n \geq 0$. There are only finitely many tableaux for this goal and all are unsuccessful. Assume that $U < V$. The starting goal is $U = V$. First we apply the $\stackrel{\epsilon}{\Longrightarrow}_0$ rule, which yields the same goal $U = V$. For this example K = 0. Next we apply the $\stackrel{a}{\Longrightarrow}_n$ rule and there are two possible transitions $\stackrel{c}{\Longrightarrow}_0$ and $\stackrel{d}{\Longrightarrow}_0$. We only examine the first of these. $\mathsf{E}(U, c, 0) = \{U, CD^*\}$ and $\mathsf{E}(V, c, 0) = \{CD^*\}$. The small elements of $c_0(U) = \{U, C, CD, CD^2\}$ and the small elements of $c_0(V) = \{C, CD, CD^2, CD^3\}$. Therefore we must find a matching of small elements with bounded elements. The easy subgoals are $C = C$, $CD = CD$, $CD^2 = CD^2$ and $CD^3 = CD^3$. The problem case is a match for the small element $U \in c_0(U)$. By Lemma 2 because U is a finite element of $\mathsf{E}(U, c, 0)$ the matching element must be CD^s where $s \leq 2$. Thus we must have one of the following subgoals (1)$U = C$, (2)$U = CD$, (3)$U = CD^2$. Assume it is (3). We apply the $\stackrel{\epsilon}{\Longrightarrow}_0$ rule to this goal, $\epsilon_0(CD^2) = \{C, CD, CD^2\}$ and $\epsilon_0(U) = \{U\}$. So in fact we must have all the subgoals (1), (2) and (3). But now we have an unsuccessful leaf (1), because $d_0(U) = \{U\}$ and $d_0(C) = \emptyset$. □

5 Proofs of the Main Lemmas

In this section we prove the two boundedness lemmas of the previous section.

Proof of Lemma 1: Assume $\alpha \stackrel{a}{\Longrightarrow}_n \alpha'$ and $\alpha' \not\approx^{\mu}_{\delta} \gamma'$ for all γ' such that $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$. Consider the sets $\mathsf{E}(\alpha, a, n)$ and $\mathsf{E}(\gamma, a, n)$. By Theorem 1 of section 3 either $\mathrm{nf}(\alpha') \in \mathsf{E}(\alpha, a, n)$ or $\alpha' = \delta A_1^{n_1} \ldots A_k^{n_k}$ and $\beta A_1^* \ldots A_k^* \in \mathsf{E}(\alpha, a, n)$ and $\mathrm{nf}(\delta) = \beta$. If the first holds then the result is proved. Assume therefore that

$\beta A_1^{n_1} \ldots A_k^{n_k} \not\approx_o \gamma'$ for all γ' such that $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$. Let $\beta_1' = \beta A_2^{n_2} \ldots A_k^{n_k}$, and consider the least m_1 such that $\beta_1' A_1^{m_1} \not\approx_o \gamma'$ for all γ' such that $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$. If $m_1 \leq |\mathsf{E}(\gamma, a, n)| + 1$ then the result is proved for $l_1 = m_1$. The argument is then repeated for other A_i. Let $\beta_2' = \beta A_1^{m_1} A_3^{n_3} \ldots A_k^{n_k}$ and consider the least m_2 such that $\beta_2' A_2^{m_2} \not\approx_o \gamma'$ for all γ' such that $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$. Therefore without loss of generality assume that $\beta_1' A_1^{m_1} \not\approx_o \gamma'$ for all γ' such that $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$ and $m_1 > |\mathsf{E}(\gamma, a, n)| + 1$. Hence there is a γ' such that $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$ and $\beta_1' A_1^{m_1 - 1} \approx_o' \gamma'$. By Theorem 1 of section 3 either $\mathrm{nf}(\gamma') \in \mathsf{E}(\gamma, a, n)$ or $\gamma' = \lambda B_1^{s_1} \ldots B_m^{s_m}$ and $\lambda' B_1^* \ldots B_m^* \in \mathsf{E}(\gamma, a, n)$ and $\mathrm{nf}(\lambda) = \lambda'$. Assume the first case, that $\mathrm{nf}(\gamma') = \gamma_0'' \in \mathsf{E}(\gamma, a, n)$. Hence $\beta_1' A_1^{m_1 - 1} \approx_o' \gamma_0''$. However $\beta_1' A_1^{m_1 - 1} \stackrel{\epsilon}{\Longrightarrow}_0 \beta_1' A_1^{m_1 - 2} \stackrel{\epsilon}{\Longrightarrow}_0 \ldots \stackrel{\epsilon}{\Longrightarrow}_0 \beta_1' A_1^0$. So therefore $\gamma_0'' \stackrel{\epsilon}{\Longrightarrow}_0 \gamma_1'' \stackrel{\epsilon}{\Longrightarrow}_0 \ldots \stackrel{\epsilon}{\Longrightarrow}_0 \gamma_{m_1 - 1}''$ and $\gamma_j'' \approx_o' \beta_1' A_1^{m_1 - (j+1)}$. By Fact 1 of the previous section there is an i, $\mathrm{nf}(\gamma_i'') = \mathrm{nf}(\gamma_{i+1}'')$ and therefore $\beta_1' A_1^{m_1 - (i+2)} \approx_o' \beta_1' A_1^{m_1 - (i+1)}$. Because $\approx_o'$ is a congruence it follows that $\beta_1' A_1^{m_1 - (i+2)} \approx_o' \beta_1' A_1^{m_1}$ which is a contradiction. Next we consider the other case, $\beta_1' A_1^{m_1 - 1} \approx_o' \gamma_0'$ and $\gamma_0' = \lambda B_1^{s_1} \ldots B_m^{s_m}$ and $\lambda' B_1^* \ldots B_1^* \in \mathsf{E}(\gamma, a, n)$ and $\mathrm{nf}(\lambda) = \lambda'$. The argument proceeds as above. $\beta_1' A_1^{m_1 - 1} \stackrel{\epsilon}{\Longrightarrow}_0 \beta_1' A_1^{m_1 - 2} \stackrel{\epsilon}{\Longrightarrow}_0 \ldots \stackrel{\epsilon}{\Longrightarrow}_0 \beta_1' A_1^0$ Therefore $\gamma_0' \stackrel{\epsilon}{\Longrightarrow}_0 \gamma_1' \stackrel{\epsilon}{\Longrightarrow}_0 \ldots \stackrel{\epsilon}{\Longrightarrow}_0 \gamma_{m_1 - 1}'$ and $\gamma_j' \approx_o' \beta_1' A_1^{m_1 - (j+1)}$. By Fact 2 of the previous section for some i, $\gamma_i' \approx \lambda_i B_1^{t_1} \ldots B_m^{t_m}$ and $\gamma_{i+1}' \approx \lambda_i' B_1^{t_1'} \ldots B_m^{t_m'}$ and $t_i' \leq t_i$ and $\mathrm{nf}(\lambda_i) = \mathrm{nf}(\lambda_i')$. $\gamma_{i+1}' \approx_o' \beta_1' A_1^{m_1 - (i+2)}$ and $\gamma_i' \approx_o' \beta_1' A_1^{m_1 - (i+1)}$. Let $\eta = B_1^{t_1 - t_1'} \ldots B_m^{t_m - t_m'}$. By congruence, $\gamma_{i+1}' \eta \approx_o' \beta_1' A_1^{m_1 - (i+2)} \eta \approx_o' \beta_1' A_1^{m_1 - (i+1)}$. Therefore by congruence $\gamma_{i+1}' \eta^{i+2} \approx_o' \beta_1' A_1^{m_1}$ which is a contradiction. □

Proof of Lemma 2: Assume $\alpha \approx \gamma$. First also assume that $\alpha \stackrel{a}{\Longrightarrow}_n \alpha'$ and $\mathrm{nf}(\alpha') \in \mathsf{E}(\alpha, a, n)$. Hence $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$ and $\mathrm{nf}(\alpha') \approx \gamma'$. By Theorem 1 of section 3 either $\mathrm{nf}(\gamma') \in \mathsf{E}(\gamma, a, n)$ and $\mathrm{nf}(\alpha') = \mathrm{nf}(\gamma')$, or $\gamma' = \lambda B_1^{s_1} \ldots B_m^{s_m}$ and $\lambda' B_1^* \ldots B_m^* \in \mathsf{E}(\gamma, a, n)$ and $\mathrm{nf}(\lambda) = \lambda'$ and $\mathrm{nf}(\alpha') = \lambda' B_1^{s_1} \ldots B_m^{s_m}$. We show that each s_i can be chosen so that $s_i \leq |\mathsf{E}(\alpha, a, n)|$. The strategy for proving this is similar to the proof of Lemma 1 above. Let $\lambda'' = \lambda' B_2^{s_2} \ldots B_m^{s_m}$ and let m_1 be the smallest index such that $\mathrm{nf}(\alpha') \approx \lambda'' B_1^{m_1}$. If $m_1 \leq |\mathsf{E}(\alpha, a, n)|$ then we let $s_1 = m_1$ and repeat the argument for the other indices s_i. Therefore assume that $m_1 > |\mathsf{E}(\alpha, a, n)|$. However $\lambda'' B_1^{m_1} \stackrel{\epsilon}{\Longrightarrow}_0 \lambda'' B_1^{m_1 - 1} \stackrel{\epsilon}{\Longrightarrow}_0 \ldots \stackrel{\epsilon}{\Longrightarrow}_0 \lambda'' B_1^0$. Therefore assuming $\alpha_0 = \mathrm{nf}(\alpha')$, $\alpha_0 \stackrel{\epsilon}{\Longrightarrow}_0 \alpha_1 \stackrel{\epsilon}{\Longrightarrow}_0 \ldots \stackrel{\epsilon}{\Longrightarrow}_0 \alpha_{m_1}$ and $\alpha_j \approx \lambda'' B_1^{m_1 - j}$. By Fact 1 of the previous section for some i, $\mathrm{nf}(\alpha_i) = \mathrm{nf}(\alpha_{i+1})$ and so $\lambda'' B_1^{m_1 - i} \approx \lambda'' B_1^{m_1 - (i+1)}$. Therefore by congruence, $\lambda'' B_1^{m_1 - (i+1)} \approx \lambda'' B_1^{m_1}$ which is a contradiction. Next assume that $\alpha \stackrel{a}{\Longrightarrow}_n \delta A_1^{n_1} \ldots A_k^{n_k}$ and $\beta A_1^* \ldots A_k^* \in \mathsf{E}(\alpha, a, n)$ and $\beta = \mathrm{nf}(\delta)$ and each $n_i \leq |\mathsf{E}(\alpha, a, n)| + 1$. Because $\alpha \approx \gamma$ it follows that $\gamma \stackrel{a}{\Longrightarrow}_n \gamma'$ and $\beta A_1^{n_1} \ldots A_k^{n_k} \approx \gamma'$. By Theorem 1 of section 3 either $\mathrm{nf}(\gamma') \in \mathsf{E}(\gamma, a, n)$ and therefore $\beta A_1^{n_1} \ldots A_k^{n_k} \approx \mathrm{nf}(\gamma')$, or $\gamma' = \lambda B_1^{s_1} \ldots B_m^{s_m}$ and $\lambda' B_1^* \ldots B_m^* \in \mathsf{E}(\gamma, a, n)$ and $\mathrm{nf}(\lambda) = \lambda'$ and $\beta A_1^{n_1} \ldots A_k^{n_k} \approx \lambda' B_1^{s_1} \ldots B_m^{s_m}$. We show that each s_i can be chosen so that $s_i \leq \sum n_i + |\mathsf{E}(\gamma, a, n)|$. Let $\lambda'' = \lambda' B_2^{s_2} \ldots B_m^{s_m}$ and let m_1 be the smallest index such that $\beta A_1^{n_1} \ldots A_k^{n_k} \approx \lambda'' B_1^{m_1}$. If $m_1 \leq \sum n_i + |\mathsf{E}(\gamma, a, n)|$ then we let

