ARCoSS

LNCS 6604

Martin Hofmann (Ed.)

Foundations of Software Science and Computational Structures

14th International Conference, FOSSACS 2011
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2011
Saarbrücken, Germany, March/April 2011, Proceedings

European Joint Conferences on
Theory
and
Practice of
Software
2011

Springer

Lecture Notes in Computer Science 6604

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Martin Hofmann (Ed.)

Foundations of Software Science and Computational Structures

14th International Conference, FOSSACS 2011
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2011
Saarbrücken, Germany, March 26–April 3, 2011
Proceedings

Springer

Volume Editor

Martin Hofmann
Ludwig-Maximilians-Universität München
Institut für Informatik
Oettingenstr. 67, 80538 München, Germany
E-mail: hofmann@ifi.lmu.de

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-19804-5 e-ISBN 978-3-642-19805-2
DOI 10.1007/978-3-642-19805-2
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 20119922505

CR Subject Classification (1998): F.3, F.1, F.4, D.3, D.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

ETAPS 2011 was the 14th 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 the usual five sister conferences (CC, ESOP, FASE, FOSSACS, TACAS), 16 satellite workshops (ACCAT, BYTECODE, COCV, DICE, FESCA, GaLoP, GT-VMT, HAS, IWIGP, LDTA, PLACES, QAPL, ROCKS, SVARM, TERMGRAPH, and WGT), one associated event (TOSCA), and seven invited lectures (excluding those specific to the satellite events).

The five main conferences received 463 submissions this year (including 26 tool demonstration papers), 130 of which were accepted (2 tool demos), giving an overall acceptance rate of 28%. Congratulations therefore to all the authors who made it to the final programme! I hope that most of the other authors will still have found a way of participating in this exciting event, and that you will all continue submitting to ETAPS and contributing to make of it the best conference on software science and engineering.

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

ETAPS is a confederation in which each event retains its own identity, with a separate Programme Committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronised 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 2011 was organised by the *Universität des Saarlandes* in cooperation with:

▷ European Association for Theoretical Computer Science (EATCS)
▷ European Association for Programming Languages and Systems (EAPLS)
▷ European Association of Software Science and Technology (EASST)

It also had support from the following sponsors, which we gratefully thank: DFG Deutsche Forschungsgemeinschaft; AbsInt Angewandte Informatik GmbH; Microsoft Research; Robert Bosch GmbH; IDS Scheer AG / Software AG; T-Systems Enterprise Services GmbH; IBM Research; gwSaar Gesellschaft für Wirtschaftsförderung Saar mbH; Springer-Verlag GmbH; and Elsevier B.V.

The organising team comprised:

General Chair:	*Reinhard Wilhelm*
Organising Committee:	*Bernd Finkbeiner, Holger Hermanns* (chair), *Reinhard Wilhelm, Stefanie Haupert-Betz,* *Christa Schäfer*
Satellite Events:	*Bernd Finkbeiner*
Website:	*Hernán Baró Graf*

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

Vladimiro Sassone (Southampton, Chair), Parosh Abdulla (Uppsala), Gilles Barthe (IMDEA-Software), Lars Birkedal (Copenhagen), Michael O'Boyle (Edinburgh), Giuseppe Castagna (CNRS Paris), Marsha Chechik (Toronto), Sophia Drossopoulou (Imperial College London), Bernd Finkbeiner (Saarbrücken) Cormac Flanagan (Santa Cruz), Dimitra Giannakopoulou (CMU/NASA Ames), Andrew D. Gordon (MSR Cambridge), Rajiv Gupta (UC Riverside), Chris Hankin (Imperial College London), Holger Hermanns (Saarbrücken), Mike Hinchey (Lero, the Irish Software Engineering Research Centre), Martin Hofmann (LMU Munich), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Barbara König (Duisburg), Shriram Krishnamurthi (Brown), Juan de Lara (Madrid), Kim Larsen (Aalborg), Rustan Leino (MSR Redmond), Gerald Luettgen (Bamberg), Rupak Majumdar (Los Angeles), Tiziana Margaria (Potsdam), Ugo Montanari (Pisa), Luke Ong (Oxford), Fernando Orejas (Barcelona), Catuscia Palamidessi (INRIA Paris), George Papadopoulos (Cyprus), David Rosenblum (UCL), Don Sannella (Edinburgh), João Saraiva (Minho), Helmut Seidl (TU Munich), Tarmo Uustalu (Tallinn), and Andrea Zisman (London).

I would like to express my sincere gratitude to all of these people and organisations, the Programme Committee Chairs and members of the ETAPS conferences, the organisers of the satellite events, the speakers themselves, the many reviewers, all the participants, and Springer for agreeing to publish the ETAPS proceedings in the ARCoSS subline.

Finally, I would like to thank the Organising Chair of ETAPS 2011, Holger Hermanns and his Organising Committee, for arranging for us to have ETAPS in the most beautiful surroundings of Saarbrücken.

January 2011

Vladimiro Sassone
ETAPS SC Chair

Preface

FoSSaCS presents original papers on the foundations of software science. The Programme Committee (PC) invited submissions on theories and methods to support analysis, synthesis, transformation and verification of programs and software systems. We received 100 full paper submissions; of these, 30 were selected for presentation at FoSSaCS and inclusion in the proceedings. Also included is an invited paper on "The Search for Structure in Quantum Computation" by Prakash Panangaden, the FoSSaCS 2011 invited speaker.

Numbers of submissions and accepted papers at the last five FoSSaCS conferences—2010 (Paphos), 2009 (York), 2008 (Budapest), 2007 (Braga), 2006 (Vienna)—were 86/25, 102/30, 124/33, 103/25, 107/28, respectively.

I thank all the authors of papers submitted to FoSSaCS 2011. I thank also the members of the PC for their excellent work, as well as the external reviewers for the expert help and reviews they provided. Throughout the phases of submission, evaluation, and production of the proceedings, we relied on the invaluable assistance of the EasyChair system; we are very grateful to its developer Andrei Voronkov and his team. Last but not least, we would like to thank the ETAPS 2011 Local Organizing Committee (chaired by Holger Hermanns) and the ETAPS Steering Committee (chaired by Vladimiro Sassone) for their efficient coordination of all the activities leading up to FoSSaCS 2011.

January 2011 Martin Hofmann

Conference Organization

Programme Committee

Amal Ahmed	University of Indiana, USA
David Basin	ETH Zurich, Switzerland
Krishnendu Chatterjee	Institute of Science and Technology, Austria
Giorgio Ghelli	University of Pisa, Italy
Daniel Hirschkoff	ENS Lyon, France
Martin Hofmann	Ludwig-Maximilians-Universität, Munich (Chair), Germany
Marieke Huisman	University of Twente, The Netherlands
Petr Jančar	Technical University of Ostrava, Czech Republic
Andrew Kennedy	Microsoft Research Cambridge, UK
Barbara König	University of Duisburg-Essen, Germany
Martin Lange	University of Kassel, Germany
Francois Laroussinie	LIAFA (Paris 7), France
Markus Lohrey	University of Leipzig, Germany
Heiko Mantel	TU Darmstadt, Germany
Marino Miculan	University of Udine, Italy
Andrzei Murawski	University of Oxford, UK
Peter O'Hearn	Queen Mary, University of London, UK
Dirk Pattinson	Imperial College London, UK
Olivier Serre	LIAFA (Paris 7 and CNRS), France
Natarajan Shankar	SRI International, Menlo Park, USA
Thomas Streicher	TU Darmstadt, Germany
Igor Walukiewicz	University of Bordeaux, France
Nobuko Yoshida	Imperial College London, UK
Greta Yorsh	IBM T.J. Watson Research Center, USA

External Reviewers

Faris Abou-Saleh
Markus Aderhold
Fabio Alessi
Thorsten Altenkirch
Andrea Asperti
Fauzi Atig
Giorgio Bacci
Bahareh Badban
David Baelde
Steffen van Bakel
Nick Benton
Josh Berdine
Federico Bergenti
Ulrich Berger
Marco Bernardo
Gavin Bierman
Udi Boker
Filippo Bonchi
Viviana Bono
Laura Bozzelli
Benjamin Braatz
Tomas Brazdil
James Brotherston
Vaclav Brozek
H.J. Sander Bruggink
Antonio Bucciarelli
Arnaud Carayol

Franck Cassez
Karlis Cerans
Iliano Cervesato
Arthur Charguéraud
Thomas Chatain
Konstantinos Chatzikokolakis
Thomas Colcombet
Sandro Coretti
Andrea Corradini
Silvia Crafa
Pieter Cuijpers
Fredrik Dahlqvist
Mohammad Torabi Dashti
Claire David
Stéphane Demri
Yuxin Deng
Agostino Dovier
Laurent Doyen
Ross Duncan
Gilberto Filè
Emmanuel Filiot
Seth Fogarty
Vojtech Forejt
Luca Fossati
Laurent Fribourg
Oliver Friedmann
Fabio Gadducci
Vashti Galpin
Pierre Ganty
Richard Gay
Alfons Geser
Dan Ghica
Pietro Di Gianantonio
Hugo Gimbert
Marco Giunti
Rob van Glabbeek
Stefan Göller
Benjamin Grégoire
Dilian Gurov
Matthew Hague
Peter Hancock
Ichiro Hasuo
Frédéric Herbreteau
Tom Hirschowitz
Dieter Hofbauer
Florian Horn
Mathias Hülsbusch
Clément Hurlin
Michael Huth
Samuel Hym
Pierre Hyvernat
Roshan James
Łukasz Kaiser
Daniel Kirsten
Felix Klaedtke
Bartek Klin
Naoki Kobayashi
Boris Koepf
Martin Kot
Vasileios Koutavas
Jean Krivine
Clemens Kupke
Manfred Kufleitner
Alexander Kurz
James Laird
Ivan Lanese
Markus Latte
Bjoern Lellmann
Serguei Lenglet
Marina Lenisa
Martin Leucker
Paul Blain Levy
Patrick Lincoln
Christof Löding
Sylvain Lombardy
Michele Loreti
Alexander Lux
Pasquale Malacaria
Bodo Manthey
Giulio Manzonetto
Radu Mardare
Nicolas Markey
Ralph Matthes
Richard Mayr
Conor McBride
Catherine Meadows
Ingmar Meinecke
Paul-André Melliès
Stefan Milius
Tri Ngo Minh
Rasmus Ejlers Møgelberg
Esfandiar Mohammadi
Maarten de Mol
Bruno Conchinha Montalto
Jean-Yves Moyen
Markus Müller-Olm
Berndt Muller
Robert Myers
Sebastian Nanz
Damian Niwinski
Claudio Orlandi
Sam Owre
Michele Pagani
Luca Paolini
Gennaro Parlato
Sophie Pinchinat
Nir Piterman
Andrew Pitts
Francesca Poggiolesi
Erik Poll
John Power
Vinayak Prabhu
Sylvain Pradalier
Karin Quaas
Julian Rathke
Jason Reed
Klaus Reinhardt
Bernhard Reus
Noam Rinetzky
Mehrnoosh Sadrzadeh
Sylvain Salvati
Arnaud Sangnier
Alexis Saurin
Zdenek Sawa
Ivan Scagnetto
Sven Schewe
Alan Schmitt
Ulrich Schöpp
Lutz Schröder
Jan Schwinghammer
Stefan Schwoon
Mihaela Sighireanu

Paweł Sobocinski
Matthieu Sozeau
Heiko Spies
Christoph Sprenger
Barbara Sprick
Jiři Srba
Artem Starostin
Sam Staton
Henning Sudbrock
Kohei Suenaga
Grégoire Sutre
Nathalie Sznajder
Claus Thrane
Mark Timmer
Simone Tini
Alwen Tiu
Ashish Tiwari
Szymon Torunczyk
Mathieu Tracol
Ashutosh Trivedi
Aaron Turon
Nikos Tzevelekos
Daniele Varacca
Jamie Vicary
Daniel Wagner
Johannes Waldmann
Robert Walters
Florian Widmann
Thomas Wies
Thomas Wilke
Glynn Winskel
James Worrell
Peng Wu
Gianluigi Zavattaro
Marc Zeitoun
Wieslaw Zielonka
Damien Zufferey

Table of Contents

Coalgebra and Computability

Type Theory

Process Calculi

Automata Theory

Semantics

Binding

Security

Program Analysis

The Search for Structure in Quantum Computation

Prakash Panangaden

School of Computer Science,
McGill University and Computing Laboratory,
University of Oxford

Abstract. I give a non-comprehensive survey of the categorical quantum mechanics program and how it guides the search for structure in quantum computation. I discuss the example of measurement-based computing which is one of the successes of such an enterprise and briefly mention topological quantum computing which is an inviting target for future research in this area.

1 Introduction

Quantum computation has attracted (and repelled!) many members of the computer science community. On the one hand, people have been excited by new possibilities: cryptography based on physics rather than on unproven complexity assumptions [1], new algorithmic paradigms [2], error correction [3], solutions to hitherto "impossible" distributed computation tasks [4] and dramatic new possibilities like teleportation [5]. On the other hand, people have been disturbed by the strangeness of quantum mechanics which has rendered many of the traditional tools of theoretical computer science inapplicable.

In this paper I will attempt to convey something of the strangeness of quantum mechanics as well as some of the attempts being made to come to grips with quantum computation. The subjects of quantum algorithms and quantum information theory are flourishing and there are dramatic new results arriving at a regular pace. For the logic and semantics community it has been a rougher ride. Defining programming languages has not been routine [6,7,8,9,10] and there are many things that we do not understand yet. Entirely new challenges have been posed for type systems. It is only this year that we have a decent definition of weak bisimulation for a quantum process algebra. New models of computation – like measurement-based computing [11] and topological quantum computing – have emerged from the physics community which have posed challenges to the theoretical computer science community to formalize properly.

I will survey some of these developments and then pose some challenges for the future.

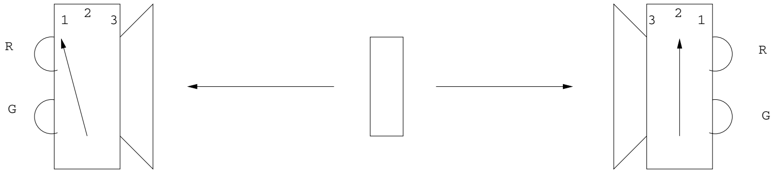
2 Strange Features of Quantum Mechanics

By now most people in theoretical computer science understand the "standard" features of quantum mechanics: the state space is a Hilbert space, the evolution

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 1–11, 2011.

of a system is described by a unitary map, observables are hermitian operators (hence their eigenvalues are real) and the outcome of a measurement is probabilistic and yields one of the eigenvalues of the observable being measured. Even the originally surprising aspects of quantum mechanics are no longer surprises: systems can be in superpositions of states and measurement outcomes are not determined.

The concept of non-locality of information continues to confound many people. Even Einstein referred to this as "spooky action at a distance." The point is that it is possible for the values of an observable to be not even defined in some states. The following thought experiment, due to Mermin [12], illustrates this dramatically. Consider the apparatus schematically shown below:



Set-up for Mermin's thought experiment

In the centre there is a source of particles that emits them in pairs travelling in opposite directions. The particles are detected by detectors that are separated far enough that signals cannot pass between them. There are two detectors each with 3 settings and 2 indicators: bulbs that flash red and green respectively. The detectors are set independently and uniformly at random. The detectors are not connected to each other or to the source.

Whatever the setting on a detector, the red or the green lights flash with equal probability, but never both at the same time. When the settings are the same the two detectors **always** agree. When the settings are different the detectors agree $\frac{1}{4}$ of the time! Why is this strange?

How could the detectors **always** agree when the settings are the same, even though the actual colour seems to be chosen at random? There must be some "hidden" property of the particles that determines which colour is chosen for each setting; the two correlated particles must be identical with respect to this property, *whether or not the switches are set the same way.* Let us write GGR mean that for the three settings, $1, 2, 3$, the detectors flash green, green and red respectively for a type GGR particle. We are assuming it is meaningful to attribute properties like GGR to a particle.

Suppose that the settings are different and we have an RRG particle: then for two of the possible settings ($1, 2$ and $2, 1$) the same colour flashes and for the other four settings the colours are different. Thus $\frac{1}{3}$ of the time the colours must match. This applies for any of the combinations: $RRG, RGR, GRR, GGR, GRG, RGG$. For particles of type RRR and GGG the colours **always** match whatever the settings. The inescapable conclusion is that *whatever the distribution of particle types* the probability that the lights match when the settings are different is at least $\frac{1}{3}$! This just ain't what we see in nature.

We made some basic assumptions about detectors:

Locality: what happens at one detector cannot alter what happens at the other,

Causality: a detector cannot predict the future sequence of particles and alter its behaviour.

No ordinary probabilistic automaton or MDP or whatever your favourite state-based model is, can reproduce the observed behaviour without breaking locality or causality. Capturing locality in an automaton means that the states of the system are the cross product of the states of each detector and the behaviour of each detector depends only on the local state.

The inequality,

$$Prob(\text{lights agree}|\text{settings different}) \geq \frac{1}{3},$$

is a simple special case of Bell's inequality. Quantum mechanics predicts that this inequality is violated. Bell's inequality has been **experimentally tested** and it is plainly violated but the experiments agree with the predictions of quantum mechanics which also predicts that the inequality is violated.

The point of this discussion is that the probabilistic nature of quantum mechanics does not arise as an abstraction of things that could be known. State is not enough to predict the outcomes of measurements; **state is enough to predict evolution to new states.**

These non-local effects are what give quantum computation its power. Teleportation is just a dramatic example of this.

3 Categorical Quantum Mechanics and Graphical Calculi

What formal techniques can be brought to bear on these kinds of systems? A key contribution of the logic and semantics community is compositionality. The whole point of denotational semantics was to provide a compositional understanding of systems. In the case of quantum mechanics we need to understand how to describe composite systems. It was known since the days of von Neumann [13] back in 1932 that the right way to combine the Hilbert spaces of two systems is by means of the tensor product. The tensor product of Hilbert spaces is quite an elaborate construction. It requires not just the construction of the tensor product of vector spaces, but the definition of an inner product followed by a completion process which is topological in nature. Ultimately, von Neumann was unhappy with the elaborate mathematical machinery of Hilbert spaces and sought to retreat to some new fundamental logical principles for axiomatizing quantum mechanics. This led to the quantum logic programme [14] where the algebra of projection operators on a Hilbert space became the inspiration for the logic.

A huge amount of work was spent on the quantum logic enterprise [15], but in the end it is fair to say that it floundered on its inability to give a clean account of compositionality. Nevertheless logical ideas are indeed fundamental and the breakthrough idea was simple: axiomatize tensor product in the simplest way possible. This is due to Abramsky and Coecke [16] in a paper which appeared in the IEEE Symposium on Logic in Computer Science in 2004. As so often happens, categorists had invented the right notions already: monoidal categories. Though it may seem to many to not be an improvement to use fancy category theory instead of fancy functional analysis, the fact is that a purely elementary account can be given based on very simple process intuitions. A very accessible account of this viewpoint is contained in a set of lecture notes by Bob Coecke appropriately entitled, "Kindergarten Quantum Mechanics" [17].

At its most basic level then quantum mechanics is about how to hook up processes, either by connecting them in sequence or placing them side by side in parallel or combinations thereof. One can model processes as arrows in a category; the objects of the categories represent the input and output types of the processes. Categorical composition models sequential composition and the tensor product models parallel composition. Indeed one can intuit the correct axioms for tensor just from this modelling.

What is so special about quantum mechanics? Surely these are the same axioms one would use for any kind of process. One can easily whip up a model of, for example, asynchronous dataflow, as a monoidal category and indeed this has been done. Clearly monoidal categories are far too general; one needs to identify other crucial ingredients of quantum mechanics and incorporate them into the axioms. The Abramsky-Coecke paper identified duality of input and output as a crucial feature and the resulting class of categories are called by them strongly compact-closed categories. It does not matter what the algebraic axioms are because the essence of this structure is captured beautifully by a graphical calculus [18].

Graphical notions are now common in physics having been famously introduced by Feynman for keeping track of certain integrals that arise in quantum electrodynamics [19,20]. Penrose introduced a beautiful graphical notation for tensor analysis [21] which was placed on a firm mathematical footing by Joyal and Street [22]. I highly recommend the excellent survey by Selinger [18] in addition to the lecture notes of Coecke mentioned above.

The fundamental theorem [22] of the graphical language states that

Theorem 1. *An equation holds between terms in the morphism language of monoidal categories if and only if it holds up to planar isotopy in the graphical language.*

This means that diagrammatic manipulations can replace algebraic gymnastics. Furthermore, many equations turn out to be trivial in the graphical language.

There are two fundamental structural aspects of quantum mechanics that are captured by the categorical formalism. The first states that "objects have duals"; in categorical jargon these are called *autonomous categories.* In diagrammatic terms it means that every object A has a dual A^* and one can reverse arrows using duality. In a closed category one *bend arrows around using the duals.* This gives quantum mechanics its reversible aspect. Finally, there is a structure called a "dagger"; this is also a way of changing the direction of an arrow and it closely corresponds to the adjoint operation in linear algebra. It is the presence of this dagger structure that signals the role of complex numbers in quantum mechanics. Analogues of the fundamental theorem hold [18] for all these richer types of monoidal categories.

There are at least three important reasons for working at this level of abstractness. First, one can explore variants of quantum mechanics that are close to but not exactly the same as standard quantum mechanics. For example, the category **Rel** of sets and binary relations is an impoverished example of "toy" quantum mechanics. One can then explore what features one really needs for various phenomena to manifest themselves and thus understand what is the essence of quantum mechanics. For example, one can ask whether teleportation could be done in **Rel**; it cannot! Another striking exploration of this kind is the work by Coecke, Edwards and Spekkens [23] on formalizing a certain toy model originally due to Spekkens and showing that there is a group-theoretic reason for the difference between the two models.

Secondly, one can explore more exotic phenomena like multipartite quantum entanglement [24] or interacting quantum observables [25] from a graphical viewpoint and even find graph theoretical characterizations of these structures. As soon as one has three or more entangled states the situation becomes much more complicated. There are some preliminary hints of the role of these states in distributed computing tasks [4] but clearly much remains to be understood and structural understanding will guide the way.

Finally, one can address foundational questions of quantum information and quantum mechanics. In very striking recent work Abramsky [26] has analyzed the issue of hidden variables in the toy relational model and has shown that many of the no-go theorems survive even when one has only a "possibilistic" view of nondeterminism.

4 Measurement-Based Computing

I now turn to a new computational model and analyze it from the viewpoint of theoretical computer science. Traditionally, the main framework to explore quantum computation has been the circuit model [27], based on unitary evolution. This is very useful for algorithmic development and complexity analysis [28]. There are other models such as quantum Turing machines [29] among a variety of others. They are all proved to be equivalent from the point of view of expressive power. For higher-order sequential programming we have the typed λ-calculus which stands out as *the* canonical programming language but there is no such language for quantum computation.

Recently physicists have introduced novel ideas based on the use of measurement and entanglement to perform computation [30,11,31]. This is very different from the circuit model where measurement is done only at the end to extract classical output. In measurement-based quantum computation the main operation to manipulate information and control computation is measurement. This is surprising because measurement creates indeterminacy, yet it is used to express deterministic computation defined by a unitary evolution.

The idea of computing based on measurements emerged from the teleportation protocol [5]. The goal of this protocol is for an agent to transmit an unknown qubit to a remote agent without actually sending the qubit. This protocol works by having the two parties share a maximally entangled state called a Bell pair. The parties perform *local* operations – measurements and unitaries – and communicate only classical bits. Remarkably, from this classical information the second party can reconstruct the unknown quantum state. In fact one can actually use this to compute via teleportation by choosing an appropriate measurement [30]. This is the key idea of measurement-based computation.

It turns out that the above method of computing is actually universal. This was first shown by Gottesman and Chuang [30] who used two-qubit measurements and given Bell pairs. The one-way computer was then invented by Raussendorf and Briegel [11,32] which used only single-qubit measurements with a particular multi-party entangled state called the cluster state.

The computation proceeds in a sequence of phases; in the first phase a collection of qubits are set up in a standard entangled state. Then measurements are applied to individual qubits and the outcomes of the measurements may be used to determine further adaptive measurements. Finally – again depending on measurement outcomes – local unitary operators, called corrections, are applied to some qubits; this allows the elimination of the indeterminacy introduced by measurements. The phrase "one-way" is used to emphasize that the computation is driven by irreversible measurements.

There are at least two reasons to take measurement-based models seriously: one conceptual and one pragmatic. The main pragmatic reason is that the *one-way* model is believed by physicists to lend itself to easier implementations [33,34,35]. Physicists have investigated various properties of the cluster state and have accrued evidence that the physical implementation is scalable and robust against decoherence [36]. Conceptually the measurement-based model highlights the role of entanglement and separates the quantum and classical aspects of computation; thus it clarifies, in particular, the interplay between classical control and the quantum evolution process.

When this model was first presented it was couched in the language of Hamiltonians and evolution of quantum states. The design of even simple gates seemed magical and an exercise in combinatorial ingenuity. Most importantly, the "proof" of universality consisted in showing that the basic gates of the circuit model could be implemented in the one-way model with the implicit understanding that any network could then be encoded. What was missing was a *compositional translation* and a proof that the *semantics* of the circuit was preserved.

Our approach to understanding the structural features of measurement-based computation was to develop a formal calculus [37]. One can think of this as an "assembly language" for measurement-based computation. It was the first programming framework specifically based on the one-way model. In our paper we developed a language for programs (we called them "patterns") as sequences of entanglements, measurements, and local corrections. We gave a careful treatment of the composition and tensor product (parallel composition) of programs and we have denotational semantics and operational semantics for these programs. In this framework we were able to give a proof of universality. In fact, we were able to modify the framework in apparently small ways but these had the effect of greatly simplifying the implementations of circuits. More precisely we had an extended notion of pattern, where inputs and outputs may overlap in any way one wants them to, and this results in more efficient – in the sense of using fewer qubits – implementations of unitaries. Specifically, our universal set consists of patterns using only 2 qubits. From it we obtained a 3 qubit realization of the R_z rotations and a 14 qubit realization for the controlled-U family: a significant reduction over the hitherto known implementations [38].

However, there were more benefits to be gained from the exploration of this structural view of measurement-based computing. We introduced a *calculus* of patterns based on the special algebraic properties of the entanglement, measurement and correction operators. These allowed local rewriting of patterns and we showed that this calculus is sound in that it preserves the interpretation of patterns. Most importantly, we derived from it a simple algorithm by which any general pattern can be put into a standard form where entanglement is done first, then measurements, then corrections. We call this *standardization.*

The consequences of the existence of such a procedure are far-reaching. Since entangling comes first, one can prepare the entire entangled state needed during the computation right at the start: one never has to do “on the fly” entanglements. Furthermore, the rewriting of a pattern to standard form reveals parallelism in the pattern computation. In a general pattern, one is forced to compute sequentially and to strictly obey the command sequence, whereas, after standardization, the dependency structure is relaxed, resulting in lower computational depth complexity [39].

Perhaps the most striking development in the theory of measurement-based computing is the discovery of the concept of *flow* by Danos and Kashefi [40]. A variety of methods for constructing measurement patterns had already been proposed that guarantee determinism by construction. They introduced a graph-theoretic condition on the states used in measurement patterns that guarantees a strong form of deterministic behavior for a class of one-way measurement patterns defined over them. Remarkably, their condition bears only on the geometric structure of the entangled graph states. This condition singles out a class of patterns with flow, which is stable under sequential and parallel compositions and is large enough to realize all unitary and unitary embedding maps.

Patterns with flow have interesting additional properties. First, they are uniformly deterministic, in the sense that no matter what the measurements are made, there is a set of corrections, which depends only on the underlying geometry, that will make the global behaviour deterministic. Second, all computation branches have equal probabilities, which means in particular, that these probabilities are independent of the inputs, and as a consequence, one can show that all such patterns implement unitary embeddings. Third, a more restricted class of patterns having both flow and reverse flow supports an operation of adjunction, corresponding to time-reversal of unitary operations. This smaller class implements all and only unitary transformations.

In the categorical quantum framework Coecke and Duncan [25] have looked at interacting quantum observables in a diagrammatic formalism. There is a very pleasing encoding of the one-way model into this framework and many of the algebraic equations of the one-way model can be done by graphical manipulations. It would be fascinating to understand flow and its relation to causality in this way.

5 Topological Quantum Computing

I would like to close with a brief mention of a wide open area: topological quantum computing. In quantum computation one is required to make excruciatingly precise manipulations of qubits while preserving entanglement when all the while the environment is trying to destroy entanglement. A very novel suggestion by Kitaev [41] proposes the use of a new type of hypothetical particle called an anyon [42,43] which has a topological character.

The mathematics and physics of anyons probe the most fundamental principles of quantum mechanics. They involve a fascinating mix of experimental phenomena (the fractional quantum Hall effect), topology (braids), algebra (Temperley-Lieb algebra, braid group and category theory) and quantum field theory. Because of their topological nature, it is hoped that one can use them as *stable* realizations of qubits for quantum computation, as proposed originally by Kitaev. The idea of topological quantum computation has been actively pursued by Freedman et al. [44,45].

There is rich algebraic structure to be understood and, as with the measurement-based model, we need a computational handle on this. We have nothing like a calculus or a standardization theorem. What is clear is that the basic interactions of the anyons can only be expressed in the categorical language. One needs a rather rich kind of categorical structure called a modular tensor category [46]. An expository account of this subject is given in [47]. Understanding topological quantum computing from the viewpoint of computational structure remains a big open problem.

Acknowledgments

I would like to thank McGill University and EPSRC for its generous support during my sabbatical leave, NSERC (Canada) and the Office of Naval Research (USA) for funding this research and the Computing Laboratory of the University of Oxford for its hospitality. I have benefitted from discussions with Samson Abramsky, Bob Coecke, Vincent Danos, Ross Duncan, Julia Evans, Elham Kashefi, Eric Paquette and Jamie Vicary.

References

1. Bennett, C.H., Brassard, G.: Quantum cryptography: Public-key distribution and coin tossing. In: Proceedings of IEEE International Conference on Computers, Systems and Signal Processing, Bangalore, India, pp. 175–179 (December 1984)
2. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Goldwasser, S. (ed.) Proc. 35nd Annual Symposium on Foundations of Computer Science, pp. 124–134. IEEE Computer Society Press, Los Alamitos (1994)
3. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Computing 26, 1484–1509 (1997)
4. D'Hondt, E., Panangaden, P.: The computational power of the W and GHZ states. Quantum Information and Computation 6(2), 173–183 (2006)
5. Bennett, C.H., Brassard, G., Crepeau, C., Josza, R., Peres, A., Wootters, W.: Teleporting an unknown quantum state via dual classical and epr channels. Phys. Rev. Lett. 70, 1895–1899 (1993)
6. Gay, S.: Quantum programming languages: Survey and bibliography. Bulletin of the EATCS 86, 176–196 (2005)
7. Selinger, P.: Towards a quantum programming language. Mathematical Structures in Computer Science 14(4), 527–586 (2004)
8. Selinger, P.: A brief survey of quantum programming languages. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998, pp. 1–6. Springer, Heidelberg (2004)
9. Selinger, P., Valiron, B.: Quantum lambda calculus. In: Gay, S., Mackie, I. (eds.) Semantic Techniques in Quantum Computation. Cambridge University Press, Cambridge (2009) (to appear)
10. van Tonder, A.: A lambda calculus for quantum computation. Siam Journal on Computing 33(5), 1109–1135 (2004)
11. Raussendorf, R., Briegel, H.J.: A one-way quantum computer. Phys. Rev. Lett. 86, 5188–5191 (2001)
12. Mermin, D.: Boojums all the way through. Cambridge University Press, Cambridge (1990)
13. von Neumann, J.: Mathematisch Grunglagen der Quantenmechanik. Springer, Heidelberg (1932); English translation. Princeton University Press, Princeton (1955)
14. Birkhoff, G., von Neumann, J.: The logic of quantum mechanics. Annals of Mathematics 37(4), 823–843 (1936)
15. Piron, C.: Foundations of quantum physics. W. A. Benjamin (1976)
16. Abramsky, S., Coecke, B.: A categorical semantics of quantum protocols. In: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science: LICS 2004, pp. 415–425. IEEE Computer Society, Los Alamitos (2004)
17. Coecke, B.: Kindergarten quantum mechanics (2005), available on the ArXivquant-ph/0510032
18. Selinger, P.: A survey of graphical languages for monoidal categories. In: New Structures for Physics, pp. 289–356. Springer, Heidelberg (2010)
19. Feynman, R.P.: The theory of positrons. Physical Review 76, 749–759 (1949)
20. Feynman, R.P.: The space-time approach to quantum electrodynamics. Physical Review 76, 769–789 (1949)

21. Penrose, R.: Applications of negative dimensional tensors. In: Welsh, D.J.A. (ed.) Combinatorial Mathematics and its Applications. Academic Press, London (1971)
22. Joyal, A., Street, R.: The geometry of tensor calculus. Advances in Mathematics 88, 55–112 (1991)
23. Coecke, B., Edwards, B., Spekkens, R.: The group theoretic origin of non-locality for qubits. Technical Report RR-09-04, OUCL (2009)
24. Coecke, B., Kissinger, A.: The compositional structure of multipartite quantum entanglement. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 297–308. Springer, Heidelberg (2010)
25. Coecke, B., Duncan, R.: Interacting quantum observables. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 298–310. Springer, Heidelberg (2008)
26. Abramsky, S.: Relational hidden variables and non-locality, arXiv:1007.2754 (July 2010)
27. Deutsch, D.: Quantum computational networks. Proc. Roy. Soc. Lond. A 425 (1989)
28. Bernstein, E., Vazirani, U.: Quantum complexity theory. SIAM Journal of Computing 5(26) (1997)
29. Deutsch, D.: Quantum theory, the Church-Turing Principle and the universal quantum computer. Proc. Roy. Soc. Lond. A 400, 97 (1985)
30. Gottesman, D., Chuang, I.L.: Quantum teleportation is a universal computational primitive. Nature 402 (1999)
31. Nielsen, M.A.: Universal quantum computation using only projective measurement, quantum memory, and preparation of the 0 state. Physical Review A 308 (2003)
32. Raussendorf, R., Browne, D.E., Briegel, H.J.: Measurement-based quantum computation on cluster states. Phys. Rev. A 68(2), 022312 (2003)
33. Nielsen, M.A.: Optical quantum computation using cluster states. Physical Review Letters 93 (2004), quant-ph/0402005
34. Childs, A.M., Leung, D.W., Nielsen, M.A.: Unified derivations of measurement-based schemes for quantum computation. Physical Review A 71 (2005), quant-ph/0404132
35. Browne, D.E., Rudolph, T.: Resource-efficient linear optical quantum computation. Physical Review Letters 95 (2005), quant-ph/0405157
36. Hein, M., Eisert, J., Briegel, H.J.: Multi-party entanglement in graph states. Physical Review A 69 (2004), quant-ph/0307130
37. Danos, V., Kashefi, E., Panangaden, P.: The measurement calculus. Journal Of The Association of Computing Machinery 52(2), article 8 (April 2007)
38. Vincent Danos, E.K., Panangaden, P.: Parsimonious and robust realizations of unitary maps in the one-way model. Physical Review A 72, 064301 (2005)
39. Broadbent, A., Kashefi, E.: Parallelizing quantum circuits. Theoretical Computer Science 410(26), 2489–2510 (2009)
40. Danos, V., Kashefi, E.: Determinism in the one-way model. Physical Review A 74(5), 6 (2006)
41. Kitaev, A.: Fault-tolerant quantum computation by anyons. Ann. Phys. 303(1), 3–20 (2003)
42. Wilczek, F.: Magnetic flux, angular momentum, and statistics. Phys. Rev. Lett. 48(17), 1144–1146 (1982)

43. Wilczek, F.: Quantum mechanics of fractional-spin particles. Phys. Rev. Lett. 49(14), 957–959 (1982)
44. Freedman, M.H., Larsen, M., Wang, Z.: A modular functor which is universal for quantum computation. Communications in Mathematical Physics 227(3), 605–622 (2002)
45. Freedman, M.H., Kitaev, A., Larsen, M., Wang, Z.: Topological quantum computing. Bulletin of the AMS 40(1), 31–38 (2003)
46. Bakalov, B., Kirillov, A.: Lectures on tensor categories and modular functors. American Mathematical Society in University Lecture Series (2001)
47. Panangaden, P., Paquette, E.: A categorical presentation of quantum computation with anyons. In: New Structures for Physics, pp. 983–1026. Springer, Heidelberg (2010)

Coalgebraic Walks, in Quantum and Turing Computation

Bart Jacobs

Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, The Netherlands
www.cs.ru.nl/B.Jacobs

Abstract. The paper investigates non-deterministic, probabilistic and quantum walks, from the perspective of coalgebras and monads. Non-deterministic and probabilistic walks are coalgebras of a monad (powerset and distribution), in an obvious manner. It is shown that also quantum walks are coalgebras of a new monad, involving additional control structure. This new monad is also used to describe Turing machines coalgebraically, namely as controlled 'walks' on a tape.

1 Introduction

Coalgebras have emerged in theoretical computer science as a generic formalism for state-based computing, covering various flavours of computation, like deterministic, non-determinstic, probabilistic *etc.* In general, a coalgebra is a transition map of the form $X \longrightarrow \boxed{\cdots X \cdots X \cdots}$, where X is the state space and the box captures the form of computation involved. For instance, it is a powerset $\mathcal{P}(X)$ in case of non-determinism; many other coalgebraic classifications of systems are described in [11,1]. More formally, this box is a functor, or often even a monad (in this paper) giving composition as monoid structure on coalgebras. A question that is open for a long time is whether Turing machines can also be modeled coalgebraically. More recently, the same question has been asked for quantum computing.

This paper addresses both these questions and provides positive answers via illustrations, starting from the notion of a random walk. Such walks exist in non-deterministic, probabilistic and quantum form. A first goal is to describe all three variants in a common (coalgebraic) framework, using monads. This effort focuses on the quantum case, and leads to a new construction for monads (see Proposition 2) that yields an appropriate monad for quantum walks, involving a separate control structure.

Since quantum computation is inherently reversible, the framework of dagger categories is needed. Examples of such categories are described in Section 5, via suitable relations that capture 'bi-coalgebraic' computations. Among the different kinds of walks, only the quantum walks give rise a unitary map.

Finally, an analogy is observed between quantum walks and Turing machines: both involve a road/tape on which movement is steered by a separate control structure. This will be captured coalgebraically, via the newly defined monads.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 12–26, 2011.

The approach of the paper is rather phenomenological, focusing on examples. However, the material is supported by two general results (Propositions 2 and 3), one of which is moved to the appendix; it describes how coalgebras of a monad, with Kleisli composition, form a monoid in categories of algebras of the monad.

2 Three Monads for Computation Types

Category theory, especially the theory of monads, plays an important role in the background of the current paper. The presentation however is intended to be accessible—to a large extent—without familiarity with monads. We do use three particular monads extensively, namely the powerset, multiset, and distribution monad, and so we describe them here explicitly—without making their monad structure explicit; *cognoscenti* will have no problem filling in this structure themselves.

The first monad is the finite powerset $\mathcal{P}_{\mathit{fin}}(X) = \{U \subseteq X \mid U \text{ is finite}\}$. Next, a multiset is like a subset except that elements may occur multiple times. Hence one needs a way of counting elements. Most generally this can be done in a semiring, but in the current setting we count in the complex numbers $\mathbb{C}$. Thus the collection of (complex-valued) multisets of a set X is defined in terms of formal linear combinations of elements of X, as in:

$$\mathcal{M}(X) = \Big\{ z_1|x_1\rangle + \cdots + z_n|x_n\rangle \;\Big|\; z_i \in \mathbb{C} \text{ and } x_i \in X \Big\}. \tag{1}$$

Such a multiset $\sum_i z_i|x_i\rangle \in \mathcal{M}(X)$ can equivalently be described as a function $X \to \mathbb{C}$ with finite support (*i.e.* with only finitely many non-zero values).

The "ket" notation $|x\rangle$, for $x \in X$, is just syntactic sugar, describing x as singleton multiset. It is Dirac's notation for vectors, that is standard in physics. The formal combinations in (1) can be added in an obvious way, and multiplied with a complex number. Hence $\mathcal{M}(X)$ is a vector space over $\mathbb{C}$, namely the free one on X.

The distribution monad $\mathcal{D}$ contains formal *convex* combinations:

$$\begin{aligned} &\mathcal{D}(X) \\ &= \Big\{ r_1|x_1\rangle + \cdots + r_n|x_n\rangle \;\Big|\; r_i \in [0,1] \text{ with } r_1 + \cdots + r_n = 1 \text{ and } x_i \in X \Big\} \end{aligned} \tag{2}$$

Such a convex combination is a discrete probability distribution on X.