$s_1 = m_1$ and repeat the argument for the other indices. Therefore assume that $m_1 > \sum n_i + |\mathsf{E}(\gamma, a, n)|$. However $\lambda'' B_1^{m_1} \stackrel{\epsilon}{\Longrightarrow}_0 \lambda'' B_1^{m_1-1} \stackrel{\epsilon}{\Longrightarrow}_0 \ldots \stackrel{\epsilon}{\Longrightarrow}_0 \lambda'' B_1^0$. Therefore let $\eta_0 = \beta A_1^{n_1} \ldots A_k^{n_k}$, and so $\eta_0 \stackrel{\epsilon}{\Longrightarrow}_0 \eta_1 \stackrel{\epsilon}{\Longrightarrow}_0 \ldots \stackrel{\epsilon}{\Longrightarrow}_0 \eta_{m_1}$ and $\eta_j \approx \lambda'' B_1^{m_1-j}$. Because $m_1 > \sum n_i + |\mathsf{E}(\gamma, a, n)|$ it follows via fact 2 of the previous section that for some i, $\eta_i \approx \eta_{i+1}$ and so $\lambda'' B_1^{m_1-(i+1)} \approx \lambda'' B_1^{m_1-i}$ and therefore by congruence $\lambda'' B_1^{m_1} \approx \lambda'' B_1^{m_1-(i+1)}$ which contradicts that m_1 is a least index. □

References

1. Baeten, J., Bergstra, J. and Klop, J. (1993). Decidability of bisimulation equivalence for processes generating context-free languages. *Journal of Association for computing Machinery*, **40**, 653-682.
2. Caucal, D. (1996). On infinite transition graphs having a decidable monadic theory. *Lecture Notes in Computer Science*, **1099**, 194-205.
3. Christensen, S. (1993). *Decidability and Decomposition in Process Algebras*. Ph.D thesis University of Edinburgh, Tech Report ECS-LFCS-93-278.
4. Christensen, S., Hirshfeld, Y. and Moller, F. (1993). Bisimulation equivalence is decidable for all basic parallel processes. *Lecture Notes in Computer Science*, **715**, 143-157.
5. Christensen, S., Hüttel, H., and Stirling, C. (1995). Bisimulation equivalence is decidable for all context-free processes. *Information and Computation*, **121**, 143-148.
6. Esparza, J. (1997). Petri nets, commutative context-free grammars, and basic parallel processes. *Fundamenta Informaticae*, **30**, 23-41.
7. Hirshfeld, Y. (1996). Bisimulation trees and the decidability of weak bisimilarity. *Electronic Notes in Theoretical Computer Science*, **5**.
8. Hirshfeld, Y., Jerrum, M. and Moller, F. (1996). A polynomial-time algorithm for deciding equivalence of normed basic parallel processes. *Journal of Mathematical Structures in Computer Science*, **6**, 251-259.
9. Hüttel, H. (1991). Silence is golden: branching bisimilarity is decidable for context-free processes. *Lecture Notes in Computer Science*, **575**, 2-12.
10. Hüttel, H., and Stirling, C. (1991). Actions speak louder than words: proving bisimilarity for context free processes. *Proceedings 6th Annual Symposium on Logic in Computer Science*, IEEE Computer Science Press, 376-386.
11. Jančar, P. (1995). Undecidability of bisimilarity for Petri nets and some related problems. *Theoretical Computer Science*, **148**, 281-301.
12. Jančar, P., Kučera, A. and Mayr, R. (1998). Deciding bisimulation-like equivalences with finite-state processes. *Lecture Notes in Computer Science*, **1443**, 200-211.
13. Milner, R. (1989) *Communication and Concurrency*. Prentice-Hall.
14. Sénizergues, G. (1998). Decidability of bisimulation equivalence for equational graphs of finite out-degree. *Procs. IEEE FOCS'98*, 120-129.
15. Stříbrná, J. (1999). *Decidability and complexity of equivalences for simple process algebras*. Ph.D thesis, University of Edinburgh, Tech. Report ECS-LFCS-99-408.
16. Stirling, C. (1998). Decidability of bisimulation equivalence for normed pushdown processes. *Theoretical Computer Science*, **195**, 113-131.

An Axiomatic Semantics for the Synchronous Language *Gentzen* ⋆

Simone Tini

Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125, Pisa, Italy

Abstract. We propose an axiomatic semantics for the synchronous language *Gentzen*, which is an instantiation of the paradigm *Timed Concurrent Constraint Programming*. We view Gentzen as a prototype of the class of state-oriented synchronous languages, since it offers the basic constructs that are shared by the languages in the class. Since synchronous concurrency cannot be simulated by arbitrary interleaving, we cannot exploit "head normal forms", on which axiomatic theories for asynchronous process calculi are based.

1 Introduction

Synchronous languages [4,11] have been proposed for programming *reactive systems* [14], i.e. systems which maintain an ongoing interaction with their environment at a rate controlled by it. These languages are based on the *synchronous hypothesis* [8], which states that a reactive system is able to react instantaneously and in no time to stimuli from the external environment, so that outputs from the system are available as soon as inputs from the environment are. The synchronous hypothesis is indeed an abstraction and amounts to requiring that the system is able to react to an input before the subsequent input arrives.

One of the main features of the synchronous hypothesis is that "it reconciles concurrency with determinism" [8], in the sense that system components running in parallel are perfectly synchronized and cannot arbitrarily interleave, because each of them reacts whenever an input from the environment arrives.

In [8,12,16,21,3] several state-oriented synchronous languages have been proposed for programming reactive systems where control handling aspects are predominant. These languages have been endowed with operational semantics [3, 7,13,16,22] and with denotational semantics [10,20,22,23]. We believe that it is worth developing also axiomatic semantics for such languages, in order to characterize behavioral equivalent programs. Possible applications of such semantics are, among others, transformation of programs and proof by rewriting.

Axiomatic theories are well established in the field of asynchronous process description languages since the eighties [6,17], and, for languages in suitable classes, algorithms have been developed to obtain axiomatizations in a syntax-driven way (as examples, see [1,2]). A central idea in the mentioned papers is

⋆ Research partially supported by ESPRIT Working Group COTIC, Project Number 23677, and by MURST Progetto Cofinanziato TOSCA.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 394–408, 2001.

that concurrency can be simulated by interleaving: concurrent processes can be reduced to *head normal forms*, i.e. nondeterministic choices of sequential processes.

Since in the synchronous setting processes running in parallel are synchronized and run at the same rate, concurrency cannot be simulated by interleaving and head normal forms for synchronous languages have no meaning. So, developing axiomatic theories for synchronous languages cannot be done by trivially adapting what has been already done in the field of asynchronous languages.

In this paper we develop an axiomatic theory for the language *Gentzen*, which is an instantiation of the paradigm *Timed Concurrent Constraint Programming* [22]. We view Gentzen as a prototype of the class of state-oriented synchronous languages, because it offers the basic constructs that are shared by all languages in this class and that are not offered by asynchronous languages. Axiomatic theories for any other state-oriented synchronous language can be obtained by extending our work to deal with constructs that characterize such a language.

1.1 Gentzen: A Prototype of State-Oriented Synchronous Languages

A Gentzen program maintains an ongoing interaction with its environment by communicating through *binary signals*. No distinction between input and output signals is done: a set of signals Π is assumed and signals in Π can be sensed and broadcast both by the program and by the environment. Gentzen assumes a discrete notion of time, i.e. time is a sequence of instants. At every instant the external environment stimulates the program by broadcasting a set of signals; the program reacts immediately and broadcasts a set of signals to the environment.

We assume the syntax for Gentzen given in [21]:

$$A ::= \texttt{skip} \mid \texttt{tell}\ a \mid \texttt{if}\ a\ \texttt{then}\ A \mid \texttt{if}\ \overline{a}\ \texttt{then}\ A \mid \texttt{next}\ A \mid A \parallel A \mid \texttt{rec}\, P.\, A \mid P$$

where a, A and P range over binary signals, *agents* and *recursion variables*, respectively. Notice that we do not use the same notation of [21]. We use `rec` instead of "μ", and we write `if` $\overline{a}$ `then` A for `if` a `else` A.

We will denote by $\equiv$ the syntactic identity over agents.

Agent `skip` does nothing at each instant: it reacts to every input by responding with the empty output. Agent `tell` a broadcasts the signal a and then terminates, i.e. it behaves as `skip`. Agent `if` a `then` A behaves as A if the environment broadcasts a, whereas `if` $\overline{a}$ `then` A behaves as A if the environment does not broadcast a. Note that if a is present (resp. absent) then `if` a `then` A (resp. `if` $\overline{a}$ `then` A) starts A immediately. Agent `next` A will start A at the next instant. Agent $A_1 \parallel A_2$ is the synchronous parallel composition of A_1 and A_2. Signals broadcast by A_1 (resp. A_2) can be detected immediately by A_2 (resp. A_1). Construct `rec` is the recursion construct. To be sure that at each instant the computation is finite, it is required that recursion be *guarded*, i.e. that recursion variables appear in bodies of `next` (see [22] for further discussion).

According to the synchronous hypothesis, Gentzen constructs take no time, except `next` which takes precisely one unit of time. So, Gentzen offers the constructs that are shared by all synchronous languages: a mechanism to detect the instantaneous presence/absence of signals (`if _ then _`), a mechanism to broadcast instantaneously signals (`tell _`), a mechanism to delay the computation for one instant (`next _`), synchronous parallel composition (with instantaneous communication between parallel components) and guarded recursion.

Despite its simplicity, Gentzen is sufficiently expressive to embed compositionally the state-oriented synchronous language Argos [16] and the dataflow synchronous language Lustre [9] (see [25] for the proof).

1.2 Technical Development of the Paper

Technically, we give a *structural operational semantics* [19] for Gentzen in terms of a *labeled transition system* (LTS), with LTS states corresponding to agents and LTS transitions corresponding to agent reactions, and then we consider the *bisimulation* on the LTS as a behavioral equivalence over Gentzen agents. Bisimulation on our LTS is a congruence and our LTS reflects the input/output behavior of agents (in the sense that the LTS reflects the operational semantics of Gentzen given in [22]). So, bisimilar agents are distinguished neither by any Gentzen context nor by the external environment. Therefore, we believe that bisimulation is a reasonable notion of behavioral equivalence.

To axiomatize Gentzen, we provide a system of axioms defining an equality relation over agents. We prove that this equality relation is *sound* and *complete* modulo bisimulation, in the sense that two arbitrary agents are equated if and only if they are bisimilar. Our axiom system consists of a *finite* set of *unconditional* axioms plus the *recursive specification principle* [5,17]. (Since concurrency cannot be simulated by interleaving, to have a finite axiomatization we do not need the auxiliary operator "left merge" [6], which, as it has been proved in [18], is needed to have finite axiomatizations of asynchronous process algebras.)

To prove the completeness of our axiomatization, we introduce a notion of normal form of agents, we prove that any agent can be transformed into a bisimilar normal form by applying our axioms, and we prove that bisimilar normal forms are equated by our axioms. The idea of exploiting normal forms to prove completeness of axiomatizations is well established since the work of Milner [17]. But, contrarily to those of Milner, our normal forms contain the construct of parallel composition. Our normal forms are of the form $A_1 \parallel \ldots \parallel A_n$, where $A_1, \ldots, A_n$ are sequential agents ("$\parallel$" is commutative and associative). The syntactic structure of our normal forms is such that, given a normal form $A_1 \parallel \ldots \parallel A_n$ and an arbitrary input from the environment, $A_1, \ldots, A_n$ react, but at most one of them will be able to react to the next input.

2 The Labeled Transition System

In this section we propose an LTS as an operational semantic model for Gentzen. LTS states correspond to agents, LTS transitions correspond to agent reactions,

and LTS labels carry information on *causality* among signals, i.e. labels highlight *causes* of signals broadcast by agents. These causes consist in the presence/absence of other signals.

We will prove that our LTS permits to recover the input/output behavior of agents (see Prop. 1, Sect. 2.2) and to detect whether instantaneous communications give rise to *nonreactivity*, i.e. the inability of an agent to react to a given input, and *nondeterminism* (see Prop. 2, Sect. 2.2).

Example 1. The agent $A \equiv \texttt{if}\ \overline{a}\ \texttt{then tell}\ a$ *is non-reactive if the environment does not supply* a*. In fact,* A *broadcasts* a *iff* a *is absent at the same instant, so no reaction is admissible. The agent* $B \equiv \texttt{if}\ \overline{a}\ \texttt{then tell}\ b \parallel \texttt{if}\ \overline{b}\ \texttt{then tell}\ a$ *is nondeterministic if the environment supplies neither* a *nor* b*. In fact, either the left branch of* B *broadcasts* b *and the right branch does not broadcast* a*, or the right branch of* B *broadcasts* a *and the left branch does not broadcast* b*.*

Both reactivity and determinism are decidable (see [22] or Prop. 2, Sect. 2.2). Only reactive and deterministic agents are accepted in [22]. (Note that [22] requires only reactivity and nondeterminism; it does not require the more restrictive property of *constructiveness* recently defined in [7].) Reactivity and determinism are required also in all state-oriented synchronous languages except the specification language Statecharts [12], which admits nondeterminism.

2.1 The Definition of the LTS

We let a, b range over the set of signals Π and π range over subsets of Π. We denote with $\overline{a}$ the absence of a and, with abuse of notation, we define $\overline{\overline{a}} = a$. We denote with $\overline{\Pi}$ the set $\{\overline{a} \mid a \in \Pi\}$, and we let γ range over $\Pi \cup \overline{\Pi}$.

An *event* E (over Π) is a subset of $\Pi \cup \overline{\Pi}$ such that there is no signal a such that both $a \in E$ and $\overline{a} \in E$. It is interpreted as the assumption that every signal a with $a \in E$ (resp. $\overline{a} \in E$) is present (resp. absent).

Definition 1. *Given an event* E *and a symbol* $u \in \Pi \cup \{n\}$*, the pair* (E, u) *is a* causality term *with* E *as* cause *and* u *as* action*.*

Let us explain Def. 1. Given a signal $a \in \Pi$, the action a refers to the act (by some agent $\texttt{tell}\ a$) of broadcasting the signal a. The action n refers to the act (by some agent $\texttt{next}$) of pausing for one instant of time. Now, given a causality term (E, u), the action $u \in \Pi \cup \{n\}$ is performed by an agent that is embedded in the body of some agent $\texttt{if}\ b\ \texttt{then}\ \_$ (resp. $\texttt{if}\ \overline{b}\ \texttt{then}\ \_$) for every $b \in E$ (resp. $\overline{b} \in E$). So, if $u = a$ then (E, u) reflects causality among signals, i.e. it reflects that a is broadcast because each $b \in E$ (resp. $\overline{b} \in E$) is present (resp. absent).

Definition 2. *A* (LTS) label *is a pair* $l = \langle \mathcal{E}_l, \mathcal{N}_l \rangle$ *such that:*

- $\mathcal{E}_l$ *is a set of causality terms s.t.* $\bigcup_{(E,u)\in\mathcal{E}_l} E \cup \bigcup_{(E,a)\in\mathcal{E}_l, a\in\Pi} \{a\}$ *is an event;*
- $\mathcal{N}_l$ *is a set of causality terms s.t. for no* $(E', u) \in \mathcal{N}_l$ *we have that* $E' \subseteq \bigcup_{(E,u)\in\mathcal{E}_l} E \cup \bigcup_{(E,a)\in\mathcal{E}_l, a\in\Pi} \{a\}$*.*

The set of labels is denoted by $\mathcal{L}$. A transition $A \xrightarrow{l} A'$ corresponds to the reaction of the agent A which performs the actions represented by the causality terms in $\mathcal{E}_l$ and does not perform the actions represented by the causality terms in $\mathcal{N}_l$. Causality terms in $\mathcal{E}_l$ are needed to recover the input/output behavior of A and to discover whether A is reactive and deterministic. The rôle of $\mathcal{N}_l$ will be explained in the following.

Table 1. The transition system specification for Gentzen.

$$\texttt{skip} \xrightarrow{\langle \emptyset, \emptyset \rangle} \texttt{skip} \quad (skip) \qquad \texttt{next}\ A \xrightarrow{\langle \{(\emptyset, n)\},\ \emptyset \rangle} A \quad (next)$$

$$\texttt{tell}\ a \xrightarrow{\langle \{(\emptyset, a)\}, \emptyset \rangle} \texttt{skip} \quad (tell)$$

$$\frac{A \xrightarrow{l} A'\ ,\ a(l) \in \mathcal{L}}{\texttt{if}\ a\ \texttt{then}\ A \xrightarrow{a(l)} A'} \quad (then\_0) \qquad \frac{A \xrightarrow{l} A'}{\texttt{if}\ a\ \texttt{then}\ A \xrightarrow{N(a,l)} \texttt{skip}} \quad (then\_1)$$

$$\frac{A \xrightarrow{l} A'\ ,\ \overline{a}(l) \in \mathcal{L}}{\texttt{if}\ \overline{a}\ \texttt{then}\ A \xrightarrow{\overline{a}(l)} A'} \quad (else\_0) \qquad \frac{A \xrightarrow{l} A'}{\texttt{if}\ \overline{a}\ \texttt{then}\ A \xrightarrow{N(\overline{a},l)} \texttt{skip}} \quad (else\_1)$$

$$\frac{A[\texttt{rec}\ P.\ A/P] \xrightarrow{l} A'}{\texttt{rec}\ P.\ A \xrightarrow{l} A'} \quad (rec) \qquad \frac{A_1 \xrightarrow{l_1} A_1'\ ,\ A_2 \xrightarrow{l_2} A_2'\ ,\ l_1 \otimes l_2 \in \mathcal{L}}{A_1 \parallel A_2 \xrightarrow{l_1 \otimes l_2} A_1' \parallel A_2'} \quad (par)$$

The LTS is defined by the transition system specification in Table 1.

Rule *skip* states that `skip` performs no action at every instant.

Rule *next* states that `next` A performs the pausing action n and will behave as A at the next instant.

Rule *tell* states that `tell` a broadcasts a and will behave as `skip`. The causality term $(\emptyset, a)$ expresses that the action of broadcasting a is performed independently of the presence/absence of signals in the environment.

Before explaining rules for `if` _ `then` _, we introduce some more notations.

Given a label $l \in \mathcal{L}$ and a symbol $\gamma \in \Pi \cup \overline{\Pi}$, we denote with $\gamma(l)$ the pair:

$$\gamma(l) = \langle \{(E \cup \{\gamma\}, b) \,|\, (E, b) \in \mathcal{E}_l\}, \{(E \cup \{\gamma\}, b) \,|\, (E, b) \in \mathcal{N}_l\} \rangle$$

and with $N(\gamma, l)$ the pair:

$$N(\gamma, l) = \langle \emptyset, \{(E \cup \{\gamma\}, b) \,|\, (E, b) \in \mathcal{E}_l \cup \mathcal{N}_l\} \rangle\,.$$

Rule *then_0* states that if a is present then `if` a `then` A reacts as A. The label $a(l)$ highlights that actions performed by agents in the body of A require the presence of a. Rule *then_1* states that if a is absent then `if` a `then` A terminates.