Coalgebra provides a generic way of modeling state-based systems, namely as maps of the form $X \to T(X)$, where T is a functor (or often a monad). Basically, we only use the terminology of coalgebras, but not associated notions like bisimilarity, finality or coalgebraic modal logic. See [11] for more information.

3 Walk the Walk

This section describes various ways of walking on a line—and not, for instance, on a graph—using non-deterministic, probabilistic or quantum decisions about next steps. Informally, one can think of a drunkard moving about. His steps are discrete, on a line represented by the integers $\mathbb{Z}$.

3.1 Non-deterministic Walks

A system for non-deterministic walks is represented as a coalgebra $s\colon \mathbb{Z} \to \mathcal{P}_{\mathit{fin}}(\mathbb{Z})$ of the finite powerset monad $\mathcal{P}_{\mathit{fin}}$. For instance, the one-step-left-one-step-right walk is represented via the coalgebra:

$$s(k) = \{k-1, k+1\}$$

In such a non-deterministic system both possible successor states $k-1$ and $k+1$ are included, without any distinction between them. The coalgebra $s\colon \mathbb{Z} \to \mathcal{P}_{\mathit{fin}}(\mathbb{Z})$ forms an endomap $\mathbb{Z} \to \mathbb{Z}$ in the Kleisli category $\mathcal{K}\ell(\mathcal{P}_{\mathit{fin}})$ of the powerset monad. Repeated composition $s^n = s \bullet \cdots \bullet s\colon \mathbb{Z} \to \mathbb{Z}$ can be defined directly in $\mathcal{K}\ell(\mathcal{P}_{\mathit{fin}})$. Inductively, one can define s^n via Kleisli extension $s^\#$ as in:

$$\begin{aligned} s^0 &= \{-\} \\ s^{n+1} &= s^\# \circ s^n \end{aligned} \qquad \text{where} \qquad \begin{array}{rl} s^\#\colon & \mathcal{P}_{\mathit{fin}}(\mathbb{Z}) \longrightarrow \mathcal{P}_{\mathit{fin}}(\mathbb{Z}) \\ \text{is} & U \longmapsto \bigcup\{s(m) \mid m \in U\}. \end{array}$$

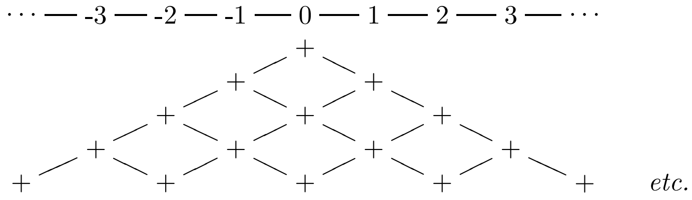
Thus, $s^n(k) \subseteq \mathbb{Z}$ describes the points that can be reached from $k \in \mathbb{Z}$ in n steps:

$$\begin{aligned} s^0(k) &= \{k\} \\ s^1(k) &= s(k) \;=\; \{k-1, k+1\} \\ s^2(k) &= \bigcup\{s(m) \mid m \in s(k)\} \;=\; s(k-1) \cup s(k+1) \\ &= \{k-2, k\} \cup \{k, k+2\} \;=\; \{k-2, k, k+2\} \\ s^3(k) &= s(k-2) \cup s(k) \cup s(k+2) \;=\; \{k-3, k-1, k+1, k+3\} \qquad \textit{etc.} \end{aligned}$$

After n iterations we obtain a set with $n+1$ elements, each two units apart:

$$s^n(k) = \{k-n, k-n+2, k-n+4, \ldots, k+n-2, k+n\}.$$

Hence we can picture the non-deterministic walk, starting at $0 \in \mathbb{Z}$ by indicating the elements of $s^n(0)$ successively by $+$ signs:



(3)

What we have used is that coalgebras $X \to \mathcal{P}_{\mathit{fin}}(X)$ carry a monoid structure given by Kleisli composition. The set $\mathcal{P}_{\mathit{fin}}(X)$ is the free join semilattice on X. The set of coalgebras $X \to \mathcal{P}_{\mathit{fin}}(X)$ then also carries a semilattice structure, pointwise. These two monoid structures (join and composition) interact appropriately, making the set of coalgebras $X \to \mathcal{P}_{\mathit{fin}}(X)$ a semiring. This follows from a quite general result about monads, see Proposition 3 in the appendix. The semiring structure is used in Section 7 when we consider matrices of coalgebras.

3.2 Probabilistic Walks

Probabilistic walks can be described by replacing the powerset monad $\mathcal{P}_{\mathit{fin}}$ by the (uniform) probability distribution monad $\mathcal{D}$, as in:

$$\mathbb{Z} \xrightarrow{d} \mathcal{D}(\mathbb{Z}) \qquad \text{given by} \qquad k \longmapsto \tfrac{1}{2}|\,k-1\,\rangle + \tfrac{1}{2}|\,k+1\,\rangle.$$

This coalgebra d is an endomap $\mathbb{Z} \to \mathbb{Z}$ in the Kleisli category $\mathcal{K}\ell(\mathcal{D})$ of the distribution monad. This yields a monoid structure, and iterations $d^n\colon \mathbb{Z} \to \mathbb{Z}$ in $\mathcal{K}\ell(\mathcal{D})$. The Kleisli extension function $d^{\#}\colon \mathcal{D}(\mathbb{Z}) \to \mathcal{D}(\mathbb{Z})$ can be described as:

$$\begin{aligned} & d^{\#}\big(r_1|\,k_1\,\rangle + \cdots + r_n|\,k_n\,\rangle\big) \\ & = \tfrac{1}{2}r_1|\,k_1-1\,\rangle + \tfrac{1}{2}r_1|\,k_1+1\,\rangle + \cdots + \tfrac{1}{2}r_n|\,k_n-1\,\rangle + \tfrac{1}{2}r_n|\,k_n+1\,\rangle, \end{aligned}$$

where on the right-hand-side we must, if needed, identify $r|\,k\,\rangle + s|\,k\,\rangle$ with $(r+s)|\,k\,\rangle$. One has $d^n = d \bullet \cdots \bullet d$, where $d \bullet d = d^{\#} \circ d$.

The iterations d^n, as functions $d^n\colon \mathbb{Z} \to \mathcal{D}(\mathbb{Z})$, yield successively:

$$\begin{aligned} d^0(k) &= 1|\,k\,\rangle \\ d^1(k) &= d(k) \;=\; \tfrac{1}{2}|\,k-1\,\rangle + \tfrac{1}{2}|\,k+1\,\rangle \\ d^2(k) &= \tfrac{1}{4}|\,k-2\,\rangle + \tfrac{1}{4}|\,k\,\rangle + \tfrac{1}{4}|\,k\,\rangle + \tfrac{1}{4}|\,k+2\,\rangle \;=\; \tfrac{1}{4}|\,k-2\,\rangle + \tfrac{1}{2}|\,k\,\rangle + \tfrac{1}{4}|\,k+2\,\rangle \\ d^3(k) &= \tfrac{1}{8}|\,k-3\,\rangle + \tfrac{1}{8}|\,k-1\,\rangle + \tfrac{1}{4}|\,k-1\,\rangle + \tfrac{1}{4}|\,k+1\,\rangle + \tfrac{1}{8}|\,k+1\,\rangle + \tfrac{1}{8}|\,k+3\,\rangle \\ &= \tfrac{1}{8}|\,k-3\,\rangle + \tfrac{3}{8}|\,k-1\,\rangle + \tfrac{3}{8}|\,k+1\,\rangle + \tfrac{1}{8}|\,k+3\,\rangle \qquad \textit{etc.} \end{aligned}$$

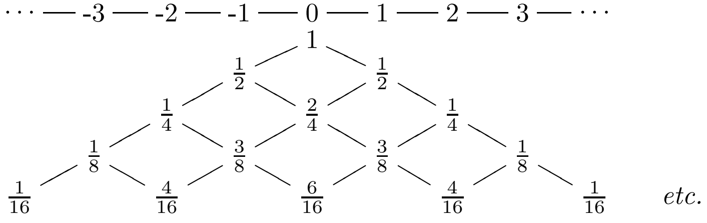
The general formula involves binomial coefficients describing probabilities:

$$\begin{aligned} d^n(k) = \tfrac{\binom{n}{0}}{2^n}|\,k-n\,\rangle + \tfrac{\binom{n}{1}}{2^n}|\,k-n+2\,\rangle + \tfrac{\binom{n}{2}}{2^n}|\,k-n+4\,\rangle + \ldots + \\ \tfrac{\binom{n}{n-1}}{2^n}|\,k+n-2\,\rangle + \tfrac{\binom{n}{n}}{2^n}|\,k+n\,\rangle. \end{aligned}$$

This provides a distribution since all probabilities involved add up to 1, because of the well-known sum formula for binomial coefficients:

$$\tbinom{n}{0} + \tbinom{n}{1} + \tbinom{n}{2} + \cdots + \tbinom{n}{n-1} + \tbinom{n}{n} = 2^n.$$

The resulting probabilistic walks starting in $0 \in \mathbb{Z}$ can be pictured like in (3), but this time with explicit probabilities:



(4)

The role of Pascal's triangle in the description of the probability distributions for such random walks is of course well-known.

3.3 Quantum Walks

In the quantum case the states $k \in \mathbb{Z}$ appear as base vectors, written as $|k\rangle \in \mathcal{M}(\mathbb{Z})$, in the free vector space $\mathcal{M}(\mathbb{Z})$ on $\mathbb{Z}$, see Section 2. Besides these vectors, one qubit, with base vectors $|\downarrow\rangle$ and $|\uparrow\rangle$, is used for the direction of the walk. Thus, the space that is typically used in physics (see [5,12]) for quantum walks is:

$$\mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z}) \qquad \text{with basis elements} \qquad |\downarrow\rangle \otimes |k\rangle,\ |\uparrow\rangle \otimes |k\rangle,$$

where we may understand $|\uparrow\rangle = \binom{1}{0} \in \mathbb{C}^2$ and $|\downarrow\rangle = \binom{0}{1} \in \mathbb{C}^2$.

A single step of a quantum walk is then written as an endomap:

$$\begin{tikzcd}
\mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z}) \arrow[r, "q"] & \mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z}) \\
|\uparrow\rangle \otimes |k\rangle \arrow[r, mapsto] & \frac{1}{\sqrt{2}}|\uparrow\rangle \otimes |k-1\rangle + \frac{1}{\sqrt{2}}|\downarrow\rangle \otimes |k+1\rangle \\
|\downarrow\rangle \otimes |k\rangle \arrow[r, mapsto] & \frac{1}{\sqrt{2}}|\uparrow\rangle \otimes |k-1\rangle - \frac{1}{\sqrt{2}}|\downarrow\rangle \otimes |k+1\rangle
\end{tikzcd} \tag{5}$$

Implicty the Hadamard transform $H = \frac{1}{\sqrt{2}}\left(\begin{smallmatrix}1 & 1\\ 1 & -1\end{smallmatrix}\right)$ is applied to the qubits in $\mathbb{C}^2$. A tree of probabilities is now obtained by repeatedly applying q, say to a start state $|\uparrow\rangle \otimes |0\rangle$, and subsequently measuring $|k\rangle$. We write $Prob_k$ for the probability of seeing $|k\rangle$ as outcome.

Thus, after one step we have:

$$q(|\uparrow\rangle \otimes |0\rangle) = \tfrac{1}{\sqrt{2}}|\uparrow\rangle \otimes |-1\rangle + \tfrac{1}{\sqrt{2}}|\downarrow\rangle \otimes |1\rangle,$$

giving probabilities $Prob_{-1} = Prob_1 = \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}$. After two steps we get:

$$\begin{aligned}
q^2(|\uparrow\rangle \otimes |0\rangle) &= \tfrac{1}{\sqrt{2}}q(|\uparrow\rangle \otimes |-1\rangle) + \tfrac{1}{\sqrt{2}}q(|\downarrow\rangle \otimes |1\rangle) \\
&= \tfrac{1}{2}|\uparrow\rangle \otimes |-2\rangle + \tfrac{1}{2}|\downarrow\rangle \otimes |0\rangle + \tfrac{1}{2}|\uparrow\rangle \otimes |0\rangle - \tfrac{1}{2}|\downarrow\rangle \otimes |2\rangle \\
&= \tfrac{1}{2}|\uparrow\rangle \otimes |-2\rangle + \tfrac{1}{2}(|\uparrow\rangle + |\downarrow\rangle) \otimes |0\rangle - \tfrac{1}{2}|\downarrow\rangle \otimes |2\rangle,
\end{aligned}$$

with probabilities:

$$Prob_{-2} = \left|\tfrac{1}{2}\right|^2 = \tfrac{1}{4} \qquad Prob_0 = \left|\tfrac{1}{2}\right|^2 + \left|\tfrac{1}{2}\right|^2 = \tfrac{1}{2} \qquad Prob_2 = \left|-\tfrac{1}{2}\right|^2 = \tfrac{1}{4}.$$

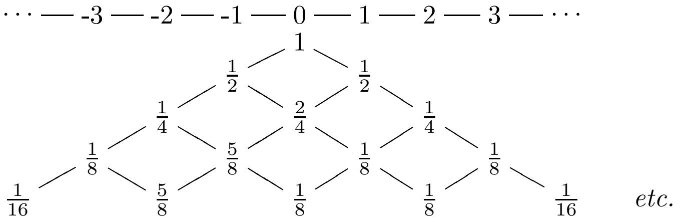
After 3 steps the outcomes begin to differ from the probabilistic outcomes, see (4), due to interference between the different summands:

$$\begin{aligned}
&q^3(|\uparrow\rangle \otimes |0\rangle) \\
&= \tfrac{1}{2}q(|\uparrow\rangle \otimes |-2\rangle) + \tfrac{1}{2}q(|\downarrow\rangle \otimes |0\rangle) \\
&\qquad + \tfrac{1}{2}q(|\uparrow\rangle \otimes |0\rangle) - \tfrac{1}{2}q(|\downarrow\rangle \otimes |2\rangle) \\
&= \tfrac{1}{2\sqrt{2}}|\uparrow\rangle \otimes |-3\rangle + \tfrac{1}{2\sqrt{2}}|\downarrow\rangle \otimes |-1\rangle + \tfrac{1}{2\sqrt{2}}|\uparrow\rangle \otimes |-1\rangle - \tfrac{1}{2\sqrt{2}}|\downarrow\rangle \otimes |1\rangle \\
&\qquad + \tfrac{1}{2\sqrt{2}}|\uparrow\rangle \otimes |-1\rangle + \tfrac{1}{2\sqrt{2}}|\downarrow\rangle \otimes |1\rangle - \tfrac{1}{2\sqrt{2}}|\uparrow\rangle \otimes |1\rangle + \tfrac{1}{2\sqrt{2}}|\downarrow\rangle \otimes |3\rangle \\
&= \tfrac{1}{2\sqrt{2}}|\uparrow\rangle \otimes |-3\rangle + \tfrac{1}{\sqrt{2}}|\uparrow\rangle \otimes |-1\rangle + \tfrac{1}{2\sqrt{2}}|\downarrow\rangle \otimes |-1\rangle \\
&\qquad - \tfrac{1}{2\sqrt{2}}|\uparrow\rangle \otimes |1\rangle + \tfrac{1}{2\sqrt{2}}|\downarrow\rangle \otimes |3\rangle,
\end{aligned}$$

leading to probabilities:

$$Prob_{-3} = Prob_1 = Prob_3 = \left|\tfrac{1}{2\sqrt{2}}\right|^2 = \tfrac{1}{8} \qquad Prob_{-1} = \left|\tfrac{1}{\sqrt{2}}\right|^2 + \left|\tfrac{1}{2\sqrt{2}}\right|^2 = \tfrac{5}{8}.$$

Thus there is a 'drift' to the left, see the following table of probabilities starting from the initial state $|\uparrow\rangle \otimes |0\rangle \in \mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z})$.



(6)

The matrix involved—Hadamard's H in this case—determines the drifting, and thus how the tree is traversed.

4 A Coalgebraic/Monadic Description of Quantum Walks

In the previous section we have seen the standard way of describing quantum walks, namely via endomaps $\mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z}) \to \mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z})$. The question arises if such walks can also be described coalgebraically, of the form $\mathbb{Z} \to T(\mathbb{Z})$, for a suitable monad T, just like for non-deterministic and probabilitistic walks in Subsections 3.1 and 3.2. This section will show how to do so. The following observation forms the basis.

Proposition 1. *1. For each $n \in \mathbb{N}$, there is an isomorphism of vector spaces:*

$$\mathbb{C}^n \otimes \mathcal{M}(X) \;\cong\; \mathcal{M}(n \cdot X),$$

natural in X—where $n \cdot X$ is the n-fold coproduct $X + \cdots + X$, also known as copower of the set X.

2. As a consequence, there is a bijective correspondence between:

$$\frac{\textit{linear maps}\quad \mathbb{C}^n \otimes \mathcal{M}(X) \longrightarrow \mathbb{C}^n \otimes \mathcal{M}(Y)}{\textit{functions}\quad X \longrightarrow \mathcal{M}(n \cdot Y)^n}$$

Proof. 1. For convenience we restrict to $n = 2$. We shall write $\oplus$ for the product of vector spaces, which is at the same time a coproduct of spaces (and hence a 'biproduct'). There is the following chain of (natural) isomorphisms

$$\begin{aligned}
\mathbb{C}^2 \otimes \mathcal{M}(X) &= (\mathbb{C} \oplus \mathbb{C}) \otimes \mathcal{M}(X) \\
&\cong \big(\mathbb{C} \otimes \mathcal{M}(Z)\big) \oplus \big(\mathbb{C} \otimes \mathcal{M}(X)\big) && \text{since } \otimes \text{ distributes over } \oplus \\
&\cong \mathcal{M}(X) \oplus \mathcal{M}(X) && \text{since } \mathbb{C} \text{ is tensor unit} \\
&\cong \mathcal{M}(X + X),
\end{aligned}$$

where the last isomorphism exists because $\mathcal{M}$ is a free functor **Sets** → **Vect**, and thus preserves coproducts.

2. Directly from the previous point, since:

$$\begin{array}{cl}
\mathbb{C}^n \otimes \mathcal{M}(X) \longrightarrow \mathbb{C}^n \otimes \mathcal{M}(Y) & \text{in } \mathbf{Vect} \\ \hline\hline
\mathcal{M}(n \cdot X) \longrightarrow \mathcal{M}(n \cdot Y) & \text{in } \mathbf{Vect}\text{, by point 1} \\ \hline\hline
n \cdot X \longrightarrow \mathcal{M}(n \cdot Y) & \text{in } \mathbf{Sets}\text{, since } \mathcal{M} \text{ is free} \\ \hline\hline
X \longrightarrow \mathcal{M}(n \cdot Y)^n & \text{in } \mathbf{Sets}
\end{array}$$

□

Corollary 1. *There is a bijective correspondence between linear endomaps*

$$\mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z}) \longrightarrow \mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z})$$

as used for quantum walks in Subsection 3.3, and coalgebras

$$\mathbb{Z} \longrightarrow \mathcal{M}(\mathbb{Z} + \mathbb{Z})^2$$

of the functor $\mathcal{M}(2 \cdot -)^2$.

The coalgebra $\mathbb{Z} \to \mathcal{M}(\mathbb{Z}+\mathbb{Z})^2$ corresponding to the linear endomap $q \colon \mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z}) \to \mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z})$ from Subsection 3.3 can be described explicitly as follows.

$$\begin{array}{l}
\mathbb{Z} \xrightarrow{\qquad\qquad\qquad\qquad\qquad\qquad} \mathcal{M}(\mathbb{Z}+\mathbb{Z})^2 \\
m \longmapsto \Big\langle \tfrac{1}{\sqrt{2}}\kappa_1|m-1\rangle + \tfrac{1}{\sqrt{2}}\kappa_2|m+1\rangle,\ \tfrac{1}{\sqrt{2}}\kappa_1|m-1\rangle - \tfrac{1}{\sqrt{2}}\kappa_2|m+1\rangle \Big\rangle
\end{array} \tag{7}$$

The κ_i, for $i = 1, 2$, are coprojections that serve as tags for 'left' and 'right' in a coproduct (disjoint union) $\mathbb{Z}+\mathbb{Z}$. Notice that in this re-description tensor spaces and their bases have disappeared completely.

Of course, at this stage one wonders if the the functor $\mathcal{M}(2 \cdot -)^2$ in Corollary 1 is also a monad—like powerset and distribution. This turns out to be the case, as an instance of the following general "monad transformer" result.

Proposition 2. *Let $\mathbf{A}$ be a category with finite powers $X^n = X \times \cdots \times X$ and copowers $n \cdot X = X + \cdots + X$. For a monad $T \colon \mathbf{A} \to \mathbf{A}$ there is for each $n \in \mathbb{N}$ a new monad $T[n] \colon \mathbf{A} \to \mathbf{A}$ by:*

$$T[n](X) = \Big(T(n \cdot X)\Big)^n$$

with unit and Kleisli extension:

$$\eta[n]_X = \langle T(\kappa_i) \circ \eta_X \rangle_{i \le n} \qquad f^{\#} = \big(\mu_{T(n \cdot Y)} \circ T([f_i]_{i \le n})\big)^n.$$

where in the latter case f is a map $f = \langle f_i \rangle_{i \le n} \colon X \to T[n](Y)$.

Proof. For convenience, and in order to be more concrete, we restrict to $n = 2$. We leave it to the reader to verify that $\eta[2]$ is natural and that its extension is the identity: $\eta[2]^{\#} = \mathrm{id}$. Of the two remaining properties of Kleisli extension, $f^{\#} \circ \eta[2] = f$ and $(g^{\#} \circ f)^{\#} = g^{\#} \circ f^{\#}$, we prove the first one:

$$\begin{aligned} f^{\#} \circ \eta[2] &= (\mu \circ T([f_1, f_2])) \times (\mu \circ T([f_1, f_2])) \circ \langle T(\kappa_1) \circ \eta, T(\kappa_2) \circ \eta \rangle \\ &= \langle \mu \circ T([f_1, f_2]) \circ T(\kappa_1) \circ \eta, \mu \circ T([f_1, f_2]) \circ T(\kappa_2) \circ \eta \rangle \\ &= \langle \mu \circ T(f_1) \circ \eta, \mu \circ T(f_2) \circ \eta \rangle \\ &= \langle \mu \circ \eta \circ f_1, \mu \circ \eta \circ f_2 \rangle \\ &= \langle f_1, f_2 \rangle \\ &= f. \end{aligned}$$

□

Kleisli extension yields the multiplication map $T[n]^2(X) \to T[n](X)$ as extension $\mathrm{id}^{\#}$ of the identity on $T[n](X)$. Concretely, it can be described as:

$$\Big[T\Big(n \cdot \big(T(n \cdot X)\big)^n\Big)\Big]^n \xrightarrow{\Big[\mu \circ T([\pi_i]_{i \leq n})\Big]^n} \Big[T(n \cdot X)\Big]^n$$

The number $n \in \mathbb{N}$ in $T[n]$ yields a form of control via n states, like in the quantum walks in Subsection 3.3 where $n = 2$ and $T = \mathcal{M}$. Indeed, there is a similarity with the state monad transformer $X \mapsto T(S \times X)^S$, for T a monad and S a fixed set of states (see *e.g.* [8]). If S is a finite set, say with size $n = |S|$, $T(S \times -)^S$ is the same as the monad $T[n] = T(n \cdot -)^n$ in Proposition 2 since the product $S \times X$ in **Sets** is the same as the copower $n \cdot X$.

Next we recall that $\mathcal{M}$ is an additive monad. This means that it maps finite coproducts to products: $\mathcal{M}(0) \cong 1$ and $\mathcal{M}(X+Y) \cong \mathcal{M}(X) \times \mathcal{M}(Y)$, in a canonical manner, see [2] for the details. This is relevant in the current setting, because the endomap for quantum walks from Subsection 3.3 can now be described also as a 4-tuple of coalgebras $\mathbb{Z} \to \mathcal{M}(\mathbb{Z})$, since:

$$\begin{array}{cl} \mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z}) \longrightarrow \mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z}) & \\ \hline\hline & \text{(by Corollary 1)} \\ \mathbb{Z} \longrightarrow \mathcal{M}(\mathbb{Z} + \mathbb{Z})^2 & \\ \hline\hline & \text{(by additivity of } \mathcal{M}) \\ \mathbb{Z} \longrightarrow \big(\mathcal{M}(\mathbb{Z}) \times \mathcal{M}(\mathbb{Z})\big)^2 & \\ \wr\| & \\ \mathcal{M}(\mathbb{Z})^4 & \end{array}$$

We shall write these four coalgebras corresponding to the endomap q in (5) as $c_{ij} \colon \mathbb{Z} \to \mathcal{M}(\mathbb{Z})$, for $i, j \in \{1, 2\}$. Explicitly, they are given as follows.

$$\begin{aligned} c_{11}(k) &= \tfrac{1}{\sqrt{2}}|k-1\rangle & \qquad c_{12}(k) &= \tfrac{1}{\sqrt{2}}|k-1\rangle \\ c_{21}(k) &= \tfrac{1}{\sqrt{2}}|k+1\rangle & \qquad c_{22}(k) &= -\tfrac{1}{\sqrt{2}}|k+1\rangle. \end{aligned}$$

As the notation already suggests, we can consider these four coalgebras as entries in a 2×2 matrix of coalgebras, in the following manner:

$$c = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} \lambda k.\, \tfrac{1}{\sqrt{2}}|k-1\rangle & \lambda k.\, \tfrac{1}{\sqrt{2}}|k-1\rangle \\ \lambda k.\, \tfrac{1}{\sqrt{2}}|k+1\rangle & \lambda k.\, -\tfrac{1}{\sqrt{2}}|k+1\rangle \end{pmatrix}. \tag{8}$$

Thus, the first column describes the output for input of the form $|\uparrow\rangle \otimes |k\rangle = \binom{|k\rangle}{0}$, and the second column describes the result for $|\downarrow\rangle \otimes |k\rangle = \binom{0}{|k\rangle}$. By multiplying this matrix with itself one achieves iteration as used in Subsection 3.3. This matrix notation is justified by the following observation.

Lemma 1. *The set $\mathcal{M}(X)^X$ of $\mathcal{M}$-coalgebras on a set X forms a semiring. Addition is done pointwise, using addition on $\mathcal{M}(X)$, and multiplication is Kleisli composition $\bullet$ for $\mathcal{M}$, given by $(d \bullet c)(x)(z) = \sum_y c(x)(y) \cdot d(y)(z)$.* □

The proof is skipped because this lemma is a special instance of a more general result, namely Proposition 3 in the appendix.

5 Reversibility of Computations

So far we have described different kinds of walks as different coalgebras $\mathbb{Z} \to \mathcal{P}_{\mathit{fin}}(\mathbb{Z})$, $\mathbb{Z} \to \mathcal{D}(\mathbb{Z})$, and $\mathbb{Z}+\mathbb{Z} \to \mathcal{M}(\mathbb{Z}+\mathbb{Z})$. Next we investigate reversibility of these coalgebras. It turns out that all these coalgebras are reversible, via a dagger operation, but only the quantum case involves a 'unitary' operation, where the dagger yields the inverse. The three dagger categories that we describe below are captured in [4] as instance of a general construction of a category of 'tame' relations. Here we only look at the concrete descriptions.

We start with the non-deterministic case. Let **BifRel** be the category of sets and bifinite relations. Object are sets X, and morphisms $X \to Y$ are relations $r: X \times Y \to 2 = \{0, 1\}$ such that:

- for each $x \in X$ the set $\{y \in Y \mid r(x, y) \neq 0\}$ is finite;
- also, for each $y \in Y$ the set $\{x \in X \mid r(x, y) \neq 0\}$ is finite.

This means that r factors both as function $X \to \mathcal{P}_{\mathit{fin}}(Y)$ and as $Y \to \mathcal{P}_{\mathit{fin}}(X)$. Relational composition and equality relations make **BifRel** a category. For a map $r: X \to Y$ there is an associated map $r^\dagger : Y \to X$ in the reverse direction, obtained by swapping arguments: $r^\dagger(y, x) = r(x, y)$. This makes **BifRel** a dagger category.

The non-deterministic walks coalgebra $s: \mathbb{Z} \to \mathcal{P}_{\mathit{fin}}(\mathbb{Z})$ from Subsection 3.1 is in fact such bifinite relation $\mathbb{Z} \to \mathbb{Z}$ in **BifRel**. Explicitly, as a map $s: \mathbb{Z}\times\mathbb{Z} \to 2$, also called s, it is given by $s(n, m) = 1$ iff $m = n-1$ or $m = n+1$. The associated dagger map $s^\dagger$, in the reverse direction, is $s^\dagger(n, m) = 1$ iff $s(m, n) = 1$ iff $n = m-1$ or $n = m+1$; it is the same relation. In general, a map f in a dagger category is unitary if $f^\dagger$ is the inverse of f. The non-deterministic walks map s is not unitary, since, for instance:

$$\begin{aligned}(s \circ s^\dagger)(n, n') = 1 &\Leftrightarrow \exists_m.\, s^\dagger(n, m) \wedge s(m, n') \\ &\Leftrightarrow s(n-1, n') \vee s(n+1, n') \\ &\Leftrightarrow n' = n-2 \vee n' = n \vee n' = n+2.\end{aligned}$$

This is not the identity map $\mathbb{Z} \to \mathbb{Z}$ given by $\mathrm{id}_{\mathbb{Z}}(n, n') = 1$ iff $n = n'$.

We turn to the probabilistic case, using a dagger category **dBisRel** of discrete bistochastic relations. Objects are sets X and morphisms $X \to Y$ are maps $r\colon X \times Y \to [0,1]$ that factor both as $X \to \mathcal{D}(Y)$ and as $Y \to \mathcal{D}(X)$. Concretely, this means that for each $x \in X$ there are only finitely many $y \in Y$ with $r(x,y) \neq 0$ and $\sum_y r(x,y) = 1$, and similarly in the other direction. These maps form a category, with composition given by matrix multiplication and identity maps by equality relations. The resulting category **dBisRel** has a dagger by reversal of arguments, like in **BifRel**.

The probabilistic walks map $d\colon \mathbb{Z} \to \mathcal{D}(\mathbb{Z})$ from Subsection 3.2 is an endomap $d\colon \mathbb{Z} \to \mathbb{Z}$ in **dBisRel**, given as:

$$\mathbb{Z} \times \mathbb{Z} \xrightarrow{d} [0,1] \qquad \text{by} \qquad d(n,m) = \begin{cases} \frac{1}{2} & \text{if } m = n-1 \text{ or } m = n+1 \\ 0 & \text{otherwise.} \end{cases}$$

Also in this case d is not unitary; for instance we do not get equality in:

$$\begin{aligned}(d \circ d^{\dagger})(n,n') &= \textstyle\sum_m d^{\dagger}(n,m) \cdot d(m,n') \\ &= \tfrac{1}{2} \cdot d(n-1,n') + \tfrac{1}{2} \cdot d(n+1,n') \\ &= \begin{cases} \frac{1}{4} & \text{if } n' = n-2 \text{ or } n' = n+2 \\ \frac{1}{2} & \text{if } n' = n \\ 0 & \text{otherwise.} \end{cases}\end{aligned}$$

Finally we turn to the quantum case, for which we use the dagger category **BifMRel** of sets and $\mathbb{C}$-valued multirelations. Objects are sets, and morphisms $r\colon X \to Y$ are maps $r\colon X \times Y \to \mathbb{C}$ which factor both as $X \to \mathcal{M}(Y)$ and as $Y \to \mathcal{M}(X)$. This means that for each x there are finitely many y with $r(x,y) \neq 0$, and similarly, for each y there are finitely many x with $r(x,y) \neq 0$. Composition and identities are as before. The dagger now not only involves argument swapping, but also conjugation in $\mathbb{C}$, as in $r^{\dagger}(y,x) = \overline{r(x,y)}$.

We have already seen that the quantum walks endomap $\mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z}) \to \mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z})$ corresponds to a coalgebra $q\colon \mathbb{Z}+\mathbb{Z} \to \mathcal{M}(\mathbb{Z}+\mathbb{Z})$. We now represent it as endo map $q\colon \mathbb{Z}+\mathbb{Z} \to \mathbb{Z}+\mathbb{Z}$ in **BifMRel** given by:

$$\big(\mathbb{Z}+\mathbb{Z}\big) \times \big(\mathbb{Z}+\mathbb{Z}\big) \xrightarrow{q} \mathbb{C} \quad \text{where} \quad \begin{cases} q(\kappa_1 n, \kappa_1(n-1)) = \frac{1}{\sqrt{2}} \\ q(\kappa_1 n, \kappa_2(n+1)) = \frac{1}{\sqrt{2}} \\ q(\kappa_2 n, \kappa_1(n-1)) = \frac{1}{\sqrt{2}} \\ q(\kappa_2 n, \kappa_2(n+1)) = -\frac{1}{\sqrt{2}} \end{cases}$$

(Only the non-zero values are described.) The dagger $q^{\dagger}$ is:

$$\begin{array}{ll} q^{\dagger}(\kappa_1(n-1), \kappa_1 n) = \frac{1}{\sqrt{2}} \qquad & q^{\dagger}(\kappa_2(n+1), \kappa_1 n) = \frac{1}{\sqrt{2}} \\ q^{\dagger}(\kappa_1(n-1), \kappa_2 n) = \frac{1}{\sqrt{2}} \qquad & q^{\dagger}(\kappa_2(n+1), \kappa_2 n) = -\frac{1}{\sqrt{2}} \end{array}$$

In the quantum case we do get a unitary map. This involves several elementary verifications, of which we present an illustration:

$$\begin{aligned}
\big(q \circ q^{\dagger}\big)(\kappa_1 m, \kappa_1 n) &= \textstyle\sum_x q^{\dagger}(\kappa_1 m, x) \cdot q(x, \kappa_1 n) \\
&= q^{\dagger}(\kappa_1 m, \kappa_1(m+1)) \cdot q(\kappa_1(m+1), \kappa_1 n) \\
&\qquad + q^{\dagger}(\kappa_1 m, \kappa_2(m+1)) \cdot q(\kappa_2(m+1), \kappa_1 n) \\
&= \begin{cases} \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} & \text{if } n = m \\ 0 & \text{otherwise} \end{cases} \\
&= \begin{cases} 1 & \text{if } \kappa_1 n = \kappa_1 m \\ 0 & \text{otherwise} \end{cases} \\
&= \mathrm{id}(\kappa_1 m, \kappa_1 n).
\end{aligned}$$

In a similar way one obtains $\big(q \circ q^{\dagger}\big)(\kappa_1 m, \kappa_2 n) = 0 = \mathrm{id}(\kappa_1 m, \kappa_2 n)$, *etc.*

6 Summary, So Far

At this stage, before proceeding, we sum up what we have seen so far. Nondeterministic and probabilistic walks are described quite naturally as coalgebras of a monad, namely of the (finite) powerset $\mathcal{P}_{\mathit{fin}}$ and distribution monad $\mathcal{D}$, respectively. Quantum walks are usually described (by physicists) as endomaps $\mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z}) \to \mathbb{C}^2 \otimes \mathcal{M}(\mathbb{Z})$. But we have illustrated that they can equivalently be described as coalgebras $\mathbb{Z} \to \mathcal{M}(2 \cdot \mathbb{Z})^2$ of a monad. Thus there is a common, generic framework in which to describe various walks, namely as coalgebras of monads. The monad yields a monoid structure on these coalgebras—via Kleisli composition—which enables iteration. This monoid structure can be described quite generally, for arbitrary monads, in the category of algebras of the monad, see Proposition 3.

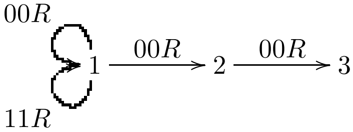
All these walks coalgebras are in fact endo maps in a suitable dagger category. Only in the quantum case the walks form a unitary morphism.

Coalgebras of the form $\mathbb{Z} \to \mathcal{M}(2 \cdot \mathbb{Z})^2$ can equivalently be described as maps $\mathbb{Z} + \mathbb{Z} \to \mathcal{M}(\mathbb{Z} + \mathbb{Z})$ or as a quadruple of coalgebras $\mathbb{Z} \to \mathcal{M}(\mathbb{Z})$. Four such coalgebras are obtained because there is a qubit (in $\mathbb{C}^2$) involved that controls the walking, see Subsection 3.3. More generally, if the control happens via $\mathbb{C}^n$, one obtains n^2 coalgebras in a $n \times n$ matrix. A next step is to observe a similarity to what happens in Turing machines: there one has a finite-state automaton that controls a head which reads/writes/moves on a tape. This similarity will be explored further in the next section, where we use the understanding of walks, using the monad construction $T[n]$ from Proposition 2, to capture Turing machines coalgebraically, as a “head walking on a tape”.

7 Turing Machines as Coalgebras

The idea we wish to explore further is that coalgebras of the form $X \to T[n](X) = T(n \cdot X)^n$ of the monad $T[n]$ from Propostion 2 can be understood as computations of type T on state space X with n auxiliary states that control the computation on X. This idea will be illustrated below for Turing machines.

We shall give a simple example of a non-deterministic Turing machine, for the finite powerset monad $T = \mathcal{P}_{\mathit{fin}}$. We use a tape with binary entries that stretches in 2 dimension, and use the integers $\mathbb{Z}$ (like in walks) as index. Thus the type $\mathbb{T}$ of tapes is given by $\mathbb{T} = 2^{\mathbb{Z}} \times \mathbb{Z}$, consisting of pairs (t, p) where $t \colon \mathbb{Z} \to 2 = \{0, 1\}$ is the tape itself and $p \in \mathbb{Z}$ the current position of the head. One could use a more general set Σ of tape symbols, and use maps $\mathbb{Z} \to \Sigma$ as tapes. Commonly one only uses a limited number of operations on a tape, given as abL or abR, with meaning: if a is read at the current position, then write b, and subsequently move one position left (or right) on the tape. Such operations can be used as labels of transitions between control states. An example non-deterministic Turing machine that can stop if it encounters two successive 0s to the right of the head can be described by the following graph with three state $1, 2, 3$.



We do not include final states explicitly, but clearly the right-most state 3 does not have any transitions and can thus be seen as final.

In line with the description of quantum walks, we shall use four equivalent ways of describing this Turing machine.

1. As an endomap, in the category of join semilattices (which is the category of algebras of the monad $\mathcal{P}_{\mathit{fin}}$ involved), described on base elements as:

$$\begin{array}{rcl}
2^3 \otimes \mathcal{P}_{\mathit{fin}}(\mathbb{T}) & \longrightarrow & 2^3 \otimes \mathcal{P}_{\mathit{fin}}(\mathbb{T}) \\
1 \otimes (t,p) & \longmapsto & \begin{cases} \big(1 \otimes (t, p+1)\big) \vee \big(2 \otimes (t, p+1)\big) & \text{if } t(p) = 0 \\ 1 \otimes (t, p+1) & \text{otherwise} \end{cases} \\
2 \otimes (t,p) & \longmapsto & \begin{cases} 3 \otimes (t, p+1) & \text{if } t(p) = 0 \\ \bot & \text{otherwise} \end{cases} \\
3 \otimes (t,p) & \longmapsto & \bot.
\end{array}$$

2. As a coalgebra of the monad $\mathcal{P}_{\mathit{fin}}(3 \cdot -)^3$, namely:

$$\begin{array}{rcl}
\mathbb{T} & \longrightarrow & \mathcal{P}_{\mathit{fin}}(\mathbb{T} + \mathbb{T} + \mathbb{T})^3 \\
(t,p) & \longmapsto & \Big\langle \{\kappa_1(t, p+1)\} \cup \{\kappa_2(t, p+1) \mid t(p) = 0\}, \\
& & \qquad \{\kappa_3(t, p+1) \mid t(p) = 0\},\ \emptyset \Big\rangle
\end{array}$$

3. As a 3×3 matrix of coalgebras $\mathbb{T} \to \mathcal{P}_{\mathit{fin}}(\mathbb{T})$, using that the monad $\mathcal{P}_{\mathit{fin}}$ is additive (see [2]), so that $\mathcal{P}_{\mathit{fin}}(\mathbb{T}+\mathbb{T}+\mathbb{T})^3 \cong \big(\mathcal{P}_{\mathit{fin}}(\mathbb{T}) \times \mathcal{P}_{\mathit{fin}}(\mathbb{T}) \times \mathcal{P}_{\mathit{fin}}(\mathbb{T})\big)^3 \cong \mathcal{P}_{\mathit{fin}}(\mathbb{T})^9$.

$$\begin{pmatrix} \lambda(t,p).\,\{(t,p+1)\} & \lambda(t,p).\,\emptyset & \lambda(t,p).\,\emptyset \\ \lambda(t,p).\,\begin{cases} \{(t,p+1)\} & \text{if } t(p)=0 \\ \emptyset & \text{otherwise} \end{cases} & \lambda(t,p).\,\emptyset & \lambda(t,p).\,\emptyset \\ \lambda(t,p).\,\emptyset & \lambda(t,p).\,\begin{cases} \{(t,p+1)\} & \text{if } t(p)=0 \\ \emptyset & \text{otherwise} \end{cases} & \lambda(t,p).\,\emptyset \end{pmatrix}$$

The entry at column i and row j describes the coalgebra for the transition from control state i to j. This matrix representation of coalgebras makes sense because the set of coalgebras $X \to \mathcal{P}_{\mathit{fin}}(X)$ forms a semiring, as remarked at the end of Subsection 3.1.

4. As endo map $3 \cdot \mathbb{T} \to 3 \cdot \mathbb{T}$ in the category **BifRel**, that is as bifinite relation $r\colon \big(\mathbb{T}+\mathbb{T}+\mathbb{T}\big) \times \big(\mathbb{T}+\mathbb{T}+\mathbb{T}\big) \to \mathbb{C}$, given by the following non-zero cases.

$$r\big(\kappa_1(t,p), \kappa_1(t,p+1)\big), \qquad r\big(\kappa_1(t,p), \kappa_2(t,p+1)\big) \text{ if } t(p)=0,$$
$$r\big(\kappa_2(t,p), \kappa_3(t,p+1)\big) \text{ if } t(p)=0.$$

Via such modelling one can iterate the mappings involved and thus calculate successor states. We give an example calculation, using the second representation $\mathbb{T} \to \mathcal{P}_{\mathit{fin}}(\mathbb{T}+\mathbb{T}+\mathbb{T})^3$. An element of $\mathbb{T}$ will be described (partially) via expressions like $\cdots 0\underline{1}011 \cdots$, where the underlining indicates the current position of the head. Starting in the first state, represented by the label κ_1, we get:

$$\begin{aligned} \kappa_1(\cdots \underline{1}01001 \cdots) &\longmapsto \{\kappa_1(\cdots 1\underline{0}1001 \cdots)\} \\ &\longmapsto \{\kappa_1(\cdots 10\underline{1}001 \cdots), \kappa_2(\cdots 10\underline{1}001 \cdots)\} \\ &\longmapsto \{\kappa_1(\cdots 101\underline{0}01 \cdots)\} \\ &\longmapsto \{\kappa_1(\cdots 1010\underline{0}1 \cdots), \kappa_2(\cdots 1010\underline{0}1 \cdots)\} \\ &\longmapsto \{\kappa_1(\cdots 10100\underline{1} \cdots), \kappa_2(\cdots 10100\underline{1} \cdots), \kappa_3(\cdots 10100\underline{1} \cdots)\} \end{aligned}$$

Etcetera. Hopefully it is clear that this coalgebraic/monadic/relational modelling of Turing machines is quite flexible. For instance, by changing the monad one gets other types of computation on a tape: by taking the multiset monad $\mathcal{M}$, and requiring unitarity, one obtains quantum Turing machines (as in [10]). For instance, coalgebraic walks like in Subsection 3.3 can be seen as a 2-state quantum Turing machine with a singleton set of symbols (and thus only the head's position forming the tape-type $\mathbb{T} = \mathbb{Z}$).

The above (equivalent) representations of a Turing machine via the monad construction $T[n]$ distinguishes between the tape and the finitely many states of a machine. In contrast, for instance in [9], a Turing machine is represented as a coalgebra of the form $X \longrightarrow \mathcal{P}_{\mathit{fin}}\big(X \times \mathit{\Gamma} \times \{\lhd, \rhd\}\big)^{\mathit{\Gamma}}$, where $\mathit{\Gamma}$ is a set of input symbols, and $\lhd, \rhd$ represent left and right moves. There is only one state space X, which implicitly combines both the tape and the states that steer the computation.

8 Conclusions

The investigation of non-deterministic, probabilistic and quantum walks has led to a coalgebraic description of quantum computation, in the form of qubits acting on a set, via a new monad construction $T[n]$. It adds n-ary steering to T-computations, not only for quantum walks but also in n-state Turing machines (as controled 'walks' on a tape). The coalgebraic approach emphasises only the one-directional aspect of computation. Via suitable categories of 'bi-coalgebraic' relations this bidirectional aspect can be made explicit, and the distinctive unitary character of quantum computation becomes explicit. For the future, the role of final coalgebras requires clarity, especially for the new monad $T[n]$, for instance for computing stationary (limiting) distributions. How to describe (unidirectional) measurements coalgebraically will be described elsewhere.

References

1. Bartels, F., Sokolova, A., de Vink, E.: A hierarchy of probabilistic system types. Theor. Comp. Sci. 327(1-2), 3–22 (2004)
2. Coumans, D., Jacobs, B.: Scalars, monads and categories (2010), http://arxiv.org/abs/1003.0585
3. Jacobs, B.: Semantics of weakening and contraction. Ann. Pure & Appl. Logic 69(1), 73–106 (1994)
4. Jacobs, B.: Dagger categories of tame relations (2011), http://arxiv.org/abs/1101.1077
5. Kempe, J.: Quantum random walks – an introductory overview. Contemporary Physics 44, 307–327 (2003)
6. Kock, A.: Bilinearity and cartesian closed monads. Math. Scand. 29, 161–174 (1971)
7. Kock, A.: Closed categories generated by commutative monads. Journ. Austr. Math. Soc. XII, 405–424 (1971)
8. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Principles of Programming Languages, pp. 333–343. ACM Press, New York (1995)
9. Pavlović, D., Mislove, M., Worrell, J.: Testing semantics: Connecting processes and process logics. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 308–322. Springer, Heidelberg (2006)
10. Perdrix, S.: Partial observation of quantum Turing machine and weaker wellformedness condition. In: Proceedings of: Quantum Physics and Logic and Development of Computational Models (2008), http://web.comlab.ox.ac.uk/people/simon.perdrix/publi/weakerQTM.pdf
11. Rutten, J.: Universal coalgebra: a theory of systems. Theor. Comp. Sci. 249, 3–80 (2000)
12. Venegas-Andraca, S.: Quantum Walks for Computer Scientists. Morgan & Claypool, San Francisco (2008)

A Coalgebras of a Monad Form a Monoid in Algebras

Let $\mathbf{A} = (\mathbf{A}, I, \otimes, \multimap)$ be a symmetric monoidal closed category. A monad $T = (T, \mu, \eta)$ is called monoidal (or commutative) if it comes with a 'double strength' natural transformation $\mathsf{dst}\colon T(X)\otimes T(Y) \to T(X\otimes Y)$ commuting appropriately with the monoidal isomorphisms and with the unit η and multiplication μ. We abbreviate $\mathsf{st} = \mathsf{dst} \circ (\mathrm{id}\otimes\eta)\colon T(X)\otimes Y \to T(X\otimes Y)$ and $\mathsf{st}' = \mathsf{dst}\circ(\eta\otimes \mathrm{id})\colon X\otimes T(Y) \to T(X\otimes Y)$. One can also express this double strength as $\mathsf{dst} = \mu \circ T(\mathsf{st}) \circ \mathsf{st}' = \mu\circ T(\mathsf{st}')\circ \mathsf{st}$, see [3] for details.

We assume that the categories $\mathbf{A}$ and $\mathit{Alg}(T)$ has enough coequalisers so that $\mathit{Alg}(T)$ is also symmetric monoidal via the canonical constructions from [7,6], with tensor $\otimes^T$ and tensor unit $I^T = T(I)$. The key property of this tensor of algebras $\otimes^T$ is that there is a bijective correspondence:

$$\begin{array}{cl} \big(TX \overset{a}{\to} X\big) \otimes^T \big(TY \overset{b}{\to} Y\big) \xrightarrow{f} \big(TZ \overset{c}{\to} Z\big) & \text{in } \mathit{Alg}(T) \\ \hline\hline X\otimes Y \xrightarrow[g]{} Z & \text{bihomomorphism} \end{array} \tag{9}$$

Such a map $g\colon X\otimes Y \to Z$ is a bihomomorphism if the following diagram commutes.

$$\begin{tikzcd} T(X)\otimes T(Y) \arrow[r, "\mathsf{dst}"] \arrow[d, "a\otimes b"'] & T(X\otimes Y) \arrow[r, "T(g)"] & T(Z) \arrow[d, "c"] \\ X\otimes Y \arrow[rr, "g"] & & Z \end{tikzcd}$$

The next result may be read as: internal T-coalgebras form a monoid in $\mathit{Alg}(T)$.

Proposition 3. *In the situation described above,*

1. *for each $X \in \mathbf{A}$, the object $T(X)^X = X \multimap T(X)$ in $\mathbf{A}$ "of T-coalgebras" carries an algebra structure $a_X\colon T\big(T(X)^X\big) \to T(X)^X$, obtained by abstraction $\Lambda(-)$ as:*

$$a_X = \Lambda\Big(T\big(T(X)^X\big)\otimes X \xrightarrow{\mathsf{st}} T\big(T(X)^X \otimes X\big) \xrightarrow{T(ev)} T^2(X) \xrightarrow{\mu} T(X)\Big).$$

2. *This algebra $a_X \in \mathit{Alg}(T)$ carries a monoid structure in $\mathit{Alg}(T)$ given by Kleisli composition, with monoid unit $u\colon I^T \to T(X)^X$ defined as:*

$$u = \Lambda\Big(T(I)\otimes X \xrightarrow{\mathsf{st}} T(I\otimes X) \xrightarrow[\cong]{T(\lambda)} T(X)\Big)$$

The monoid multiplication $m\colon T(X)^X \otimes^T T(X)^X \to T(X)^X$ is obtained via the correspondence (9) from the bihomomorphism $T(X)^X\otimes T(X)^X \to T(X)^X$ that one gets by abstraction from:

$$\begin{tikzcd} \big(T(X)^X\otimes T(X)^X\big)\otimes X \arrow[d, "\alpha^{-1}"', "\cong"] & & & T(X) \\ T(X)^X\otimes\big(T(X)^X\otimes X\big) \arrow[r, "\mathrm{id}\otimes ev"] & T(X)^X\otimes T(X) \arrow[r, "\mathsf{st}'"] & T\big(T(X)^X\otimes X\big) \arrow[r, "T(ev)"] & T^2(X) \arrow[u, "\mu"'] \end{tikzcd}$$

Similarity Quotients as Final Coalgebras

Paul Blain Levy*

University of Birmingham, UK
P.B.Levy@cs.bham.ac.uk

Abstract. We give a general framework connecing a branching time relation on nodes of a transition system to a final coalgebra for a suitable endofunctor. Examples of relations treated by our theory include bisimilarity, similarity, upper and lower similarity for transition systems with divergence, similarity for discrete probabilistic systems, and nested similarity. Our results describe firstly how to characterize the relation in terms of a given final coalgebra, and secondly how to construct a final coalgebra using the relation.

Our theory uses a notion of "relator" based on earlier work of Thijs. But whereas a relator must preserve binary composition in Thijs' framework, it only laxly preserves composition in ours. It is this weaker requirement that allows nested similarity to be an example.

1 Introduction

A series of influential papers including [1,11,17,18,19] have developed a coalgebraic account of bisimulation, based on the following principles.

- A transition system may be regarded as a coalgebra for a suitable endofunctor F on **Set** (or another category).
- Bisimulation can be defined in terms of an operation on relations, called a "relational extension" or "relator".
- This operation may be obtained directly from F, if F preserves quasi-pullbacks [3].
- Given a final F-coalgebra, two nodes of transition systems are bisimilar iff they have the same *anamorphic image*–i.e. image in the final coalgebra.
- Any coalgebra can be quotiented by bisimilarity to give an *extensional* coalgebra—one in which bisimilarity is just equality.
- One may construct a final coalgebra by taking the extensional quotient of a sufficiently large coalgebra.

Thus a final F-coalgebra provides a "universe of processes" according to the viewpoint that bisimilarity is the appropriate semantic equivalence.

More recently [2,4,7,12,13,22] there have been several coalgebraic studies of simulation, in which the final F-coalgebra carries a preorder. This is valuable for someone who wants to study bisimilarity and similarity together: equality

* Supported by EPSRC Advanced Research Fellowship EP/E056091.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 27–41, 2011.

represents bisimilarity, and the preorder represents similarity. But someone who is exclusively interested in similarity will want the universe of processes to be a poset: if two nodes are mutually similar, they should be equal. In this paper we shall see that such a universe is also a final coalgebra, for a suitable endofunctor H on the category of posets.

For example, consider countably branching transition systems. In this case, we shall see that H maps a poset A to the set of countably generated lower sets, ordered by inclusion. A final H-coalgebra is a universe for similarity, in two senses.

- On the one hand, we can *use* a final H-coalgebra to characterize similarity, by regarding a transition system as a discretely ordered H-coalgebra.
- On the other hand, we can *construct* a final H-coalgebra, by taking a sufficiently large transition system and quotienting by similarity.

We give this theory in Sect. 4. But first, in Sect. 3, we introduce the notion of relator, which gives many notions of simulation, e.g. for transition systems with divergence and Markov chains. Finally, in Sect. 5 we look at the example of 2-nested simulation; this requires a generalization of our theory where relations are replaced by indexed families of relations.

2 Mathematical Preliminaries

Definition 1. *(Relations)*

1. *For sets X and Y, we write* $X \overset{\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} Y$ *when $\mathcal{R}$ is a relation from X to Y, and* $\mathrm{Rel}(X,Y)$ *for the complete lattice of relations ordered by inclusion.*
2. $X \overset{(=_X)}{\relbar\joinrel\mapstochar\joinrel\rightarrow} X$ *is the equality relation on X.*
3. *Given relations* $X \overset{\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} Y \overset{\mathcal{S}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} Z$ *, we write* $X \overset{\mathcal{R};\mathcal{S}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} Z$ *for the composite.*
4. *Given functions* $Z \xrightarrow{f} X$ *and* $W \xrightarrow{g} Y$ *, and a relation* $X \overset{\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} Y$ *, we write* $Z \overset{(f,g)^{-1}\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} W$ *for the inverse image* $\{(z,w) \in Z \times W \mid f(z) \mathrel{\mathcal{R}} g(w)\}$.
5. *Given a relation* $X \overset{\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} Y$ *, we write* $Y \overset{\mathcal{R}^{\mathsf{c}}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} X$ *for its converse. $\mathcal{R}$ is* difunctional *when* $\mathcal{R};\mathcal{R}^{\mathsf{c}};\mathcal{R} \subseteq \mathcal{R}$.

Definition 2. *(Preordered sets)*

1. *A* preordered set *A is a set A_0 with a preorder $\leqslant_A$. It is a* poset *(*setoid, discrete setoid*) when $\leqslant_A$ is a partial order (an equivalence relation, the equality relation).*
2. *We write* **Preord** *(***Poset**, **Setoid**, **DiscSetoid***) for the category of preordered sets (posets, setoids, discrete setoids) and monotone functions.*
3. *The functor* $\Delta : \mathbf{Set} \longrightarrow \mathbf{Preord}$ *maps X to $(X, =_X)$ and* $X \xrightarrow{f} Y$ *to f. This gives an isomorphism* $\mathbf{Set} \cong \mathbf{DiscSetoid}$.

4. *Let A and B be preordered sets. A* bimodule $A \xrightarrow{\mathcal{R}} B$ *is a relation such that $(\leqslant_A);\mathcal{R};(\leqslant_B) \subseteq \mathcal{R}$. We write* $\mathrm{Bimod}(A,B)$ *for the complete lattice of bimodules, ordered by inclusion. For an arbitrary relation $A_0 \xrightarrow{\mathcal{R}} B_0$, its* bimodule closure $A \xrightarrow{\overline{\mathcal{R}}} B$ *is $(\leqslant_A);\mathcal{R};(\leqslant_B)$.*

Definition 3. *(Quotienting)*

1. *Let A be a preordered set. For $x \in A$, its* principal lower set $[x]_A$ *is $\{y \in A \mid y \leqslant_A x\}$. The* quotient poset QA *is $\{[x]_A \mid x \in A\}$ ordered by inclusion. (This is isomorphic to the quotient of A by the equivalence relation $(\leqslant_A) \cap (\geqslant_A)$.) We write $A \xrightarrow{p_A} QA$ for the function $x \mapsto [x]_A$.*
2. *Let A and B be preordered sets and $A \xrightarrow{f} B$ a monotone function. The monotone function $QA \xrightarrow{Qf} QB$ maps $[x]_A \mapsto [f(x)]_B$.*
3. *Let A and B be preordered sets and $A \xrightarrow{\mathcal{R}} B$ a bimodule. The bimodule $QA \xrightarrow{Q\mathcal{R}} QB$ relates $[x]_A$ to $[y]_B$ iff $x\ \mathcal{R}\ y$.*

We give some examples of endofunctors on **Set**.

Definition 4. *1. For any set X and class K of cardinals, we write $\mathcal{P}^K X$ for the set of subsets X with cardinality in K. $\mathcal{P}$ is the endofunctor on* **Set** *mapping X to the set of subsets of X and $X \xrightarrow{f} Y$ to $u \mapsto \{f(x) \mid x \in u\}$. It has subfunctors $\mathcal{P}^{[0,\kappa)}$ and $\mathcal{P}^{[1,\kappa)}$ where κ is a cardinal or ∞.*

2. Maybe *is the endofunctor on* **Set** *mapping X to $X+1 = \{\mathrm{Just}\ x \mid x \in X\} \cup \{\Uparrow\}$ and $X \xrightarrow{f} Y$ to* $\mathrm{Just}\ x \mapsto \mathrm{Just}\ f(x), \Uparrow \mapsto \Uparrow$.
3. *A* discrete subprobability distribution *on a set X is a function $d : X \longrightarrow [0,1]$ such that $\sum_{x \in X} d_x \leqslant 1$ (so d is countably supported). For any $U \subseteq X$ we write $dU \overset{\mathrm{def}}{=} \sum_{x \in U} d_x$, and we write $d \Uparrow \overset{\mathrm{def}}{=} 1 - d(X)$. D is the endofunctor on* **Set** *mapping X to the set of discrete subprobability distributions on X and $X \xrightarrow{f} Y$ to $d \mapsto (y \mapsto d(f^{-1}\{y\}))$.*

Definition 5. *Let $\mathcal{C}$ be a category.*

1. *Let F be an endofunctor on $\mathcal{C}$. An F-*coalgebra *M is a $\mathcal{C}$-object $M^{\cdot}$ and morphism $M^{\cdot} \xrightarrow{\zeta_M} FM^{\cdot}$. We write* $\mathrm{Coalg}(\mathcal{C}, F)$ *for the category of F-coalgebras and homomorphisms.*
2. *Let F and G be endofunctors on $\mathcal{C}$, and $F \xrightarrow{\alpha} G$ a natural transformation. We write* $\mathrm{Coalg}(\mathcal{C}, \alpha) : \mathrm{Coalg}(\mathcal{C}, F) \longrightarrow \mathrm{Coalg}(\mathcal{C}, G)$ *for the functor mapping M to $(M^{\cdot}, \zeta_M; \alpha_{M^{\cdot}})$ and $M \xrightarrow{f} N$ to f.*

Examples of coalgebras:

- a *transition system* is a $\mathcal{P}$-coalgebra
- a *countably branching transition system* is a $\mathcal{P}^{[0,\aleph_0]}$-coalgebra
- a *transition system with divergence* is a $\mathcal{P}$Maybe-coalgebra
- a *partial Markov chain* is a D-coalgebra.

There are also easy variants for labelled systems.

Lemma 1. *[8] Let $\mathcal{C}$ be a category and $\mathcal{B}$ a reflective replete (i.e. full and isomorphism-closed) subcategory of $\mathcal{C}$.*

1. *Let $A \in \mathsf{ob}\ \mathcal{C}$. Then A is a final object of $\mathcal{C}$ iff it is a final object of $\mathcal{B}$.*
2. *Let F be an endofunctor on $\mathcal{C}$. Then* $\mathrm{Coalg}(\mathcal{B}, F)$ *is a reflective replete subcategory of* $\mathrm{Coalg}(\mathcal{C}, F)$*.*

Examples of reflective replete subcategories:

- **Poset** of **Preord**, and **DiscSetoid** of **Setoid**. In each case the reflection is given by Q with unit p.
- **Setoid** of **Preord**. At A, the reflection is $(A_0, \equiv)$, where $\equiv$ is the least equivalence relation containing $\leqslant_A$, with unit id_{A_0}.

3 Relators

3.1 Relators and Simulation

Any notion of simulation depends on a way of transforming a relation. For example, given a relation $X \xrightarrow{\mathcal{R}} Y$, we define

- $\mathcal{P}X \xrightarrow{\mathrm{Sim}\mathcal{R}} \mathcal{P}Y$ to relate u to v when $\forall x \in u. \exists y \in v.\ x\ \mathcal{R}\ y$
- $\mathcal{P}X \xrightarrow{\mathrm{Bisim}\mathcal{R}} \mathcal{P}Y$ to relate u to v when $\forall x \in u. \exists y \in v.\ x\ \mathcal{R}\ y$ and $\forall y \in v.\ \exists x \in u.\ x\ \mathcal{R}\ y$.

for simulation and bisimulation respectively. In general:

Definition 6. *Let F be an endofunctor on* **Set***. An F-*relator *maps each relation $X \xrightarrow{\mathcal{R}} Y$ to a relation $FX \xrightarrow{\Gamma\mathcal{R}} FY$ in such a way that the following hold.*

- *For any relations $X \xrightarrow{\mathcal{R},\mathcal{S}} Y$, if $\mathcal{R} \subseteq \mathcal{S}$ then $\Gamma\mathcal{R} \subseteq \Gamma\mathcal{S}$.*
- *For any set X we have $(=_{FX}) \subseteq \Gamma(=_X)$*
- *For any relations $X \xrightarrow{\mathcal{R}} Y \xrightarrow{\mathcal{S}} Z$ we have $(\Gamma\mathcal{R});(\Gamma\mathcal{S}) \subseteq \Gamma(\mathcal{R};\mathcal{S})$*
- *For any functions $Z \xrightarrow{f} X$ and $W \xrightarrow{g} Y$, and any relation $X \xrightarrow{\mathcal{R}} Y$, we have $\Gamma(f,g)^{-1}\mathcal{R} = (Ff, Fg)^{-1}\Gamma\mathcal{R}$.*

An F-relator Γ is conversive *when $\Gamma(\mathcal{R}^{\mathsf{c}}) = (\Gamma\mathcal{R})^{\mathsf{c}}$ for every relation $X \xrightarrow{\mathcal{R}} Y$.*

For example: Sim is a $\mathcal{P}$-relator, and Bisim is a conversive $\mathcal{P}$-relator.

We can now give a general definition of simulation.

Definition 7. *Let F be an endofunctor on* **Set**, *and let Γ be an F-relator. Let M and N be F-coalgebras.*

1. *A* Γ-simulation *from M to N is a relation $M^{\cdot} \overset{\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} N^{\cdot}$ such that $\mathcal{R} \subseteq (\zeta_M, \zeta_N)^{-1}\Gamma\mathcal{R}$.*
2. *The largest Γ-simulation is called* Γ-similarity *and written $\lesssim^{\Gamma}_{M,N}$.*
3. *M is* Γ-encompassed *by N, written $M \preccurlyeq^{\Gamma} N$, when for every $x \in M$ there is $y \in N$ such that $x \lesssim^{\Gamma}_{M,N} y$ and $y \lesssim^{\Gamma}_{N,M} x$.*

For example: a Sim-simulation is an ordinary simulation, and a Bisim-simulation is a bisimulation.

The basic properties of simulations are as follows.

Lemma 2. *Let F be an endofunctor on* **Set**, *and Γ an F-relator.*

1. *Let M be an F-coalgebra. Then $M \overset{(=_{M^{\cdot}})}{\relbar\joinrel\mapstochar\joinrel\rightarrow} M$ is a Γ-simulation. Moreover $\lesssim^{\Gamma}_{M,M}$ is a preorder on $M^{\cdot}$—an equivalence relation if Γ is conversive.*
2. *Let M, N, P be F-coalgebras. If $M \overset{\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} N \overset{\mathcal{S}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} P$ are Γ-simulations then so is $M \overset{\mathcal{R};\mathcal{S}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} P$. Moreover $(\lesssim^{\Gamma}_{M,N});(\lesssim^{\Gamma}_{N,P}) \sqsubseteq (\lesssim^{\Gamma}_{M,P})$.*
3. *Let M and N be F-coalgebras, and let Γ be conversive. If $M \overset{\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} N$ is a simulation then so is $N \overset{\mathcal{R}^{\mathsf{c}}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} M$. Moreover $(\lesssim^{\Gamma}_{M,N})^{\mathsf{c}} = (\lesssim^{\Gamma}_{N,M})$ and $\lesssim^{\Gamma}_{M,N}$ is difunctional.*
4. *Let $M \xrightarrow{f} N$ and $M' \xrightarrow{g} N'$ be F-coalgebra morphisms. If $N \overset{\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} N'$ is a Γ-simulation then so is $M \overset{(f,g)^{-1}\mathcal{R}}{\relbar\joinrel\relbar\joinrel\mapstochar\joinrel\relbar\joinrel\rightarrow} M'$. Moreover $(f,g)^{-1}(\lesssim^{\Gamma}_{N,N'}) = (\lesssim^{\Gamma}_{M,M'})$.*
5. *$\preccurlyeq^{\Gamma}$ is a preorder on the class of F-coalgebras.*
6. *Let $M \xrightarrow{f} N$ be an F-coalgebra morphism. Then x and $f(x)$ are mutually Γ-similar for all $x \in M^{\cdot}$. Hence $M \preccurlyeq N$, and if f is surjective then also $N \preccurlyeq M$.*

An F-coalgebra is *all-Γ-encompassing* when it is greatest in the $\preccurlyeq^{\Gamma}$ preorder. For example, take the disjoint union of all transition systems carried by an initial segment of $\mathbb{N}$. This is an all-Bisim-encompassing $\mathcal{P}^{[0,\aleph_0]}$-coalgebra, because every node of a $\mathcal{P}^{[0,\aleph_0]}$-coalgebra has only countably many descendants.

3.2 Relators Preserving Binary Composition

Definition 8. *Let F be an endofunctor on* **Set**. *An F-relator Γ is said to* preserve binary composition *when for all sets X, Y, Z and relations $X \overset{\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} Y \overset{\mathcal{S}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} Z$ we have $\Gamma(\mathcal{R};\mathcal{S}) = (\Gamma\mathcal{R});(\Gamma\mathcal{S})$. If we also have $\Gamma(=_X) = (=_{FX})$ for every set X, then F is* functorial.

For example, Sim preserves binary composition and Bisim is functorial. We shall examine relators preserving binary composition using the following notions.

Definition 9

1. *A commutative square* $$\begin{tikzcd} Z \arrow[r, "g"] \arrow[d, "f"'] & Y \arrow[d, "k"] \\ X \arrow[r, "h"'] & W \end{tikzcd}$$ *in* **Set** *is a* quasi-pullback *when*
$$\forall x \in X.\ \forall y \in Y.\ h(x) = k(y) \Rightarrow \exists z \in Z.\ x = f(z) \wedge g(z) = y$$
2. *A commutative square* $$\begin{tikzcd} C \arrow[r, "g"] \arrow[d, "f"'] & B \arrow[d, "k"] \\ A \arrow[r, "h"'] & D \end{tikzcd}$$ *in* **Preord** *is a* preorder-quasi-pullback *when* $\forall x \in A.\ \forall y \in B.\ h(x) \leqslant_D k(y) \Rightarrow \exists z \in C.\ x \leqslant_A f(z) \wedge g(z) \leqslant_B y$

Definition 10. *(adapted from [13]) Let F be an endofunctor on* **Set**. *A* stable preorder *on F is a functor G :* **Set** $\longrightarrow$ **Preord** *that makes*

$$\begin{tikzcd} & \mathbf{Preord} \arrow[d, "(-)_0"] \\ \mathbf{Set} \arrow[ur, "G"] \arrow[r, "F"'] & \mathbf{Set} \end{tikzcd}$$

commute and sends quasi-pullbacks to preorder-quasi-pullbacks. It is a stable equivalence relation *on F when it is a functor* **Set** $\longrightarrow$ **Setoid**.

For any relation $\begin{tikzcd} X \arrow[r, "\mathcal{R}", "\shortmid"{marking}] & Y \end{tikzcd}$, we write $\begin{tikzcd} X & \mathcal{R} \arrow[l, "\pi_{\mathcal{R}}"'] \arrow[r, "\pi'_{\mathcal{R}}"] & Y \end{tikzcd}$ for the two projections. We can now give our main result.

Theorem 1. *Let F be an endofunctor on* **Set**. *There is a bijection between*

- *F-relators preserving binary composition*
- *stable preorders on F*

described as follows.

- *Given an F-relator Γ preserving binary composition, we define the stable preorder $\tilde{\Gamma}$ on F to map X to $(FX, \Gamma(=_X))$ and $X \xrightarrow{f} Y$ to Ff.*
- *Given a stable preorder G on F, we define the F-relator $\hat{G}$ to map a relation $\begin{tikzcd} X \arrow[r, "\mathcal{R}", "\shortmid"{marking}] & Y \end{tikzcd}$ to*
$$\{(x, y) \in FX \times FY \mid \exists z \in F\mathcal{R}.\ x \leqslant_{GX} (F\pi_{\mathcal{R}})z \wedge (F\pi'_{\mathcal{R}})z \leqslant_{GY} y\}$$

It restricts to a bijection between

- *conversive F-relators preserving binary composition*
- *stable equivalence relations on F.*

Corollary 1. *[3] Let F be an endofunctor on* **Set***.*

1. *Suppose F preserves quasi-pullbacks. Then we obtain a conversive functorial F-relator $\hat{F}$ mapping a relation $X \xrightarrow{\mathcal{R}} Y$ to*
$$\{(x, y) \in FX \times FY \mid \exists z \in F\mathcal{R}.\ x = (F\pi_{\mathcal{R}})z \wedge (F\pi'_{\mathcal{R}})z = y\}$$
2. *Let Γ be a functorial F-relator. Then F preserves quasi-pullbacks and $\Gamma = \hat{F}$.*

3.3 Further Examples of Relators

We first note several ways of constructing relators.

Lemma 3. *1. Let F be an endofunctor on* **Set***, and $(\Gamma_j)_{j\in J}$ a family of F-relators. Then*
$$\prod_{j\in J} \Gamma_j \ :\ (\ X \xrightarrow{\mathcal{R}} Y\ ) \ \mapsto\ \bigcap_{j\in J} \Gamma_j \mathcal{R}$$
is an F-relator. If M and N are F-coalgebras, then $M^{\cdot} \xrightarrow{\mathcal{R}} N^{\cdot}$ is a $\prod_{j\in J}\Gamma_j$-simulation from M to N iff, for all $j \in J$, it is a Γ_j-simulation from M to N.

2. *Let F be an endofunctor on* **Set***, and Γ an F-relator. Then*
$$\Gamma^{\mathsf{c}} \ :\ (\ X \xrightarrow{\mathcal{R}} Y\ ) \ \mapsto\ (\Gamma\mathcal{R}^{\mathsf{c}})^{\mathsf{c}}$$
is an F-relator. If M and N are F-coalgebras, then $M^{\cdot} \xrightarrow{\mathcal{R}} N^{\cdot}$ is a Γ^{c}-simulation from M to N iff $\mathcal{R}^{\mathsf{c}}$ is a Γ-simulation from N to M; hence $(\lesssim^{\Gamma^{\mathsf{c}}}_{M,N}) = (\lesssim^{\Gamma}_{N,M})^{\mathsf{c}}$.
3. *Let F and G be endofunctors on* **Set** *and $F \xrightarrow{\alpha} G$ a natural transformation. Let Γ be an G-relator. Then*
$$\alpha^{-1}\Gamma \ :\ (\ X \xrightarrow{\mathcal{R}} Y\ ) \ \mapsto\ (\alpha_X, \alpha_Y)^{-1}\Gamma\mathcal{R}$$
is an F-relator. If M and N are F-coalgebras, then $M^{\cdot} \xrightarrow{\mathcal{R}} N^{\cdot}$ is an $\alpha^{-1}\Gamma$-simulation from M to N iff it is a Γ-simulation from $\mathrm{Coalg}(\mathbf{Set}, \alpha)M$ *to* $\mathrm{Coalg}(\mathbf{Set}, \alpha)N$*; hence $(\lesssim^{\alpha^{-1}\Gamma}_{M,N}) = (\lesssim^{\Gamma}_{\mathrm{Coalg}(\mathbf{Set},\alpha)M, \mathrm{Coalg}(\mathbf{Set},\alpha)N})$.*
4. *The identity operation on relations is an $\mathsf{id}_{\mathbf{Set}}$-relator.*
5. *Let F and F' be endofunctors on* **Set***. If Γ is an F-relator and Γ' an F'-relator, then $\Gamma'\Gamma$ is an $F'F$-relator.*

Note that $\Gamma \sqcap \Gamma^{\mathsf{c}}$ is the greatest conversive relator contained in Γ.

We give some relators for our examples:

- Via Def. 3(3), Sim and Bisim are $\mathcal{P}^{[0,\kappa)}$-relators and $\mathcal{P}^{[1,\kappa)}$-relators where κ is a cardinal or ∞. Moreover Sim preserves binary composition, and if $\kappa \leqslant 3$ or $\kappa \geqslant \aleph_0$ then Bisim is functorial. But for $4 \leqslant \kappa < \aleph_0$, the functors $\mathcal{P}^{[0,\kappa)}$ and $\mathcal{P}^{[1,\kappa)}$ do not preserve quasi-pullbacks, so Bisim does not preserve binary composition over them.

- We define $\mathcal{P}$Maybe-relators, all preserving binary composition. For a relation $X \overset{\mathcal{R}}{\longrightarrow\!\!\!\!\!\!\!+\;\;} Y$,

$$
\begin{aligned}
\text{LowerSim}\mathcal{R} &\stackrel{\text{def}}{=} \{(u,v) \in \mathcal{P}\text{Maybe}X \times \mathcal{P}\text{Maybe}Y \mid \\
&\qquad \forall x \in \text{Just}^{-1}u.\ \exists y \in \text{Just}^{-1}v.\ (x,y) \in \mathcal{R}\} \\
\text{UpperSim}\mathcal{R} &\stackrel{\text{def}}{=} \{(u,v) \in \mathcal{P}\text{Maybe}X \times \mathcal{P}\text{Maybe}Y \mid \Uparrow \notin u \Rightarrow \\
&\qquad \Uparrow \notin v \\
&\qquad \wedge \forall y \in \text{Just}^{-1}v.\ \exists x \in \text{Just}^{-1}u.\ (x,y) \in \mathcal{R})\} \\
\text{ConvexSim} &\stackrel{\text{def}}{=} \text{LowerSim} \sqcap \text{UpperSim} \\
\text{SmashSim}\mathcal{R} &\stackrel{\text{def}}{=} \{(u,v) \in \mathcal{P}\text{Maybe}X \times \mathcal{P}\text{Maybe}Y \mid \Uparrow \notin u \Rightarrow \\
&\qquad \Uparrow \notin v \\
&\qquad \wedge \forall y \in \text{Just}^{-1}v.\ \exists x \in \text{Just}^{-1}u.\ (x,y) \in \mathcal{R} \\
&\qquad \wedge \forall x \in \text{Just}^{-1}u.\ \exists y \in \text{Just}^{-1}v.\ (x,y) \in \mathcal{R}\} \\
\text{InclusionSim}\mathcal{R} &\stackrel{\text{def}}{=} \{(u,v) \in \mathcal{P}\text{Maybe}X \times \mathcal{P}\text{Maybe}Y \mid \\
&\qquad \forall x \in \text{Just}^{-1}u.\ \exists y \in \text{Just}^{-1}v.\ (x,y) \in \mathcal{R}\} \\
&\qquad \wedge \Uparrow \in u \Rightarrow \Uparrow \in v\}
\end{aligned}
$$

We respectively obtain notions of *lower*, *upper*, *convex*, *smash* and *inclusion simulation* on transiton systems with divergence [10,20]. By taking converses and intersections of these relators, we obtain—besides $\top$—nineteen different relators of which three are conversive. A more systematic analysis that includes these is presented in [16].

- We define D-relators. For a relation $X \overset{\mathcal{R}}{\longrightarrow\!\!\!\!\!\!\!+\;\;} Y$

$$
\begin{aligned}
\text{ProbSim}\mathcal{R} &\stackrel{\text{def}}{=} \{(d,d') \in DX \times DY \mid \forall U \subseteq X. dU \leqslant d'\mathcal{R}(U)\} \\
\text{ProbBisim}\mathcal{R} &\stackrel{\text{def}}{=} \{(d,d') \in DX \times DY \mid \forall U \subseteq X. dU \leqslant d'\mathcal{R}(U) \wedge d(\Uparrow) \leqslant d'(\Uparrow)\}
\end{aligned}
$$

where $\mathcal{R}(U) \stackrel{\text{def}}{=} \{y \in Y \mid \exists x \in U.\ (x,y) \in \mathcal{R}\}$. In fact ProbBisim is the greatest conversive relator contained in ProbSim. We obtain notions of simulation and bisimulation on partial Markov chains as in [5,6,21,15,22]. By Thm. 1 of [14], ProbSim preserves binary composition and ProbBisim is functorial.

4 Theory of Simulation and Final Coalgebras

Throughout this section, F is an endofunctor on **Set** and Γ is an F-relator.

4.1 QF_Γ-Coalgebras

Definition 11. *F_Γ is the endofunctor on* **Preord** *that maps A to $(FA_0, \Gamma(\leqslant_A))$ and $A \xrightarrow{f} B$ to Ff.*

Thus we obtain an endofunctor QF_Γ on **Preord**. It restricts to **Poset** and also, if Γ is conversive, to **Setoid** and to **DiscSetoid**.

For example, if A is a preordered set, then $Q\mathcal{P}^{[0,\aleph_0]}_{\mathrm{Sim}}A$ is (isomorphic to) the set of countably generated lower sets, ordered by inclusion. The probabilistic case is unusual: D_{ProbSim} is already an endofunctor on **Poset**, so applying Q makes no difference (up to isomorphism). This reflects the fact that, for partial Markov chains, mutual similarity is bisimilarity [6].

A QF_Γ-coalgebra M is said to be *final* when the following equivalent conditions hold:

- M is final in Coalg(**Preord**$, QF_\Gamma$)
- M is final in Coalg(**Poset**$, QF_\Gamma$).

If Γ is conversive, the following are equivalent to the above:

- M is final in Coalg(**Setoid**$, QF_\Gamma$)
- M is final in Coalg(**DiscSetoid**$, QF_\Gamma$).

These equivalences follow from Lemma 1.

We adapt Def. 7 and Lemma 2 from F-coalgebras to QF_Γ-coalgebras.

Definition 12. *Let M and N be QF_Γ-coalgebras.*

1. *A* simulation *from M to N is a bimodule $M^{\cdot} \xrightarrow{\mathcal{R}} N^{\cdot}$ such that $\mathcal{R} \subseteq (\zeta_M, \zeta_N)^{-1}Q\Gamma\mathcal{R}$.*
2. *The greatest simulation is called* similarity *and written $\lesssim_{M,N}$.*
3. *M is* encompassed *by N, written $M \preccurlyeq N$, when for every $x \in M$ there is $y \in N$ such that $x \lesssim_{M,N} y$ and $y \lesssim_{N,M} x$.*

Lemma 4. *Let F be an endofunctor on* **Set***, and Γ an F-relator.*

1. *Let M be a QF_Γ-coalgebra. Then $M \xrightarrow{(\leqslant_{M^{\cdot}})} M$ is a simulation. Moreover $\lesssim^{\Gamma}_{M,M}$ is a preorder on $M_0^{\cdot}$—an equivalence relation if Γ is conversive—that contains $\leqslant_{M^{\cdot}}$.*
2. *Let M, N, P be QF_Γ-coalgebras. If $M \xrightarrow{\mathcal{R}} N \xrightarrow{\mathcal{S}} P$ are simulations then so is $M \xrightarrow{\mathcal{R};\mathcal{S}} P$. Moreover $(\lesssim_{M,N});(\lesssim_{N,P}) \sqsubseteq (\lesssim_{M,P})$.*
3. *Let M and N be QF_Γ-coalgebras, and let Γ be conversive. If $M \xrightarrow{\mathcal{R}} N$ is a simulation then so is $N \xrightarrow{\overline{\mathcal{R}^{\mathsf{c}}}} M$ —recall that this is $(\leqslant_{N^{\cdot}});\mathcal{R}^{\mathsf{c}};(\leqslant_{M^{\cdot}})$. Moreover $(\lesssim_{M,N})^{\mathsf{c}} = (\lesssim_{N,M})$ and $\lesssim_{M,N}$ is difunctional.*
4. *Let $M \xrightarrow{f} N$ and $M' \xrightarrow{g} N'$ be QF_Γ-coalgebra morphisms. If $N \xrightarrow{\mathcal{R}} N'$ is a simulation then so is $M \xrightarrow{(f,g)^{-1}\mathcal{R}} M'$. Moreover $(\lesssim_{M,M'}) = (f,g)^{-1}(\lesssim_{N,N'})$.*
5. *$\preccurlyeq$ is a preorder on the class of QF_Γ-coalgebras.*

6. *Let* $M \xrightarrow{f} N$ *be an* QF_{Γ}*-coalgebra morphism. Then* x *and* $f(x)$ *are mutually similar for all* $x \in M^{\cdot}$*. Hence* $M \preccurlyeq N$*, and if* f *is surjective then also* $N \preccurlyeq M$*.*

We can also characterize coalgebra morphisms.