Example 2. *Given* $A \equiv$ `if` a `then tell` b, *rule then_0 implies* $A \xrightarrow{l_0}$ `skip`, $l_0 = \langle \{(\{a\}, b)\}, \emptyset \rangle$, *and rule then_1 implies* $A \xrightarrow{l_1}$ `skip`, $l_1 = \langle \emptyset, \{(\{a\}, b)\} \rangle$.

Now, we are able to explain the rôle of the second component of labels. Let us consider the label l_1 of Example 2. The causality term $(\{a\}, b)$ in $\mathcal{N}_{l_1}$ keeps track of the fact that the reaction represented by the transition labeled by l_1 is performed because a is absent. In principle, one could keep track of this information in other ways. As an example, it could seem to be reasonable to remove $\mathcal{N}_{l_1}$ and to add the causality term $(\{\overline{a}\}, s)$ to $\mathcal{E}_{l_1}$, with s denoting some idle action. But, with a similar choice, agents `if` a `then if` b `then` A and `if` b `then if` a `then` A would not be bisimilar, despite they are intuitively equivalent. On the contrary, our choice guarantees that these agents are bisimilar.

Rules *else_0* and *else_1* are analogous to *then_0* and *then_1*, respectively.

Rule *par* states that $A_1 \parallel A_2$ performs both reactions of A_1 and A_2. The partial function $\otimes : \mathcal{L} \times \mathcal{L} \rightharpoonup \mathcal{L}$ is defined as follows:

$$l_1 \otimes l_2 = \langle \mathcal{E}_{l_1} \cup \mathcal{E}_{l_2}, \mathcal{N}_{l_1} \cup \mathcal{N}_{l_2} \rangle .$$

Finally, rule *rec* is a standard recursion rule.

2.2 Correspondence with the Operational Semantics of [22]

We say that a set of signals $\pi \subseteq \Pi$ *triggers* a transition $A \xrightarrow{l} A'$ if the transition represents the reaction of A to an environment prompting π. Formally, let us assume a label l such that the causality terms in $\mathcal{E}_l$ having actions in Π are $(E_1, a_1), \ldots, (E_n, a_n)$. We say that π triggers $A \xrightarrow{l} A'$ iff there exists a sequence of sets of signals $\pi_1 \subseteq \ldots \subseteq \pi_{n+1}$ such that:

1. $\pi_1 = \pi$;
2. $\pi_{i+1} = \pi_i \cup \{a_i\}$, $1 \leq i \leq n$;
3. $(E_i \cap \Pi) \subseteq \pi_i$ and $\overline{E_i \cap \overline{\Pi}} \cap \pi_{n+1} = \emptyset$, $1 \leq i \leq n$;
4. both $E \cap \Pi \subseteq \pi_{n+1}$ and $\overline{E \cap \overline{\Pi}} \cap \pi_{n+1} = \emptyset$, for every $(E, u) \in \mathcal{E}_l$;
5. either $E \cap \Pi \not\subseteq \pi_{n+1}$ or $\overline{E \cap \overline{\Pi}} \cap \pi_{n+1} \neq \emptyset$, for every $(E, u) \in \mathcal{N}_l$.

Intuitively, π triggers $A \xrightarrow{l} A'$ if π enables a chain of agents `tell` a_i, $1 \leq i \leq n$, so that signals in $\pi_{n+1} = \pi \cup \bigcup_{1 \leq i \leq n} \{a_i\}$ are in the environment (cond. 1–3), all pausing actions represented by causality terms (E, n) in $\mathcal{E}_l$ are enabled by π_{n+1} (cond. 4), and no action represented by a causality term (E, u) (with $u \in \{n\} \cup \Pi$) in $\mathcal{N}_l$ is enabled by π_{n+1} (cond. 5).

The following propositions have been proved in [26].

Proposition 1. *A set of signals π triggers a transition $A \xrightarrow{l} A'$ if and only if the agent A reacts to the set of signals π so that it will behave as A' at the next instant and the resulting environment contains signals $\pi \cup \bigcup_{(E,a) \in \mathcal{E}_l, a \in \Pi} \{a\}$.*

Proposition 2. *An agent A is reactive and deterministic if and only if, for any agent B such that $A \xrightarrow{l_1} \ldots \xrightarrow{l_n} B$, $n \geq 0$, every set of signals π triggers precisely one transition $B \xrightarrow{l} B'$.*

Example 3. Given the agent A of Example 1, we have that $A \xrightarrow{l} \texttt{skip}$, with $l = \langle \emptyset, \{(\{\overline{a}\}, a)\}\rangle$, is the unique transition from A. (Note that $A \not\xrightarrow{l'}$ for $l' = \langle \{(\{\overline{a}\}, a)\}, \emptyset\rangle$, since l' violates the first condition of Def. 2 and is not in $\mathcal{L}$.) No set $\pi \subseteq \Pi$ with $a \not\in \pi$ triggers the transition. It follows that A is non-reactive. Given the agent B of Example 1, we have that $B \xrightarrow{l_1} \texttt{skip}$ and $B \xrightarrow{l_2} \texttt{skip}$, with $l_1 = \langle \{(\{\overline{a}\}, b)\}, \{(\{\overline{b}\}, a)\}\rangle$ and $l_2 = \langle \{(\{\overline{b}\}, a)\}, \{(\{\overline{a}\}, b)\}\rangle$. The empty set of signals triggers both transitions. It follows that B is nondeterministic.

Note that the part of LTS reachable from a given state A is finite. This fact and Prop. 2 imply that reactivity and determinism of agents are decidable.

2.3 Removing Redundant Information from Labels

Although the LTS we have defined serves our purpose in reflecting the operational semantics of agents given in [22], we are not satisfied with it. In fact, it discriminates agents that are intuitively equivalent. As an example, agents A and $A \parallel \texttt{if}\ a\ \texttt{then}\ A$ are not bisimilar. Our aim is now to remedy this situation.

Let $del : \mathcal{L} \to \mathcal{L}$ be the function removing redundant information from labels such that:

- *del* removes from $\mathcal{E}_l \cup \mathcal{N}_l$ each (E, u) s.t. $(E', u) \in \mathcal{E}_l \cup \mathcal{N}_l$ and $E \supset E'$;
- *del* removes from $\mathcal{E}_l \cup \mathcal{N}_l$ every pair of causality terms $(E \cup \{a\}, n)$ and $(E \cup \{\overline{a}\}, n)$ and adds the causality term (E, n) either to $\mathcal{E}_l$, if $(E \cup \{a\}, n)$ or $(E \cup \{\overline{a}\}, n)$ were in $\mathcal{E}_l$, or to $\mathcal{N}_l$, if both $(E \cup \{a\}, n)$ and $(E \cup \{\overline{a}\}, n)$ were in $\mathcal{N}_l$.
- *del* replaces each causality term $(E \cup \{\gamma\}, n) \in \mathcal{E}_l \cup \mathcal{N}_l$ by (E, n), provided that $(E' \cup \{\overline{\gamma}\}, n) \in \mathcal{E}_l \cup \mathcal{N}_l$ and $E' \subset E$.

If both (E, u) and (E', u) appear in l and $E \supset E'$, then the former causality term carries redundant information. In fact, it is sufficient that the environment prompts signals as assumed by E' to have the action represented by u. Intuitively, this corresponds to have that A is equivalent to $A \parallel \texttt{if}\ \gamma\ \texttt{then}\ A$.

If both $(E \cup \{a\}, n)$ and $(E \cup \{\overline{a}\}, n)$ appear in l, then they are replaced by (E, n) because it is sufficient that the environment prompts signals as assumed by E to have the pausing action represented by n. Intuitively, this corresponds to have that $\texttt{next}\ A$ is equivalent to $\texttt{if}\ a\ \texttt{then}\ \texttt{next}\ A \parallel \texttt{if}\ \overline{a}\ \texttt{then}\ \texttt{next}\ A$.

Note that $A \equiv \texttt{if}\ a\ \texttt{then}\ \texttt{tell}\ b \parallel \texttt{if}\ \overline{a}\ \texttt{then}\ \texttt{tell}\ b$ is not equivalent to $A' \equiv \texttt{tell}\ b$. In fact, agent $B \equiv \texttt{if}\ b\ \texttt{then}\ \texttt{tell}\ a$ discriminates them, since $A' \parallel B$ is deterministic and reactive, whereas $A \parallel B$ is not. For this reason, given $b \in \Pi$, *del* does not replace $(E \cup \{a\}, b)$ and $(E \cup \{\overline{a}\}, b)$ by (E, b).

If both $(E \cup \{\gamma\}, n)$ and $(E' \cup \{\overline{\gamma}\}, n)$ appear in l and $E' \subset E$, then the former causality term is replaced by (E, n), because it is sufficient that the environment prompts signals as assumed by E to have the action represented by n.

Now, we will consider the LTS obtained by applying function *del* to every label. This corresponds to replace $a(l)$ by $del(a(l))$, $\overline{a}(l)$ by $del(\overline{a}(l))$, $N(a, l)$ by $del(N(a, l))$, $N(\overline{a}, l)$ by $del(N(\overline{a}, l))$ and $l_1 \otimes l_2$ by $del(l_1 \otimes l_2)$ in Table 1.

It is immediate that Prop. 1 and Prop. 2 are still valid.

2.4 Bisimulation

In order to relate agents having the same input/output behavior, we consider the bisimulation on the states of the LTS. Given agents A_1 and A_2 such that $A_1 \approx A_2$, the external environment is not able to distinguish between them. In fact, at each instant, if A_1 and A_2 are stimulated with the same set of signals, then they respond by broadcasting the same signals. Moreover, A_1 is reactive and deterministic if and only if A_2 is. (These facts are consequences of Prop. 1 and Prop. 2.) We show now that no Gentzen context is able to discriminate them, i.e. Gentzen constructs preserve bisimulation.

Theorem 1. *The bisimulation on Gentzen agents is a congruence.*

Proof. Table 1 contains a recursion rule and only rules in the well known *de Simone format* [24]. So, Gentzen is a *de Simone language* and, as it has been proved in [24], this guarantees that bisimulation is a congruence. □

2.5 Some Remarks

The structure of our LTS labels could seem too heavy. One may wonder whether LTS labels consisting of pairs ⟨"received signals", "produced signals"⟩ would be sufficient. Unfortunately, such a simple solution must be rejected, since labels do not carry information on signal causality, which is needed to define a semantics of a synchronous language in a compositional way (see [15] for a detailed discussion).