Lemma 5. *Let* M *and* N *be* QF_{Γ}*-coalgebras. For any function* $M^{\cdot}_0 \xrightarrow{f} N^{\cdot}_0$ *, the following are equivalent.*

1. $M \xrightarrow{f} N$ *is a* QF_{Γ}*-coalgebra morphism.*
2. $\begin{tikzcd} M \arrow[r, "(f,N^{\cdot}_0)^{-1}(\leqslant_{N^{\cdot}})", "\shortmid" marking] & N \end{tikzcd}$ *and* $\begin{tikzcd} N \arrow[r, "(N^{\cdot}_0,f)^{-1}(\leqslant_{N^{\cdot}})", "\shortmid" marking] & M \end{tikzcd}$ *are both simulations.*

A QF_{Γ}-coalgebra N is *all-encompassing* when it is encompasses every $M \in \text{Coalg}(\mathbf{Preord}, QF_{\Gamma})$, or equivalently every $M \in \text{Coalg}(\mathbf{Poset}, QF_{\Gamma})$, or equivalently—if Γ is conversive—every $M \in \text{Coalg}(\mathbf{Setoid}, QF_{\Gamma})$ or every $M \in \text{Coalg}(\mathbf{Setoid}, QF_{\Gamma})$. These equivalences follow from the surjectivity of the units of the reflections.

4.2 Extensional Coalgebras

Definition 13. *An* extensional coalgebra *is* $M \in \text{Coalg}(\mathbf{Poset}, QF_{\Gamma})$ *such that* $(\lesssim_{M,M}) = (\leqslant_{M^{\cdot}})$*. We write* $\text{ExtCoalg}(\Gamma)$ *for the category of extensional coalgebras and coalgebra morphisms.*

These coalgebras enjoy several properties.

Lemma 6. *Let* N *be an extensional coalgebra.*

1. *If* Γ *is conversive, then* $N^{\cdot}$ *is a discrete setoid.*
2. *Let* M *be a* QF_{Γ}*-coalgebra and* $N \xrightarrow{f} M$ *a coalgebra morphism. Then* f *is order-reflecting and injective.*
3. *Let* M *be a* QF_{Γ}*-coalgebra and* $M \xrightarrow{f} N$ *an order-reflecting, injective coalgebra morphism. Then* M *is extensional.*
4. *Let* M *be a* QF_{Γ}*-coalgebra such that* $M \preccurlyeq N$*. Then there is a unique* QF_{Γ}*-coalgebra morphism* $M \xrightarrow{f} N$ *.*

Thus $\text{ExtCoalg}(\Gamma)$ is just a preordered class. It is a replete subcategory of $\text{Coalg}(\mathbf{Poset}, QF_{\Gamma})$ and also—if Γ is conversive—of $\text{Coalg}(\mathbf{DiscSetoid}, QF_{\Gamma})$. We next see that is reflective within $\text{Coalg}(\mathbf{Preord}, QF_{\Gamma})$.

Lemma 7. *(Extensional Quotient) Let* M *be a* QF_{Γ}*-coalgebra, and define* $\mathbf{p}_M \overset{\text{def}}{=} p_{(M^{\cdot}_0, \lesssim_{M,M})}$*.*

1. *There is a* QF_{Γ}*-coalgebra* $\mathbf{Q}M$ *carried by* $Q(M^{\cdot}_0, \lesssim_{M,M})$*, uniquely characterized by the fact that* $M \xrightarrow{\mathbf{p}_M} \mathbf{Q}M$ *is a coalgebra morphism.*

2. $\mathbf{Q}M$*, with unit* $\mathbf{p}_M$*, is a reflection of* M *in* $\mathrm{ExtCoalg}(\Gamma)$*.*

More generally, a QF_Γ-coalgebra M can be quotiented by any $(\leqslant_{M^\cdot})$-containing preorder that is an endosimulation on M; but we shall not need this.

Lemma 8. *Let* M *be a* QF_Γ*-coalgebra. The following are equivalent.*

1. M *is a final* QF_Γ*-coalgebra.*
2. M *is all-encompassing and extensional.*
3. M *is extensional, and encompasses all extensional* QF_Γ*-coalgebras.*

Lemma 9. *Let* M *be a* QF_Γ*-coalgebra. The following are equivalent.*

1. M *is all-encompassing.*
2. M *encompasses all extensional coalgebras.*
3. $\mathbf{Q}M$ *is a final* QF_Γ*-coalgebra.*

4.3 Relating F-Coalgebras and QF_Γ-Coalgebras

We have studied F-coalgebras and QF_Γ-coalgebras separately, but now we connect them: each F-coalgebra gives rise to a QF_Γ-coalgebra, and the converse is also true in a certain sense.

Definition 14. *The functor* $\Delta^\Gamma : \mathrm{Coalg}(\mathbf{Set}, F) \longrightarrow \mathrm{Coalg}(\mathbf{Preord}, QF_\Gamma)$ *maps*

- *an* F*-coalgebra* $M = (M^\cdot, \zeta_M)$ *to the* QF_Γ*-coalgebra with carrier* $\Delta M^\cdot$ *and structure* $\Delta M^\cdot \xrightarrow{\zeta_M} F_\Gamma \Delta M^\cdot \xrightarrow{p_{F_\Gamma \Delta M^\cdot}} QF_\Gamma \Delta M^\cdot$
- *an* F*-coalgebra morphism* $M \xrightarrow{f} N$ *to* f*.*

Lemma 10. *Let* M *and* N *be* F*-coalgebras. Then a* Γ*-simulation from* M *to* N *is precisely a simulation from* $\Delta^\Gamma M$ *to* $\Delta^\Gamma N$*. Hence* $(\lesssim_{\Delta^\Gamma M, \Delta^\Gamma N}) = (\lesssim^\Gamma_{M,N})$*, and* $M \preccurlyeq^\Gamma N$ *iff* $\Delta^\Gamma M \preccurlyeq \Delta^\Gamma N$*.*

We are thus able to use a final QF_Γ-coalgebra to characterize similarity in F-coalgebras.

Theorem 2. *Let* M *be a final* QF_Γ*-coalgebra; for any* QF_Γ*-coalgebra* P *we write* $P \xrightarrow{a_P} M$ *for its anamorphism. Let* N *and* N' *be* F*-coalgebras. Then*

$$(\lesssim^\Gamma_{N,N'}) = (a_{\Delta^\Gamma N}, a_{\Delta^\Gamma N'})^{-1}(\leqslant_{M^\cdot})$$

Our other results require moving from a QF_Γ-coalgebra to an F-coalgebra.

Lemma 11. *Let* M *be a* QF_Γ*-coalgebra. Then there is an* F*-coalgebra* N *and a surjective* QF_Γ*-coalgebra morphism* $\Delta^\Gamma N \xrightarrow{f} M$ *.*

Theorem 3

1. *Let* M *be an* F*-coalgebra. Then* $\mathbf{Q}\Delta^\Gamma M$ *is a final* QF_Γ*-coalgebra iff* M *is all-*Γ*-encompassing.*
2. *Any final* QF_Γ*-coalgebra is isomorphic to one of this form.*

5 Beyond Similarity

5.1 Multiple Relations

We recall from [9] that a *2-nested simulation* from M to N (transition systems) is a simulation contained in the converse of similarity. Let us say that a *nested preordered set* is a set equipped with two preorders $\leqslant_{\mathsf{n}}$ (think 2-nested similarity) and $\leqslant_{\mathsf{o}}$ (think converse of similarity) such that $(\leqslant_{\mathsf{n}}) \subseteq (\leqslant_{\mathsf{o}})$ and $(\leqslant_{\mathsf{n}}) \subseteq (\geqslant_{\mathsf{o}})$. It is a *nested poset* when $\leqslant_{\mathsf{n}}$ is a partial order. By working with these instead of preordered sets and posets, we can obtain a characterization of 2-nested similarity as a final coalgebra.

We fix a set I. For our example of 2-nested simulation, it would be $\{\mathsf{n}, \mathsf{o}\}$.

Definition 15. *(I-relations)*

1. *For any sets X and Y, an I-*relation $X \xrightarrow{\mathcal{R}} Y$ *is an I-indexed family $(\mathcal{R}_i)_{i \in I}$ of relations from X to Y. We write* $\mathrm{Rel}_I(X, Y)$ *for the complete lattice of I-relations ordered pointwise.*
2. *Identity I-relations $(=_X)$ and composite I-relations $\mathcal{R};\mathcal{S}$ are defined pointwise, as are inverse image I-relations $(f, g)^{-1}\mathcal{R}$ for functions f and g.*

We then obtain analogues of Def. 2 and 3. In particular, an *I-preordered set* A is a set A_0 equipped with an I-indexed family of preorders $(\leqslant_{A,i})_{i \in I}$, and it is an *$I$-poset* when $\bigcap_{i \in I}(\leqslant_i)$ is a partial order. We thus obtain categories $\mathbf{Preord}_I$ and $\mathbf{Poset}_I$, whose morphisms are *monotone* functions, i.e. monotone in each component. Given an I-preordered set A, the *principal lower set* of $x \in A$ is $\{y \in A \mid \forall i \in I.\ y \leqslant_{A,i} x\}$. The *quotient I-poset* QA is $\{[x]_A \mid x \in A\}$ with ith preorder relating $[x]_A$ to $[y]_A$ iff $x \leqslant_{A,i} y$, and we write $A \xrightarrow{p_A} QA$ for the function $x \mapsto [x]_A$. Thus $\mathbf{Poset}_I$ is a reflective replete subcategory of $\mathbf{Preord}_I$.

Returning to our example, a nested preordered set is a $\{\mathsf{n}, \mathsf{o}\}$-preordered set, subject to some constraints that we ignore until Sect. 5.2.

For the rest of this section, let F be an endofunctor on **Set**, and Λ an *F-relator I-matrix*, i.e. an $I \times I$-indexed family of F-relators $(\Lambda_{i,j})_{i,j \in I}$. This gives us an operation on I-relations as follows.

Definition 16. *For any I-relation $FX \xrightarrow{\mathcal{R}} FY$, we define the I-relation $FX \xrightarrow{\Lambda\mathcal{R}} FY$ as $(\bigcap_{j \in I} \Lambda_{i,j}\mathcal{R}_j)_{i \in I}$.*

For our example, we take the $\mathcal{P}$-relator $\{\mathsf{n}, \mathsf{o}\}$-matrix TwoSim

$$\begin{array}{ll} \mathrm{TwoSim}_{\mathsf{n},\mathsf{n}} \stackrel{\mathrm{def}}{=} \mathrm{Sim} & \mathrm{TwoSim}_{\mathsf{n},\mathsf{o}} \stackrel{\mathrm{def}}{=} \mathrm{Sim}^{\mathsf{c}} \\ \mathrm{TwoSim}_{\mathsf{o},\mathsf{n}} \stackrel{\mathrm{def}}{=} \top & \mathrm{TwoSim}_{\mathsf{o},\mathsf{o}} \stackrel{\mathrm{def}}{=} \mathrm{Sim}^{\mathsf{c}} \end{array}$$

We can see that the operation $\mathcal{R} \mapsto \Lambda\mathcal{R}$ has the same properties as a relator.

Lemma 12

1. *For any I-relations $X \xrightarrow{\mathcal{R},\mathcal{S}} Y$, if $\mathcal{R} \sqsubseteq \mathcal{S}$ then $\Lambda\mathcal{R} \sqsubseteq \Lambda\mathcal{S}$.*
2. *For any set X we have $(=_{FX}) \sqsubseteq \Lambda(=_X)$*

3. *For any I-relations* $X \xrightarrow{\mathcal{R}} Y \xrightarrow{\mathcal{S}} Z$ *we have* $(\Lambda\mathcal{R});(\Lambda\mathcal{S}) \sqsubseteq \Lambda(\mathcal{R};\mathcal{S})$
4. *For any functions* $X' \xrightarrow{f} X$ *and* $Y' \xrightarrow{g} Y$ *and any I-relation* $X \xrightarrow{\mathcal{R}} Y$ *, we have* $\Lambda(f,g)^{-1}\mathcal{R} = (Ff,Fg)^{-1}\Lambda\mathcal{R}$.

Note by the way that TwoSim as a $\mathcal{P}$-relator matrix does not preserve binary composition. Now we adapt Def. 7.

Definition 17. *Let M and N be F-coalgebras.*

1. *A* Λ-simulation *from M to N is an I-relation* $M^{\cdot} \xrightarrow{\mathcal{R}} N^{\cdot}$ *such that for all $i,j \in I$ we have $\mathcal{R}_i \in (\zeta_M,\zeta_N)^{-1}\Lambda_{i,j}\mathcal{R}_j$, or equivalently $\mathcal{R} \sqsubseteq \Lambda(\zeta_M,\zeta_N)^{-1}\mathcal{R}$.*
2. *The largest Λ-simulation is called* Λ-similarity *and written $\lesssim^{\Lambda}_{M,N}$.*
3. *N is said to* Λ-encompass *M when for every $x \in M$ there is $y \in N$ such that, for all $i \in I$, we have $x\ (\lesssim^{\Gamma}_{M,N,i})\ y$ and $y\ (\lesssim^{\Gamma}_{N,M,i})\ x$.*

In our example, the n-component of $\lesssim^{\mathrm{TwoSim}}_{M,N}$ is 2-nested similarity, and the o-component is the converse of similarity from N to M.

The rest of the theory in Sect. 4 goes through unchanged, using Lemma 12.

5.2 Constraints

We wish to consider not all I-preordered sets (for a suitable indexing set I) but only those that satisfy certain constraints. These constraints are of two kinds:

- a "positive constraint" is a pair (i,j) such that we require $(\leqslant_i) \subseteq (\leqslant_j)$
- a "negative constraint" is a pair (i,j) such that we require $(\leqslant_i) \subseteq (\geqslant_j)$.

Furthermore the set of constraints should be "deductively closed". For example, if $(\leqslant_i) \subseteq (\geqslant_j)$ and $(\leqslant_j) \subseteq (\geqslant_k)$ then $(\leqslant_i) \subseteq (\leqslant_k)$.

Definition 18. *A* constraint theory *on I is a pair $\gamma = (\gamma^+,\gamma^-)$ of relations on I such that γ^+ is a preorder and $\gamma^+;\gamma^-;\gamma^+ \subseteq \gamma^-$ and $\gamma^-;\gamma^- \subseteq \gamma^+$.*

For our example, let γ_{nest} be the constraint theory on $\{\mathsf{n},\mathsf{o}\}$ given by

$$\gamma^+_{\mathrm{nest}} = \{(\mathsf{n},\mathsf{n}),(\mathsf{n},\mathsf{o}),(\mathsf{o},\mathsf{o})\} \quad \gamma^-_{\mathrm{nest}} = \{(\mathsf{n},\mathsf{o})\}$$

A constraint theory γ gives rise to two operations γ^{+L} and γ^{-L} on relations (where L stands for "lower adjoint"). They are best understood by seeing how they are used in the rest of Def. 19.

Definition 19. *Let γ be a constraint theory on I.*

1. *For an I-relation* $X \xrightarrow{\mathcal{R}} Y$ *, we define I-relations*
   - $X \xrightarrow{\gamma^{+L}\mathcal{R}} Y$ *as* $(\bigcup_{j \in I (j,i) \in \gamma^+} \mathcal{R}_j)_{i \in I}$
   - $Y \xrightarrow{\gamma^{-L}\mathcal{R}} X$ *as* $(\bigcup_{j \in I (j,i) \in \gamma^-} \mathcal{R}^{\mathsf{c}}_j)_{i \in I}$.

2. *An I-endorelation $X \overset{\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} X$ is* γ-symmetric *when*
   - *for all $(j,i) \in \gamma^{+}$ we have $\mathcal{R}_j \subseteq \mathcal{R}_i$, or equivalently $\gamma^{+L}\mathcal{R} \sqsubseteq \mathcal{R}$*
   - *for all $(j,i) \in \gamma^{-}$ we have $\mathcal{R}_j^{\mathsf{c}} \subseteq \mathcal{R}_i$, or equivalently $\gamma^{-L}\mathcal{R} \sqsubseteq \mathcal{R}$.*
3. *We write $\mathbf{Preord}_{\gamma}$ ($\mathbf{Poset}_{\gamma}$) for the category of γ-symmetric I-preordered sets (I-posets) and monotone functions.*
4. *An I-relation $X \overset{\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} Y$ is* γ-difunctional *when*
   - *for all $(j,i) \in \gamma^{+}$ we have $\mathcal{R}_j \subseteq \mathcal{R}_i$, or equivalently $\gamma^{+L}\mathcal{R} \sqsubseteq \mathcal{R}$*
   - *for all $(j,i) \in \gamma^{-}$ we have $\mathcal{R}_i;\mathcal{R}_j^{\mathsf{c}};\mathcal{R}_i \subseteq \mathcal{R}_i$, or equivalently $\mathcal{R};\gamma^{-L}\mathcal{R};\mathcal{R} \sqsubseteq \mathcal{R}$.*

For our example, $\mathbf{Preord}_{\gamma_{\mathrm{nest}}}$ and $\mathbf{Poset}_{\gamma_{\mathrm{nest}}}$ are the categories of nested preordered sets and nested posets respectively. In general, $\mathbf{Poset}_{\gamma}$ is a reflective replete subcategory of $\mathbf{Preord}_{\gamma}$ and $\mathbf{Preord}_{\gamma}$ of $\mathbf{Preord}_{I}$.

Now let F be an endofunctor and Λ an F-relator I-matrix.

Definition 20. *Let γ be a constraint theory on I. Then Λ is* γ-conversive *when*

$$\bigsqcap_{\substack{l \in I \\ (l,k) \in \gamma^{+}}} \Lambda_{j,l} \sqsubseteq \Lambda_{i,k} \quad \textit{for all } (j,i) \in \gamma^{+} \textit{ and } k \in I$$

$$\bigsqcap_{\substack{l \in I \\ (l,k) \in \gamma^{-}}} \Lambda_{j,l}^{\mathsf{c}} \sqsubseteq \Lambda_{i,k} \quad \textit{for all } (j,i) \in \gamma^{-} \textit{ and } k \in I$$

For our example, it is clear that the matrix TwoSim is γ_{nest}-conversive.

Lemma 13. *Let γ be a constraint theory on I such that Λ is γ-conversive. For every I-relation $X \overset{\mathcal{R}}{\relbar\joinrel\mapstochar\joinrel\rightarrow} Y$ we have $\gamma^{+L}\Lambda\mathcal{R} \sqsubseteq \Lambda\gamma^{+L}\mathcal{R}$ and $\gamma^{-L}\Lambda\mathcal{R} \sqsubseteq \Lambda\gamma^{-L}\mathcal{R}$.*

5.3 Generalized Theory of Simulation and Final Coalgebras (Sketch)

All the results of Sect. 4, in particular Thms. 2–3, generalize to the setting of a set I with a constraint theory γ. We replace "conversive" by "γ-conversive".

In our nested simulation example, we thus obtain an endofunctor $\mathcal{P}^{[0,\aleph_0]}_{\mathrm{TwoSim}}$ on $\mathbf{Preord}_{\gamma_{\mathrm{nest}}}$ that maps a nested preordered set $A = (A_0, (\leqslant_{A,\mathsf{n}}), (\leqslant_{A,\mathsf{o}}))$ to $(\mathcal{P}^{[0,\aleph_0]}A_0, \mathrm{Sim}(\leqslant_{A,\mathsf{n}}) \cap \mathrm{Sim}^{\mathsf{c}}(\leqslant_{A,\mathsf{o}}), \mathrm{Sim}^{\mathsf{c}}(\leqslant_{A,\mathsf{o}}))$. We conclude:

- (from Thm. 2) Given a final $Q\mathcal{P}^{[0,\aleph_0]}_{\mathrm{TwoSim}}$-coalgebra M, we can use $(\leqslant_{M\cdot,\mathsf{n}})$ and $(\geqslant_{M\cdot,\mathsf{o}})$ to characterize 2-nested similarity and similarity, respectively, in countably branching transition systems.
- (from Thm. 3) Given a countably branching transition system that is all-Bisim-encompassing (and hence all-TwoSim-encompassing), we can quotient it by 2-nested similarity to obtain a final $Q\mathcal{P}^{[0,\aleph_0]}_{\mathrm{TwoSim}}$-coalgebra.

Acknowledgements. I am grateful to Jean Goubault-Larrecq, Bartek Klin, Alexander Kurz, Sam Staton and James Worrell for their help.

References

1. Aczel, P., Mendler, P.F.: A final coalgebra theorem. In: Dybjer, P., Pitts, A.M., Pitt, D.H., Poigné, A., Rydeheard, D.E. (eds.) Category Theory and Computer Science. LNCS, vol. 389, pp. 357–365. Springer, Heidelberg (1989)
2. Baltag, A.: A logic for coalgebraic simulation. ENTCS 33 (2000)
3. Carboni, A., Kelly, G.M., Wood, R.J.: A 2-categorical approach to change of base and geometric morphisms I. Cah. Topologie Géom. Différ. Catégoriques 32(1), 47–95 (1991), http://www.numdam.org/item?id=CTGDC_1991__32_1_47_0
4. Cîrstea, C.: A modular approach to defining and characterising notions of simulation. Inf. Comput. 204(4), 469–502 (2006)
5. Danos, D., Laviolette, P.: Bisimulation and cocongruence for probabilistic systems. Information and Computation 204 (2006)
6. Desharnais, J.: A logical characterization of bisimulation for labelled Markov processes. In: Proceedings of the 2nd International Workshop on Probabilistic Methods in Verification, Univ. of Birmingham Technical Report CS-99-8, pp. 33–48 (1999)
7. Fábregas, I., de Frutos Escrig, D., Palomino, M.: Non-strongly stable orders also define interesting simulation relations. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 221–235. Springer, Heidelberg (2009)
8. Freyd, P.J.: Algebraically complete categories. In: Proc. 1990 Como Category Theory Conf., Berlin. Lecture Notes in Mathematics, vol. 1488, pp. 95–104 (1991)
9. Groote, J.F., Vaandrager, F.: Structured operational semantics and bisimulation as a congruence. Information and Computation 100(2), 202–260 (1992)
10. Hennessy, M., Plotkin, G.D.: A term model for CCS. In: Dembiński, P. (ed.) MFCS 1980. LNCS, vol. 88, pp. 261–274. Springer, Heidelberg (1980)
11. Hermida, C., Jacobs, B.: Structural induction and coinduction in a fibrational setting. Information and Computation 145(2), 107–152 (1998)
12. Hesselink, W.H., Thijs, A.: Fixpoint semantics and simulation. TCS 238(1-2), 275–311 (2000)
13. Hughes, J., Jacobs, B.: Simulations in coalgebra. TCS 327(1-2), 71–108 (2004)
14. Kamae, T., Krengel, U., O'Brien, G.L.: Stochastic inequalities on partially ordered spaces. The Annals of Probability 5(6), 899–912 (1977)
15. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Information and Computation 94(1), 1–28 (1991)
16. Levy, P.B.: Boolean precongruences (2009) (journal submission)
17. Rutten, J.: Universal coalgebra: a theory of systems. TCS 249(1), 3–80 (2000)
18. Sokolova, A.: Coalgebraic analysis of probabilistic systems. Ph.D. thesis, Technische Universiteit Eindhoven (2005)
19. Staton, S.: Relating coalgebraic notions of bisimulation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 191–205. Springer, Heidelberg (2009)
20. Ulidowski, I.: Equivalences on observable processes. In: Scedrov, A. (ed.) Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science, pp. 148–161. IEEE Computer Society Press, Santa Cruz (June 1992)
21. de Vink, E.P., Rutten, J.J.M.M.: Bisimulation for probabilistic transition systems: a coalgebraic approach. Theoretical Computer Science 221 (1999)
22. Worrell, J.: Coinduction for recursive data types: partial orders, metric spaces and ω-categories. ENTCS 33 (2000)

What Do Reversible Programs Compute?

Holger Bock Axelsen and Robert Glück

DIKU, Department of Computer Science, University of Copenhagen
funkstar@diku.dk, glueck@acm.org

Abstract. Reversible computing is the study of computation models that exhibit both forward and backward determinism. Understanding the fundamental properties of such models is not only relevant for reversible programming, but has also been found important in other fields, *e.g.*, bidirectional model transformation, program transformations such as inversion, and general static prediction of program properties.

Historically, work on reversible computing has focussed on *reversible simulations of irreversible computations*. Here, we take the viewpoint that the property of reversibility itself should be the starting point of a computational theory of reversible computing. We provide a novel semantics-based approach to such a theory, using *reversible Turing machines* (RTMs) as the underlying computation model.

We show that the RTMs can compute *exactly all injective, computable functions*. We find that the RTMs are *not* strictly classically universal, but that they support another notion of universality; we call this *RTM-universality*. Thus, even though the RTMs are sub-universal in the classical sense, they are powerful enough as to include a self-interpreter. Lifting this to other computation models, we propose *r-Turing completeness* as the 'gold standard' for computability in reversible computation models.

1 Introduction

The computation models that form the basis of programming languages are usually deterministic in *one direction* (forward), but non-deterministic in the opposite (backward) direction. Most other well-studied programming models exhibit non-determinism in both computation directions. Common to both of these classes is that they are information lossy, because generally a previous computation state cannot be recovered from a current state. This has implications on the analysis and application of these models. *Reversible computing* is the study of computation models wherein all computations are organized *two-way* deterministically, without any logical information loss.

Reversible computation models have been studied in widely different areas ranging from cellular automata [11], program transformation concerned with the inversion of programs [19], reversible programming languages [3,21], the view-update problem in bidirectional computing and model transformation [14,6], static prediction of program properties [15], digital circuit design [18,20], to quantum computing [5]. However, between all these cases, the definition and use of reversibility varies significantly (and subtly), making it difficult to apply

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 42–56, 2011.

results learned in one area to others. For example, even though reversible Turing machines were introduced several decades ago [4], the authors have found that there has been a blurring of the concepts of *reversibility* and *reversibilization*, which makes it difficult to ascertain exactly what is being computed, in later publications.

This paper aims to establish the foundational computability aspects for reversible computation models from a formal semantics viewpoint, using reversible Turing machines (RTMs, [4]) as the underlying computation model, to answer the question: *What do reversible programs compute?*

In reversible computation models, each atomic computation step *must* be reversible. This might appear as too restrictive to allow general and useful computations in reversible computation models. On the other hand, it might appear from the seminal papers by Landauer [9] and Bennett [4], that reversibility is not restrictive at all, and that all computations can be performed reversibly. We show that both of these viewpoints are wrong, under the view that the functional (semantical) behavior of a reversible machine should be logically reversible.

This paper brings together many different streams of work in an integrated semantics formalism that makes reversible programs accessible to a precise analysis, as a stepping stone for future work that makes use of reversibility. Thus, this paper is also an attempt to give a precise structure and basis for a foundational computability theory of reversible languages, in the spirit of the semantics approach to computability of Jones [8].

We give a formal presentation of the reversible Turing machines (Sect. 2), and, using a semantics-based approach (Sect. 3), outline the foundational results of reversible computing (Sect. 4). We show the computational robustness of the RTMs under reductions of the number of symbols and tapes (Sect. 5). Following a proof tactic introduced by Bennett, we show that the RTMs can compute exactly all injective, computable functions (Sect. 6). We study the question of universality, and give a novel interpretation of the concept (*RTM-universality*) that applies to RTMs, and prove constructively that the RTMs are RTM-universal (Sect. 7). We propose *r-Turing completeness* (Sect. 8) as the measure for computability of reversible computation models. Following a discussion of related work (Sect. 9) we present our conclusions (Sect. 10).

2 Reversible Triple-Format Turing Machines

The computation model we shall consider here is the *Turing machine* (TM). Recall that a Turing machine consists of a (doubly-infinite) *tape* of cells along which a *tape head* moves in discrete steps, reading and writing on the tape according to an *internal state* and a fixed *transition relation*. We shall here adopt a *triple format* for the rules which is similar to Bennett's quadruple format [4], but has the advantage of being slightly easier to work with[1].

[1] It is straightforward to translate back and forth between triple, quadruple, and the usual quintuple formats.

Definition 1 (Turing machine). *A TM T is a tuple $(Q, \Sigma, \delta, b, q_s, q_f)$ where Q is a finite set of states, Σ is a finite set of tape symbols, $b \in \Sigma$ is the blank symbol,*

$$\delta \subseteq (Q \times [(\Sigma \times \Sigma) \cup \{\leftarrow, \downarrow, \rightarrow\}] \times Q) = \Delta$$

is a partial relation defining the transition relation, $q_s \in Q$ is the starting state, and $q_f \in Q$ is the final state. There must be no transitions leading out of q_f nor into q_s. Symbols $\leftarrow$, $\downarrow$, $\rightarrow$ represent the three shift directions (left, stay, right).

The form of a triple in δ is either a *symbol rule* $(q, (s, s'), q')$ or a *shift rule* (q, d, q') where $q, q' \in Q$, $s, s' \in \Sigma$, and $d \in \{\leftarrow, \downarrow, \rightarrow\}$. Intuitively, a symbol rule says that in state q, if the tape head is reading symbol s, write s' and change into state q'. A shift rule says that in state q, move the tape head in direction d and change into state q'. It is easy to see how to extend the definition to k-tape machines by letting

$$\delta \subseteq (Q \times [(\Sigma \times \Sigma)^k \cup \{\leftarrow, \downarrow, \rightarrow\}^k] \times Q) \ .$$

Definition 2 (Configuration). *The* configuration *of a TM is a tuple $(q, (l, s, r)) \in Q \times (\Sigma^* \times \Sigma \times \Sigma^*) = \mathcal{C}$, where $q \in Q$ is the internal state, $l, r \in \Sigma^*$ are the parts of the tape to the left and right of the tape head represented as strings, and $s \in \Sigma$ is the symbol being scanned by the tape head*[2].

Definition 3 (Computation step). *A TM $T = (Q, \Sigma, \delta, b, q_s, q_f)$ in configuration $C \in \mathcal{C}$* leads to *configuration $C' \in \mathcal{C}$, written as $T \vdash C \leadsto C'$, defined for $s, s' \in \Sigma$, $l, r \in \Sigma^*$ and $q, q' \in Q$ by*

$$\begin{array}{llll}
T \vdash (q, (l, s, r)) & \leadsto (q', (l, s', r)) & \textit{if} & (q, (s, s'), q') \in \delta \ , \\
T \vdash (q, (ls', s, r)) & \leadsto (q', (l, s', sr)) & \textit{if} & (q, \leftarrow, q') \quad\ \in \delta \ , \\
T \vdash (q, (l, s, r)) & \leadsto (q', (l, s, r)) & \textit{if} & (q, \downarrow, q') \quad\ \in \delta \ , \\
T \vdash (q, (l, s, s'r)) & \leadsto (q', (ls, s', r)) & \textit{if} & (q, \rightarrow, q') \quad\ \in \delta \ .
\end{array}$$

Definition 4 (Local forward/backward determinism). *A TM $T = (Q, \Sigma, \delta, b, q_s, q_f)$ is* locally forward deterministic *iff for any distinct pair of transition rule triples (q_1, a_1, q'_1), $(q_2, a_2, q'_2) \in \delta$, if $q_1 = q_2$ then $a_1 = (s_1, s'_1)$ and $a_2 = (s_2, s'_2)$, and $s_1 \neq s_2$. A TM T is* locally backward deterministic *iff for any distinct pair of triples $(q_1, a_1, q'_1), (q_2, a_2, q'_2) \in \delta$, if $q'_1 = q'_2$ then $a_1 = (s_1, s'_1)$ and $a_2 = (s_2, s'_2)$, and $s'_1 \neq s'_2$.*

As an example, the pair $(\mathtt{q}, (\mathtt{a}, \mathtt{b}), \mathtt{p})$ and $(\mathtt{q}, (\mathtt{a}, \mathtt{c}), \mathtt{p})$ respects backward determinism (but not forward determinism); the pair $(\mathtt{q}, (\mathtt{a}, \mathtt{b}), \mathtt{p})$ and $(\mathtt{r}, (\mathtt{c}, \mathtt{b}), \mathtt{p})$ is not backward deterministic; and neither is the pair $(\mathtt{q}, (\mathtt{a}, \mathtt{b}), \mathtt{p})$ and $(\mathtt{r}, \rightarrow, \mathtt{p})$[3].

Definition 5 (Reversible Turing machine). *A TM T is* reversible *iff it is locally forward and backward deterministic.*

[2] When describing tape contents we shall use the empty string ε to denote the infinite string of blanks b^ω, and shall usually omit it when unambiguous.

[3] When we use `typewriter` font we usually refer to concrete instances, rather than variables. Thus, in this example `q` and `p` refers to different concrete states.

The reversible Turing machines (RTMs) are thus a proper subset of the set of all Turing machines, with an easily decidable property. We need the following important lemma. Note that this applies to *each* computation step.

Lemma 1. *If T is a reversible Turing machine, then the induced computation step relation $T \vdash \cdot \rightsquigarrow \cdot$ is an injective function on configurations.*

3 Semantics for Turing Machines

What do Turing machines compute? In other words, what is the codomain and definition of the semantics function $[\![\cdot]\!] : \text{TMs} \rightarrow ?$ for Turing machines? This might seem an odd question seeing as we have just defined how TMs work, but the answer depends on a concrete semantical choice, and has a profound effect on the computational strength of the RTMs. (Note: For the rest of this paper, we shall consider the relationship mainly between *deterministic* and *reversible* Turing machines. Thus, all TMs are assumed to be fwd deterministic).

At this point, the expert reader might object that the original results by Landauer [9] and Bennett [4] (cf. Lemmas 4 and 5) show exactly how we can "reversibilize" any computation, and that the RTMs should therefore be able to compute exactly what the TMs in general can compute. Furthermore, Morita and Yamaguchi [13] exhibited a universal reversible Turing machine, so the universality of the RTMs should already be established. As we shall see, however, if one takes reversibility as also including the input/output behaviour of the machines, neither of these claims hold: Reversibilization is *not* semantics preserving, and the RTMs are *not* universal in the classical sense.

There are several reasons for considering the extensional behavior of RTMs to itself be subject to reversibility.

- The *reversible* machines, computation models and programming languages, form a much *larger* class than just the *reversibilized* machines of Landauer and Bennett performing *reversible simulations of irreversible machines.*
- It leads to a richer and more elegant (functional) theory for reversible programs: Program composition becomes function composition, program inversion becomes function inversion, etc., and we are able to use such properties directly and intuitively in the construction of new reversible programs. This is *not* the case for reversible simulations.
- If we can *ad hoc* dismiss part of the output configuration, there seems to be little to constrain us from allowing such irreversibilities as part of the computation process as well.

In order to talk about input/output behavior on tapes in a regular fashion, we use the following definition.

Definition 6 (Standard configuration). *A tape containing a finite, blank-free string $s \in (\Sigma\backslash\{b\})^*$ is said to be given in* standard configuration *for a TM $(Q, \Sigma, \delta, b, q_s, q_f)$ iff the tape head is positioned to the* immediate left *of s on the tape, i.e. if for some $q \in Q$, the configuration of the TM is $(q, (\varepsilon, b, s))$.*

We shall consider the tape input/output (function) behavior. Here, the semantic function of a Turing machine is defined by its effect on the entire configuration.

Definition 7 (String transformation semantics). *The* semantics $[\![T]\!]$ *of a TM* $T = (Q, \Sigma, \delta, b, q_s, q_f)$ *is given by the relation*

$$[\![T]\!] = \{(s, s') \in ((\Sigma\backslash\{b\})^* \times (\Sigma\backslash\{b\})^*) \mid T \vdash (q_s, (\varepsilon, b, s)) \leadsto^* (q_f, (\varepsilon, b, s'))\}.$$

Intuitively, a computation is performed as follows. In starting state q_s, with input s given in standard configuration $(q_s, (\varepsilon, b, s))$, repeatedly apply $\leadsto$, until the machine halts (if it halts) in standard configuration $(q_f, (\varepsilon, b, s'))$. To differentiate between semantics and mechanics, we shall write $T(x)$ to mean the computation of $[\![T]\!](x)$ by the specific machine T. We say that T *computes* function f iff $[\![T]\!] = f$. Thus, the *string transformation semantics* of a TM T has type

$$[\![T]\!] : \Sigma^* \rightharpoonup \Sigma^* \ .$$

Under this semantics there is a one-to-one correspondence between input/output strings and the configurations that represent them, so the machine input/output behaviour is logically reversible. In contrast to this, the (implicit) semantics used for decision problems (*language recognition*) gives us programs of type

$$[\![T]\!]_{dp} : \Sigma^* \rightharpoonup \{accept, reject\} \ ,$$

where halting configurations are projected down to a single bit. It is well known that for classical Turing machines it does not matter computability-wise which of these two semantics we choose. (There is a fairly straightforward translation from languages to functions and *vice versa*.) Anticipating the Landauer embedding of Lemma 4 it is easy to see that under the language recognition semantics then the RTMs are universal: Given a TM T recognizing language L, there exists an RTM T' that also recognizes L. However, under the string transformation semantics the RTMs *cannot* be universal.

Theorem 1. *If* T *is an RTM, then* $[\![T]\!]$ *is injective.*

Proof. By induction, using Lemma 1. □

It thus makes little sense to talk about what RTMs compute without explicitly specifying the semantics.

4 Foundations of Reversible Computing

At this point it becomes necessary to recast the foundational results of reversible computing in terms of the strict semantical interpretation above.

4.1 Inversion

If f is a computable injective function, is the inverse function f^{-1} computable?

Lemma 2 (TM inversion, McCarthy [10]). *Given a TM T computing an injective function $[\![T]\!]$, there exists a TM $M(T)$, such that $[\![M(T)]\!] = [\![T]\!]^{-1}$.*

It is interesting to note that McCarthy's generate-and-test approach [10] does not actually give the *program inverter* (computing the transformation M), but rather an *inverse interpreter*, cf. [1]. However, we can turn an inverse interpreter into a program inverter by specialization [7], so the transformation M is computable.

The generate-and-test method used by McCarthy is sufficient to show the existence of an inverse interpreter, but unsuitable for practical usage as it is very inefficient. For the RTMs there is an appealing alternative.

Lemma 3 (RTM inversion, Bennett [4]). *Given an RTM $T = (Q, \Sigma, \delta, b, q_s, q_f)$, the RTM $T^{-1} \overset{\text{def}}{=} (Q, \Sigma, inv(\delta), b, q_f, q_s)$ computes the inverse function of $[\![T]\!]$, i.e. $[\![T^{-1}]\!] = [\![T]\!]^{-1}$, where $inv : \Delta \to \Delta$ is defined as*

$$\begin{array}{lcl@{\qquad}lcl} inv(q,(s,s'),q') &=& (q',(s',s),q) & inv(q,\leftarrow,q') &=& (q',\rightarrow,q) \\ inv(q,\downarrow,q') &=& (q',\downarrow,q) & inv(q,\rightarrow,q') &=& (q',\leftarrow,q)\ . \end{array}$$

This remarkably simply transformation is one of the great insights in Bennett's seminal 1973 paper [4] that may superficially seem trivial. Here, we have additionally shown that it is an example of local (peephole) program inversion. Note that the transformation *only* works as a program inversion under the string transformation semantics, and *not* under language recognition. In the following, we shall make heavy use of program inversion, so the direct coupling between the mechanical and semantical transformation is significant.

4.2 Reversibilization

How can irreversible TMs computing (possibly non-injective) functions be *reversibilized*, i.e., transformed into RTMs?

Lemma 4 (Landauer embedding [9]). *Given a 1-tape TM $T = (Q, \Sigma, \delta, b, q_s, q_f)$, there is a 2-tape RTM $L(T)$ such that $[\![L(T)]\!] : \Sigma^* \rightharpoonup \Sigma^* \times R^*$, and*

$$[\![L(T)]\!] = \lambda x.([\![T]\!](x), trace(T,x)),$$

where $trace(T,x)$ is a complete trace of the specific rules from δ (enumerated as R) that are applied during the computation $T(x)$.