Example 4. Assume agents $A \equiv \texttt{if}\ a\ \texttt{then tell}\ b \parallel \texttt{if}\ c\ \texttt{then tell}\ d$ and $A' \equiv \texttt{if}\ a\ \texttt{then if}\ c\ \texttt{then}\ (\texttt{tell}\ b \parallel \texttt{tell}\ d)$. Let $A \xrightarrow{l} \texttt{skip} \parallel \texttt{skip}$ and $A' \xrightarrow{l} \texttt{skip} \parallel \texttt{skip}$, with $l = \langle \{a, c\}, \{b, d\} \rangle$, be the transitions representing the reactions of A and A' to input $\{a, c\}$, respectively. Let $B \equiv \texttt{if}\ b\ \texttt{then tell}\ c$ and $B \xrightarrow{l'} \texttt{skip}$, with $l' = \langle \{b\}, \{c\} \rangle$, be the transition representing the reaction of B to input $\{b\}$. The transition $A \parallel B \xrightarrow{l''} (\texttt{skip} \parallel \texttt{skip}) \parallel \texttt{skip}$, with $l'' = \langle \{a\}, \{b, c, d\} \rangle$, must be obtained by combining $A \xrightarrow{l} \texttt{skip} \parallel \texttt{skip}$ with $B \xrightarrow{l'} \texttt{skip}$. Therefore, also the transition $A' \parallel B \xrightarrow{l''} (\texttt{skip} \parallel \texttt{skip}) \parallel \texttt{skip}$ is in the LTS. This is wrong, since $A' \parallel B$ reacts to $\{a\}$ by broadcasting no signal.

We have said in the introduction of this paper that synchronous concurrency cannot be reduced to sequentiality. If this sentence were wrong, it would be possible to simulate agent A of Example 4 by a sequential agent $\hat{A}$. Now, $\hat{A}$ should broadcast both b and d iff both a and c are present. So, both $\texttt{tell}\ b$ and $\texttt{tell}\ d$ should appear in the body of both $\texttt{if}\ a\ \texttt{then}\ \_$ and $\texttt{if}\ c\ \texttt{then}\ \_$. In this case, the Gentzen context $[\_ \parallel B]$, with B as in Example 4, discriminates A and $\hat{A}$, thus proving that $\hat{A}$ cannot simulate A.

3 The Axiomatization

In this section we present an axiomatization over Gentzen which is sound and complete modulo bisimulation.

3.1 The Axiom System

Let us denote by "=" the least reflexive, symmetric and transitive congruence over Gentzen agents satisfying the axioms in Table 2.

Table 2. An axiom system for Gentzen.

$$A_1 \parallel A_2 = A_2 \parallel A_1 \quad (\parallel_1) \qquad A_0 \parallel (A_1 \parallel A_2) = (A_0 \parallel A_1) \parallel A_2 \quad (\parallel_2)$$

$$A \parallel \mathtt{skip} = A \quad (\parallel_3) \qquad A \parallel A = A \quad (\parallel_4)$$

$$\mathtt{if}\ \gamma\ \mathtt{then}\ (A_1 \parallel A_2) = \mathtt{if}\ \gamma\ \mathtt{then}\ A_1 \parallel \mathtt{if}\ \gamma\ \mathtt{then}\ A_2 \qquad (if\_1)$$

$$\mathtt{if}\ \gamma\ \mathtt{then}\ \mathtt{if}\ \gamma'\ \mathtt{then}\ A = \mathtt{if}\ \gamma'\ \mathtt{then}\ \mathtt{if}\ \gamma\ \mathtt{then}\ A \qquad (if\_2)$$

$$\mathtt{if}\ \gamma\ \mathtt{then}\ \mathtt{if}\ \gamma\ \mathtt{then}\ A = \mathtt{if}\ \gamma\ \mathtt{then}\ A \qquad (if\_3)$$

$$A = A \parallel \mathtt{if}\ \gamma\ \mathtt{then}\ A \qquad (if\_4)$$

$$\mathtt{if}\ \gamma\ \mathtt{then}\ \mathtt{skip} = \mathtt{skip} \qquad (if\_5)$$

$$\mathtt{next}\ A = \mathtt{if}\ a\ \mathtt{then}\ \mathtt{next}\ A \parallel \mathtt{if}\ \overline{a}\ \mathtt{then}\ \mathtt{next}\ A \qquad (if\_6)$$

$$\mathtt{next}\ A_1 \parallel \mathtt{next}\ A_2 = \mathtt{next}\ (A_1 \parallel A_2) \qquad (next)$$

$$\mathtt{rec}\, P.\, A = A[\mathtt{rec}\, P.\, A/P] \qquad (rec\_1)$$

$$B = A[B/P] \text{ and } P \text{ guarded in } A \text{ imply } B = \mathtt{rec}\, P.A \qquad (rec\_2)$$

Axioms $\parallel_1 - \parallel_3$, $if\_1 - if\_5$, $next$ and rec_1, rec_2 are very intuitive.

Axiom $\parallel_4$ deserves attention. Given transitions $A \xrightarrow{l_1} A_1$ and $A \xrightarrow{l_2} A_2$ with $l_1 \neq l_2$, it holds that $l_1 \otimes l_2 \notin \mathcal{L}$(this fact can be proved by induction over A). Moreover, given a label $l \in \mathcal{L}$, it holds that $l \otimes l = l$. From these facts it follows that $A \parallel A \approx A$, i.e. axiom $\parallel_4$ is sound.

We have explained in Section 2 that one expects axiom if_6, and that one does not expect the following axiom:

$$\mathtt{tell}\ b = \mathtt{if}\ a\ \mathtt{then}\ \mathtt{tell}\ b \parallel \mathtt{if}\ \overline{a}\ \mathtt{then}\ \mathtt{tell}\ b\,.$$

The following result follows from definitions of $\gamma(l)$, $N(\gamma, l)$, $l_1 \otimes l_2$ and $del(l)$.

Lemma 1 (Soundness). *Given agents A and A', $A = A'$ implies $A \approx A'$.*

Our axiom system is *finite*; all axioms except the *recursion specification principle* [5,17] (i.e. axiom rec_2) are *unconditional*. We have not enriched the language by any auxiliary operator. In particular, it is worth noting that we do not need the auxiliary operator "left merge" [6], which, on the contrary, is needed to have finite axiomatizations for asynchronous process algebras (see [18]).

3.2 Normal Forms

Given a string $\vartheta \in (\Pi \cup \overline{\Pi})^*$, we denote by $\texttt{if}\ \vartheta\ \texttt{then}\ A$ the agent:

$$\texttt{if}\ \vartheta\ \texttt{then}\ A \equiv \begin{cases} A & \text{if } \vartheta = \epsilon \text{ } (\epsilon \text{ denotes the empty string}) \\ \texttt{if}\ \gamma\ \texttt{then if}\ \phi\ \texttt{then}\ A & \text{if } \vartheta = \gamma\phi, \phi \in (\Pi \cup \overline{\Pi})^* \end{cases}$$

Given a string $\vartheta \in (\Pi \cup \overline{\Pi})^*$, we denote by $|\vartheta|$ the set $\{\gamma \,|\, \gamma \text{ appears in } \vartheta\}$.

Definition 3. *An agent A is a* normal form *if either $A \equiv \texttt{skip}$ or there exist agents $A_1, \ldots, A_n$ such that:*

1. $A \equiv A_1 \parallel \ldots \parallel A_n$;
2. $A_i \equiv \texttt{if}\ \vartheta_i\ \texttt{then}\ B_i$, *and either* $B_i \equiv \texttt{tell}\ a_i$, *or* $B_i \equiv \texttt{next}\ A'_i$, *or* $B_i \equiv P_i$, *with a_i a signal, A'_i an arbitrary agent and P_i a recursion variable;*
3. *if* $B_i \equiv \texttt{next}\ A'_i$ *then, for each $a \in \Pi$, either a or $\overline{a}$ appears in ϑ_i;*
4. *if* $B_i \equiv \texttt{next}\ A'_i$ *then every symbol $\gamma \in \Pi \cup \overline{\Pi}$ appears at most once in ϑ_i;*
5. *if* $B_i \equiv \texttt{next}\ A'_i$ *and* $B_j \equiv \texttt{next}\ A'_j$ *then there exists some $\gamma \in \Pi \cup \overline{\Pi}$ such that γ appears in ϑ_i and $\overline{\gamma}$ appears in ϑ_j;*
6. *if* $B_i \equiv \texttt{tell}\ a$ *then there exists no j such that* $B_j \equiv \texttt{tell}\ a$ *and* $|\vartheta_j| \subseteq |\vartheta_i|$.

Let $A \equiv A_1 \parallel \ldots \parallel A_n$ be a normal form. Conditions 1, 2 and 3 of Def. 3 will be exploited in proving the completeness of our axiomatization.
Condition 4 is a reasonable request and can be easily respected by transforming agents by axioms if_2 and if_3.
Condition 5 implies that if $A_i \equiv \texttt{if}\ \vartheta_i\ \texttt{then next}\ A'_i$ and each signal a such that $a \in |\vartheta_i|$ (resp. $\overline{a} \in |\vartheta_i|$) is present (resp. absent), then A will behave as A'_i at the next instant. More precisely, at the next instant the agent $\texttt{skip} \parallel \ldots \parallel \texttt{skip} \parallel A'_i \parallel \texttt{skip} \parallel \ldots \parallel \texttt{skip}$ will run.
Condition 6 implies that redundant agents broadcasting signals do not appear as parallel components of A. Moreover, if $A_i \equiv \texttt{if}\ \vartheta_i\ \texttt{then tell}\ a$, then $(|\vartheta_i|, a)$ appears in any label l such that $A \xrightarrow{l}$, since it is not removed by function *del*.

We prove now that each agent can be transformed into a normal form, having only normal forms as derivatives, by applying axioms of Table 2.