The Landauer embedding is named in honor of Rolf Landauer, who suggested the idea of a trace to ensure reversibility [9]. It is historically the first example of what we call "reversibilization," the addition of *garbage data* to the output in order to guarantee reversibility. The Landauer embedding shows that any computable function can be injectivized such that it is computable by a reversible TM.

The size of the garbage data $trace(T,x)$ is of order of the number of steps in the computation $T(x)$, which makes it in general unsuited for practical programming. The trace is also machine-specific: Given functionally equivalent TMs T_1 and T_2, i.e., $[\![T_1]\!] = [\![T_2]\!]$, it will almost always be the case that $[\![L(T_1)]\!] \neq [\![L(T_2)]\!]$. The addition of the trace also changes the space consumption of the original program.

It is preferable that an injectivization generates *extensional* garbage data (specific to the function) rather than *intensional* garbage data (specific to the machine), since we would like to talk about semantics and ignore the mechanics. This is attained in the following Lemma, known colloquially as "Bennett's trick."

Lemma 5 (Bennett's method [4]). *Given a 1-tape TMT* $= (Q, \Sigma, \delta, b, q_s, q_f)$*, there exists a 3-tape RTM* $B(T)$*, s.t.*

$$[\![B(T)]\!] = \lambda x.(x, [\![T]\!](x)) \ .$$

While the construction (shown below) is defined for 1-tape machines, it can be extended to Turing machines with an arbitrary number of tapes. It is important to note that neither Landauer embedding nor Bennett's method are semantics preserving as both reversibilizations lead to garbage:

$$[\![L(T)]\!] \neq [\![T]\!] \neq [\![B(T)]\!] \, .$$

4.3 Reversible Updates

Bennett's method implements a special case of a *reversible update* [3], where D (below) is a simple "copying machine", and the second input is initially blank:

Theorem 2. *Assume that* $\odot : (\Sigma^* \times \Sigma^*) \rightarrow \Sigma^*$ *is a (computable) operator injective in its first argument: If* $b \odot a = c \odot a$*, then* $b = c$*. Let* D *be an RTM computing the injective function* $\lambda(a,b).(a, b \odot a)$*, and let* T *be any TM. Let* $L_1(T)$ *be an RTM that applies* $L(T)$ *to the first argument* x *of a pair* (x,y) *(using an auxiliary tape for the trace.) We have*[4]

$$[\![L_1(T)^{-1} \circ D \circ L_1(T)]\!] = \lambda(x,y).(x, y \odot [\![T]\!](x)) \ .$$

Reversible updates models many reversible language constructs [21], and is also useful when designing reversible circuits [16,17]. We found this generalization to be of great practical use in the translation from high-level to low-level reversible languages [2], as it directly suggests a translation strategy for reversible updates.

5 Robustness

The Turing machines are remarkably computationally robust. Using multiple symbols, tapes, heads etc. has no impact on computability. Above, we have been silently assuming that the same holds true for the RTMs: The Landauer embedding takes n-tape machine to $n+1$-tape machines, the Bennett trick takes 1-tape machines to 3-tape machines, etc.

[4] The mechanical concatenation of two machines $T_2 \circ T_1$ is straightforward, and it is an easy exercise to see that $[\![T_2 \circ T_1]\!] = [\![T_2]\!] \circ [\![T_1]\!]$.

Are these assumptions justified? We have seen that a precise characterization of the semantics turned out to have a huge impact on computational expressiveness (limiting us to injective functions.) It would not be unreasonable to expect the RTMs to suffer additional restrictions *wrt* the parameters of machine space.

First, we consider the question of multiple symbols. Morita *et al.* [12] showed how to simulate a 1-tape, 32-symbol RTM by a 1-tape 2-symbol RTM. One can generalize this result to an arbitrary number of symbols. Furthermore, we also need to adapt it to work when applying our string transformation semantics such that the encodings can be efficient[5].

Lemma 6 (m-symbol RTM to 3-symbol RTM). *Given a 1-tape, m-symbol RTM $T = (Q, \Sigma, \delta, b, q_s, q_f)$, $|\Sigma| = m$, there is a 1-tape, 3-symbol RTM $T' = (Q', \{b, \mathit{0}, \mathit{1}\}, \delta', b, q_s, q_f)$ s.t. $[\![T]\!](x) = y$ iff $[\![T']\!](e(x)) = e(y)$, where $e : (\Sigma \backslash \{b\})^* \to \{\mathit{0}, \mathit{1}\}^*$ is an injective binary encoding of (blank-free) strings, with b encoded by a sequence of blanks.*

Thus, the number of symbols in a tape alphabet is not important, and a fixed-size alphabet (with at least 3 distinct symbols) can be used for all computations.

We now turn to the question of multiple tapes.

Lemma 7 (2-tape RTM to 1-tape RTM). *Given a 2-tape RTM T, there exists a 1-tape RTM T' s.t. $[\![T]\!](x, y) = (u, v)$ iff $[\![T']\!](\langle x, y\rangle) = \langle u, v\rangle$, where $\langle x, y\rangle = x_1 y_1, x_2 y_2, \ldots$ is the pairwise character sequence (convolution) of strings x and y (with blanks at the end of the shorter string as necessary.)*

The main difficulty in proving this is that the original 2-tape machine may allow halting configurations where the tape heads end up displaced an unequal number of cells from their starting positions. Thus "zipping" the tapes into one tape will not necessarily give the convolution of the outputs in standard configuration. This is corrected by realigning the simulated tapes for each rule where the original tape heads move differently.

This result generalizes to an arbitrary number of tapes. Combining these two lemmas yields the following statement of robustness.

Theorem 3 (Robustness of the RTMs). *Let T be a k-tape, m-symbol RTM. Then there exists a 1-tape, 3-symbol RTM T' s.t.*

$$[\![T]\!](x_1, \ldots, x_k) = (y_1, \ldots, y_k) \quad \textit{iff} \quad [\![T']\!](e(\langle x_1, \ldots, x_k\rangle)) = e(\langle y_1, \ldots y_k\rangle)\,,$$

where $\langle\cdot\rangle$ is the convolution of tape contents, and $e(\cdot)$ is a binary encoding.

This retroactively justifies the use of the traditional transformational approaches.

6 Exact Computational Expressiveness of the RTMs

We have outlined the two classical reversibilizations that turn TMs into RTMs. However, they are not semantics-preserving, and do not tell us anything about

[5] A 2-symbol machine can only have unary input strings in standard configuration, as one of the two symbols must be the blank symbol.

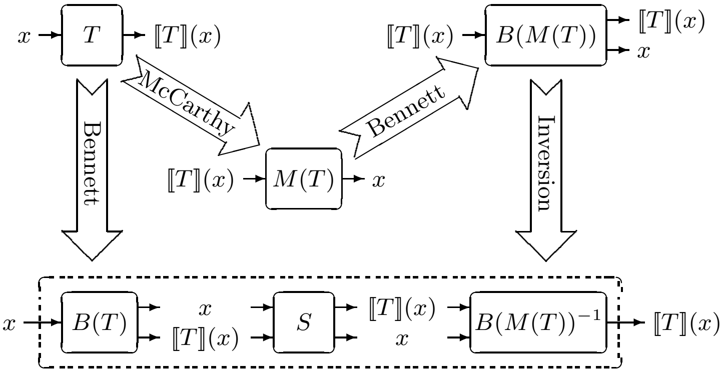


Fig. 1. Generating an RTM computing (injective) $[\![T]\!]$ from an irreversible TM T

the *a priori* computational expressiveness of RTMs. By Theorem 1 the RTMs compute only injective functions. How many such functions *can* they compute?

Theorem 4 (Reversibilizing injections, Bennett [4]). *Given a 1-tape TM S_1 s.t. $[\![S_1]\!]$ is injective, and given a 1-tape TM S_2 s.t. $[\![S_2]\!] = [\![S_1]\!]^{-1}$, there exists a 3-tape RTM T s.t. $[\![T]\!] = [\![S_1]\!]$.*

We can use this to establish the exact computational expressiveness of the RTMs.

Theorem 5 (Expressiveness). *The RTMs can compute* exactly all *injective computable functions. That is, given a 1-tape TM T such that $[\![T]\!]$ is an injective function, there is a 3-tape RTM T' such that $[\![T]\!] = [\![T']\!]$.*

This theorem then follows from the existence of a TM inverter (Lemma 2) and Theorem 4. We make use of the construction used by Bennett to prove Theorem 4, but now operating purely at the semantics level, making direct use of the transformations, and without assuming that an inverse for T is given *a priori.*

Proof. We construct and concatenate three RTMs (see Fig. 1 for a graphical representation.) First, construct $B(T)$ by applying Lemma 5 directly to T:

$$[\![B(T))]\!] = \lambda x.(x, [\![T]\!](x)) \ , \quad B(T) \in \text{RTMs}$$

Second, construct the machine $B(M(T))^{-1}$ by successively applying the transformations of Lemmas 2, 5 and 3 to T:

$$[\![B(M(T))^{-1}]\!] = (\lambda y.(y, [\![T]\!]^{-1}(y)))^{-1} \ , \quad B(M(T))^{-1} \in \text{RTMs}$$

Third, we can construct an RTM S, s.t. $[\![S]\!] = \lambda(a, b).(b, a)$, that is, a machine to exchange the contents of two tapes (in standard configuration). To see that $[\![B(M(T))^{-1} \circ S \circ B(T)]\!] = [\![T]\!]$, we apply the machine to an input, x:

$$\begin{aligned}[\![B(M(T))^{-1} \circ S \circ B(T)]\!](x) &= [\![B(M(T))^{-1} \circ [\![S]\!]]\!]\ (x, [\![T]\!](x)) \\ &= [\![B(M(T))^{-1}]\!]\ ([\![T]\!](x), x)) \\ &= (\lambda y.(y, [\![T]\!]^{-1}(y)))^{-1}\ ([\![T]\!](x), x) \\ &= (\lambda y.(y, [\![T]\!]^{-1}(y)))^{-1}\ ([\![T]\!](x), [\![T]\!]^{-1}([\![T]\!](x))) \\ &= [\![T]\!](x)\ . \qquad \square\end{aligned}$$

Thus, the RTMs can compute exactly all the injective computable functions. This suggests that the RTMs have the maximal computational expressiveness we could hope for in *any* (effective) reversible computing model.

7 Universality

Having characterized the computational *expressiveness* of the RTMs, an obvious next question is that of computation *universality.* A universal machine is a machine that can simulate the functional behaviour of any other machine. For the classical, irreversible Turing machines, we have the following definition.

Definition 8 (Classical universality). *A TM U is* classically universal *iff for all TMs T, all inputs $x \in \Sigma^*$, and Gödel number $\ulcorner T \urcorner \in \Sigma^*$ representing T:*

$$[\![U]\!](\ulcorner T \urcorner, x) = [\![T]\!](x)\ .$$

The actual Gödel numbering $\ulcorner\cdot\urcorner$: TMs → Σ^* for a given universal machine is not important, but we do require that it is computable and injective (up to renaming of symbols and states).

Because $[\![U]\!]$ in this definition is a *non-injective* function, it is clear that *no classically universal RTM exists!* Bennett [4] suggests that if U is a (classically) universal machine, $B(U)$ is a machine for reversibly simulating any irreversible machine. However, $B(U)$ is not itself universal, $[\![B(U)]\!] \neq [\![U]\!]$, and furthermore we should not use *reversible simulation of irreversible machines* as a benchmark.

The appropriate question to ask is whether the RTMs are classically universal for just their own class, i.e. where the interpreted machine T is restricted to being an RTM. The answer is, again, no: Different programs may compute the same function, so there exists RTMs $T_1 \neq T_2$ such that $[\![T_1]\!](x) = [\![T_2]\!](x)$, so $[\![U]\!]$ is *inherently* non-injective, and therefore not computable by any RTM.

Classical universality is thus unsuitable if we want to capture a similar notion wrt RTMs. We propose that a universal machine should be allowed to remember which machine it simulates.

Definition 9 (Universality). *A TM U_{TM} is* universal *iff for all TMs T and all inputs $x \in \Sigma^*$,*

$$[\![U_{TM}]\!](\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, [\![T]\!](x))\ .$$

This is equivalent to the original definition of classical universality[6]. Importantly, it now suggests a concept of universality that *can* apply to RTMs.

Definition 10 (RTM-universality). *An RTM U_{RTM} is* RTM-universal *iff for all RTMs T and all inputs $x \in \Sigma^*$,*

$$[\![U_{RTM}]\!](\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, [\![T]\!](x)) \ .$$

Now, is there an RTM-universal reversible Turing machine, a *URTM*?

Theorem 6 (URTM existence). *There exists an RTM-universal RTM U_R.*

Proof. We show that an RTM U_R exits, such that for all RTMs T, $[\![U_R]\!](\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, [\![T]\!](x))$. Clearly, $[\![U_R]\!]$ is computable, since T is a TM (so $[\![T]\!]$ is computable), and $\ulcorner T \urcorner$ is given as input. We show that $[\![U_R]\!]$ is injective: Assuming $(\ulcorner T_1 \urcorner, x_1) \neq (\ulcorner T_2 \urcorner, x_2)$ we show that $(\ulcorner T_1 \urcorner, [\![T_1]\!](x_1)) \neq (\ulcorner T_2 \urcorner, [\![T_2]\!](x_2))$. Either $\ulcorner T_1 \urcorner \neq \ulcorner T_2 \urcorner$ or $x_1 \neq x_2$ or both. Because the program text is passed through to the output, the first and third cases are trivial. Assuming that $x_1 \neq x_2$ and $\ulcorner T_1 \urcorner = \ulcorner T_2 \urcorner$, we have that $[\![T_1]\!] = [\![T_2]\!]$, i.e. T_1 and T_2 are the same machine, and so compute the same function. Because they are RTMs this function is injective (by Theorem 1), so $x_1 \neq x_2$ implies that $[\![T_1]\!](x_1) \neq [\![T_2]\!](x_2)$. Therefore, $[\![U_R]\!]$ is injective, and by Theorem 5 computable by some RTM U_R. □

We remark that this works out very nicely: RTM-universality is now simply universality restricted to interpreting the RTMs, and while general universality is non-injective, RTM-universality becomes exactly injective by virtue of the semantics of RTMs. Also, by interpreting just the RTMs, we remove the redundancy (and reliance on reversibilization) inherent in the alternatives.

Given an irreversible TM computing the function of RTM-universality, Theorem 5 provides us with a possible construction for an RTM-universal RTM. However, we do not actually directly have such machines in the literature, and in any case the construction uses the very inefficient generate-and-test inverter by McCarthy. We can do better.

Lemma 8. *There exists an RTM pinv, such that pinv is a program inverter for RTM programs,*

$$[\![pinv]\!](\ulcorner T \urcorner) = \ulcorner T^{-1} \urcorner \ .$$

This states that the RTMs are expressive enough to perform the program inversion of Lemma 3. For practical Gödelizations this will only take linear time.

Theorem 7 (UTM to URTM). *Given a classically universal TM U s.t. $[\![U]\!](\ulcorner T \urcorner, x) = [\![T]\!](x)$, the RTM U_R defined as follows is RTM-universal.*

$$U_R = pinv_1 \circ (B(U))^{-1} \circ S_{23} \circ pinv_1 \circ B(U) \, ,$$

where $pinv_1$ is an RTM that applies RTM program inversion on its first argument, $[\![pinv_1]\!](p, x, y) = ([\![pinv]\!]p, x, y)$, and S_{23} is an RTM that swaps its second and third arguments, $[\![S_{23}]\!] = \lambda(x, y, z).(x, z, y)$.

[6] Given U_{TM} universal by Definition 9, $snd \circ U_{TM}$ is classically universal, where snd is a TM s.t. $[\![snd]\!] = \lambda(x, y).y$. The converse is analogous.

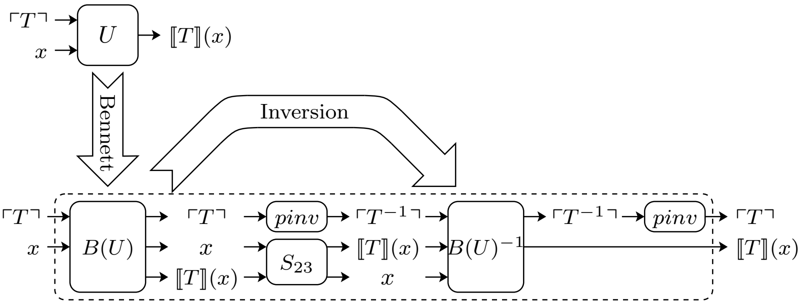


Fig. 2. Constructing an RTM-universal RTM U_R from a classically universal TM U

Proof. We must show that $\llbracket U_R \rrbracket(\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, \llbracket T \rrbracket(x))$ for any RTM T. To show this, we apply U_R to an input $(\ulcorner T \urcorner, x)$. Fig. 2 shows a graphical representation of the proof.

$$\begin{aligned}
\llbracket U_R \rrbracket(\ulcorner T \urcorner, x) &= \llbracket pinv_1 \circ (B(U))^{-1} \circ S_{23} \circ pinv_1 \circ B(U) \rrbracket(\ulcorner T \urcorner, x) \\
&= \llbracket pinv_1 \circ (B(U))^{-1} \circ S_{23} \circ pinv_1 \rrbracket(\ulcorner T \urcorner, x, \llbracket T \rrbracket(x)) \\
&= \llbracket pinv_1 \circ (B(U))^{-1} \circ S_{23} \rrbracket(\ulcorner T^{-1} \urcorner, x, \llbracket T \rrbracket(x)) \\
&= \llbracket pinv_1 \circ (B(U))^{-1} \rrbracket(\ulcorner T^{-1} \urcorner, \llbracket T \rrbracket(x), x) \\
&= \llbracket pinv_1 \rrbracket(\ulcorner T^{-1} \urcorner, \llbracket T \rrbracket(x)) \\
&= (\ulcorner T \urcorner, \llbracket T \rrbracket(x)) \ . \qquad \square
\end{aligned}$$

Note that this implies that RTMs can simulate themselves exactly as time-efficiently as the TMs can simulate themselves, but the space usage of the constructed machine will, by using Bennett's method, be excessive. However, there is nothing that forces us to start with an irreversible (universal) machine, when constructing an RTM-universal RTM, nor are reversibilizations necessarily required (as will be seen below).

A first principles approach to an RTM-universal reversible Turing machine, which does not rely on reversibilization, remains for future work.

8 r-Turing Completeness

With a theory of the computational expressiveness and universality of the RTMs at hand, we shall lift the discussion to computation models in general. What, then, do reversible programs compute, and what can they compute?

Our fundamental assumption is that the RTMs (with the given semantics) are a good and exhaustive model for reversible computing. Thus, for every program $\mathtt{p}$ in a reversible programming language R, we assume there to be an RTM $T_{\mathtt{p}}$, s.t. $\llbracket \mathtt{p} \rrbracket_R = \llbracket T_{\mathtt{p}} \rrbracket$. Thus, because the RTMs are restricted to computing injective functions, reversible programs too compute injective functions only. On the other hand, we have seen that the RTMs are maximally expressive wrt these functions, and support a natural notion of universality. For this reason we propose the following standard of computability for reversible computing.

Definition 11 (r-Turing completeness). *A (reversible) programming language R is called* r-Turing complete *iff for all RTMs T computing function $[\![T]\!]$, there exists a program* $\mathtt{p} \in R$*, such that* $[\![\mathtt{p}]\!]_R = [\![T]\!]$.

Note that we are here quite literal about the semantics: Given an RTM T, it will *not* suffice to compute a Landauer embedded version of $[\![T]\!]$, or apply Bennett's trick, or, indeed any injectivization of $[\![T]\!]$. Program $\mathtt{p}$ *must* compute $[\![T]\!]$, *exactly.* Only if this is respected can we truly say that a reversible computation model can compute *all* the injective, computable functions, i.e. is as computationally expressive as we can expect reversible computation models to be.

Demonstrating r-Turing Completeness. A common approach to proving that a language, or computational model, is Turing-complete, is to demonstrate that a classically universal TM (a TM interpreter) can be implemented, and specialized to any TM T. However, that is for *classically* universal machines and (in general) irreversible languages, which compute non-injective functions. What about *our* notion of RTM-universality (Definition 10) and reversible languages?

Assume that $u \in R$ (where R is a reversible programming language) is an R-program computing an RTM-universal interpreter $[\![u]\!]_R(\ulcorner T \urcorner, x) = (\ulcorner T \urcorner, [\![T]\!](x))$. Assume also that R is expressive enough to guarantee the existence of programs $w_T \in R$ s.t. $[\![w_T]\!]_R = \lambda x.(\ulcorner T \urcorner, x)$, (whose sole purpose is to *hardcode* $\ulcorner T \urcorner$ as an input for u) and its inverse $w_T^{-1} \in R$, $[\![w_T^{-1}]\!]_R = [\![w_T]\!]_R^{-1}$, for any RTM T. Note that $[\![w_T]\!]_R$ is injective, so we do not violate our rule of R computing injective functions by assuming w_T and its inverse. Now $[\![u \circ w_T]\!]_R = \lambda x.(\ulcorner T \urcorner, [\![T]\!](x)) \neq [\![T]\!]$, because it leaves the representation $\ulcorner T \urcorner$ as part of the output. To complete the specialization, we need to apply w_T^{-1} as well. Thus, $[\![w_T^{-1} \circ u \circ w_T]\!]_R = [\![T]\!]$.

Therefore, completely analogous to the classical case, we may demonstrate r-Turing completeness of a reversible computation model by implementing RTM-universality (essentially, an RTM interpreter), keeping in mind that we must respect the semantics exactly (by *clean simulation* that doesn't leave *garbage*).

The authors have done exactly that to demonstrate r-Turing completeness of the imperative, high level, reversible language *Janus*, and for reversible flowchart languages in general, cf. [21,22] (where the r-Turing completeness concept was informally proposed.) In these cases, we were able to exploit the reversibility of the interpreted machines directly, and did not have to rely on reversibilization of any kind, which eased the implementation greatly. Furthermore, the RTM-interpreters are complexity-wise *robust*, in that they preserve the space and time complexities of the simulated machines, which no reversibilization is liable to do.

9 Related Work

Morita *et al.* have studied reversible Turing machines [13,12] and other reversible computation models, including cellular automata [11], with a different approach to semantics (and thus different results wrt computational strength) than the present paper. Most relevant here is the universal RTM proposed in [13]. With our strict semantics viewpoint, the construction therein does *not* directly demonstrate neither RTM-universality nor classical universality, but rather a sort of "traced" universality: Given a program for a *cyclic tag system* (a Turing complete formalism) and an input, the halting configuration encompasses both the program and output, but also the entire string produced by the tag system along the way. We believe that this machine could possibly be transformed fairly straightforwardly into a machine computing a function analogous to $[\![B(U)]\!]$. However, it is not clear that cyclic tag systems should have a notion of reversibility, so the construction in Fig. 2 is therefore not immediately applicable.

10 Conclusion

The study of reversible computation models complements that of deterministic and non-deterministic computation models. We have given a foundational treatment of a computability theory for reversible computing using a strict semantics-based approach (where input/output behaviour must also be reversible), taking reversible Turing machines as the underlying computational model. By formulating the classical transformational approaches to reversible computation in terms of this semantics, we hope to have clarified the distinction between *reversibility* and *reversibilization*, which may previously have been unclear.

We found that starting directly with reversibility leads to a clearer, cleaner, and more useful functional theory for RTMs. Natural (mechanical) *program* transformations such as composition and inversion now correspond directly to the (semantical) *function* transformations. This carries over to other computation models as well.

We showed that the RTMs compute exactly all injective, computable functions, and are thus not classically universal. We also showed that they are expressive enough to be universal for their own class, with the concept of *RTM-universality.* We introduced the concept of *r-Turing completeness* as the measure of the computational expressiveness in reversible computing. As a consequence, a definitive practical criterion for deciding the computation universality of a reversible programming computation model is now in place: Implement an RTM-interpreter, in the sense of an RTM-universal machine.

Acknowledgements. The authors wish to thank Michael Kirkedal Thomsen for help with the figures and Tetsuo Yokoyama for discussions on RTM-computability.

References

1. Abramov, S., Glück, R.: Principles of inverse computation and the universal resolving algorithm. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 269–295. Springer, Heidelberg (2002)

2. Axelsen, H.B.: Clean translation of an imperative reversible programming language. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 144–163. Springer, Heidelberg (2011)
3. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible machine code and its abstract processor architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007)
4. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development 17, 525–532 (1973)
5. Feynman, R.: Quantum mechanical computers. Optics News 11, 11–20 (1985)
6. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. ACM Trans. Prog. Lang. Syst. 29(3), article 17 (2007)
7. Glück, R., Sørensen, M.: A roadmap to metacomputation by supercompilation. In: Danvy, O., Thiemann, P., Glück, R. (eds.) Partial Evaluation. LNCS, vol. 1110, pp. 137–160. Springer, Heidelberg (1996)
8. Jones, N.D.: Computability and Complexity: From a Programming Language Perspective. In: Foundations of Computing. MIT Press, Cambridge (1997)
9. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development 5(3), 183–191 (1961)
10. McCarthy, J.: The inversion of functions defined by Turing machines. In: Automata Studies, pp. 177–181. Princeton University Press, Princeton (1956)
11. Morita, K.: Reversible computing and cellular automata — A survey. Theoretical Computer Science 395(1), 101–131 (2008)
12. Morita, K., Shirasaki, A., Gono, Y.: A 1-tape 2-symbol reversible Turing machine. Trans. IEICE, E 72(3), 223–228 (1989)
13. Morita, K., Yamaguchi, Y.: A universal reversible turing machine. In: Durand-Lose, J., Margenstern, M. (eds.) MCU 2007. LNCS, vol. 4664, pp. 90–98. Springer, Heidelberg (2007)
14. Mu, S.-C., Hu, Z., Takeichi, M.: An algebraic approach to bi-directional updating. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 2–20. Springer, Heidelberg (2004)
15. Schellekens, M.: MOQA; unlocking the potential of compositional static average-case analysis. Journal of Logic and Algebraic Programming 79(1), 61–83 (2010)
16. Thomsen, M.K., Axelsen, H.B.: Parallelization of reversible ripple-carry adders. Parallel Processing Letters 19(2), 205–222 (2009)
17. Thomsen, M.K., Glück, R., Axelsen, H.B.: Reversible arithmetic logic unit for quantum arithmetic. Journal of Physics A: Mathematics and Theoretical 42(38), 2002 (2010)
18. Toffoli, T.: Reversible computing. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 632–644. Springer, Heidelberg (1980)
19. van de Snepscheut, J.L.A.: What computing is all about. Springer, Heidelberg (1993)
20. Van Rentergem, Y., De Vos, A.: Optimal design of a reversible full adder. International Journal of Unconventional Computing 1(4), 339–355 (2005)
21. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Proceedings of Computing Frontiers, pp. 43–54. ACM Press, New York (2008)
22. Yokoyama, T., Axelsen, H.B., Glück, R.: Reversible flowchart languages and the structured reversible program theorem. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 258–270. Springer, Heidelberg (2008)

Irrelevance in Type Theory with a Heterogeneous Equality Judgement

Andreas Abel

Department of Computer Science
Ludwig-Maximilians-University Munich
andreas.abel@ifi.lmu.de

Abstract. Dependently typed programs contain an excessive amount of static terms which are necessary to please the type checker but irrelevant for computation. To obtain reasonable performance of not only the compiled program but also the type checker such static terms need to be erased as early as possible, preferably immediately after type checking. To this end, Pfenning's type theory with irrelevant quantification, that models a distinction between static and dynamic code, is extended to universes and large eliminations. Novel is a heterogeneously typed implementation of equality which allows the smooth construction of a universal Kripke model that proves normalization, consistency and decidability.

Keywords: dependent types, proof irrelevance, heterogeneously typed equality, algorithmic equality, logical relation, universal Kripke model.

1 Introduction and Related Work

Dependently typed programming languages such as Agda [9], Coq [13], and Epigram [15] allow the programmer to express in one language programs, their types, rich invariants, and even proofs of these invariants. Besides code executed at run-time, dependently typed programs contain much code needed only to pass the type checker, which is a the same time the verifier of the proofs woven into the program.

Program extraction takes type-checked terms and discards parts that are irrelevant for execution. Augustsson's dependently typed functional language Cayenne [6] erases *types* using a universe-based analysis. Coq's extraction procedure has been designed by Paulin-Mohring and Werner [21] and Letouzey [14] and discards not only types but also proofs. The erasure rests on Coq's universe-based separation between propositional (Prop) and computational parts (Set/Type). The rigid Prop/Set distinction has the drawback of code duplication: A structure which is sometimes used statically and sometimes dynamically needs to be coded twice, once in Prop and once in Set.

An alternative to the fixed Prop/Set-distinction is to let the usage context decide whether a term is a proof or a program. Besides whole-program analyses such as data flow, some type-based analyses have been put forward. One of them is Pfenning's modal type theory of *Intensionality, Extensionality, and Proof Irrelevance* [22] which introduces functions with irrelevant arguments that play the role of proofs. Not only can these arguments be erased during extraction, they can also be disregarded in type conversion tests during type checking. This relieves the user of unnecessary proof burden

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 57–71, 2011.

(proving that two proofs are equal). Furthermore, proofs can not only be discarded during program extraction but directly after type checking, since they will never be looked at again during type checking subsequent definitions.

In principle, we have to distinguish "post mortem" program extraction, let us call it *external erasure*, and proof disposal during type checking, let us call it *internal erasure*. External erasure deals with closed expressions, programs, whereas internal erasure deals with open expressions that can have free variables. Such free variables might be assumed proofs of (possibly false) equations and block type casts, or (possibly false) proofs of well-foundedness and prevent recursive functions from unfolding indefinitely. For type checking to not go wrong or loop, those proofs can only be externally erased, thus, the Prop/Set distinction is not for internal erasure. In Pfenning's type theory, proofs can never block computations even in open expressions (other than computations on proofs), thus, internal erasure is sound.

Miquel's Implicit Calculus of Constructions (ICC) [17] goes further than Pfenning and considers also *parametric* arguments as irrelevant. These are arguments which are irrelevant for function execution but relevant during type conversion checking. Such arguments may only be erased in function application but not in the associated type instantiation. Barras and Bernardo [8] and Mishra-Linger and Sheard [19] have build decidable type systems on top of ICC, but both have not fully integrated inductive types and types defined by recursion (large eliminations). Barras and Bernardo, as Miquel, have inductive types only in the form of their impredicative encodings, Mishra-Linger [20] gives introduction and elimination principles for inductive types by example, but does not show normalization or consistency.

Our long-term goal is to equip Agda with internal and external erasure. To this end, a type theory for irrelevance is needed that supports user-defined data types and functions and types defined by pattern matching. Experiments with my prototype implementation MiniAgda [2] have revealed some issues when combining Miquel-style irrelevance with large eliminations (see Ex. 2 in Sec. 2). Since it is unclear whether these issues can be resolved, I have chosen to scale Pfenning's notion of proof irrelevance up to inductive types.

In this article, we start with the "extensionality and proof irrelevance" fragment of Pfenning's type theory in Reed's version [23,24]. We extend it by a hierarchy of predicative universes, yielding *Irrelevant Intensional Type Theory* IITT (Sec. 2). Based on a heterogeneous algorithmic equality which compares two expressions, each in its own context at its own type (Sec. 3), we smoothly construct a Kripke model that is both sound and complete for IITT (Sec. 4). It allows us to prove soundness and completeness of algorithmic equality, normalization, subject reduction, consistency, and decidability of typing in one go (Sec. 5). The model is ready for data types, large eliminations, types with extensionality principles, and internal erasure (Sec. 6).

The novel technical contributions of this work are a heterogeneous formulation of equality in the specification of type theory, and the universal Kripke model that yields all interesting meta-theoretic results at once.

The Kripke model is inspired by previous work on normalization by evaluation [3]. There we have already observed that a heterogeneous treatment of algorithmic equality solves the problem of defining a Kripke logical relation that shows completeness of algorithmic equality. Harper and Pfenning [12] hit the same problem, and their fix was to erase dependencies in types. In weak type theories like the logical framework erasure is possible, but it does not scale to large eliminations.

Related to our present treatment of IITT is Goguen's *Typed Operational Semantics* [11]. He proves meta-theoretic properties such as normalization, subject reduction, and confluence by a Kripke logical predicate of well-typed terms. However, his notion of equality is based on reduction and not a step-wise algorithm.

Awodey and Bauer [7] give a categorical treatment of proof irrelevance which is very similar to Pfenning and Reed's. However, they work in the setting of Extensional Type Theory with undecidable type checking, I could not directly use their results for this work.

Due to lack of space, proofs have been mostly omitted; more proofs are available in an extended version of this article on the author's home page.

2 Irrelevant Intensional Type Theory

In this section, we present *Irrelevant Intensional Type Theory* IITT which features two of Pfenning's function spaces [22], the ordinary "extensional" $(x : U) \to T$ and the proof irrelevant $(x{\div}U) \to T$. The main idea is that the argument of a $(x{\div}U) \to T$ function is counted as a proof and can neither be returned nor eliminated on, it can only be passed as argument to another proof irrelevant function or data constructor. Technically, this is realized by annotating variables as relevant, $x : U$, or irrelevant, $x \div U$, in the typing context, to restrict the use of irrelevant variables to use in irrelevant arguments.

Expression and context syntax. We distinguish between relevant ($t\,^{:}u$ or simply $t\;u$) and irrelevant application ($t\,^{\div}u$). Accordingly, we have relevant ($\lambda x : U.\, T$) and irrelevant abstraction ($\lambda x{\div}U.\, T$). Our choice of typed abstraction is not fundamental; a bidirectional type-checking algorithm [10] can reconstruct type and relevance annotations at abstractions and applications.

$$\begin{array}{llll}
\mathsf{Var} \ni x, y, X, Y & & \\
\mathsf{Sort} \ni s & ::= \mathsf{Set}_k \;\; (k \in \mathbb{N}) & \text{universes} \\
\mathsf{Ann} \ni \star & ::= \div \mid : & \text{annotation: irrelevant, relevant} \\
\mathsf{Exp} \ni t, u, T, U & ::= s \mid (x{\star}U) \to T & \text{sort, (ir)relevant function type} \\
 & \quad\mid x \mid \lambda x{\star}U.\, t \mid t\,^{\star}u & \text{lambda-calculus} \\
\mathsf{Cxt} \ni \Gamma, \Delta & ::= \diamond \mid \Gamma.\, x{\star}T & \text{empty, (ir)relevant extension}
\end{array}$$

Expressions are considered modulo α-equality, we write $t \equiv t'$ when we want to stress that t and t' identical (up to α).

Sorts. IITT is a pure type system (PTS) with infinite hierarchy of predicative universes $\mathsf{Set}_0 : \mathsf{Set}_1 : ....$ The universes are not cumulative. We have the PTS axioms $\mathsf{Axiom} = \{(\mathsf{Set}_i, \mathsf{Set}_{i+1}) \mid i \in \mathbb{N}\}$ and the rules $\mathsf{Rule} = \{(\mathsf{Set}_i, \mathsf{Set}_j, \mathsf{Set}_{\max(i,j)}) \mid i, j \in \mathbb{N}\}$. As customary, we will write the side condition $(s, s') \in \mathsf{Axiom}$ just as (s, s') and likewise $(s_1, s_2, s_3) \in \mathsf{Rule}$ just as (s_1, s_2, s_3). IITT is a full and functional PTS, which means that for all s_1, s_2 there is exactly one s_3 such that (s_1, s_2, s_3). As a consequence, there is no subtyping, types are unique up to equality.

Substitutions. σ are maps from variables to expressions. We require that the domain $\mathsf{dom}(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite. We write id for the identity substitution and $[u/x]$ for the singleton substitution σ with $\mathsf{dom}(\sigma) = \{x\}$ and $\sigma(x) = u$. Capture avoiding parallel substitution of σ in t is written as juxtaposition $t\sigma$.

Contexts. Γ feature two kinds of bindings, relevant $(x : U)$ and irrelevant $(x \div U)$ ones. Only relevant variables are in scope in an expression. Resurrection $\Gamma^{\oplus}$ turns all irrelevant bindings $x \div T$ into relevant $x : T$ ones [22]. It is the tool to make irrelevant variables, also called proof variables, available in proofs. Extending context Γ by some bindings to context Δ is written $\Delta \leq \Gamma$.

Judgements of IITT

$\vdash \Gamma$	context Γ is well-formed
$\vdash \Gamma = \ \vdash \Gamma'$	contexts Γ and Γ' are well-formed and equal
$\Gamma \vdash t : T$	in context Γ, expression t has type T
$\Gamma \vdash t : T = \Gamma' \vdash t' : T'$	typed expressions t and t' are equal

Derived judgements

$$\begin{array}{lcl}
\Gamma \vdash t \div T & \iff & \Gamma^{\oplus} \vdash t : T \\
\Gamma \vdash t \div T = \Gamma' \vdash t' \div T' & \iff & \Gamma \vdash t \div T \text{ and } \Gamma' \vdash t' \div T' \\
\Gamma \vdash t = t' \star T & \iff & \Gamma \vdash t \star T = \Gamma \vdash t' \star T \\
\Gamma \vdash T & \iff & \Gamma \vdash T : s \text{ for some } s \\
\Gamma \vdash T = \Gamma' \vdash T' & \iff & \Gamma \vdash T : s = \Gamma' \vdash T' : s' \text{ for some } s, s' \\
\Gamma \vdash T = T' & \iff & \vdash \Gamma \text{ and } T \equiv s \equiv T' \text{ or } \Gamma \vdash T = \Gamma \vdash T'
\end{array}$$

Context well-formedness and typing. $\vdash \Gamma$ and $\Gamma \vdash t : T$, extending Reed [23] to PTS style. Note that there is no variable rule for irrelevant bindings $(x \div U) \in \Gamma$.

$$\frac{}{\vdash \diamond} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash T : s}{\vdash \Gamma.\, x{\star}T}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash s : s'}\,(s, s') \qquad \frac{\Gamma \vdash U : s_1 \qquad \Gamma.\, x{\star}U \vdash T : s_2}{\Gamma \vdash (x{\star}U) \to T : s_3}\,(s_1, s_2, s_3)$$

$$\frac{\vdash \Gamma \qquad (x{:}U) \in \Gamma}{\Gamma \vdash x : U} \qquad \frac{\Gamma.\, x{\star}U \vdash t : T}{\Gamma \vdash \lambda x{\star}U.\, t : (x{\star}U) \to T}$$

$$\frac{\Gamma \vdash t : (x{\star}U) \to T \qquad \Gamma \vdash u \star U}{\Gamma \vdash t^{\star}u : T[u/x]} \qquad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash T = T'}{\Gamma \vdash t : T'}$$

When we apply an irrelevant function $\Gamma \vdash t : (x{\div}U) \to T$ to u, the argument u is typed in the resurrected context $\Gamma^{\oplus} \vdash u : U$. This means that u is treated as a proof and the proof variables become available.