Lemma 2 (Reducibility to normal forms). *Given an agent A, there exist normal forms $A_1, \ldots, A_m$ such that:*

- $A = A_1$;
- *either* $A_i = \texttt{skip}$, *or* $A_i = A_{i,1} \parallel \ldots \parallel A_{i,n_i}$, $A_{i,j} \equiv \texttt{if}\ \vartheta_{i,j}\ \texttt{then}\ B_{i,j}$, *and, if* $B_{i,j} \equiv \texttt{next}\ A_{f(i,j)}$, *then* $f_{(i,j)} \in \{1, \ldots, m\}$.

Proof. By structural induction on A.
Base case: if either $A \equiv \texttt{skip}$ or $A \equiv \texttt{tell}\ a$ or $A \equiv P$ then the thesis is immediate, because A is a normal form.
Induction step: Assume that, given agents A' and A'', there exist normal forms $A_{1'}, \ldots, A_{m'}$ and $A_{1''}, \ldots, A_{m''}$, either $A_{i'} = \texttt{skip}$ or $A_{i'} = A_{i',1} \parallel \ldots \parallel A_{i',n_{i'}}$, either $A_{i''} = \texttt{skip}$ or $A_{i''} = A_{i'',1} \parallel \ldots \parallel A_{i'',n_{i''}}$, $A_{i',j} \equiv \texttt{if}\ \vartheta_{i',j}\ \texttt{then}\ B_{i',j}$, $A_{i'',j} \equiv \texttt{if}\ \vartheta_{i'',j}\ \texttt{then}\ B_{i'',j}$, $A' = A_{1'}$, $A'' = A_{1''}$.
We consider the following cases:

- $A \equiv \texttt{next}\ A'$: Since $A' = A_{1'}$ and $=$ is a congruence, we can rewrite A into $\texttt{next}\ A_{1'}$, which does not satisfy Condition 3 of Def. 3. Now, $\texttt{next}\ A_{1'}$ is rewritten into a normal form by axiom if_6, and the thesis follows.
- $A \equiv A' \parallel A''$: Since $A' = A_{1'}$ and $A'' = A_{1''}$, we infer $A = A_{1'} \parallel A_{1''}$ by the fact that $=$ is a congruence. Let us denote with $\mathcal{I} \subseteq \{1', \ldots, m'\} \times \{1'', \ldots, m''\}$ the least set such that:
  - $(1', 1'') \in \mathcal{I}$;
  - if $(i', i'') \in \mathcal{I}$, $A_{i',j'} \equiv \texttt{if}\ \vartheta_{i',j'}\ \texttt{then next}\ A_{f(i',j')}$, $A_{i'',j''} \equiv \texttt{if}\ \vartheta_{i'',j''}\ \texttt{then next}\ A_{f(i'',j'')}$ and $|\vartheta_{i',j'}| = |\vartheta_{i'',j''}|$ then $(f(i', j'), f(i'', j'')) \in \mathcal{I}$.

  The set $\mathcal{I}$ contains (i', i'') if A reaches a state in which it behaves as $A_{i'} \parallel A_{i''}$. The thesis follows if we infer $A_{i'} \parallel A_{i''} = A_{i',i''}$, with $A_{i',i''}$ an arbitrary normal form having only normal forms as derivatives, for every $(i', i'') \in \mathcal{I}$.
  If $A_{i'} = \texttt{skip}$ (resp. $A_{i''} = \texttt{skip}$) then we delete it by axiom $\|_3$, and the thesis follows. Otherwise, agent $A_{i'} \parallel A_{i''}$ satisfies Cond. 1–4 of Def. 3. Let us assume that it does not satisfy Cond. 5, i.e. $A_{i',j'} \equiv \texttt{if}\ \vartheta_{i',j'}\ \texttt{then next}\ A_{f(i',j')}$, $A_{i'',j''} \equiv \texttt{if}\ \vartheta_{i'',j''}\ \texttt{then next}\ A_{f(i'',j'')}$ and $|\vartheta_{i',j'}| = |\vartheta_{i'',j''}|$, for some $1 \leq j' \leq n_{i'}$, $1 \leq j'' \leq n_{i''}$. By axioms $\|_1, \|_2$ we rewrite $A_{i'} \parallel A_{i''}$ so that $A_{i',j'}$ and $A_{i'',j''}$ appear as adjacent. Now, by axiom if_2 we rewrite $\texttt{if}\ \vartheta_{i'',j''}\ \texttt{then next}\ A_{f(i'',j'')}$ into $\texttt{if}\ \vartheta_{i',j'}\ \texttt{then next}\ A_{f(i'',j'')}$, and, by axioms if_1 and $next$, we rewrite $A_{i',j'} \parallel \texttt{if}\ \vartheta_{i',j'}\ \texttt{then next}\ A_{f(i'',j'')}$ into $\texttt{if}\ \vartheta_{i',j'}\ \texttt{then next}\ (A_{f(i',j')} \parallel A_{f(i'',j'')})$. Since this rewriting preserves conditions 1–4 of Def. 3 and can be applied to every pair $(A_{i',j'}, A_{i'',j''})$ such that $|\vartheta_{i',j'}| = |\vartheta_{i'',j''}|$, we can transform $A_{i'} \parallel A_{i''}$ into an agent satisfying Cond. 1–5 of Def. 3. If the agent so obtained violates Cond. 6 of Def. 3, we can remedy to this situation by applying axioms $\|_1, \|_2$ as before, and then axioms if_2 and if_4.
- $A \equiv \texttt{if}\ \gamma\ \texttt{then}\ A'$: Since $A' = A_{1'}$, we infer $A = \texttt{if}\ \gamma\ \texttt{then}\ A_{1'}$ by the fact that $=$ is a congruence. If $A_{1'} = \texttt{skip}$ then we rewrite $\texttt{if}\ \gamma\ \texttt{then}\ A_{1'}$ into $\texttt{skip}$ by axiom if_5 and the thesis follows. Otherwise, since $A_{1'}$ is a normal form, we apply axiom if_1 to $\texttt{if}\ \gamma\ \texttt{then}\ A_{1'}$ and we obtain an agent satisfying conditions 1, 2, 3 and 5 of Def. 3. By axioms if_2 and if_3 we can rewrite every parallel component of this agent so that it satisfies also Cond. 4, while preserving other conditions. The agent so obtained can be transformed into an agent satisfying Cond. 6 of Def. 3 as in the previous case, and the thesis follows.
- $A \equiv \texttt{rec}\, P.A'$: If $A' \equiv \texttt{skip}$ then $A = \texttt{skip}$ by axiom rec_1 and the thesis follows. Otherwise, let $H_{1'}, \ldots, H_{m'}$ be agents s.t. $H_{i'} \equiv A_{i'}[A/P]$.
  We have $H_{i'} \equiv A_{i'}[A/P] \equiv (A_{i',1} \parallel \ldots \parallel A_{i',n_{i'}})[A/P] \equiv$
  $A_{i',1}[A/P] \parallel \ldots \parallel A_{i',n_{i'}}[A/P] \equiv$
  $A_{i',1}[H_{f(i',1)}/A_{f(i',1)}][A/P] \parallel \ldots \parallel A_{i',n_{i'}}[H_{f(i',n_{i'})}/A_{f(i',n_{i'})}][A/P] =$
  $A_{i',1}[H_{f(i',1)}/A_{f(i',1)}][A'[A/P]/P] \parallel \ldots$
  $\ldots \parallel A_{i',n_{i'}}[H_{f(i',n_{i'})}/A_{f(i',n_{i'})}][A'[A/P]/P] =$
  $A_{i',1}[H_{f(i',1)}/A_{f(i',1)}][A_{1'}[A/P]/P] \parallel \ldots$
  $\ldots \parallel A_{i',n_{i'}}[H_{f(i',n_{i'})}/A_{f(i',n_{i'})}][A_{1'}[A/P]/P] \equiv$
  $A_{i',1}[H_{f(i',1)}/A_{f(i',1)}][H_{1'}/P] \parallel \ldots \parallel A_{i',n_{i'}}[H_{f(i',n_{i'})}/A_{f(i',n_{i'})}][H_{1'}/P],$

where the equalities are inferred by axiom *rec_1* and by the fact that $=$ is a congruence. Now, $H_{i'}$ has been rewritten as a parallel composition of normal forms having only normal forms as derivatives. Let us call $K_{i'}$ such agent. Every $K_{i'}$ can be transformed in a normal form as in the case for "$\|$". Finally, the thesis follows since $A = A'[A/P] = A_{1'}[A/P] \equiv H_{1'} = K_{1'}$.

□

3.3 The Proof of Completeness

First of all we recall the notion of guarded recursive specification and we prove that every guarded recursive specification has a solution which is unique modulo $=$. Then, we exploit this result and Lemma 2 to prove that two arbitrary agents A and A' such that $A \approx A'$ are equated by our axioms. In fact, we prove that $A = B$ and $A' = B'$, where A' and B' are normal forms and solutions of the same guarded recursive specification.

A *recursive specification* over variables $\vec{P} = P_1, \ldots, P_m$ is a set of equations

$$P_i = B_i \qquad\qquad 1 \leq i \leq m \tag{1}$$

where B_i is an agent, $1 \leq i \leq m$. It is *guarded* if and only if $P_1, \ldots, P_m$ are guarded in $B_1, \ldots, B_m$, i.e. $P_1, \ldots, P_m$ appear in $B_1, \ldots, B_m$ in bodies of `next` _.

A *solution* of a recursive specification as (1) is a set of agents $\vec{A} \equiv A_1, \ldots, A_m$ such that $A_i = B_i[\vec{A} \,/\, \vec{P}]$, $1 \leq i \leq m$.

The following result, which states that every guarded recursive specification has a solution unique modulo "=", has been presented in [17]. (Note that axiom *rec_2* plays a central rôle in the proof.)

Lemma 3. *Every guarded recursive specification*

$$P_i = B_i \qquad\qquad 1 \leq i \leq m$$

has a solution $\vec{A} \equiv A_1, \ldots, A_m$. *Given any other solution* $\vec{A'} \equiv A'_1, \ldots, A'_m$, *it holds that* $A'_i = A_i$, $1 \leq i \leq m$.

We show now that two arbitrary bisimilar agents are equated by our axioms.