Parallel computation (β) and extensionality (η)

$$\frac{\Gamma.\, x{\star}U \vdash t : T = \Gamma'.\, x{\star}U' \vdash t' : T' \qquad \Gamma \vdash u \star U = \Gamma' \vdash u' \star U'}{\Gamma \vdash (\lambda x{\star}U.\, t)\,^{\star}u : T[u/x] = \Gamma' \vdash t'[u'/x] : T'[u'/x]}$$

$$\frac{\Gamma \vdash t : (x{\star}U) \to T = \Gamma' \vdash t' : (x{\star}U') \to T'}{\Gamma \vdash t : (x{\star}U) \to T = \Gamma' \vdash \lambda x{\star}U'.\, t'\,^{\star}x : (x{\star}U') \to T'}$$

Equivalence rules

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t : T = \Gamma \vdash t : T} \qquad \frac{\Gamma \vdash t : T = \Gamma' \vdash t' : T'}{\Gamma' \vdash t' : T' = \Gamma \vdash t : T}$$

$$\frac{\Gamma_1 \vdash t_1 : T_1 = \Gamma_2 \vdash t_2 : T_2 \qquad \Gamma_2 \vdash t_2 : T_2 = \Gamma_3 \vdash t_3 : T_3}{\Gamma_1 \vdash t_1 : T_1 = \Gamma_3 \vdash t_3 : T_3}$$

Compatibility rules

$$\frac{\vdash \Gamma = \vdash \Gamma'}{\Gamma \vdash s : s' = \Gamma' \vdash s : s'}\,(s, s') \qquad \frac{(x\!:\!U) \in \Gamma \quad \Gamma \vdash U : s = \Gamma' \vdash U' : s' \quad (x\!:\!U') \in \Gamma'}{\Gamma \vdash x : U = \Gamma' \vdash x : U'}$$

$$\frac{\begin{array}{c}\Gamma \vdash U : s_1 = \Gamma' \vdash U' : s_1' \\ \Gamma.\, x{\star}U \vdash T : s_2 = \Gamma'.\, x{\star}U' \vdash T' : s_2'\end{array}}{\Gamma \vdash (x{\star}U) \to T : s_3 = \Gamma' \vdash (x{\star}U') \to T' : s_3'}$$

$$\frac{\Gamma.\, x{\star}U \vdash t : T = \Gamma'.\, x{\star}U' \vdash t' : T'}{\Gamma \vdash \lambda x{\star}U.\, t : (x{\star}U) \to T = \Gamma' \vdash \lambda x{\star}U'.\, t' : (x{\star}U') \to T'}$$

$$\frac{\begin{array}{c}\Gamma \vdash t\!:\!(x\!:\!U) \to T = \Gamma' \vdash t'\!:\!(x\!:\!U') \to T' \\ \Gamma \vdash u : U = \Gamma' \vdash u' : U'\end{array}}{\Gamma \vdash t\, u : T[u/x] = \Gamma' \vdash t'\, u' : T'[u'/x]} \quad \frac{\begin{array}{c}\Gamma \vdash t\!:\!(x\div U) \to T = \Gamma' \vdash t'\!:\!(x\div U') \to T' \\ \Gamma^{\oplus} \vdash u : U \qquad \Gamma'^{\oplus} \vdash u' : U'\end{array}}{\Gamma \vdash t\,^{\div}u : T[u/x] = \Gamma' \vdash t'\,^{\div}u' : T'[u'/x]}$$

Conversion rule

$$\frac{\Gamma_1 \vdash t_1 : T_1 = \Gamma_2 \vdash t_2 : T_2 \qquad \Gamma_2 \vdash T_2 = T_2'}{\Gamma_1 \vdash t_1 : T_1 = \Gamma_2 \vdash t_2 : T_2'}$$

Fig. 1. Rules of heterogeneous equality

Equality. Figure 1 presents the rules to construct the judgement $\Gamma \vdash t : T = \Gamma' \vdash t' : T'$. The novelty is the heterogeneous typing: we do not locally enforce that equal terms must have equal types, but we will show it globally in Sec. 5. Note that in the compatibility rule for irrelevant application, the function arguments may be completely unrelated.

In heterogeneous judgements such as equality, we maintain the invariant that the two contexts Γ and Γ' have the same shape, i. e., bind the same variables with the same irrelevance status. Only the types bound to the variables maybe different in Γ and Γ'.

Context equality $\vdash \Gamma = \ \vdash \Gamma'$ is a partial equivalence relation (PER), i. e., a symmetric and transitive relation, given inductively by the following rules:

$$\frac{}{\vdash \diamond = \ \vdash \diamond} \qquad \frac{\vdash \Gamma = \ \vdash \Gamma' \qquad \Gamma \vdash U = \Gamma' \vdash U'}{\vdash \Gamma.\, x{\star}U = \ \vdash \Gamma'.\, x{\star}U'}$$

Typing and equality are closed under weakening. Typing enjoys the usual inversion properties. To show substitution we introduce judgements $\Delta \vdash \sigma : \Gamma$ for substitution typing and $\Delta \vdash \sigma : \Gamma = \Delta' \vdash \sigma : \Gamma'$ for substitution equality which are given inductively by the following rules:

$$\frac{\vdash \Delta}{\Delta \vdash \sigma : \diamond} \qquad \frac{\Delta \vdash \sigma : \Gamma \qquad \Gamma \vdash U \qquad \Delta \vdash \sigma(x) \star U\sigma}{\Delta \vdash \sigma : \Gamma.\, x{\star}U}$$

$$\frac{\vdash \Delta = \ \vdash \Delta'}{\Delta \vdash \sigma : \diamond = \Delta' \vdash \sigma' : \diamond} \qquad \frac{\begin{array}{c}\Delta \vdash \sigma : \Gamma = \Delta' \vdash \sigma' : \Gamma' \qquad \Gamma \vdash U = \Gamma' \vdash U' \\ \Delta \vdash \sigma(x) \star U\sigma = \Delta' \vdash \sigma'(x) \star U'\sigma'\end{array}}{\Delta \vdash \sigma : \Gamma.\, x{\star}U = \Delta \vdash \sigma' : \Gamma'.\, x{\star}U'}$$

Lemma 1 (Substitution). *Substitution equality is a PER. Further:*

1. *If $\Delta \vdash \sigma : \Gamma$ and $\Gamma \vdash t : T$ then $\Delta \vdash t\sigma : T\sigma$.*
2. *If $\Delta \vdash \sigma : \Gamma = \Delta' \vdash \sigma' : \Gamma'$. and $\Gamma \vdash t : T = \Gamma' \vdash t' : T'$ then $\Delta \vdash t\sigma : T\sigma = \Delta' \vdash t'\sigma' : T'\sigma'$.*

Example 1 (Algebraic structures). [1] In type theory, we can model an algebraic structure over a carrier set A by a record of operations and proofs that the operations have the relevant properties. Consider an extension of IITT by tuples and Leibniz equality:

$(x{\star}A) \times B$	$: \mathsf{Set}_{\max(i,j)}$	for $A : \mathsf{Set}_i$ and $x \star A \vdash B : \mathsf{Set}_j$
(a, b)	$: (x{\star}A) \times B$	for $a : A$ and $b : B[a/x]$
$\mathsf{let}\ (x, y) = p\ \mathsf{in}\ t$	$: C$	for $p : (x{\star}A) \times B$ and $x \star A, y{:}B \vdash t : C$
$a \equiv b$	$: \mathsf{Set}_i$	for $A : \mathsf{Set}_i$ and $a, b : A$
refl	$: a \equiv a$	for $A : \mathsf{Set}_i$ and $a : A$
$\mathsf{sym}\ p$	$: b \equiv a$	for $p : a \equiv b$

In the presence of a unit type $1 : \mathsf{Set}_i$ with constructor $() : 1$, the class $\mathsf{SemiGrp}$ of semigroups over a fixed $A : \mathsf{Set}_0$ can be defined as

$$\begin{array}{ll}\mathsf{Assoc} & : (A \to A \to A) \to \mathsf{Set}_0 \\ \mathsf{Assoc}\, m & = (a, b, c : A) \to m\ (m\ a\ b)\ c \equiv m\ a\ (m\ b\ c)\end{array}$$

$$\begin{array}{ll}\mathsf{SemiGrp} & : \mathsf{Set}_0 \\ \mathsf{SemiGrp} & = (m : A \to A \to A) \times (assoc \div \mathsf{Assoc}\, m) \times 1.\end{array}$$

[1] Inspired by the 2010-09-23 message of Andrea Vezzosi on the Agda mailing list.

We have marked the component $assoc$ as irrelevant which means that two $\mathsf{SemiGrp}$ structures over A are already equal when they share the operation m; the shape of the associativity proofs might differ. For instance, consider the flip operator (in a slightly sugared definition):

$$\begin{array}{ll}\mathsf{flip} & : \mathsf{SemiGrp} \to \mathsf{SemiGrp} \\ \mathsf{flip}\ (m, (assoc, u)) & = (\lambda a{:}A.\lambda b{:}A.\, m\, b\, a, (\mathsf{sym}\ assoc, ())) \\[1ex] \mathsf{thm} & : (s : \mathsf{SemiGrp}) \to \mathsf{flip}\ (\mathsf{flip}\ s) \equiv s \\ \mathsf{thm}\ s & = \mathsf{refl}\end{array}$$

A proof thm that flip cancels itself is now trivial, since $\lambda ab.\, (\lambda ab.\, m\, b\, a)\, b\, a = m$ by $\beta\eta$-equality and the $assoc$-component is irrelevant. This saves us from constructing a proof of $\mathsf{sym}\ (\mathsf{sym}\ assoc) \equiv assoc$ and the type checker from validating it. While the saving is small for this small example, it illustrates the principle.

Example 2 (Large Eliminations).[2] The ICC* [8] or EPTS [19] irrelevant function type $(x \div A) \to B$ allows x to appear *relevantly* in B. This extra power raises some issues with large eliminations. Consider

$$\begin{array}{lll}\mathsf{T} & : & \mathsf{Bool} \to \mathsf{Set}_0 \\ \mathsf{T\ true} & = & \mathsf{Bool} \to \mathsf{Bool} \\ \mathsf{T\ false} & = & \mathsf{Bool} \\[1ex] t & = & \lambda F : (b{\div}\mathsf{Bool}) \to (\mathsf{T}\, b \to \mathsf{T}\, b) \to \mathsf{Set}_0. \\ & & \lambda g : F\ \mathsf{false}\ (\lambda x : \mathsf{Bool}.\, x) \to \mathsf{Bool}. \\ & & \lambda a : F\ \mathsf{true}\ (\lambda x : \mathsf{Bool} \to \mathsf{Bool}.\lambda y : \mathsf{Bool}.\, x\, y).\ g\, a.\end{array}$$

The term t is well-typed in ICC* + T because the domain type of g and the type of a are $\beta\eta$-equal after erasure $(-)^*$ of type annotations and irrelevant arguments:

$$\begin{aligned}(F\ \mathsf{false}\ (\lambda x : \mathsf{Bool}.\, x))^* &= F\ (\lambda x x) \\ &=_{\beta\eta} F\ (\lambda x \lambda y.\, x\, y) = (F\ \mathsf{true}\ (\lambda x : \mathsf{Bool} \to \mathsf{Bool}.\lambda y : \mathsf{Bool}.\, x\, y))^*\end{aligned}$$

While a Curry view supports this, it is questionable whether identity functions at different types should be viewed as one. It is unclear how a type-directed equality algorithm (see Sec. 3) should proceed here; it needs to recognize that $x : \mathsf{Bool}$ is equal to $\lambda y{:}\mathsf{Bool}.\, x\, y : \mathsf{Bool} \to \mathsf{Bool}$. This situation is amplified by a unit type 1 with extensional equality. When we change $\mathsf{T\ true}$ to 1 and the type of a to $F\ \mathsf{true}\ (\lambda x : 1.\, ())$ then t should still type-check, because $\lambda x.\, ()$ is the identity function on 1. However, η-equality for 1 cannot checked without types, and a type-directed algorithm would end up checking $x : \mathsf{Bool}$ for equality with $() : 1$. This can never work, because by transitivity we would get that any two booleans are equal.

Summarizing, we may conclude that the type of F bears trouble and needs to be rejected. IITT does this because it forbids the irrelevant b in relevant positions such as $\mathsf{T}\, b$; ICC* lacks T altogether. Extensions of ICC* should at least make sure that b is never eliminated, such as in $\mathsf{T}\, b$. Technically, T would have to be put in a separate class of *recursive* functions, those that actually compute with their argument. We leave the interaction of the three different function types to future research.

[2] Inspired by discussions with Ulf Norell during the 11th Agda Implementor's Meeting.

3 Algorithmic Equality

The algorithm for checking equality in IITT is inspired by Harper and Pfenning [12]. Like theirs, it is type-directed, but in our case each term comes with its own type in its own typing context. The algorithm proceeds stepwise, by alternating weak head normalization and head symbol comparison. Weak head normal forms (whnfs) are given by the following grammar:

$$\begin{array}{lll} \mathsf{Whnf} \ni a, b, f, A, B, F & ::= s \mid (x{\star}U) \to T \mid \lambda x{\star}U.\, t \mid n & \text{whnf} \\ \mathsf{Wne} \;\ni n, N & ::= x \mid n\,{}^{\star}u & \text{neutral whnf} \end{array}$$

Weak head evaluation. $t \searrow a$ and active application $f \mathbin{@^\star} u \searrow a$ are given by the following rules.

$$\frac{t \searrow f \qquad f \mathbin{@^\star} u \searrow a}{t\,{}^{\star}u \searrow a} \qquad \frac{}{a \searrow a} \qquad \frac{t[u/x] \searrow a}{(\lambda x{\star}U.\, t) \mathbin{@^\star} u \searrow a} \qquad \frac{}{n \mathbin{@^\star} u \searrow n\,{}^{\star}u}$$

Instead of writing the propositions $t \searrow a$ and $P[a]$ we will sometimes simply write $P[{\downarrow}t]$. Similarly, we might write $P[f \mathbin{@^\star} u]$ instead of $f \mathbin{@^\star} u \searrow a$ and $P[a]$. In rules, it is understood that the evaluation judgement is always an extra premise, never an extra conclusion.

Type equality. $\Delta \vdash A \iff \Delta' \vdash A'$, for weak head normal forms, and $\Delta \vdash T \mathrel{\widehat{\iff}} \Delta' \vdash T'$, for arbitrary well-formed types, checks that two given types are equal in their respective contexts.

$$\frac{}{\Delta \vdash s \iff \Delta' \vdash s} \qquad \frac{\Delta \vdash N : s \longleftrightarrow \Delta' \vdash N' : s'}{\Delta \vdash N \iff \Delta' \vdash N'} \qquad \frac{\Delta \vdash {\downarrow}T \iff \Delta' \vdash {\downarrow}T'}{\Delta \vdash T \mathrel{\widehat{\iff}} \Delta' \vdash T'}$$

$$\frac{\Delta \vdash U \mathrel{\widehat{\iff}} \Delta' \vdash U' \qquad \Delta.\,x\!:\!U \vdash T \mathrel{\widehat{\iff}} \Delta'.\,x\!:\!U' \vdash T'}{\Delta \vdash (x{\star}U) \to T \iff \Delta' \vdash (x{\star}U') \to T'}$$

Structural equality. $\Delta \vdash n : A \longleftrightarrow \Delta' \vdash n' : A'$ and $\Delta \vdash n : T \mathrel{\widehat{\longleftrightarrow}} \Delta' \vdash n' : T'$ checks the neutral expressions n and n' for equality and at the same time infers their types, which are returned as output.

$$\frac{\Delta \vdash n : T \mathrel{\widehat{\longleftrightarrow}} \Delta' \vdash n' : T'}{\Delta \vdash n : {\downarrow}T \longleftrightarrow \Delta' \vdash n' : {\downarrow}T'} \qquad \frac{(x\!:\!T) \in \Delta \qquad (x\!:\!T') \in \Delta'}{\Delta \vdash x : T \mathrel{\widehat{\longleftrightarrow}} \Delta' \vdash x : T'}$$

$$\frac{\begin{array}{c}\Delta \vdash n : (x\!:\!U) \to T \longleftrightarrow \Delta' \vdash n' : (x\!:\!U') \to T' \\ \Delta \vdash u : U \mathrel{\widehat{\iff}} \Delta' \vdash u' : U'\end{array}}{\Delta \vdash n\,u : T[u/x] \mathrel{\widehat{\longleftrightarrow}} \Delta' \vdash n'\,u' : T'[u'/x]}$$

$$\frac{\Delta \vdash n : (x{\div}U) \to T \longleftrightarrow \Delta' \vdash n' : (x{\div}U') \to T'}{\Delta \vdash n\,{}^{\div}u : T[u/x] \mathrel{\widehat{\longleftrightarrow}} \Delta' \vdash n'\,{}^{\div}u' : T'[u'/x]}$$

Note that the inferred types $T[u/x]$ and $T'[u'/x]$ in the last rule are a priori different, even if T is equal to T'. This motivates a heterogeneously-typed algorithmic equality.

Type-directed equality. $\Delta \vdash t : A \Longleftrightarrow \Delta' \vdash t' : A'$ and $\Delta \vdash t : T \mathrel{\widehat{\Longleftrightarrow}} \Delta' \vdash t' : T'$ checks terms t and t' for equality and proceeds by the common structure of the supplied types, to account for η.

$$\frac{\Delta \vdash T \mathrel{\widehat{\Longleftrightarrow}} \Delta' \vdash T'}{\Delta \vdash T : s \Longleftrightarrow \Delta' \vdash T' : s'}$$

$$\frac{\Delta.\, x{\star}U \vdash t\,{}^\star x : T \mathrel{\widehat{\Longleftrightarrow}} \Delta'.\, x{\star}U' \vdash t'\,{}^\star x : T'}{\Delta \vdash t : (x{\star}U) \to T \Longleftrightarrow \Delta' \vdash t' : (x{\star}U') \to T'}$$

$$\frac{\Delta \vdash {\downarrow} t : T \mathrel{\widehat{\longleftrightarrow}} \Delta' \vdash {\downarrow} t' : T'}{\Delta \vdash t : N \Longleftrightarrow \Delta' \vdash t' : N'} \qquad \frac{\Delta \vdash t : {\downarrow} T \Longleftrightarrow \Delta' \vdash t' : {\downarrow} T'}{\Delta \vdash t : T \mathrel{\widehat{\Longleftrightarrow}} \Delta' \vdash t' : T'}$$

Note that in the but-last rule we do not check that the inferred type T of ${\downarrow} t$ equals the ascribed type N. Since algorithmic equality is only invoked for well-typed t, we now that this must always be the case. Skipping this test is a conceptually important improvement over Harper and Pfenning [12].

Lemma 2 (Algorithmic equality is a Kripke PER). *$\longleftrightarrow$, $\widehat{\longleftrightarrow}$, $\Longleftrightarrow$, and $\widehat{\Longleftrightarrow}$ are symmetric and transitive and closed under weakening.*

Extending structural equality to irrelevance, we let

$$\frac{\Delta^{\oplus} \vdash n : A \longleftrightarrow \Delta^{\oplus} \vdash n : A \qquad \Delta'^{\oplus} \vdash n' : A' \longleftrightarrow \Delta'^{\oplus} \vdash n' : A'}{\Delta \vdash n \div A \longleftrightarrow \Delta' \vdash n' \div A'}$$

and analogously for $\Delta \vdash n \div T \mathrel{\widehat{\longleftrightarrow}} \Delta' \vdash n' \div T'$.

4 A Universal Kripke Model for IITT

In this section we build, based on algorithmic equality, a universal Kripke model of typed terms that is both sound and complete for IITT. Following Goguen [11] and previous work [3], we first define a semantic universe hierarchy $\mathcal{T}_i$ whose sole purpose is to provide a measure for defining a logical relation and proving some of its properties. The limit $\mathcal{T}_\omega$ corresponds to the proof-theoretic strength or ordinal of IITT.

4.1 An Induction Measure

We denote sets of expressions by $\mathcal{A}, \mathcal{B}$ and functions from expressions to sets of expressions by $\mathcal{F}$. Let $\widehat{\mathcal{A}} = \{t \mid {\downarrow} t \in \mathcal{A}\}$ denote the closure of $\mathcal{A}$ by weak head expansion. Dependent function space is defined as $\Pi\, \mathcal{A}\, \mathcal{F} = \{f \in \mathsf{Whnf} \mid \forall u \in \widehat{\mathcal{A}}.\, f \mathbin{@^\star} u \in \mathcal{F}(u)\}$.

By recursion on $i \in \mathbb{N}$ we define inductively sets $\mathcal{T}_i \subseteq \mathsf{Whnf} \times \mathcal{P}(\mathsf{Whnf})$ as follows [3, Sec. 5.1]:

$$\frac{}{(N, \mathsf{Wne}) \in \mathcal{T}_i} \qquad \frac{}{(\mathsf{Set}_j, |\mathcal{T}_j|) \in \mathcal{T}_i}\ (\mathsf{Set}_j, \mathsf{Set}_i) \in \mathsf{Axiom}$$

$$\frac{(U, \mathcal{A}) \in \widehat{\mathcal{T}_i} \qquad \forall u \in \widehat{\mathcal{A}}.\, (T[u/x], \mathcal{F}(u)) \in \widehat{\mathcal{T}_i}}{((x{\star}U) \to T, \Pi\, \mathcal{A}\, \mathcal{F}) \in \mathcal{T}_i}$$

Herein, $\widehat{\mathcal{T}_i} = \{(T, \mathcal{A}) \mid (\downarrow T, \mathcal{A}) \in \mathcal{T}_i\}$ and $|\mathcal{T}_j| = \{A \mid (A, \mathcal{A}) \in \mathcal{T}_j$ for some $\mathcal{A}\}$. The induction measure $A \in \mathsf{Set}_i$ shall now mean the minimum height of a derivation of $(A, \mathcal{A}) \in \mathcal{T}_i$ for some $\mathcal{A}$. Note that due to universe stratification, $A \in \mathsf{Set}_i$ is smaller than $\mathsf{Set}_i \in \mathsf{Set}_j$.

4.2 A Heterogeneously Typed Kripke Logical Relation

By induction on the maximum of the measures $A \in \mathsf{Set}_i$ and $A' \in \mathsf{Set}_{i'}$ we define two Kripke relations

$$\begin{array}{c} \Delta \vdash A : \mathsf{Set}_i \circledS \Delta' \vdash A' : \mathsf{Set}_{i'} \\ \Delta \vdash a : A \circledS \Delta' \vdash a' : A'. \end{array}$$

together with their respective closures $\widehat{\circledS}$ and the generalization to $\star$. The clauses are given in rule form.

$$\frac{\begin{array}{c}\Delta \vdash N \iff \Delta' \vdash N' \\ \Delta \vdash N : \mathsf{Set}_i = \Delta' \vdash N' : \mathsf{Set}_{i'}\end{array}}{\Delta \vdash N : \mathsf{Set}_i \circledS \Delta' \vdash N' : \mathsf{Set}_{i'}} \qquad \frac{\begin{array}{c}\Delta \vdash n : \_ \mathrel{\widehat{\longleftrightarrow}} \Delta' \vdash n' : \_ \\ \Delta \vdash n : N = \Delta' \vdash n' : N'\end{array}}{\Delta \vdash n : N \circledS \Delta' \vdash n' : N'}$$

$$\frac{\Delta \vdash \mathsf{Set}_i : \mathsf{Set}_{i+1} = \Delta' \vdash \mathsf{Set}_i : \mathsf{Set}_{i+1}}{\Delta \vdash \mathsf{Set}_i : \mathsf{Set}_{i+1} \circledS \Delta' \vdash \mathsf{Set}_i : \mathsf{Set}_{i+1}}$$

$$\frac{\begin{array}{l}\Delta \vdash U : \mathsf{Set}_i \mathrel{\widehat{\circledS}} \Delta' \vdash U' : \mathsf{Set}_{i'} \\ \forall (\Gamma, \Gamma') \leq (\Delta, \Delta'),\ \Gamma \vdash u \star U \mathrel{\widehat{\circledS}} \Gamma' \vdash u' \star U' \implies \\ \qquad\qquad \Gamma \vdash T[u/x] : \mathsf{Set}_i \mathrel{\widehat{\circledS}} \Gamma' \vdash T'[u'/x] : \mathsf{Set}_{i'} \\ \ \Delta \vdash (x{\star}U) \to T : \mathsf{Set}_i = \Delta' \vdash (x{\star}U') \to T' : \mathsf{Set}_{i'}\end{array}}{\Delta \vdash (x{\star}U) \to T : \mathsf{Set}_i \circledS \Delta' \vdash (x{\star}U') \to T' : \mathsf{Set}_{i'}}$$

$$\frac{\begin{array}{l}\forall (\Gamma, \Gamma') \leq (\Delta, \Delta'),\ \Gamma \vdash u \star U \mathrel{\widehat{\circledS}} \Gamma' \vdash u' \star U' \implies \\ \qquad\qquad \Gamma \vdash f\,{}^{\star}u : T[u/x] \mathrel{\widehat{\circledS}} \Gamma' \vdash f'\,{}^{\star}u' : T'[u'/x] \\ \qquad \Delta \vdash f : (x{\star}U) \to T = \Delta' \vdash f' : (x{\star}U') \to T'\end{array}}{\Delta \vdash f : (x{\star}U) \to T \circledS \Delta' \vdash f' : (x{\star}U') \to T'}$$

$$\frac{\begin{array}{c}t \searrow a \qquad \Delta \vdash t = a : \downarrow T \qquad \Delta' \vdash t' = a' : \downarrow T' \qquad t' \searrow a' \\ \Delta \vdash a : \downarrow T \circledS \Delta' \vdash a' : \downarrow T'\end{array}}{\Delta \vdash t : T \mathrel{\widehat{\circledS}} \Delta' \vdash t' : T'}$$

$$\frac{\Delta^{\oplus} \vdash a : A \circledS \Delta^{\oplus} \vdash a : A \qquad \Delta'^{\oplus} \vdash a' : A' \circledS \Delta'^{\oplus} \vdash a' : A'}{\Delta \vdash a \div A \circledS \Delta' \vdash a' \div A'}$$

$$\frac{\Delta^{\oplus} \vdash t : T \mathrel{\widehat{\circledS}} \Delta^{\oplus} \vdash t : T \qquad \Delta'^{\oplus} \vdash t' : T' \mathrel{\widehat{\circledS}} \Delta'^{\oplus} \vdash t' : T'}{\Delta \vdash t \div T \mathrel{\widehat{\circledS}} \Delta' \vdash t' \div T'}$$

It is immediate that the logical relation contains only well-typed terms, is symmetric, transitive, and closed under weakening.

Lemma 3 (Type and context conversion). *If* $\Delta \vdash t : T \mathrel{\widehat{\textcircled{S}}} \Delta' \vdash t' : T'$ *and* $\Delta' \vdash T' : s' \mathrel{\widehat{\textcircled{S}}} \Delta'' \vdash T'' : s''$ *then* $\Delta \vdash t : T \mathrel{\widehat{\textcircled{S}}} \Delta'' \vdash t' : T''$.

Lemma 4 (Escape from the logical relation). *Let* $\Delta \vdash T : \mathsf{Set}_i \mathrel{\widehat{\textcircled{S}}} \Delta' \vdash T' : \mathsf{Set}_{i'}$

1. $\Delta \vdash T \mathrel{\widehat{\Longleftrightarrow}} \Delta \vdash T'$.
2. *If* $\Delta \vdash t : T \mathrel{\widehat{\textcircled{S}}} \Delta' \vdash t' : T'$ *then* $\Delta \vdash t : T \mathrel{\widehat{\Longleftrightarrow}} \Delta' \vdash t' : T'$.
3. *If* $\Delta \vdash n \star T \mathrel{\widehat{\longleftrightarrow}} \Delta' \vdash n' \star T'$ *and* $\Delta \vdash n \star T = \Delta \vdash n' \star T'$ *then* $\Delta \vdash n \star T \mathrel{\widehat{\textcircled{S}}} \Delta \vdash n' \star T'$.

4.3 Validity in the Model

Simultaneously and by induction on the length of Γ we define the PERs $\Vdash \Gamma = \Vdash \Gamma'$ and $\Delta \vdash \sigma : \Gamma \mathrel{\widehat{\textcircled{S}}} \Delta' \vdash \sigma' : \Gamma'$ which presupposes the former. In rule notation this reads:

$$\frac{}{\Vdash \diamond = \Vdash \diamond} \qquad \frac{\Vdash \Gamma = \Vdash \Gamma' \qquad \Gamma \Vdash U = \Gamma' \Vdash U'}{\Vdash \Gamma.\, x{\star}U = \Vdash \Gamma'.\, x{\star}U'}$$

$$\frac{}{\Delta \vdash \sigma : \diamond \mathrel{\widehat{\textcircled{S}}} \Delta' \vdash \sigma' : \diamond}$$

$$\frac{\Delta \vdash \sigma : \Gamma \mathrel{\widehat{\textcircled{S}}} \Delta' \vdash \sigma' : \Gamma' \qquad \Delta \vdash \sigma(x) \star U\sigma \mathrel{\widehat{\textcircled{S}}} \Delta' \vdash \sigma'(x) \star U'\sigma'}{\Delta \vdash \sigma : \Gamma.\, x{\star}U \mathrel{\widehat{\textcircled{S}}} \Delta' \vdash \sigma' : \Gamma'.\, x{\star}U'}$$

Again at the same time, we define the following abbreviations, also given in rule notation:

$$\frac{}{\Gamma \Vdash s = \Gamma' \Vdash s} \qquad \frac{\Gamma \Vdash T : s = \Gamma' \Vdash T' : s'}{\Gamma \Vdash T = \Gamma' \Vdash T'}$$

$$\frac{\begin{array}{c}\Vdash \Gamma = \Vdash \Gamma' \qquad \Gamma \Vdash T = \Gamma' \Vdash T' \\ \forall \Delta \vdash \sigma : \Gamma \mathrel{\widehat{\textcircled{S}}} \Delta' \vdash \sigma' : \Gamma' \implies \Delta \vdash t\sigma : T\sigma \mathrel{\widehat{\textcircled{S}}} \Delta' \vdash t'\sigma' : T'\sigma'\end{array}}{\Gamma \Vdash t : T = \Gamma' \Vdash t' : T'}$$

Finally, let $\Gamma \Vdash t : T \iff \Gamma \Vdash t : T = \Gamma \Vdash t : T$ and $\Vdash \Gamma \iff \Vdash \Gamma = \Vdash \Gamma$.

Lemma 5 (Context satisfiable). *For the identity substitution* id *and* $\Vdash \Gamma = \Vdash \Gamma'$ *we have* $\Gamma \vdash \mathsf{id} : \Gamma \mathrel{\widehat{\textcircled{S}}} \Gamma' \vdash \mathsf{id} : \Gamma'$.

Theorem 1 (Completeness of IITT rules). *If* $\Gamma \Vdash t : T = \Gamma' \Vdash t' : T'$ *then* $\Gamma \vdash t : T = \Gamma' \vdash t' : T'$ *and* $\Gamma \vdash T = \Gamma' \vdash T'$.

Theorem 2 (Fundamental theorem of logical relations)

1. *If* $\vdash \Gamma$ *then* $\Vdash \Gamma$.
2. *If* $\vdash \Gamma = \vdash \Gamma'$ *then* $\Vdash \Gamma = \Vdash \Gamma'$.
3. *If* $\Gamma \vdash t : T$ *then* $\Gamma \Vdash t : T$.
4. *If* $\Gamma \vdash t : T = \Gamma' \vdash t' : T'$ *then* $\Gamma \Vdash t : T = \Gamma' \Vdash t' : T'$.

5 Meta-theoretic Consequences of the Model Construction

After doing hard work in the construction of a universal model, the rest of the meta-theory of IITT falls into our lap like a ripe fruit.

Normalization and subject reduction. An immediate consequence of the model construction is that each term has a weak head normal form and that typing and equality is preserved by weak head normalization.

Theorem 3 (Normalization and subject reduction). *If $\Gamma \vdash t : T$ then $t \searrow a$ and $\Gamma \vdash t = a : T$.*

Correctness of algorithmic equality. Algorithmic equality is correct, i. e., sound, complete, and terminating. Together, this entails decidability of equality in IITT. Algorithmic equality is built into the model at every step, thus, completeness is immediate:

Theorem 4 (Completeness of algorithmic equality). *If $\Gamma \vdash t : T = \Gamma' \vdash t' : T'$ then $\Gamma \vdash t : T \mathrel{\widehat{\Longleftrightarrow}} \Gamma' \vdash t' : T'$.*

Termination of algorithmic equality is a consequence of full normalization, which we have not defined explicitly, but which is implicit in the model.

Theorem 5 (Termination of algorithmic equality). *If $\Delta \vdash t : T$ and $\Delta' \vdash t' : T'$ then the query $\Delta \vdash t : T \mathrel{\widehat{\Longleftrightarrow}} \Delta' \vdash t' : T'$ terminates.*

Soundness of the equality algorithm is a consequence of subject reduction.

Theorem 6 (Soundness of algorithmic equality). *Let $\Delta \vdash t : T$ and $\Delta' \vdash t' : T'$ and $\Delta \vdash T = \Delta' \vdash T'$. If $\Delta \vdash t : T \mathrel{\widehat{\Longleftrightarrow}} \Delta' \vdash t' : T'$ then $\Delta \vdash t : T = \Delta' \vdash t' : T'$.*

Homogeneity. Although we defined IITT-equality heterogeneously, we can now show that the heterogeneity was superficial, i. e., in fact do equal terms have equal types. This was already implicit in the formulation of the equality algorithm which only compares terms at types of the same shape. By rather than building homogeneity into the definition of equality, we obtain it as a global result.

Theorem 7 (Homogeneity). *If $\Gamma \vdash t : T = \Gamma' \vdash t' : T'$ then $\vdash \Gamma = \vdash \Gamma'$ and $\Gamma \vdash T = \Gamma' \vdash T'$.*

Consistency. Importantly, not every type is inhabited in IITT, thus, it can be used as a logic. A prerequisite is that types can be distinguished, which follows immediately from completeness of algorithmic equality.

Theorem 8 (Consistency). $X : \mathsf{Set}_0 \not\vdash t : X$.

Decidability. To round off, we show that typing in IITT is decidable. Type checking algorithms such as bidirectional checking [10] rely on injectivity of function type constructors, which is built into the definition of $\textcircled{S}$:

Theorem 9 (Function type injectivity). *If $\Gamma \vdash (x{\star}U) \to T : s = \Gamma' \vdash (x{\star}U') \to T' : s'$ then $\Gamma \vdash U : s = \Gamma' \vdash U' : s'$ and $\Gamma.\, x{\star}U \vdash T : s = \Gamma'.\, x{\star}U' \vdash T' : s'$.*

Theorem 10 (Decidability of IITT). *Equality $\Gamma \vdash t : T = \Gamma' \vdash t' : T'$ and typing $\Gamma \vdash t : T$ are decidable.*

6 Extensions

Data types and recursion. The semantics of IITT is ready to cope with inductive data types like the natural numbers and the associated recursion principles. Recursion into types, aka known as large elimination, is also accounted for since we have universes and a semantics which does not erase dependencies (unlike Pfenning's model [22]).

Types with extensionality principles. The purpose of having a typed equality algorithm is to handle η-laws that are not connected to the shape of the expression (like η-contraction for functions) but to the shape of the type only. Typically these are types T with at most one inhabitant, i. e., the empty type, the unit type, singleton types or propositions[3]. For such T we have the η-law

$$\frac{\Gamma \vdash t, t' : T}{\Gamma \vdash t = t' : T}$$

which can only be checked in the presence of type T. Realizing such η-laws gives additional "proof" irrelevance which is not covered by Pfenning's irrelevant quantification $(x{\div}U) \to T$.

Internal erasure. Terms $u \div U$ in irrelevant position are only there to please the type checker, they are ignored during equality checking. This can be inferred from the substitution principle: If $\Gamma.\, x{\div}U \vdash T$ and $\Gamma \vdash u, u' \div U$, then $\Gamma \vdash T[u/x] = T[u'/x]$; the type T has the same shape regardless of u, u'. Hence, terms like u serve the sole purpose to prove some proposition and could be replaced by a dummy $\bullet$ immediately after type-checking. This is an optimization which in the first place saves memory, but if expressions are written to interface files and reloaded later, it also saves disk space and execution time of saving and loaded. First experiments with an implementation of internal erasure in Agda [9] shows that savings are huge, like in formalizing category theory and algebra which uses structures with embedded proofs (see Example 1).

Internal erasure can be realized by making $\Gamma \vdash t \div T$ a judgement (as opposed to just a notation for $\Gamma^{\oplus} \vdash t : T$) and adding the rule

$$\frac{\Gamma \vdash t \div T}{\Gamma \vdash \bullet \div T}.$$

The rule states that if there is already a proof t of T, then $\bullet$ is a new proof of T. This preserves provability while erasing the proof terms. Conservativity of this rule can be proven as in joint work of the author with Coquand and Pagano [4].

Proof modality. Pfenning [22] suggests a modality $\triangle$ formed by the rule

$$\frac{\Gamma \vdash t \div T}{\Gamma \vdash t : \triangle T}.$$

which for instance allows the definition of the subset type $\{x : U \mid T\,x\}$ as $\Sigma x : U.\, \triangle(T\,x)$. Such a modality has been present in Nuprl as *Squash* type [20] and it is also known as the type of proofs of (proposition) T [4,8]. Using the extensions of Example 1, we can encode it as $\triangle T = (\_{\div}T) \times 1$.

[3] Some care is necessary for the type of Leibniz equality [1,25].

7 Conclusions

We have extended Pfenning's notion of irrelevance to a type theory IITT with universes that accommodates types defined by recursion. A heterogeneous algorithmic equality judgement has allowed a smooth construction of a universal Kripke model that is both sound and complete for IITT, yielding normalization, consistency and decidability of IITT. Inspired by a heterogeneously typed semantics, we have formulated the declarative system with a heterogeneous equality judgement as well and demonstrated that this also yields a sound specification of type theory.

Integrating irrelevance and data types in dependent type theory does not seem without challenges. We have succeeded to treat Pfenning's notion of irrelevance, but our proof does not scale directly to *parametric* function types, a stronger notion of irrelevant function types called implicit quantification by Miquel [18][4]. Two more type theories build on Miquel's calculus [17], Barras and Bernardo's ICC* [8] and Mishra-Linger and Sheard's *Erasure Pure Type Systems* (EPTS) [19], but none has offered a satisfying account of large eliminations yet. Miquel's model [16] features data types only as impredicative encodings. For irrelevant, parametric, and recursive functions to coexist it seems like three different function types are necessary, e. g., in the style of Pfenning's *irrelevance, extensionality and intensionality*. We would like to solve this puzzle in future work, not least to implement high-performance languages with dependent types.

Acknowledgments. Thanks to Bruno Barras, Bruno Bernardo, Thierry Coquand, Dan Doel, Hugo Herbelin, Conor McBride, and Ulf Norell for discussions on irrelevance in type theory. Thanks to Jason Reed, Conor McBride, and the anonymous referees for suggesting improvements on the text. Work on a previous paper has been carried out while being invited researcher at PPS, Paris, in the INRIA πr^2 team headed by Pierre-Louis Curien and Hugo Herbelin. I have benefited from presentations at JFLA 2010, the Agda Implementer's Meeting in March 2010 and the DTP workshop in 2010.

References

1. Abel, A.: Extensional normalization in the logical framework with proof irrelevant equality. In: Danvy, O. (ed.) 2009 Wksh. on Normalization by Evaluation (2009)
2. Abel, A.: MiniAgda: Integrating sized and dependent types. In: Bove, A., Komendantskaya, E., Niqui, M. (eds.) Wksh. on Partiality And Recursion in Interactive Theorem Provers (PAR 2010) (2010)
3. Abel, A., Coquand, T., Dybjer, P.: Verifying a semantic $\beta\eta$-conversion test for martin-löf type theory. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 29–56. Springer, Heidelberg (2008)
4. Abel, A., Coquand, T., Pagano, M.: A modular type-checking algorithm for type theory with singleton types and proof irrelevance. In: Curien, P.-L. (ed.) TLCA 2009. LNCS, vol. 5608, pp. 5–19. Springer, Heidelberg (2009)
5. Amadio, R.M. (ed.): FOSSACS 2008. LNCS, vol. 4962. Springer, Heidelberg (2008)

[4] A function argument is parametric if it is irrelevant for computing the function result while the type of the result may depend on it. In Pfenning's notion, the argument must also be irrelevant in the type.

6. Augustsson, L.: Cayenne - a language with dependent types. In: Proc. of the 3rd ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 1998). SIGPLAN Notices, vol. 34, pp. 239–250. ACM Press, New York (1999)
7. Awodey, S., Bauer, A.: Propositions as [Types]. J. Log. Comput. 14(4), 447–471 (2004)
8. Barras, B., Bernardo, B.: The implicit calculus of constructions as a programming language with dependent types. In: Amadio [5], pp. 365–379
9. Bove, A., Dybjer, P., Norell, U.: A Brief Overview of Agda – A Functional Language with Dependent Types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer, Heidelberg (2009)
10. Coquand, T.: An algorithm for type-checking dependent types. In: Proc. of the 3rd Int. Conf. on Mathematics of Program Construction, MPC 1995. Sci. Comput. Program., vol. 26, pp. 167–177. Elsevier, Amsterdam (1996)
11. Goguen, H.: A Typed Operational Semantics for Type Theory. PhD thesis, University of Edinburgh, Available as LFCS Report ECS-LFCS-94-304 (August 1994)
12. Harper, R., Pfenning, F.: On equivalence and canonical forms in the LF type theory. ACM Transactions on Computational Logic 6(1), 61–101 (2005)
13. INRIA. The Coq Proof Assistant Reference Manual. INRIA, version 8.2 edition (2008), http://coq.inria.fr/
14. Letouzey, P.: A new extraction for coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003)
15. McBride, C., McKinna, J.: The view from the left. J. Func. Program (2004)
16. Miquel, A.: A model for impredicative type systems, universes, intersection types and subtyping. In: Proc. of the 15th IEEE Symp. on Logic in Computer Science (LICS 2000), pp. 18–29 (2000)
17. Miquel, A.: The Implicit Calculus of Constructions. In: Abramsky, S. (ed.) TLCA 2001. LNCS, vol. 2044, pp. 344–359. Springer, Heidelberg (2001)
18. Miquel, A.: Le Calcul des Constructions implicite: syntaxe et sémantique. PhD thesis, Université Paris 7 (December 2001)
19. Mishra-Linger, N., Sheard, T.: Erasure and polymorphism in pure type systems. In: Amadio [5], pp. 350–364
20. Mishra-Linger, R.N.: Irrelevance, Polymorphism, and Erasure in Type Theory. PhD thesis, Portland State University (2008)
21. Paulin-Mohring, C., Werner, B.: Synthesis of ML programs in the system Coq. J. Symb. Comput. 15(5/6), 607–640 (1993)
22. Pfenning, F.: Intensionality, extensionality, and proof irrelevance in modal type theory. In: LICS 2001: IEEE Symposium on Logic in Computer Science (June 2001)
23. Reed, J.: Proof irrelevance and strict definitions in a logical framework, Senior Thesis, published as Carnegie-Mellon University technical report CMU-CS-02-153 (2002)
24. Reed, J.: Extending Higher-Order Unification to Support Proof Irrelevance. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 238–252. Springer, Heidelberg (2003)
25. Werner, B.: On the strength of proof-irrelevant type theories. Logical Meth. in Comput. Sci. 4 (2008)

When Is a Type Refinement an Inductive Type?

Robert Atkey, Patricia Johann, and Neil Ghani[*]

University of Strathclyde
{Robert.Atkey,Patricia.Johann,Neil.Ghani}@cis.strath.ac.uk

Abstract. Dependently typed programming languages allow sophisticated properties of data to be expressed within the type system. Of particular use in dependently typed programming are indexed types that refine data by computationally useful information. For example, the ℕ-indexed type of vectors refines lists by their lengths. Other data types may be refined in similar ways, but programmers must produce purpose-specific refinements on an *ad hoc* basis, developers must anticipate which refinements to include in libraries, and implementations often store redundant information about data and their refinements. This paper shows how to generically derive inductive characterisations of refinements of inductive types, and argues that these characterisations can alleviate some of the aforementioned difficulties associated with *ad hoc* refinements. These characterisations also ensure that standard techniques for programming with and reasoning about inductive types are applicable to refinements, and that refinements can themselves be further refined.

1 Introduction

One of the key aims of current research in functional programming is to reduce the *semantic gap* between what programmers know about computational entities and what the types of those entities can express about them. One particularly promising approach is to parameterise, or *index*, types by extra information that can be used to express properties of data having those types. For example, most functional languages support a standard list data type parameterised over the type of the data the lists contain, but for some applications it is also crucial to know the length of a list. We may wish, for instance, to ensure that the list argument to the `tail` function has non-zero length — i.e., is non-empty — or that the lengths of the two list arguments to `zip` are the same.

A data type that equips each list with its length can be defined in the dependently typed language Agda [26] by

```
data Vector (B : Set) : Nat -> Set where
  VNil  : Vector B Z
  VCons : (n : Nat) -> B -> Vector B n -> Vector B (S n)
```

This declaration inductively defines a data type `Vector` which, for each choice of element type `B`, is indexed by natural numbers and has two constructors: `VNil`,

[*] This work was funded by EPSRC grant EP/G068917/1.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 72–87, 2011.

which constructs a vector of B-data of length zero (i.e., Z), and VCons, which constructs from an index n, an element of B, and a vector of B-data of length n, a new vector of B-data of length $n+1$ (i.e., S n). The inductive type Vector can be used to define functions on lists which are "length-aware" in a way that functions which process data of standard list types cannot be. For example, length-aware tail and zip functions can be given via the following types and definitions:

```
tail : (n : Nat) -> Vector B (S n) -> Vector B n
tail (VCons b bs) = bs

zip  : (n : Nat) -> Vector B n -> Vector C n -> Vector (B x C) n
zip  VNil         VNil        = VNil
zip (VCons b bs) (VCons c cs) = VCons (b , c) (zip bs cs)
```

Examples such as those above suggest that indexing types by computationally relevant information has great potential. However, for this potential to be realised, we must better understand how indexed types can be constructed. Moreover, since we want to ensure that all of the techniques developed for structured programming with and principled reasoning about inductive types — such as those championed in the Algebra of Programming [6] literature — are applicable to the resulting indexed types, we also want these types to be inductive. This paper therefore asks the following fundamental question:

> *Can elements of inductive types be systematically augmented with computationally relevant information to give indexed inductive types that store computationally relevant information in their indices? If so, how?*

That is, how can we *refine* a given inductive type to get a new such type, called a *refinement*, that associates with each element of the given type its index?

One straightforward way to refine an inductive type is to use a refinement function to compute the index for each of its elements, and then to associate these indices to their corresponding elements. To refine lists by their lengths, for example, we would start with the standard list data type and length function:

```
data List (B : Set) : Set where   length : List B -> Nat
  Nil  : List B                   length Nil        = Z
  Cons : B -> List B -> List B    length (Cons _ l) = S (length l)
```

and construct the following refinement type of indexed lists:

$$\mathtt{IdxList\,B\,n} \cong \{\mathtt{x : List\ B} \mid \mathtt{length\ x = n}\} \tag{1}$$

This construction is *global* in that both the data type and the collection of indices exist *a priori*, and the refinement is obtained by assigning, *post facto*, an appropriate index to each data type element. But the construction suffers from a serious drawback: the resulting refinement — IdxList here — need not be inductive, and so is not a solution to the fundamental question posed above.

We propose an alternative construction of refinements that provides a comprehensive answer to the fundamental question raised above in the case when the given refinement function is computed by structural recursion over the data type to be refined. This is often the case in practice. More specifically, we construct, for each inductive type μF and each F-algebra α whose fold computes the desired refinement function, a functor F^α whose least fixed point μF^α is the desired refinement. The characterisation of the refinement of μF by α as the inductive type μF^α allows the entire arsenal of structured programming techniques to be brought to bear on them. This construction is also *local* in that the indices of recursive substructures are readily available *at the time a structurally recursive program is written*, rather than needing to be computed by inversion from the index of the input data structure.

The functor F^α that we construct is intimately connected with the generic structural induction rule for the inductive type μF [16,18]. This is perhaps not surprising: structural induction proves properties of functions defined by structural recursion on elements of inductive types. If the values of those functions are abstracted into the indices of associated indexed inductive types, then the computation of those values need no longer be performed during inductive proofs. In essence, we have shifted work away from computation and onto data. Refinement thus supports reasoning by structural induction "up to" the index of a term.

We adopt a semantic approach based on category theory because it allows a high degree of abstraction and economy, and exposes structure that might be lost were a specific programming notation to be used. Although we have developed our theory in the abstract setting of fibrations [20], we specialise to the families fibration over the category of sets to improve accessibility and give concrete intuitions. A type-theoretic answer to the fundamental question posed above has already been given by McBride [23] using his notion of ornamenting a data type (see section 7).

The remainder of this paper is structured as follows. In section 2 we recall basic categorical preliminaries. In section 3 we introduce a framework within which refinement may be developed [16,18]. We describe and illustrate our basic refinement technique in section 4. In section 5 we show how to refine inductive types which are themselves indexed. In section 6 we further extend our basic refinement technique to allow partial refinement, in which indexed types are constructed from inductive types not all of whose elements have indices. Finally, section 7 discusses applications to dependently typed programming, library development, and implementation, as well as future and related work.

2 Inductive Types and F-Algebras

A data type is *inductive (in a category* $\mathbb{C}$*)* if it is the least fixed point μF of an endofunctor on $\mathbb{C}$. For example, if Set denotes the category of sets and functions, $\mathbb{Z}$ is the set of integers, and $+$ and $\times$ denote coproduct and product, respectively, then the following data type of binary trees with integer leaves is $\mu F_{\mathtt{Tree}}$ for the endofunctor $F_{\mathtt{Tree}} X = \mathbb{Z} + X \times X$ on Set:

```
data Tree : Set where
    Leaf : Integer -> Tree
    Node : (Tree x Tree) -> Tree
```

Inductive types can also be understood in terms of the categorical notion of an F-algebra. If $\mathbb{C}$ is a category and $F : \mathbb{C} \to \mathbb{C}$ is a functor, then an *F-algebra* is a pair $(A, \alpha : FA \to A)$ comprising an object A of $\mathbb{C}$ and a morphism $\alpha : FA \to A$ in $\mathbb{C}$. The object A is called the *carrier* of the F-algebra, and the morphism α is called its *structure map.* We usually refer to an F-algebra solely by its structure map, since the carrier is present in the type of this map.

An *F-algebra homomorphism* from $(\alpha : FA \to A)$ to $(\alpha' : FB \to B)$ is a morphism $f : A \to B$ of $\mathbb{C}$ such that $f \circ \alpha = \alpha' \circ Ff$. An F-algebra $(\alpha : FA \to A)$ is *initial* if, for any F-algebra $(\alpha' : FB \to B)$, there exists a unique F-algebra morphism from α to α'. The initial F-algebra is unique up to isomorphism, and Lambek's Lemma further ensures that it is itself an isomorphism. Its carrier is thus the least fixed point μF of F. We write $(\mathit{in}_F : F(\mu F) \to \mu F)$ for the initial F-algebra, and $\llparenthesis \alpha \rrparenthesis_F : \mu F \to A$ for the unique morphism from $(\mathit{in}_F : F(\mu F) \to \mu F)$ to any F-algebra $(\alpha : FA \to A)$. We write $\llparenthesis - \rrparenthesis$ for $\llparenthesis - \rrparenthesis_F$ when F is clear from context. Of course, not all functors have least fixed points. For instance, the functor $FX = (X \to 2) \to 2$ on Set does not have any fixed point at all.

In light of the above, the data type `Tree` can be interpreted as the carrier of the initial $F_{\texttt{Tree}}$-algebra. In functional programming terms, if $(\alpha : \mathbb{Z} + A \times A \to A)$ is an $F_{\texttt{Tree}}$-algebra, then $\llparenthesis \alpha \rrparenthesis : \texttt{Tree} \to A$ is exactly the application of the standard iteration function `fold` for trees to α (actually, to an "unbundling" of α into replacement functions, one for each of $F_{\texttt{Tree}}$'s constructors). More generally, for each functor F, the map $\llparenthesis - \rrparenthesis_F : (FA \to A) \to \mu F \to A$ is the iteration function for μF.

If F is a functor on $\mathbb{C}$, we write Alg_F for the category of all F-algebras and F-algebra homomorphisms between them. Identities and composition in Alg_F are taken directly from $\mathbb{C}$. The existence of initial F-algebras is equivalent to the existence of initial objects in Alg_F. Recall that an *adjunction* between two categories $\mathbb{C}$ and $\mathbb{D}$ consists of a left adjoint functor L and a right adjoint functor R and an isomorphism natural in A and X between the set $\mathbb{C}(LA, X)$ of morphisms in $\mathbb{C}$ from LA to X and the set $\mathbb{D}(A, RX)$ of morphisms in $\mathbb{D}$ from A to RX. We say that the functor L is *left adjoint* to R, and that the functor R is *right adjoint* to L, and write $L \dashv R$.

We will make much use of the following theorem from [18]:

Theorem 1. *If $F : \mathbb{C} \to \mathbb{C}$ and $G : \mathbb{D} \to \mathbb{D}$ are functors, $L \dashv R$, and $FL \cong LG$ is a natural isomorphism, then*

$$\begin{tikzcd} \mathbb{C} \arrow[r, bend right, "R"'] \arrow[r, phantom, "\perp"] & \mathbb{D} \arrow[l, bend right, "L"'] \end{tikzcd}$$

lifts to

$$\begin{tikzcd} \mathrm{Alg}_F \arrow[r, bend right, "R'"'] \arrow[r, phantom, "\perp"] & \mathrm{Alg}_G \arrow[l, bend right, "L'"'] \end{tikzcd}$$

.

Theorem 1 will be useful in conjunction with the fact that left adjoints preserve colimits, and thus preserve initial objects. In the setting of the theorem, if G has an initial algebra, then so does F. To compute the initial F-algebra in concrete situations we need to know that $L'(k : GA \to A) = Lk \circ p_A$ where p is (one half of) the natural isomorphism between FL and LG. Then the initial F-algebra is given by applying L' to the initial G-algebra, and so $\mu F = L(\mu G)$.

3 A Framework for Refinement

An object of Fam(Set) is a pair (A, P) comprising a set A and a function $P : A \to$ Set; such a pair is called a *family* of sets. A morphism $(f, f^\sim) : (A, P) \to (B, Q)$ of Fam(Set) is a pair of functions $f : A \to B$ and $f^\sim : \forall a.\ Pa \to Q(fa)$. From a programming perspective, a family (A, P) is an A-indexed type P, with Pa representing the collection of data with index a. An alternative, logical view is that (A, P) is a predicate representing a property P of data of type A, and that Pa represents the collection of proofs that P holds for a. When Pa is inhabited, P is said to hold for a. When Pa is empty, P is said not to hold for a.

The *families fibration* U : Fam(Set) $\to$ Set is the functor mapping each family (A, P) to A and each morphism $(f, f^\sim)$ to f. For each set A, the category Fam(Set)_A consists of families (A, P) and morphisms $(f, f^\sim)$ between them such that $f = id_A$. We call Fam(Set)_A the *fibre* of the families fibration over A. A function $f : A \to B$ contravariantly generates a *re-indexing functor* $f^* : \text{Fam(Set)}_B \to \text{Fam(Set)}_A$ which maps (B, Q) to $(A, Q \circ f)$.

3.1 Truth and Comprehension

Each fibre Fam(Set)_A has a terminal object $(A, \lambda a : A.\ 1)$, where 1 is the canonical singleton set. This object is called the *truth predicate* for A. The mapping of objects to their truth predicates extends to a functor K_1 : Set $\to$ Fam(Set), called the *truth functor*. In addition, for each family (A, P) we can define the *comprehension* of (A, P), denoted $\{(A, P)\}$, to be the set $\{(a, p) \mid a \in A, p \in Pa\}$. The mapping of families to their comprehensions extends to a functor $\{-\}$: Fam(Set) $\to$ Set, called the *comprehension functor*, and we end up with the following pleasing collection of adjoint relationships:

$$\begin{tikzcd}
\text{Fam(Set)} \arrow[d, "U"'] \arrow[d, phantom, shift left=2, "\dashv"] \arrow[d, bend left=70, "\{-\}"] \arrow[d, phantom, shift left=9, "\dashv"] \\
\text{Set} \arrow[u, bend right=20, "K_1"']
\end{tikzcd} \qquad (2)$$

The families fibration U is thus a *comprehension category with unit* [19,20]. Like every comprehension category with unit, U supports a natural transformation $\pi : \{-\} \to U$ such that $\pi_{(A,P)}(a, p) = a$ for all (a, p) in $\{(A, P)\}$. In fact, U is *full*, i.e., the functor from Fam(Set) to $\text{Set}^{\to}$ induced by π is full and faithful.

3.2 Indexed Coproducts and Indexed Products

For each function $f : A \to B$ and family (A, P), we can form the family $(B, \lambda b.\ \Sigma_{a \in A}.\ (b = fa) \times Pa)$, called the *indexed coproduct of* (A, P) *along* f. The mapping of each family to its indexed coproduct along f extends to a functor $\Sigma_f : \text{Fam(Set)}_A \to \text{Fam(Set)}_B$ which is left adjoint to the re-indexing functor f^*. In the abstract setting of fibrations, a fibration with the property that each re-indexing functor f^* has a left adjoint Σ_f is called a *bifibration*, and the functors Σ_f are called *op-re-indexing* functors. These functors are often subject to the Beck-Chevalley condition for coproducts, which is well-known to hold for the families fibration. This condition ensures that in certain circumstances op-re-indexing commutes with re-indexing [20]. A bifibration which is also a full comprehension category with unit is called a *full cartesian Lawvere category* [19].

For each function $f : A \to B$ and family (A, P) we can also form the family $(B, \lambda b.\ \Pi_{a \in A}.(b = fa) \to Pa)$, called the *indexed product of* (A, P) *along* f. The mapping of each family to its indexed product along f extends to a functor $\Pi_f : \mathrm{Fam(Set)}_A \to \mathrm{Fam(Set)}_B$ which is right adjoint to f^*. This gives the following collection of relationships for each function $f : A \to B$:

$$\begin{tikzcd}
\mathrm{Fam(Set)}_B \arrow[r, "f^*" description] & \mathrm{Fam(Set)}_A \arrow[l, bend right, "\Sigma_f"'] \arrow[l, bend left, "\Pi_f"] \arrow[l, phantom, shift right=3, "\perp"] \arrow[l, phantom, shift left=3, "\perp"]
\end{tikzcd}$$

Like its counterpart for coproducts, the Beck-Chevalley condition for products is often required. However, we do not make use of this condition in this paper.

At several places below we make essential use of the fact that the families fibration has strong coproducts, i.e., that in the diagram

$$\begin{tikzcd}
\{(A,P)\} \arrow[r, "\{\psi\}"] \arrow[d, "\pi_{(A,P)}"'] & \{(B, \Sigma_f(A,P))\} \arrow[d, "\pi_{(B,\Sigma_f(A,P))}"] \\
A \arrow[r, "f"] & B
\end{tikzcd} \tag{3}$$

where ψ is the obvious map of families of sets over f, $\{\psi\}$ is an isomorphism. This definition of strong coproducts naturally generalises the usual one [20], and imposes a condition which is standard in models of type theory.

3.3 Liftings

A *lifting* of a functor $F : \mathrm{Set} \to \mathrm{Set}$ is a functor $\hat{F} : \mathrm{Fam(Set)} \to \mathrm{Fam(Set)}$ such that $FU = U\hat{F}$. A lifting is *truth-preserving* if it satisfies $K_1 F \cong \hat{F} K_1$. Truth-preserving liftings for all polynomial functors — i.e., for all functors built from identity functors, constant functors, coproducts, and products — are given in [18]. Truth-preserving liftings were established for arbitrary functors in [16]. The truth-preserving lifting $\hat{F}$ is defined on objects by

$$\hat{F}(A,P) = (FA, \lambda a.\ \{x : F\{(A,P)\} \mid F\pi_{(A,P)} x = a\}) = \Sigma_{F\pi_{(A,P)}} K_1(F\{(A,P)\}) \tag{4}$$

The final expression is written point-free using the constructions of Sections 3.1 and 3.2.

Since $\hat{F}$ is an endofunctor on Fam(Set), the category $\mathrm{Alg}_{\hat{F}}$ of $\hat{F}$-algebras exists. The families fibration $U : \mathrm{Fam(Set)} \to \mathrm{Set}$ extends to a fibration $U^{\mathsf{Alg}} : \mathrm{Alg}_{\hat{F}} \to \mathrm{Alg}_F$, called the *algebras fibration* induced by U. Moreover, writing K_1^{Alg}

and $\{-\}^{\mathsf{Alg}}$ for the truth and comprehension functors, respectively, for U^{Alg}, the adjoint relationships from Diagram 2 all lift to give $U^{\mathsf{Alg}} \dashv K_1^{\mathsf{Alg}} \dashv \{-\}^{\mathsf{Alg}}$. The two adjunctions here follow from Theorem 1 using the fact that $\hat{F}$ is a lifting and preserves truth. That left adjoints preserve initial objects can now be used to establish the following fundamental result from [16,18]:

Theorem 2. *$K_1(\mu F)$ is the carrier $\mu\hat{F}$ of the initial $\hat{F}$-algebra.*

4 From Liftings to Refinements

In this section we show that the refinement of an inductive type μF by an F-algebra $(\alpha : FA \to A)$, i.e., the family

$$(A, \lambda a : A.\ \{x : \mu F \mid (\!|\alpha|\!) x = a\}) \tag{5}$$

is inductively characterised as μF^{α} where $F^{\alpha} : \mathrm{Fam}(\mathrm{Set})_A \to \mathrm{Fam}(\mathrm{Set})_A$ is

$$F^{\alpha} = \Sigma_{\alpha}\hat{F} \tag{6}$$

An alternative, set-theoretic presentation of F^{α} is:

$$F^{\alpha}(A, P) = (A, \lambda a.\ \{x : F\{(A, P)\} \mid \alpha(F\pi_{(A,P)}x) = a\}) \tag{7}$$

That is, $F^{\alpha}(A, P)$ is obtained by first building the FA-indexed type $\hat{F}(A, P)$ (cf. Equation 4), and then restricting membership to those elements whose α-values are correctly computed from those of their immediate subterms. The proof consists of the following three theorems, which are, as far as we are aware, new.

Theorem 3. *For each F-algebra $(\alpha : FA \to A)$, $(\mathrm{Alg}_{\hat{F}})_{\alpha} \cong \mathrm{Alg}_{F^{\alpha}}$.*

Proof. First note that $(\mathrm{Alg}_{\hat{F}})_{\alpha}$ is isomorphic to the category $(\hat{F} \downarrow \alpha^*)$ whose objects are morphisms from $\hat{F}(A, P)$ to $\alpha^*(A, P)$ in $\mathrm{Fam}(\mathrm{Set})_{FA}$ and whose morphisms are commuting squares. Then $(\hat{F} \downarrow \alpha^*) \cong \mathrm{Alg}_{F^{\alpha}}$ because $\Sigma_{\alpha} \dashv \alpha^*$.

Theorem 3 can be used to prove the following key result:

Theorem 4. *$U^{\mathsf{Alg}} : \mathrm{Alg}_{\hat{F}} \to \mathrm{Alg}_F$ is a bifibration.*

Proof. That U^{Alg} is a fibration, indeed a comprehension category with unit, is proved in [18]. Next, let f be an F-algebra morphism from $\alpha : FA \to A$ to $\beta : FB \to B$. We must show that the reindexing functor $f^{*\mathsf{Alg}}$ in U^{Alg} has a left adjoint Σ_f^{Alg}. Such an adjoint can be defined using Theorem 1, Theorem 3, and $F^{\beta}\Sigma_f \cong \Sigma_f F^{\alpha}$. By Equation 6, the latter is equivalent to $\Sigma_{\beta}\hat{F}\Sigma_f \cong \Sigma_f\Sigma_{\alpha}\hat{F}$. From the definition of $\hat{F}$, we must show that for all (A, P) in $\mathrm{Fam}(\mathrm{Set})_A$,

$$\Sigma_{\beta}\Sigma_{F\pi_{\Sigma_f(A,P)}}K_1F\{\Sigma_f(A, P)\} \;\cong\; \Sigma_f\Sigma_{\alpha}\Sigma_{F\pi_{(A,P)}}K_1F\{(A, P)\} \tag{8}$$

To see that this is the case, consider the following diagram:

$$\begin{tikzcd}
F\{(A,P)\} \arrow[r, "F\pi_{(A,P)}"] \arrow[d, "F\{\psi\}"'] & FA \arrow[r, "\alpha"] \arrow[d, "Ff"] & A \arrow[d, "f"] \\
F\{\Sigma_f(A,P)\} \arrow[r, "F\pi_{\Sigma_f(A,P)}"'] & FB \arrow[r, "\beta"'] & B
\end{tikzcd}$$

The left-hand square commutes because it is obtained by applying F to the naturality square for π, and the right-hand square commutes because f is an F-algebra morphism. Then $\Sigma_\beta \Sigma_{F\pi_{\Sigma_f(A,P)}} \Sigma_{F\{\psi\}} \cong \Sigma_f \Sigma_\alpha \Sigma_{F\pi_{(A,P)}}$ because opre-indexing preserves composition. Equation 8 now follows by applying both of these functors to $K_1 F\{(A,P)\}$, and then observing that $F\{\psi\}$ is an isomorphism since $\{\psi\}$ is one by assumption (cf. Diagram 3), so that $\Sigma_{F\{\psi\}}$ is a right adjoint (as well as a left adjoint) and thus preserves terminal objects.

We can now give an explicit characterisation for μF^α. We have

Theorem 5. *The functor F^α has an initial algebra with carrier $\Sigma_{\llparenthesis \alpha \rrparenthesis} K_1(\mu F)$.*

Proof. The category $\mathrm{Alg}_{\hat{F}}$ has initial object whose carrier is $K_1(\mu F)$ by Theorem 2. Since U^{Alg} is a left adjoint and hence preserves initial objects, Proposition 9.2.2 of [20] ensures that the fibre $(\mathrm{Alg}_{\hat{F}})_\alpha$ — and so, by Theorem 3, that Alg_{F^α} — has an initial object whose carrier is $\Sigma_{\llparenthesis \alpha \rrparenthesis} K_1(\mu F)$.

Instantiating Theorem 5 for Fam(Set) gives exactly the inductive characterisation of refinements we set out to find, namely that in Equation 5.

4.1 Some Specific Refinements

The following explicit formulas are used to compute refinements in the examples below. In the expression B^α, B is the constantly B-valued functor.

$$\begin{aligned}
Id^\alpha(A,P) &= (A, \lambda a.\{x : \{(A,P)\} \mid \alpha\ (\pi_{(A,P)} x) = a\}) \\
B^\alpha(A,P) &= (A, \lambda a.\{x : B \mid \alpha\ x = a\}) \\
(G+H)^\alpha(A,P) &= (A, \lambda a\{x : G\ \{(A,P)\} \mid \alpha(\mathsf{inl}(G\pi_{(A,P)} x)) = a\} \\
&\qquad\qquad + \{x : H\ \{(A,P)\} \mid \alpha(\mathsf{inr}(H\pi_{(A,P)} x)) = a\}) \\
&= (A, \lambda a.\ G^{\alpha \circ \mathsf{inl}} Pa + H^{\alpha \circ \mathsf{inr}} Pa) \\
(G \times H)^\alpha(A,P) &= (A, \lambda a.\ \{x_1 : G\ \{(A,P)\},\ x_2 : H\ \{(A,P)\} \mid \\
&\qquad\qquad \alpha(G\pi_{(A,P)} x_1, H\pi_{(A,P)} x_2) = a\}
\end{aligned}$$

Refinements of the identity and constant functors are as expected. Refinement splits coproducts of functors into two cases, specialising the refining algebra for each summand. It is not possible to decompose the refinement of a product of functors $G \times H$ into refinements of G and H (possibly by algebras other than α). This is because α may need to relate multiple elements to the overall index.

Example 1. *The inductive type of lists of elements of type B can be specified by the functor $F_{\mathit{List}} X = 1 + B \times X$. Writing* Nil *for the left injection and* Cons *for the right injection into the coproduct $F_{\mathit{List}} X$, the F_{List}-algebra $\mathit{lengthalg} : F_{\mathit{List}} \mathbb{N} \to \mathbb{N}$ that computes the lengths of lists is*

$$\begin{aligned} \mathit{lengthalg}\ \mathsf{Nil} &= 0 \\ \mathit{lengthalg}\ (\mathsf{Cons}(b, n)) &= n + 1 \end{aligned}$$

The refinement of μF_{List} by the algebra lengthalg is the least fixed point of

$$F_{\mathit{List}}^{\mathit{lengthalg}}(\mathbb{N}, P) = (\mathbb{N}, \lambda n.(n = 0)\ +\ \{n_1 : \mathbb{N}, x_1 : B, x_2 : P n_1 \mid n = n_1 + 1\})$$

This formulation of $\mu F_{\mathit{List}}^{\mathit{lengthalg}}$ is essentially the declaration `Vector` *from the introduction with the implicit equality constraints in that definition made explicit.*

Example 2. *We can similarly refine μF_{Tree} by the F_{Tree}-algebra*

$$\begin{aligned} \mathit{sum} &: F_{\mathit{Tree}} \mathbb{Z} \to \mathbb{Z} \\ \mathit{sum}\ (\mathsf{Leaf}\ z) &= z \\ \mathit{sum}\ (\mathsf{Node}\ (l, r)) &= l + r \end{aligned}$$

which sums the values in a tree. This gives the refinement $\mu F_{\mathit{Tree}}^{\mathit{sum}}$, where

$$F_{\mathit{Tree}}^{\mathit{sum}}(\mathbb{Z}, P) = (\mathbb{Z}, \lambda n.\ \{z : \mathbb{Z} \mid z = n\} + \{l, r : \mathbb{Z}, x_1 : P l, x_2 : P r \mid n = l + r\ \})$$

This corresponds to the Agda declaration

```
data IdxTree : Integer -> Set where
  IdxLeaf : (z : Integer) -> IdxTree z
  IdxNode : (l r : Integer) ->
                IdxTree l -> IdxTree r -> IdxTree (l + r)
```

Refinement by the initial algebra $(\mathit{in}_F : F(\mu F) \to \mu F)$ gives a μF-indexed type inductively characterised by $F^{\mathit{in}} = \Sigma_{\mathit{in}} \hat{F}$. Since in is an isomorphism, Σ_{in} is as well. Thus $F^{\mathit{in}} \cong \hat{F}$, so that $\mu F^{\mathit{in}} = \mu \hat{F} = K_1(\mu F)$, and each term is its own index. By contrast, refinement by the final algebra $(! : F1 \to 1)$ (which always exists because 1 is the terminal object of Set) gives a 1-indexed type inductively characterised by $F^{!}$. Since $F^{!} \cong F$, we have $\mu F^{!} = \mu F$, and all terms have index 1. Refining by the initial algebra thus has maximal discriminatory power, while refining by the terminal algebra has no discriminatory power.

5 Starting with Already Indexed Types

The development in section 4 assumes the type being refined is the initial algebra of an endofunctor F on Set. This seems to preclude refining an inductive type that is already indexed. But since we carefully identified the abstract structure of Fam(Set) we needed, our results can be extended to *any* fibration having that structure. In particular, we can refine indexed types using a *change-of-base* [20]:

$$\begin{tikzcd}
\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set}) \arrow[r] \arrow[d, "U^A"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mathrm{Fam}(\mathrm{Set}) \arrow[d, "U"] \\
\mathrm{Fam}(\mathrm{Set})_A \arrow[r, "\{-\}"] & \mathrm{Set}
\end{tikzcd}$$

This diagram, which is a pullback in Cat, the (large) category of categories and functors, generates a new fibration U^A from the functors U and $\{-\}$. The objects of $\mathrm{Fam}(\mathrm{Set})_A \times_{\mathrm{Set}} \mathrm{Fam}(\mathrm{Set})$ are (dependent) pairs $((A, P), (\{(A, P)\}, Y))$ of predicates. Thus, Y is "double indexed" by both $a \in A$ and $x \in Pa$.

The following theorem states that all the structure we require for our constructions is preserved by the change-of-base construction. It also generalises to any full cartesian Lawvere category with strong coproducts, so the change-of-base construction may be iterated.

Theorem 6. *U^A is a full cartesian Lawvere category with strong coproducts.*

Proof. First, U^A is a fibration by construction. The truth functor is defined by $K_1^A(A, P) = ((A, P), K_1\{(A, P)\})$ and the comprehension functor is defined by $\{((A, P), (\{(A, P)\}, Y))\}^A = \Sigma_{\pi_{(A,P)}}(\{(A, P)\}, Y)$. Coproducts are defined directly using the coproducts of U.

Example 3. *To demonstrate the refinement of an already indexed inductive type we consider a small expression language of well-typed terms. Let $\mathbb{T} = \{\mathsf{int}, \mathsf{bool}\}$ be the set of possible base types. The language is μF_{wtexp} for the functor F_{wtexp} : $\mathrm{Fam}(\mathrm{Set})_{\mathbb{T}} \to \mathrm{Fam}(\mathrm{Set})_{\mathbb{T}}$ given by*

$$\begin{aligned}
F_{\mathit{wtexp}}(\mathbb{T}, P) = (\mathbb{T},\ \lambda t : \mathbb{T}.\ & \{z : \mathbb{Z} \mid t = \mathsf{int}\} + \{b : \mathbb{B} \mid t = \mathsf{bool}\} \\
& + \{t_1, t_2 : \mathbb{T},\ x_1 : Pt_1,\ x_2 : Pt_2 \mid t_1 = t_2 = t = \mathsf{int}\} \\
& + \{t_1, t_2, t_3 : \mathbb{T},\ x_1 : Pt_1,\ x_2 : Pt_2,\ x_3 : Pt_3 \mid \\
& \qquad\qquad\qquad\qquad t_1 = \mathsf{bool}, t_2 = t_3 = t\})
\end{aligned}$$

For any t, write IntConst*,* BoolConst*,* Add*, and* If *for the four injections into $(\mathit{snd}\,(F_{\mathit{wtexp}}(\mathbb{T}, P))\,t$. Letting $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ denoting the set of booleans, and assuming there exist a $\mathbb{T}$-indexed family T such that T $\mathsf{int} = \mathbb{Z}$ and T $\mathsf{bool} = \mathbb{B}$, we have a semantic interpretation of the extended language's types. This can be used to specify a "tagless" interpreter by giving an F_{wtexp}-algebra:*

$$\begin{aligned}
&\mathit{eval} :\ F_{\mathit{wtexp}}(\mathbb{T}, T) \to (\mathbb{T}, T) \\
&\mathit{eval} = (\mathit{id}, \lambda x : \mathbb{T}.\ \lambda t : \mathit{snd}\,(F_{\mathit{wtexp}}(\mathbb{T}, T))\,x.\ \mathit{case}\ t\ \mathit{of} \\
&\qquad\qquad \mathsf{IntConst}\ z \qquad\qquad\quad \Rightarrow z \\
&\qquad\qquad \mathsf{BoolConst}\ b \qquad\qquad \Rightarrow b \\
&\qquad\qquad \mathsf{Add}\ (\mathsf{int}, \mathsf{int}, z_1, z_2) \quad \Rightarrow z_1 + z_2 \\
&\qquad\qquad \mathsf{If}\ (\mathsf{bool}, t, t, b, x_1, x_2) \Rightarrow \mathit{if}\ b\ \mathit{then}\ x_1\ \mathit{else}\ x_2)
\end{aligned}$$

Refining μF_{wtexp} by eval yields a type indexed by $\Sigma t : \mathbb{T}.\ Tt$, i.e., by $\{(\mathbb{T}, T)\}$. This type associates to every well-typed expression that expression's semantics.

6 Partial Refinement

In Sections 4 and 5 we assumed that every element of an inductive type has an index that can be assigned to it. Every list has a length, every tree has a number of leaves, every well-typed expression has a semantic meaning, and so on. But how can an inductive type be refined if only *some* data have values by which we want to index? For example, how can the inductive type of well-typed expressions of Example 3 be obtained by refining a data type of untyped expressions by an algebra for type assignment? And how can the inductive type of red-black trees be obtained by refining a data type of coloured trees by an algebra enforcing the well-colouring properties? As these examples show, the problem of refining subsets of inductive types is a common and naturally occurring one. Partial refinement is a technique for solving this problem.

The key idea underlying the required generalisation of our theory is to move from algebras to partial algebras. If F is a functor, then a *partial F-algebra* is a pair $(A, \alpha : FA \to (1 + A))$ comprising a carrier A and a structure map $\alpha : FA \to (1+A)$. We write $\mathsf{ok} : A \to 1+A$ and $\mathsf{fail} : 1 \to 1+A$ for the injections into $1 + A$, and often refer to a partial algebra solely by its structure map. The functor $MA = 1 + A$ is (the functor part of) the *error monad.*

Example 4. *The inductive type of expressions is μF_{exp} for the functor $F_{exp}X = \mathbb{Z}+\mathbb{B}+(X\times X)+(X\times X\times X)$. Letting $\mathbb{T} = \{\mathsf{int}, \mathsf{bool}\}$ as in Example 3 and using the obvious convention for naming the injections into $F_{exp}X$, such expressions can be type-checked using the following partial F_{exp}-algebra:*

$$\begin{aligned}
&\mathit{tyCheck} && : F_{exp}\mathbb{T} \to 1 + \mathbb{T}\\
&\mathit{tyCheck}\ (\mathsf{IntConst}\ z) && = \mathsf{ok}\ \mathsf{int}\\
&\mathit{tyCheck}\ (\mathsf{BoolConst}\ b) && = \mathsf{ok}\ \mathsf{bool}\\
&\mathit{tyCheck}\ (\mathsf{Add}\ (t_1, t_2)) && = \begin{cases}\mathsf{ok}\ \mathsf{int} & \mathit{if}\ t_1 = \mathsf{int}\ \mathit{and}\ t_2 = \mathsf{int}\\ \mathsf{fail} & \mathit{otherwise}\end{cases}\\
&\mathit{tyCheck}\ (\mathsf{If}\ (t_1, t_2, t_3)) && = \begin{cases}\mathsf{ok}\ t_2 & \mathit{if}\ t_1 = \mathsf{bool}\ \mathit{and}\ t_2 = t_3\\ \mathsf{fail} & \mathit{otherwise}\end{cases}
\end{aligned}$$

Example 5. *Let $\mathbb{S} = \{\mathsf{R}, \mathsf{B}\}$ be a set of colours. The inductive type of coloured trees is μF_{ctree} for the functor $F_{\mathsf{ctree}}X = 1 + \mathbb{S} \times X \times X$. We write* Leaf *and* Br *for injections into $F_{\mathsf{ctree}}X$. Red-black trees [11] are coloured trees satisfying the following constraints:*

1. *Every leaf is black;*
2. *Both children of a red node are black;*
3. *For every node, all paths to leaves contain the same number of black nodes.*

We can check whether or not a coloured tree is a red-black tree using the following partial F_{ctree}-algebra. Its carrier $\mathbb{S} \times \mathbb{N}$ records the colour of the tree in the first component and the number of black nodes to any leaf, assuming this number is the same for every leaf, in the second.

$$
\begin{aligned}
&\mathit{checkRB} &&: F_{ctree}(\mathbb{S}\times\mathbb{N}) \to 1 + (\mathbb{S}\times\mathbb{N})\\
&\mathit{checkRB}\ \mathsf{Leaf} &&= \mathsf{ok}\ (\mathsf{B}, 1)\\
&\mathit{checkRB}\ (\mathsf{Br}\ (\mathsf{R}, (s_1, n_1), (s_2, n_2))) &&= \begin{cases} \mathsf{ok}\ (\mathsf{R}, n_1) & \textit{if } s_1 = s_2 = \mathsf{B} \textit{ and } n_1 = n_2\\ \mathsf{fail} & \textit{otherwise}\end{cases}\\
&\mathit{checkRB}\ (\mathsf{Br}\ (\mathsf{B}, (s_1, n_1), (s_2, n_2))) &&= \begin{cases} \mathsf{ok}\ (\mathsf{B}, n_1 + 1) & \textit{if } n_1 = n_2\\ \mathsf{fail} & \textit{otherwise}\end{cases}
\end{aligned}
$$

The process of (total) refinement described in section 4 constructs, from a functor F with initial algebra $(in_F : F(\mu F) \to \mu F)$ and an F-algebra $\alpha : FA \to A$, a functor F^α such that μF^α associates to each $x : \mu F$ its index $(\!|\alpha|\!)x$. If we can compute an index for each element of μF from a partial F-algebra, then we can apply the same technique to partially refine μF. The key to doing this is to turn every partial F-algebra into a (total) F-algebra. Let λ be any distributive law for the error monad M over the functor F. Then λ respects the unit and multiplication of M (see [4] for details), and

Lemma 1. *Every partial F-algebra $\kappa : FA \to 1 + A$ generates an F-algebra $\overline{\kappa} : F(1 + A) \to (1 + A)$ defined by $\overline{\kappa} = [\mathsf{fail}, \kappa] \circ \lambda_A$.*

Here, $[\mathsf{fail}, \kappa]$ is the cotuple of the functions fail and κ. Refining μF by the F-algebra $\overline{\kappa}$ using the techniques of section 4 would result in an inductive type indexed by $1 + A$. But, as our examples show, what we actually want is an A-indexed type that inductively describes only those terms having values of the form $\mathsf{ok}\ a$ for some $a \in A$. Partial refinement constructs, from a functor F with initial algebra $(in_F : F(\mu F) \to \mu F)$ and a partial F-algebra $\kappa : FA \to 1 + A$, a functor $F^{?\kappa}$ such that $\mu F^{?\kappa}$ is the A-indexed type

$$
(A, \lambda a.\ \{x : \mu F \mid (\!|\overline{\kappa}|\!)x = \mathsf{ok}\ a\}) = \mathsf{ok}^* \Sigma_{(\!|\overline{\kappa}|\!)} K_1(\mu F) = \mathsf{ok}^* \mu F^{\overline{\kappa}} \tag{9}
$$

As we will see in Theorem 7, if

$$
F^{?\kappa} = \mathsf{ok}^* \Sigma_{\overline{\kappa}} \hat{F} \tag{10}
$$

then $\mu F^{?\kappa} = (A, \lambda a.\ \{x : \mu F \mid (\!|\overline{\kappa}|\!)x = \mathsf{ok}\ a\})$. Indeed, since left adjoints preserve initial objects, we can prove $\mu F^{?\kappa} \cong \mathsf{ok}^* \mu F^{\overline{\kappa}}$ by lifting the following adjunction to an adjunction between $\mathrm{Alg}_{F^{?\kappa}}$ and $\mathrm{Alg}_{F^{\overline{\kappa}}}$ via Theorem 1:

$$
\begin{tikzcd}
\mathrm{Fam}(\mathrm{Set})_A \arrow[r, bend right, "\Pi_{\mathsf{ok}}"'] \arrow[r, phantom, "\perp"] & \mathrm{Fam}(\mathrm{Set})_{1+A} \arrow[l, bend right, "\mathsf{ok}^*"']
\end{tikzcd}
$$

To satisfy the precondition of Theorem 1, we prove that $\mathsf{ok}^* F^{\overline{\kappa}} \cong F^{?\kappa}\mathsf{ok}^*$ by first observing that if F preserves pullbacks, then $\hat{F}$ preserves re-indexing, i.e., for every function f, $\hat{F} f^* \cong (Ff)^* \hat{F}$. This is proved by direct calculation. Thus if F preserves pullbacks, and if

$$
\mathsf{ok}^* \Sigma_{\overline{\kappa}} \cong \mathsf{ok}^* \Sigma_\kappa (F\mathsf{ok})^* \tag{11}
$$

then $\mathsf{ok}^* F^{\overline{\kappa}} = \mathsf{ok}^* \Sigma_{\overline{\kappa}} \hat{F} \cong \mathsf{ok}^* \Sigma_\kappa (F\mathsf{ok})^* \hat{F} \cong \mathsf{ok}^* \Sigma_\kappa \hat{F} \mathsf{ok}^* = F^{?\kappa} \mathsf{ok}^*$. The first equality is by Equation 6, the first isomorphism is by Equation 11, the second isomorphism is by the preceding observation assuming that F preserves pullbacks, and the final equality is by Equation 10. All container functors [1], and hence all polynomial functors, preserve pullbacks. Finally, to verify Equation 11, we require that the distributive law λ for M over F satisfies the following property, which we call *non-introduction of failure*: for all $x : F(1 + A)$ and $y : FA$, $\lambda_A\, x = \mathsf{ok}\ y$ if and only if $x = F\,\mathsf{ok}\, y$. This property strengthens the usual unit axiom for λ in which the implication holds only from right to left. It ensures that if applying λ does not result in failure, then no failures were present in the data to which it was applied. In an arbitrary category, this property is formulated as requiring the following square (i.e., the unit axiom for λ) to be a pullback:

$$\begin{tikzcd}
FA \arrow[r, "F\mathsf{ok}"] \arrow[d, "\mathrm{id}"'] & F(1+A) \arrow[d, "\lambda_A"] \\
FA \arrow[r, "\mathsf{ok}"] & 1 + FA
\end{tikzcd}$$

Every container functor has a canonical distributive law for M satisfying the non-introduction of failure property. We now have

Lemma 2. *If the distributive law λ satisfies non-introduction of failure, then Equation 11 holds.*

Proof. Given $(F(1 + A), P : F(1 + A) \to \mathrm{Set})$, we have

$$\begin{aligned}
&(\mathsf{ok}^* \circ \Sigma_{\overline{\kappa}})(F(1 + A), P) \\
&= (A, \lambda a : A.\ \{(x_1 : F(1 + A), x_2 : P x_1) \mid [\mathsf{fail}, \kappa](\lambda_A x_1) = \mathsf{ok}\ a\}) \\
&\cong (A, \lambda a : A.\ \{x_1 : FA, x_2 : P(F\,\mathsf{ok}\, x_1) \mid \kappa x_1 = \mathsf{ok}\ a\}) \\
&\cong (A, \mathsf{ok}^* \circ \Sigma_\kappa \circ (F\,\mathsf{ok})^*(F(1 + A), P))
\end{aligned}$$

And, putting everything together, we get the correctness of partial refinement:

Theorem 7. *If λ is a distributive law for M over F with the non-introduction of failure property, and if F preserves pullbacks, then $F^{?\kappa}$ has an initial algebra whose carrier is given by Equation 9.*

In fact, Theorem 7 holds in the more general setting of full cartesian Lawvere category whose coproducts satisfy the Beck-Chevalley condition and whose base categories satisfy extensivity [8]. Moreover, Theorem 6 extends to show that these properties are also preserved by the change-of-base construction provided all fibres of the original fibration satisfy extensivity.