Lemma 4 (Completeness). *Given agents A and A', $A \approx A'$ implies $A = A'$.*

Proof. By Lemma 2, there are normal forms $A_1, \ldots, A_m$ and $A_{1'}, \ldots, A_{m'}$ s.t.:

- $A = A_1$, $A' = A_{1'}$
- either $A_i = \texttt{skip}$ or $A_i = A_{i,1} \parallel \ldots \parallel A_{i,n_i}$, $1 \leq i \leq m$
- either $A_{i'} = \texttt{skip}$ or $A_{i'} = A_{i',1} \parallel \ldots \parallel A_{i',n_{i'}}$, $1 \leq i' \leq m'$
- $A_{i,j} \equiv \texttt{if } \vartheta_{i,j} \texttt{ then } B_{i,j}$, $1 \leq i \leq m$, $1 \leq j \leq n_i$
- $A_{i',j'} \equiv \texttt{if } \vartheta_{i',j'} \texttt{ then } B_{i',j'}$, $1 \leq i' \leq m'$, $1 \leq j' \leq n_{i'}$.

Moreover, by axioms $\|_4$ we can assume that $A_{i,j} \not\equiv A_{i,h}$ for $1 \leq j < h \leq n_i$ and $A_{i',j'} \not\equiv A_{i',h'}$ for $1 \leq j' < h' \leq n_{i'}$.
Now, to prove the thesis it is sufficient to prove that $A_1 = A_{1'}$.
Let us denote by $\mathcal{I} \subseteq \{1, \ldots, m\} \times \{1', \ldots, m'\}$ the set of pairs (i, i') such that $A_i \approx A_{i'}$. Since $A \approx A'$, $A = A_1$ and $A' = A_{1'}$, and $=$ is sound modulo $\approx$, we have $(1, 1') \in \mathcal{I}$.
We prove now that, given a pair $(i, i') \in \mathcal{I}$, and an arbitrary index $1 \leq j \leq n_i$, there exists an index $1 \leq j' \leq n_{i'}$ such that:

- $\vartheta_{i',j'} = \vartheta_{i,j}$;
- either $B_{i,j} \equiv B_{i',j'}$, or $B_{i,j} \equiv \texttt{next}\ A_{f(i,j)}$, $B_{i',j'} \equiv \texttt{next}\ A_{f(i',j')}$ and $(f(i,j), f(i',j')) \in \mathcal{I}$.

We begin with proving that there is a $1 \leq j' \leq n_{i'}$ such that $|\vartheta_{i,j}| = |\vartheta_{i',j'}|$ and the second condition above is satisfied. We consider the following cases:

- $B_{i,j} \equiv \texttt{tell}\ a$: Since A_i is a normal form and, in particular, it satisfies Cond. 6 of Def. 3, $(|\vartheta_{i,j}|, a)$ appears in any label l such that $A \xrightarrow{l}$. Since $A_i \approx A_{i'}$, we have $A_{i'} \xrightarrow{l}$. Since $(|\vartheta_{i,j}|, a)$ appears in l, there exists $1 \leq j' \leq n_{i'}$ such that $|\vartheta_{i',j'}| = |\vartheta_{i,j}|$ and $B_{i',j'} \equiv \texttt{tell}\ a$.
- $B_{i,j} \equiv \texttt{next}\ A_{f(i,j)}$: Since A_i is a normal form and, in particular, it satisfies Cond. 5 of Def. 3, we have $A \xrightarrow{l} A_{f(i,j)}$ for some label l. Since $A \approx A'$, there exists an agent $A_{f(i',j')}$ such that $A \xrightarrow{l} A_{f(i',j')}$ and $A_{f(i,j)} \approx A_{f(i',j')}$.

Now, if $|\vartheta_{i,j}| = |\vartheta_{i',j'}|$ but $\vartheta_{i,j} \neq \vartheta_{i',j'}$, we can apply axiom if_2 to $A_{i',j'}$ while preserving conditions of Def. 3. So, we can rewrite $\texttt{if}\ \vartheta_{i',j'}\ \texttt{then}\ B_{i',j'}$ into $\texttt{if}\ \vartheta_{i,j}\ \texttt{then}\ B_{i',j'}$. Now, we reapply our reasoning from the beginning.

Let us consider now the recursive specification

$$P_{i,i'} = A_{i,i'} \qquad\qquad (i, i') \in \mathcal{I}$$

where $A_{i,i'}$ is the agent such that $A_{i,i'} \equiv H_{i,i',1} \parallel \ldots \parallel H_{i,i',n_i}$, with
either $H_{i,i',j} \equiv \texttt{if}\ \vartheta_{i,j}\ \texttt{then}\ \texttt{next}\ P_{f(i,j),f(i',j')}$,
if both $A_{i,j} \equiv \texttt{if}\ \vartheta_{i,j}\ \texttt{then}\ \texttt{next}\ A_{f(i,j)}$ and $A_{i',j'} \equiv \texttt{if}\ \vartheta_{i,j}\ \texttt{then}\ \texttt{next}\ A_{f(i',j')}$,
or $H_{i,i',j} \equiv A_{i,j} \equiv A_{i',j'}$, otherwise.
This recursive specification is guarded and has both $\vec{G}$ and $\vec{G'}$ as solutions, where $G_{i,i'} \equiv A_i$ and $G'_{i,i'} \equiv A_{i'}$ for each $(i, i') \in \mathcal{I}$. Now, by Lemma 3 it follows that $A_i = A_{i'}$, $(i, i') \in \mathcal{I}$. Since $(1, 1') \in \mathcal{I}$, we have $A_1 = A_{1'}$, as required. □

The following theorem states that axioms in Table 2 give an axiomatization sound and complete modulo bisimulation on Gentzen agents.

Theorem 2 (Soundness and completeness). *Given arbitrary agents A and A', $A = A'$ if and only if $A \approx A'$.*

Proof "If": by Lemma 4. "Only if": by Lemma 1.

4 Conclusions and Future Work

We have presented an axiomatization of the synchronous language Gentzen, in order to characterize behaviorally equivalent agents.

The axiomatization of Gentzen cannot be done by trivially adapting techniques developed in the field of asynchronous concurrency. Axiomatizations of asynchronous process description languages are based on the transformation of processes into "head normal forms". In order to transform concurrent processes into head normal forms, concurrency must be simulated by interleaving. This is impossible in the synchronous setting, because processes running in parallel are perfectly synchronized and cannot interleave.

The axiomatization of Gentzen should be intended as an initial step toward the development of axiomatizations for state-oriented synchronous languages. In fact, Gentzen offers all basic constructs that are shared by the languages in this class and that distinguish synchronous languages w.r.t. asynchronous ones.

Given any other state-oriented synchronous language, we believe that it can be axiomatized by simply extending our work to deal with those constructs that are typical in such a language and that are not offered by Gentzen. In particular, we believe that such a language can be axiomatized by exploiting the notion of normal form proposed in this paper. To support this affirmation, we have developed an axiomatic semantics for the language Esterel [8] in [26]. The axiomatization of Esterel is derived by extending the technique proposed in the present paper to deal with the interruption mechanism "`trap`", the suspension mechanism "`suspend`", the construct of instantaneous sequencing ";" and the property of *constructiveness* [7]. The technique adopted to axiomatize these constructs is to express them in terms of constructs of Gentzen.

The approach of [26] can be generalized to any synchronous language: given a superset of Gentzen, it is sufficient to express any construct in terms of constructs of Gentzen. So, Gentzen could play in the synchronous setting the rôle of T_{fintree} (i.e. a subset of CCS suitable for expressing finite LTSs) in the asynchronous setting, where an axiomatization of T_{fintree} has been proposed in [17], and supersets of T_{fintree} are axiomatized by expressing any construct in terms of constructs of T_{fintree}. Following [1,2], where algorithmic techniques to compute axiomatizations of classes of supersets of T_{fintree} have been proposed, one could develop algorithmic techniques to compute axiomatizations of classes of supersets of Gentzen.

References

1. Aceto, L.: *Deriving complete inference systems for a class of GSOS languages generating regular behaviors*. Proc. of CONCUR '94, Springer LNCS 836, 1994.
2. Aceto, L., Bloom, B. and Vaandrager, F.: *Turning SOS rules into equations*. Information and Computation **111**, 1994.
3. André, C.: *Representation and analysis of reactive behaviors: a synchronous approach*. Presented at CESA '96, IEEE-SMC, Lille, France, 1996.

4. Benveniste, A. and Berry, G. (editors): *Another look at real-time systems*. Proceedings of the IEEE **79**, 1991.
5. Bergstra, J.A. and Klop, J.W.: *A complete inference system for regular processes with silent moves*. Proc. of Logic Colloquium 1986, North Holland, 1988.
6. Bergstra, J.A. and Klop, J.W.: *Process algebra for synchronous communication*. Information and Computation **60**, 1984.
7. Berry, G.: *The constructive semantics of pure Esterel*. Version 3.0, 1999. URL: `http://www.inria.fr/meije/personnel/Gerard.Berry.html`.
8. Berry, G. and Gonthier, G.: *The Esterel synchronous programming language: design, semantics, implementation*. Science of Computer Programming **19**, 1992.
9. Caspi, P., Halbwachs, N., Pilaud, P. and Plaice, J.: *Lustre, a declarative language for programming synchronous systems*. Proc. of POPL '87, ACM Press, 1987.
10. Gonthier, G: *Sémantique et modèles d'exécution des langages réactifs synchrones; application à Esterel*. Thèse d' informatique, Univesité d'Orsay, France, 1988.
11. Halbwachs, N.: *Synchronous programming of reactive systems*. Kluwer Academic Publishers, Dordrecht, 1993.
12. Harel, D.: *Statecharts: A visual formalism for complex systems*. Science of Computer Programming **8**, 1987.
13. Harel, D. and Naamad, A.: *The Statemate semantics of Statecharts*. ACM Transactions on Software Engineering Methodologies **5**, 1996.
14. Harel, D. and Pnueli, A.: *On the development of reactive systems*. In K.R. Apt editor, Logic and Models of Concurrent Systems, NATO, ASI-13, Springer, 1985.
15. Huizing, C. and Gerth, R.: *Semantics of reactive systems in abstract time*. Proc. of REX Workshop "Real Time: Theory in Practice", Springer LNCS 600, 1992.
16. Maraninchi, F.: *Operational and compositional semantics of synchronous automaton composition*. Proc. of CONCUR '92, Springer LNCS 630, 1992.
17. Milner, R.: *A complete inference system for a class of regular behaviors*. Journal of Computer and System Sciences **28**, 1984.
18. Moller, F.: *The importance of the left merge operator in process algebras*. Proc. of ICALP '90, Springer LNCS 443, 1990.
19. Plotkin, G.: *A structural approach to operational semantics*. Technical Report DAIMI FN-19, University of Aarhus, Denmark, 1981.
20. Pnueli, A. and Shalev, M.: *What is a step: On the semantics for Statecharts*. Proc. of TACS '91, Springer LNCS 526, 1991.
21. Saraswat, V.A., Jagadeesan, R. and Gupta, V.: *Default timed concurrent constraint programming*. Proc. of POPL '95, ACM Press, 1995.
22. Saraswat, V.A., Jagadeesan, R. and Gupta, V.: *Timed default concurrent constraint programming*. Journal of Symbolic Computation **11**, 1996.
23. Scholtz, P., Nazareth, D. and Regensburger, F.: *Mini-Statecharts: A compositional way to model parallel systems*. In 9^{th} Int. Conf. on Parallel and Distributed Computing Systems, Dijon, France, 1997.
24. de Simone, R.: *Higher level synchronizing devices in SCCS-Meije*. Theoretical Computer Science **37**, 1985.
25. Tini, S.: *On the expressiveness of Timed Concurrent Constraint Programming*. Proc. of Express '99, Electronic Notes in Theoretical Computer Science **27**, 1999.
26. Tini, S.: *Structural Operational Semantics for Synchronous Languages*. Ph.D. Thesis, Dipartimento di Informatica, University of Pisa, Italy, 2000.

MARRELLA and the Verification of an Embedded System

Dominique Ambroise[1], Patrick Augé[1], Kamel Bouchefra[2], and Brigitte Rozoy[1]

[1] Laboratoire de Recherche en Informatique,
Université Paris 11, Bât. 490, 91405 Orsay Cedex France
{ambroise, auge, rozoy}@lri.fr
[2] Laboratoire d'Informatique de Paris Nord, Université Paris 13,
Avenue Jean-Baptiste Clément, 93430 Villetaneuse, France
kb@lipn.univ-paris13.fr

Abstract. We present the architecture of MARRELLA, a tool designed for simulation and verification of distributed systems. The input of the tool is an event driven language whereas the underlying model is event structures. It gives the possibilities of generating one, some or all the executions of any distributed program. We have tested the tool for the verification of an embedded system. The corresponding results are reported here as well as those obtained with equivalent tools applied to the same case study. These experimental results show then the efficiency of MARRELLA.

1 Introduction

It is now well known and recognized that one execution of a distributed program may be symbolized by a partial order. In the same spirit, prime event structures can be used to exhibit in a single object all possible executions. Thus, their use may become of a great help for formal specifications, simulations and verifications of such programs. Inherently, our method is based on these prime event structures and exploits techniques that unfold the behaviors of programs into acyclic transition systems, the nodes of which are global states. Then we exploit the properties of the structure to reduce the graph.

Another version of this idea has already been used for verification purposes : unfolding of occurrence Petri nets are constructed. Numerous other techniques have also been used for reducing the size of the graph and for on the fly verification [Esp. Röm. 99]. In case of distributed programs and as it partially avoids the famous state explosion problem, this model is well adapted for efficient simulations, even exhaustive, thus for verifications. Substantial efficiencies are obtained as the enumeration of all possible interleaving is avoided.

2 MARRELLA

The tool may be viewed either as a simulator or as a model checker: starting with a concrete distributed algorithm, it efficiently constructs the associated graph of states. It is theoretically based on prime event structures with binary conflicts.

F. Honsell and M. Miculan (Eds.): FOSSACS 2001, LNCS 2030, pp. 409–412, 2001.

The prime event structure S constitutes the abstract level, not necessarily implemented but used to derive properties of the graph. Whereas a partial order representation stands for one execution of a distributed program, an event structure exhibits in a single object all its possible executions. In that sense event structures are close to Petri Nets with which they have strong connections [Wins. Niel. 95], while prime event structures are a special case related to traces [Maz. 87]. This set of configurations satisfies certain important domains properties not far from those of lattices : it may be called a budding lattice. Roughly speaking, its regularity allows efficient traversal algorithms.

First, the input is a distributive reactive asynchronous algorithm written in an Estelle or in a Sdl like language [Est. 88, SDL 88]. Then Marrella implements an object, some kind of network the nodes of which are automaton equipped with buffers and communicating by channels [Amb. Roz. 96]. It allows constructing $G(R)$, the unfolded-labeled graph of states of the system, actually implemented totally or piece by piece in case it is too big.

This later graph is shown to be isomorphic to the graph of configurations of a prime event structure : $G(R) \approx Conf(S)$. Therefore, using this result, efficient constructions and traversals of this graph may be implemented. Latter on, depending on the properties to be shown, this graph is folded into a smaller equivalent one.

3 Construction and Observation Strategies

As it benefits by the steady structure of the budding lattice, Marrella uses strategies to represent, reduce and examine the graph in an efficient way. First, the collection of events generated by processes is constructed on the fly, together with their dependence relation (a partial order) and their conflict relation (a binary irreflexive relation). Then two strategies are possible, depending on the properties to be tested.

Instead of the classical interleaving construction, that is heavy in space and time, we use an economical strategy that constructs a tree that covers the graph and is linear in the number of states. Here every state is constructed, thus the gain is not in the number of states, but in the number of comparisons : using an enumeration that labels the event on the fly, the algorithm "knows" which state has already been constructed and does not try to re-construct an already existent one. It is a breadth first construction with the following shape :

- Constructing the states of depth (ρ+1) starting with those of depth (ρ).
- Initialize S(ρ+1) at ϕ.

```
for all state s in S(ρ) do
    for all event e allowed in s do
        if label(s) ≪ e then
            | add s+e to S(ρ + 1)
            | label(s+e) ⟵ e
        fi
    od
od
```

The full version can be found in [Amb. Roz. 96]. The figure 1.a gives an example of such graph construction. A second strategy, called COMAX, uses concurrent steps and constructs a tree too. It consists in maximizing the number of independent transitions that are executed in parallel. This tree has less states than a covering tree, and is such that each reachable state of the original graph is covered by at least one state of the constructed tree : thus detection of deadlocks or stable properties is available. It is not already published, but a preliminary version can be found in (`http://lri.fr/~ambroise/marrella.html`). Its main new idea is that, at each step, several independent processes may act together. Then a careful investigation of the conflict relation allows to backtrack some actions when necessary. An example is given in Fig. 1.b.

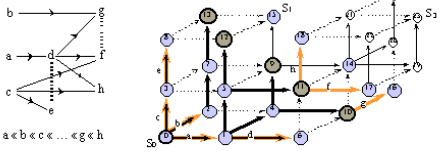


Fig. 1. (a) Atomic construction

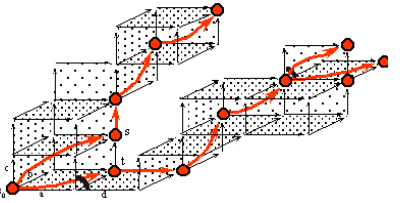


Fig. 1. (b) Traversing of the graph

4 Experiment with MARRELLA on an Embedded System

Collaboration has been initiated by electronic researchers and computer scientists. The goal was to re-write and verify the ASTRE architecture [Bou. 97], an agent based embedded system for avoiding road collision risks. This system processes data acquisition issued from the road traffic scenery, constructs a model of the environment and outputs diagnostics on collisions risks. Together, we have specified new architectures, focussing on the control scheme. The first one is a rather centralized control, corresponding to architecture with hard constraints. The second one is distributed, with relaxed constraints.

These new specifications of the ASTRE architecture have been verified with MARRELLA as well as with other tools: OBJECTGEODE, a standard commercial tool, CADP, that is based on bisimulation, SPIN, based on static partial order reductions and SPIN ULG, based on dynamic partial order reductions. We give here the results and compare them. To make pertinent comparisons, both architectures have been first described with the SDL description language. Then translators, based on IF, are used to obtain the descriptions in the language supported by each tool. The following table reports the results when using respectively OBJECTGEODE, MARRELLA, CADP, SPIN and SPIN ULG. We compare here the number of states generated by each tool in the same conditions. We give them here for the two architectures and various data flows.

Table 1. Simulation results, number of states constructed by each tool

	Architecture 1			Architecture 2		
Tools	Flow 1	Flow 2	Flow 3	Flow 1	Flow 2	Flow 3
ObjectGEODE	2028	1530	4321	19 372	475 285	4 000 000
Cadp	1 579	1 448	2 618	7 543	59 486	325 195
Spin	215	274	483	4 315	75 828	1 034 899
Spin Ulg	85	169	375	1 388	48 699	636 218
Marrella	105	78	152	81	29 486	137 863

Thus, considering the number of states, Marrella (more or less together with Spin Ulg and Cadp) is clearly among the best, the reduction ratio going from 3 to 40.

5 Conclusion

Although being only a toy written with a Lisp like language, Marrella has been carefully implemented, thus spares memory. Moreover, precise and neat algorithms benefit from the trace properties of prime event structures and thus gain in avoiding the enumeration of equivalent interleaving.

Tested comparatively with other known tools, Marrella has proved its efficiency. The question now is to evaluate whether it is easy to use for a non-specialist !

References

[Amb. Roz. 96] Ambroise, D., Rozoy, B. Marrella : a tool to analyse the graph of states. In: Parallel Processing Letters. Vol 6, n^{o}4, 583-594. (1996)

[Bou. 97] K. Bouchefra, Formal model for road traffic collision risk avoidance. In: IFAC-IFIP Conference on Control of Industrial Systems (Crujic, Borne, Ferney, Ed.). Elsevier Science Ltd. Belfort, France (1997).

[Esp. Röm. 99] J. Esparza, S. Römer, An unfolding algorithm for synchronous product of transition systems, In: Proceedings of the 12^{th} International Conference on Concurrency Theory, Concur'99, Eindhoven, The Netherlands, August 1999, LNCS. n^o1664, 2-20, 1999

[EST. 88] ESTELLE: A Formal Description Based on an Extended State Transition Model. ISO-9074, Geneva (1988)

[Maz. 87] A. Mazurkiewicz, Trace Theory, Advanced Course on Petri Nets, In: Bad Honnef, Germany LNCS n^o254, 269-324.

[SDL 88] SDL : Recommendation Z. 100, Specification and Description Language. ITU-T, Geneva (1988)

[Wins. Niel. 95] G. Winskel, M. Nielsen, Models for concurrency, In: Handbook of Logic in Computer Science (S. Abramsky, DM. Gabbay, TSE. Maibaum eds.).

Author Index