7 Conclusions, Applications, Related and Future Work

We have given a clean semantic framework for deriving refinements of inductive types which store computationally relevant information within the indices of refined types. We have also shown how indexed types can be refined further, and how refined types can be derived even when some elements of the original type do not have indices. In addition to its theoretical clarity, the theory of refinement we have developed has potential applications in the following areas:

Dependently Typed Programming: Often a user is faced with a choice between building properties of elements of types into more sophisticated types, or stating these properties externally as, say, pre- and post-conditions. While the former is clearly preferable because properties can then be statically type checked, it also incurs an overhead which can deter its adoption. Supplying the programmer with infrastructure to produce refined types as needed can reduce this overhead.
Libraries: Library implementors need no longer provide a comprehensive collection of data types, but rather methods for defining new data types. Similarly, our results suggest that library implementors need not guess which refinements of data types will prove useful to programmers, and can instead focus on providing useful abstractions for creating more sophisticated data types from simpler ones.
Implementation: Current implementations of types such as `Vector` store all index information. For example, a vector of length 3 will store the lengths 3, 2, and 1 of its subvectors. Brady [7] seeks to determine when this information can be generated "on the fly" rather than stored. Our work suggests that the refinement μF^{α} can be implemented by simply implementing the underlying type μF, since programs requiring indices can reconstruct these as needed. We thus provide a user-controllable tradeoff between space and time efficiency.

Related Work: The work closest to that reported here is McBride's work on ornaments [23]. McBride defines a type of descriptions of inductive data types along with a notion of one description "ornamenting" another. Despite the differences between our fibrational approach and his type theoretic approach, the notion of refinement presented in Sections 4 and 5 is very similar to his notion of an algebraic ornament.

A line of research allowing the programmer to give refined types to constructors of inductive data types was initiated by Freeman and Pfenning [15] and later developed by Xi [27], Davies [12] and Dunfield [14] for ML-like languages, and by Pfenning [24] and Lovas and Pfenning [22] for LF. The work of Kawaguchi *et al.* [21] is also similar. This research begins with an existing type system and aims to provide the programmer with a means of expressing richer properties of values that are well-typeable in that type system. It is thus similar to the work reported here, although we are working in (a syntax-free presentation of) an intrinsically typed language, and this affects pragmatic considerations such as decidability of type checking. We also formally prove that each refinement is isomorphic to the richer, property-expressing data type it is intended to capture, rather than leaving this to the programmer to justify on a refinement-by-refinement basis.

Refinement types have also been used elsewhere to give more precise types to programs in existing programming languages (but not specifically to inductive types). For example, Denney [13] and Gordon and Fournet [17] use subset types to refine the type systems of ML-like languages. Subset types are also used heavily in the PVS theorem prover [25].

Our results extend the systematic code reuse delivered by generic programming [2,3,5]: in addition to generating new programs we can also generate new types from existing types. This area is being explored in Epigram [9], in which codes for data types can be represented within a predicative intensional system so that programs can generate new data types. It should be possible to implement our refinement process using similar techniques.

Aside from the specific differences between our work and that discussed above, a distinguishing feature of our work is the semantic methodology we use to develop refinement. We believe that this methodology is new. We also believe that a semantic approach is important: it can serve as a principled foundation for refinement, as well as provide a framework in which to compare different implementations. It may also lead to new algebraic insights into refinement which complement the logical perspective of previous work.

Finally, we are interested in a number of extensions to the work reported here. Many readers will wonder about the possibility of a more general monadic refinement using, for example, Kleisli categories. We are working on this, but due to space limitations have chosen to concentrate in this paper on partial refinement, which is already sufficient to show that refinement is applicable to sophisticated programming problems. In addition, many more indexed inductive data types exist than can be produced by the refinement process described in this paper. We leave it to future work to discover to what extent this developing world of dependently typed data structures can be organised and characterised by processes like refinement and its extensions.

Acknowledgements. We would like to thank Conor McBride, Frank Pfenning, and Pierre-Evariste Dagand for helpful comments on this work.

References

1. Abbott, M., Altenkirch, T., Ghani, N.: Containers - constructing strictly positive types. Theoretical Computer Science 342, 3–27 (2005)
2. Altenkirch, T., McBride, C., Morris, P.: Generic Programming with Dependent Types. In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) SSDGP 2006. LNCS, vol. 4719, pp. 209–257. Springer, Heidelberg (2007)
3. Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.): SSDGP 2006. LNCS, vol. 4719. Springer, Heidelberg (2007)
4. Barr, M., Wells, C.: Toposes, Triples and Theories. Springer, Heidelberg (1983)
5. Benke, M., Dybjer, P., Jansson, P.: Universes for generic programs and proofs in dependent type theory. Nordic Journal of Computing 10(4), 265–289 (2003)
6. Bird, R.S., de Moor, O.: Algebra of Programming. Prentice Hall, Englewood Cliffs (1997)
7. Brady, E., McBride, C., McKinna, J.: Inductive Families Need Not Store Their Indices. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 115–129. Springer, Heidelberg (2004)
8. Carboni, A., Lack, S., Walters, R.F.C.: Introduction to extensive and distributive categories. Journal of Pure and Applied Algebra 84, 145–158 (1993)
9. Chapman, J., Dagand, P.-E., McBride, C., Morris, P.: The gentle art of levitation. In: Proc. ICFP, pp. 3–14 (2010)

10. Chuang, T.-R., Lin, J.-L.: An algebra of dependent data types. Technical Report TR-IIS-06-012, Institute of Information Science, Academia Sinica, Taiwan (2006)
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press and McGraw-Hill (2001)
12. Davies, R.: Practical Refinement-Type Checking. PhD thesis, Carnegie Mellon University, available as Technical Report CMU-CS-05-110 (2005)
13. Denney, E.: Refinement types for specification. In: Proc. PROCOMET, pp. 148–166. Chapman and Hall, Boca Raton (1998)
14. Dunfield, J.: A Unified System of Type Refinements. PhD thesis, Carnegie Mellon University, available as Technical Report CMU-CS-07-129 (2007)
15. Freeman, T., Pfenning, F.: Refinement types for ML. In: Proc. Symposium on Programming Language Design and Implementation, pp. 268–277 (June 1991)
16. Ghani, N., Johann, P., Fumex, C.: Fibrational Induction Rules for Initial Algebras. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 336–350. Springer, Heidelberg (2010)
17. Gordon, A.D., Fournet, C.: Principles and applications of refinement types. Technical Report MSR-TR-2009-147, Microsoft Research (October 2009)
18. Hermida, C., Jacobs, B.: Structural induction and coinduction in a fibrational setting. Information and Computation 145(2), 107–152 (1998)
19. Jacobs, B.: Comprehension categories and the semantics of type dependency. Theoretical Computer Science 107, 169–207 (1993)
20. Jacobs, B.: Categorical Logic and Type Theory. Studies in Logic and the Foundations of Mathematics, vol. 141. North Holland, Amsterdam (1999)
21. Kawaguchi, M., Rondon, P.M., Jhala, R.: Type-based data structure verification. In: PLDI, pp. 304–315 (2009)
22. Lovas, W., Pfenning, F.: Refinement types for logical frameworks and their interpretation as proof irrelevance. Log. Meth, in Comp. Sci. (2010) (to appear)
23. McBride, C.: Ornamental algebras, algebraic ornaments (2010) (unpublished note)
24. Pfenning, F.: Refinement types for logical frameworks. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 285–299. Springer, Heidelberg (1994)
25. Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: Predicate subtyping in pvs. IEEE Transactions on Software Engineering 24(9), 709–720 (1998)
26. The Agda Team (2010), http://wiki.portal.chalmers.se/agda
27. Xi, H.: Dependently typed data structures. Revision after WAAAPL 1999 (2000)

Complexity of Strongly Normalising λ-Terms via Non-idempotent Intersection Types

Alexis Bernadet[1,2] and Stéphane Lengrand[1,3]

[1] École Polytechnique, France
[2] École Normale Supérieur de Cachan, France
[3] CNRS, France
{lengrand,bernadet}@lix.polytechnique.fr

Abstract. We present a typing system for the λ-calculus, with non-idempotent intersection types. As it is the case in (some) systems with idempotent intersections, a λ-term is typable if and only if it is strongly normalising. Non-idempotency brings some further information into typing trees, such as a bound on the longest β-reduction sequence reducing a term to its normal form.

We actually present these results in Klop's extension of λ-calculus, where the bound that is read in the typing tree of a term is refined into an exact measure of the longest reduction sequence.

This complexity result is, for longest reduction sequences, the counterpart of de Carvalho's result for linear head-reduction sequences.

1 Introduction

Intersection types were introduced in [CD78], extending the simply-typed λ-calculus with a notion of finite polymorphism. This is achieved by a new construct $A \cap B$ in the syntax of types and new typing rules such as:

$$\frac{M : A \quad M : B}{M : A \cap B}$$

where $M : A$ denotes that a term M is of type A.

One of the motivations was to characterise strongly normalising (SN) λ-terms, namely the property that a λ-term can be typed if and only if it is strongly normalising. Variants of systems using intersection types have been studied to characterise other evaluation properties of λ-terms and served as the basis of corresponding semantics [Lei86, Ghi96, DCHM00, CS07].

This paper refines with quantitative information the property that typability characterises strong normalisation. Since strong normalisation ensures that all reduction sequences are finite, we are naturally interested in identifying the length of the longest reduction sequence. We do this with a typing system that is very sensitive to the usage of resources when λ-terms are reduced.

This system results from a long line of research inspired by Linear Logic [Gir87]. The usual logical connectives of, say, classical and intuitionistic logic, are decomposed therein into finer-grained connectives, separating a *linear* part from a part

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 88–107, 2011.

that controls how and when the structural rules of *contraction* and *weakening* are used in proofs. This can be seen as resource management when hypotheses, or more generally logical formulae, are considered as resource.

The Curry-Howard correspondence, which originated in the context of intuitionistic logic [How80], can be adapted to Linear Logic [Abr93, BBdH93], whose resource-awareness translates to a control of resources in the execution of programs (in the usual computational sense). From this, have emerged some versions of linear logic that capture polytime functions [BM03, Laf04, GR07]. Also from this has emerged a theory of λ-calculus with resource, with semantical support (such as the differential λ-calculus) [ER03, BEM10]. In this line of research, de Carvalho [dC05, dC09] obtained interesting measures of reduction lengths in the λ-calculus by means of *non-idempotent* intersection types (as pionnered by [KW99, NM04]).

Intersections were originally introduced as idempotent, with the equation $A \cap A = A$ either as an explicit quotient or as a consequence of the system. This corresponds to the understanding of the judgement $M : A \cap B$ as follows: M can be used as data of type A or data of type B. But the meaning of $M : A \cap B$ can be strengthened in that M **will** be used **once** as data of type A and **once** as data of type B. With this understanding, $A \cap A \neq A$, and dropping idempotency of intersections is thus a natural way to study control of resources and complexity. Using this, de Carvalho [dC09] has shown a correspondence between the size of the typing derivation tree and the number of steps taken by a Krivine machine to reduce the term. This relates to the length of linear head-reductions, but if we remain in the realm of intersection systems that characterise strong normalisation, then the more interesting measure is the length of the longest reduction sequence. In this paper we get a result similar to de Carvalho's, but with the measure corresponding to strong normalisation.

First we define a system with non-idempotent intersection types. Then we prove that if a term is typable then it is SN (soundness) and that if a term is SN then it is typable (correctness). As opposed to idempotent intersection types, the proof of correctness is very direct: we use a simple measure on typing trees (the easy proof differs from that in [Val01], in that the typing system itself does not perform β-reduction).

The proof of soundness gives us immediatly a bound of the maximum number of β-reductions. So we have an inequality result for complexity. We would like an equality result.

One of the reasons why we only have an inequality result is because, in a β-reduction, the argument of the β-redex may disappear (we call this weakening). One simple way to avoid the weakening problem without blocking computation is to use Klop's extension of λ-calculus:

$$M, N ::= \ \ldots \mid [M, N]$$

In $[M, N]$, N is a sub-term that was meant to be erased. To avoid weakenings, we can replace every term $\lambda x.M$ such that $x \notin FV(M)$ with $\lambda x.[M, x]$.

We refer the reader to [Sør97, Xi97] for a survey on different techniques based on the λI-calculus to infer normalisation properties. Intersection types in the framework of Church-Klop's calculus have been studied in e.g. [DCT07], but, to our knowledge, they were always considered idempotent. Hence, the quantitative analysis provided by non-idempotency was not carried out.

In order to obtain the complexity result we want, we still have to expect a property on typing trees: *optimality*. This property is mostly on the "interface types". It is not too restrictive because the completeness theorem produces such typing trees, but it is still quite so because we can prove that for each term, the shape of such a derivation tree is unique (*principality*).

We can then prove that if π is a optimal-principal typing tree of M then we can read off π the **exact** length of the longest reduction sequence starting from M.

2 Syntax and Typing

In this section we present the calculus and the typing system that we are going to use.

2.1 Lambda Calculus with Klop's Extension

As said in the introduction, the language that we are going to type is the pure λ-calculus extended with Klop's construct [Klo80]:

Definition 1 (Syntax and reduction rules)

- *Terms are defined by the following grammar*

$$M, N ::= \ x \mid \lambda x.M \mid MN \mid [M, N]$$

Unless otherwise stated, we do not consider any restriction on this syntax. Sometimes we write $N\ \overrightarrow{M_i}$ for $N\ M_1 \ldots M_n$ (when $\overrightarrow{M_i}$ is the vector of terms $M_1 \ldots M_n$).

The free variables $fv(M)$ of a term M are defined as usual and terms are considered up to α-equivalence.

- *The reduction rules are β-reduction and π-reduction (see e.g. [Klo80]):*

$$\begin{array}{lll} \beta & (\lambda x.M)\ N \longrightarrow & M\{x := N\} \\ \pi & [M_1, N]M_2 \longrightarrow & [M_1\ M_2, N] \end{array}$$

If S is a rule (such as β or π), or a system of rules (like $\beta\pi$), we write $M \longrightarrow_S N$ for the congruent closure of the (system of) rule(s).

An interesting fragment of the calculus is Church-Klop's λI [Klo80]:

Definition 2 (λI). *A term M is in the λI-fragment if any abstraction $\lambda x.N$ occurring in M is such that $x \in fv(N)$.*

Remark 1. The λI-fragment is stable under substitution and under $\longrightarrow_{\beta\pi}$. If $M \longrightarrow_{\beta\pi} N$ then $fv(M) = fv(N)$ and $P\{x := M\} {\longrightarrow^+}_{\beta\pi} P\{x := N\}$ provided that $x \in fv(P)$.

Another obvious fragment is the pure λ-calculus[1]. Klop's construct is only useful here for the complexity results of section 5; we do not need it for soundness or completeness (section 3). So if one is interested only in the pure λ-calculus, it is possible to follow the proofs of these theorems while ignoring Klop's construct. As we will see later some theorems are not true for every reduction (for example, the subject expansion property), so we define the reduction $M \hookrightarrow_N M'$, where N is either a term or ϵ (a dummy placeholder) as follows:

Definition 3 (A restricted reduction relation)

$$\frac{x \in FV(M)}{(\lambda x.M)N \hookrightarrow_\epsilon M\{x := N\}} \qquad \frac{x \notin FV(M)}{(\lambda x.M)N \hookrightarrow_N M}$$

$$\frac{M_1 \hookrightarrow_N M_1'}{M_1 M_2 \hookrightarrow_N M_1' M_2} \qquad \frac{M_2 \hookrightarrow_N M_2'}{M_1 M_2' \hookrightarrow_N M_1 M_2'}$$

$$\frac{M \hookrightarrow_N M \;\; x \notin FV(N)}{\lambda x.M \hookrightarrow_N \lambda x.M} \qquad \frac{}{[M_1, N]M_2 \hookrightarrow_\epsilon [M_1 \;\; M_2, N]}$$

$$\frac{M_1 \hookrightarrow_N M_1'}{[M_1, M_2] \hookrightarrow_N [M_1', M_2]} \qquad \frac{M_2 \hookrightarrow_N M_2'}{[M_1, M_2] \hookrightarrow_N [M_1, M_2']}$$

Fig. 1. $\hookrightarrow$-reduction

Fig. 1 formalises the reduction relation that can be be intuitively described as follows: $M \hookrightarrow_N M'$ *if and only if* $M \longrightarrow_{\beta\pi} M'$ *such that:*

- *either the reduction* $M \longrightarrow_{\beta\pi} M'$ *does not erase anything in which case* $N = \epsilon$.
- *or the reduction* $M \longrightarrow_{\beta\pi} M'$ *erases the term* N *within* M *and the free variables of* N *are not bound by binders in* M.

Remark 2. If M is a λI term then every reduction is in $\hookrightarrow_\epsilon$.

2.2 Intersection Types and Contexts

Definition 4 (Intersection types)
Our intersection types are defined by the following grammar:

$$\begin{array}{rcll} A, B & ::= & F \mid A \cap B & \\ F, G & ::= & \tau \mid A \to F & \\ U, V & ::= & \omega \mid A & \quad \textit{(General types)} \end{array}$$

[1] This motivates our choice of sticking to the reduction rules of [Klo80] rather than opting for the variant in [Bou03] where β-reduction can generate new instances of Klop's construct.

We consider the types up to associativity and commutativity of intersections. The notation $U \cap V$ extends the intersection construct to general types using the following equations:

$$A \cap \omega = \omega \cap A = A \qquad \omega \cap \omega = \omega$$

As opposed to [CD78, CD80] we do not have idempotency $A = A \cap A$.

Remark 3

- It would be possible to formalise the results of this paper without using general types, but then in many definitions and proofs we would have to deal with two or three cases instead of one.
- Notice that, by construction, if $A \to B$ is a type then B is not an intersection. This limitation, which corresponds to the *strict types* of [vB92], is useful for the key property of separation (Lemma 1).

Definition 5 (Type inclusion). *We define inclusion on types as follows:*

$U \subseteq V$ if $V = \omega$, or $U = V$, or $U = A$ and $V = B$ with $A = B \cap C$ for some C.

Notice that this definition of inclusion is weaker than the traditional one originating from [BCDC83]. Indeed, inclusions between domains and co-domains of function types do not induce inclusions between function types.

Remark 4. $\subseteq$ is a partial order.

Definition 6 (Contexts)

- *A context Γ is a total map from variables $(x, y, z, \ldots)$ to general types (U, V, $\ldots$) that has finite support, i.e. such that $\{x \mid \Gamma(x) \neq \omega\}$ is finite.*
- *The expression $(x_1 : U_1, ..., x_n : U_n)$, where for all i, j, $x_i \neq x_j$, is defined as the context Γ such that:*
  - *For all i, $\Gamma(x_i) = U_i$*
  - *$\Gamma(y) = \omega$ for any other y*
- *Given two contexts Γ and Δ,*
  $\Gamma \cap \Delta$ is defined pointwise as: for all x, $(\Gamma \cap \Delta)(x) = \Gamma(x) \cap \Delta(x)$, and we write $\Gamma \subseteq \Delta$ for either $\Gamma = \Delta$ or there exists Γ' such that $\Gamma = \Gamma' \cap \Delta$

Remark 5

- $\subseteq$ is a partial order on contexts.
- $\Gamma \subseteq \Delta$ if and only if for all x, $\Gamma(x) \subseteq \Delta(x)$

2.3 Typing System and Its Basic Properties

Definition 7 (Typing system)

- *Fig. 2 inductively defines the derivability of typing judgements, of the form $\Gamma \vdash M : U$. Typing trees will be denoted π, $\pi' \ldots$*

- *To prove strong normalisation we will use a simple measure on typing trees: we just count the number of occurences of rule* (*App*).

  So we write $\Gamma \vdash^{n} M:U$ *(resp.* $\Gamma \vdash^{\leq n} M:U$*, resp.* $\Gamma \vdash^{<n} M:U$*) if there exists a typing tree* π *concluding* $\Gamma \vdash M:U$ *and in* π *there are exactly (resp. less than or equal to, resp. less than)* n *occurences of rule* (*App*).
- *We say that a term* M *is typable if there exist* Γ *and* A *such that* $\Gamma \vdash M:A$

$$\frac{}{x:F \vdash x:F} \qquad \frac{\Gamma \vdash M:A \quad \Delta \vdash M:B}{\Gamma \cap \Delta \vdash M:A \cap B}$$

$$\frac{\Gamma, x:U \vdash M:F \quad A \subseteq U}{\Gamma \vdash \lambda x.M:A \to F} \qquad \frac{\Gamma \vdash M:A \to B \quad \Delta \vdash N:A}{\Gamma \cap \Delta \vdash MN:B}\,(App)$$

$$\frac{\Gamma \vdash M:F \quad \Delta \vdash N:A}{\Gamma \cap \Delta \vdash [M,N]:F} \qquad \frac{}{\vdash M:\omega}$$

Fig. 2. Typing system

In the rest of this section and for the proof of subject reduction we can ignore everything about the measure: but taking it into account does not complicate the proofs.

Remark 6. If $\Gamma \vdash^{n} M:U$ and $\Delta \vdash^{m} M:V$ then $\Gamma \cap \Delta \vdash^{n+m} M:U \cap V$

Lemma 1 (Separation). *If* $\Gamma \vdash^{n} M:U_1 \cap U_2$ *then there exist* Γ_1, Γ_2, n_1 *and* n_2 *such that* $\Gamma = \Gamma_1 \cap \Gamma_2$, $n = n_1 + n_2$, *and* $\Gamma_1 \vdash^{n_1} M:U_1$ *and* $\Gamma_2 \vdash^{n_2} M:U_2$.

Proof. By induction on the typing tree.

- If $U_1 = \omega$ or $U_2 = \omega$ it is trivial.
- If $\dfrac{\Gamma \vdash^{n} M:C \quad \Delta \vdash^{m} M:D}{\Gamma \cap \Delta \vdash^{n+m} M:C \cap D}$ and $C \cap D = A_1 \cap A_2$ then there exist U_1, U_2, V_1, V_2 such that $C = U_1 \cap U_2$, $D = V_1 \cap V_2$, $A_1 = U_1 \cap V_1$ and $A_2 = U_2 \cap V_2$.

  By induction, there exist n_1, n_2, m_1, m_2, Γ_1, Γ_2, Δ_1, Δ_2 such that $n = n_1 + n_2$, $m = m_1 + m_2$, $\Gamma = \Gamma_1 \cap \Gamma_2$, $\Delta = \Delta_1 \cap \Delta_2$, $\Gamma_1 \vdash^{n_1} M : U_1$, $\Gamma_2 \vdash^{n_2} M:U_2$, $\Delta_1 \vdash^{m_1} M:V_1$, $\Delta_2 \vdash^{m_2} M:V_2$.

  So we have $\Gamma_1 \cap \Delta_1 \vdash^{n_1+m_1} M:A_1$ and $\Gamma_2 \cap \Delta_2 \vdash^{n_2+m_2} M:A_2$.
- All the other cases are impossible, especially the application rule: if $A \to B$ is a type, then B is not an intersection. This is why we used this restriction in the begining.

A useful property can be inferred from the separation property:

Corollary 1 (Weakening). *If* $\Gamma \vdash^{n} M:U$ *and* $U \subseteq V$ *then there exists* Γ' *such that* $\Gamma \subseteq \Gamma'$ *and* $\Gamma' \vdash^{\leq n} M:V$.

3 Soundness and Completeness of the Typing System w.r.t. Strong Normalisation

In this section we prove that a term is typable if and only if it is strongly normalising.

3.1 Soundness

Here we prove that typed terms are strongly normalising.

Lemma 2 (Typing of substitution). *If $\Gamma, x\!:\!U \vdash^{n_1} M\!:\!V$ and $\Delta \vdash^{n_2} N\!:\!U$ then $\Gamma \cap \Delta \vdash^{n_1+n_2} M\{x := N\}\!:\!V$.*

Proof. By induction on the typing tree of M.

- For the variable rule it is trivial.
- For $\vdash^0 M\!:\!\omega$ with $(\Gamma, x\!:\!U) = ()$, $n_1 = 0$, $V = \omega$. So we have $U = \omega$, so $n_2 = 0$ and $\Delta = ()$. So we can conclude.
- For $\dfrac{\Gamma, x\!:\!U, y\!:\!W \vdash^{n_1} M_1\!:\!F \quad A \subseteq W}{\Gamma, x\!:\!U \vdash^{n_1} \lambda y.M_1\!:\!A \to F}$ with $M = \lambda y.M_1$, $V = A \to F$, $x \neq y$, $y \notin FV(N)$. From the induction hypothesis we have $\Gamma, y\!:\!W \vdash^{n_1+n_2} : M_1\{x := N\}F$ so we can conclude.
- For $\dfrac{\Gamma_1, x\!:\!U_1 \vdash^{m_1} M\!:\!A \quad \Gamma_2, x\!:\!U_2 \vdash^{m_2} M\!:\!B}{\Gamma_1 \cap \Gamma_2, x\!:\!U_1 \cap U_2 \vdash^{m_1+m_2} M\!:\!A \cap B}$ with $n_1 = m_1 + m_2$, $U = U_1 \cap U_2$, $V = A \cap B$. By the use the Lemma 1, there exists Δ_1, Δ_2, m_3, m_4, such that $n_2 = m_3 + m_4$, $\Delta = \Delta_1 \cap \Delta_2$, $\Delta_1 \vdash^{m_3} N\!:\!U_1$, $\Delta_2 \vdash^{m_4} N\!:\!U_2$. By the induction hypothesis we have $\Gamma_1 \cap \Delta_1 \vdash^{m_1+m_3} M\{x := N\} : A$ and $\Gamma_2 \cap \Delta_2 \vdash^{m_2+m_4} M\{x := N\}\!:\!B$ so we can conclude.
- For the application rule and Klop' construct rule we adapt the proof for the intersection rule.

Theorem 1 (Subject Reduction for β)
If $\Gamma \vdash^n M : A$ and $M \longrightarrow_\beta M'$ then there exists Γ' such that $\Gamma' \vdash^{<n} M' : A$ and $\Gamma \subseteq \Gamma'$.

Proof. First by induction on $M \longrightarrow_\beta M'$ then by induction on A.

- If A is an intersection then we use the Lemma 1.
- For $\dfrac{}{(\lambda x.M_1)M_2 \longrightarrow_\beta M_1\{x := M_2\}}$ with A not an intersection then there exist B, Γ_1, Γ_2, n_1, n_2 such that $n = n_1+n_2+1$, $\Gamma = \Gamma_1 \cap \Gamma_2$, $\Gamma_1 \vdash^{n_1} \lambda x.M_1 : B \to A$ and $\Gamma_2 \vdash^{n_2} M_2 : B$. So there exists U such that $B \subseteq U$ and $\Gamma_1, x : U \vdash^{n_1} M_1 : A$. So, from Corollary 1, there exists Δ such that $\Gamma_2 \subseteq \Delta$ and $\Delta \vdash^{\leq n_2} M_2\!:\!U$. So by the use of the previous lemma we can conclude.

- If $\dfrac{M \longrightarrow_\beta M'}{\lambda x.M \longrightarrow_\beta \lambda x.M'}$ with A is not an intersection, then there exist U, A_2, A_3 such that $A = A_2 \rightarrow A_3$, $A_2 \subseteq U$ $\Gamma, x : U \vdash^n M : A_3$. By the induction hyptothesis there exist m, Δ, V such that $\Gamma \subseteq \Delta$, $m < n$, $U \subseteq V$ and $\Delta, x : V \vdash^m M' : A_3$. So we have $A_2 \subseteq V$, so we have $\Delta \vdash^m \lambda x.M' : A_2 \rightarrow A_3$.
- The other cases are straightforward.

Lemma 3 (Subject Reduction for π). *If $\Gamma \vdash^n M : A$ and $M \longrightarrow_\pi M'$ then $\Gamma \vdash^n M' : A$.*

Proof. Again, by induction first on A then on $M \longrightarrow_\pi M'$. All cases are straightforward.

Corollary 2 (Soundness). *If $\Gamma \vdash M : A$ then M is SN.*

Proof. The measure is decreased by β-reduction (Theorem 1), is invariant under π-reduction (Lemma 3), and π-reduction on its own terminates. Strong normalisation follows by considering the corresponding lexicographic order.

Notice that in the particular case of $\hookrightarrow$, Subject Reduction can be stated more precisely:

Theorem 2 (Subject Reduction for $\hookrightarrow$)
Assume $\Gamma \vdash M : B$ and $M \hookrightarrow_N M'$.
If $N \neq \epsilon$ there exist Δ and A such that $\Delta \vdash N : A$ otherwise let $\Delta = ()$.
There exists Γ' such that $\Gamma' \vdash M' : B$ with $\Gamma = \Gamma' \cap \Delta$.

Proof. By investigating the proof of Theorem 1.

In the even more particular case of the λI-fragment, where $\hookrightarrow_\epsilon = \longrightarrow_{\beta\pi}$, Subject Reduction does not modify the context.

3.2 Completeness

We now prove that strongly normalising terms can be typed.

Lemma 4 (Typing of substitution). *If $\Gamma \vdash M\{x := N\} : B$ then there exist Γ_1, Γ_2, U such that $\Gamma = \Gamma_1 \cap \Gamma_2$ and $\Gamma_1, x : U \vdash M : B$ and $\Gamma_2 \vdash N : U$.*

Proof. By induction on the typing tree for M. If $x \notin fv(M)$ we take $U = \omega$ and $\Gamma_2 = ()$.

Theorem 3 (Subject Expansion)
Assume $\Gamma \vdash M' : B$ and $M \hookrightarrow_N M'$.
Assume $\Delta \vdash N : A$ if $N \neq \epsilon$, otherwise let $\Delta = ()$.
We have $\Gamma \cap \Delta \vdash M : B$.

Proof. First by induction on B then by induction on $M \hookrightarrow_N M'$. The cases are exactly those of Subject Reduction.

Remark 7. This is not true with general β-reduction. For example, $(\lambda z.a)(\lambda y.yy)$ is typable but $(\lambda z.(\lambda x.a)(zz))(\lambda y.yy)$ is not (it is not SN).

Lemma 5 (Shape of $\hookrightarrow$-normal forms). *If a term cannot be reduced by $\hookrightarrow$ then it is of one of the following forms:*

- $\lambda x.M$
- $[M, N]$
- $xM_1 \dots M_n$

Proof. Straightforward.

Theorem 4 (Completeness). *If M is SN then there exist F and Γ such that $\Gamma \vdash M{:}F$.*

Proof. First by induction on the length of the longest $\longrightarrow_{\beta\pi}$ -reduction sequence starting from M then by induction on the size of M.

- If there exists M' such that $M \hookrightarrow_N M'$ then we use the induction hypothesis on M' (and if $N \neq \epsilon$ we use it on N too, which is a strict sub-term of M). Then we conclude with Theorem 3.
- If not then M is of one of these forms:
  - If $M = \lambda x.N$ then we apply the induction hypothesis on N.
  - If $M = [M_1, M_2]$ then we apply the induction hypothesis on M_1 and M_2
  - If $M = xM_1 \dots M_n$ then we apply the induction hypothesis on $M_1, \dots,$ M_n to get $\Gamma_1 \vdash M_1 : F_1, \dots, \Gamma_n \vdash M_n : F_n$, so we pick a fresh atomic type τ and we get:

$$(x{:}F_1{\to}\cdots{\to}F_n{\to}\tau) \cap \Gamma_1 \cap \ldots \cap \Gamma_n \vdash xM_1 \dots M_n{:}\tau$$

In the particular case of the λI-fragment, typing is preserved by arbitrary expansions, with no modification of context. Since normal forms can be typed, any weakly normalising term can be typed, and are thus strongly normalising. Equivalence between weak normalisation and strong normalisation in λI is a well-known theorem that finds here a simple proof.

3.3 Corollaries

Corollary 3 (Characterisation of Strong Normalisation). *M is typable if and only if M is SN.*

With the characterisation of strong normalisation and the subject expansion theorem we have the following corollary.

Corollary 4. *If $M \hookrightarrow_N M'$ and N is SN and M' is SN then M is SN.*

This is a useful result, used for instance in many proofs of strong normalisation (e.g. by reducibility candidates) for λ-terms that are typed in various systems. It can also be seen as a generalisation the theorem that in λI, weak normalisation is equivalent to strong normalisation.

4 Optimal and Principal Typing

As we showed in the previous section, the typing system of Fig. 2 above characterises strongly normalising terms. Even better, the measure defined on the typing tree of a term gives a bound on the length of longest reduction sequence. But the typing system is too coarse to improve on that bound, so in order to get a better result about complexity (as established in section 5) it needs to be refined.

4.1 Optimal Typing

In this section we first notice that the typing trees produced by the proof of completeness all satisfy a particular property that we call the *optimal* property.

This property involves the following notions:

Definition 8 (Subsumption and forgotten types)

- *If π is a typing tree, we say that π* uses subsumption *if it features an occurrence of the abstraction rule where the condition $A \subseteq U$ is neither $A \subseteq A$ nor $A \subseteq \omega$.*
- *If π is a typing tree with no subsumption then we say that a type A is* forgotten *in π if:*
  - *π features an occurence of the abstraction rule where the condition is $A \subseteq \omega$,*
  - *or π features an occurence of the typing rule for Klop's construct $[M, N]$ where A is the type of N.*

  The multiset of forgotten types in π is written $\mathsf{forg}(\pi)$.

The optimal property also involves refining the grammar of types:

Definition 9 (Refined intersection types). *A^+, A^- and A^{--} are defined by the following grammar:*

$$\begin{array}{lcl} A^+, B^+ & ::= & \tau \mid A^{--} \to B^+ \\ A^{--}, B^{--} & ::= & A^- \mid A^{--} \cap B^{--} \\ A^-, B^- & ::= & \tau \mid A^+ \to B^- \end{array}$$

The degree *of a type of the form A^+ is the number of arrows in negative positions:*

$$\begin{array}{|lcl|} \hline \delta^+(\tau) & := & 0 \\ \delta^+(A^{--} \to B^+) & := & \delta^-(A^{--}) + \delta^+(B^+) + 1 \\ \hline \delta^-(A^{--} \cap B^{--}) & := & \delta^-(A^{--}) + \delta^-(B^{--}) \\ \hline \delta^-(\tau) & := & 0 \\ \delta^-(A^+ \to B^-) & := & \delta^+(A^+) + \delta^-(B^-) \\ \hline \end{array}$$

We can finally define the optimal property:

Definition 10 (Optimal typing). *A typing tree π concluding $\Gamma \vdash M : A$ is optimal if*

- *There is no subsumption in* π
- A *is of the form* A^+
- *For every* $(x : B) \in \Gamma$, B *is of the form* B^{--}
- *For every forgotten type* B *in* π, B *is of the form* B^+.

We write $\Gamma \vdash_{\mathsf{opt}} M\!:\!A^+$ *if there exists such* π.

The degree *of such a typing tree is defined as*

$$\delta(\pi) = \delta^+(A^+) + \Sigma_{x\,:\,B^{--}\in\Gamma}\delta^-(B^{--}) + \Sigma_{C^+\in \mathsf{forg}(\pi)}\delta^+(C^+)$$

In this definition, A^+ is an output type, A^- is a basic input type (i.e. for a variable to be used once), and A^{--} is the type of a variable that can be used several times. The intuition behind this asymmetric grammar can be found in linear logic:

Remark 8. A simple type T can be translated as a type T^* of linear logic [Gir87] as follows:

$$\begin{array}{|lcl|}\hline \tau^* & := & \tau \\ (T \to S)^* & := & !S^* \multimap T^* \\ \hline\end{array}$$

It can also be translated as T^+ and T^- as follow:

$$\begin{array}{|lcl|lcl|}\hline \tau^+ & := & \tau & \tau^- & := & \tau \\ (T \to S)^+ & := & !T^- \multimap S^+ & (T \to S)^- & := & T^+ \multimap S^- \\ \hline\end{array}$$

And we have in linear logic: $T^- \vdash T^*$ and $T^* \vdash T^+$

Now we can establish Subject Expansion for optimal trees:

Theorem 5 (Subject Expansion, optimal case)
Assume $\Gamma \vdash_{\mathsf{opt}} M'\!:\!B$ *and* $M \hookrightarrow_N M'$.
Assume $\Delta \vdash_{\mathsf{opt}} N\!:\!A$ *if* $N \neq \epsilon$, *otherwise let* $\Delta = ()$.
We have $\Gamma \cap \Delta \vdash_{\mathsf{opt}} M\!:\!B$.

Proof. By investigating the proof of Theorem 3, noticing that, in Lemma 4:

- if the typing tree of $\Gamma \vdash M\{x := N\}\!:\!B$ does not use subsumption, neither do those of $\Gamma_1, x\!:\!U \vdash M\!:\!B$ and $\Gamma_2 \vdash N\!:\!U$;
- the forgotten types in the typing tree of $\Gamma \vdash M\{x := N\} : B$ are exactly those in the typing trees of $\Gamma_1, x\!:\!U \vdash M\!:\!B$ and $\Gamma_2 \vdash N\!:\!U$.

The multiset of forgotten types in the proof of $\Gamma \cap \Delta \vdash_{\mathsf{opt}} M : B$ is that in the proof of $\Gamma \vdash_{\mathsf{opt}} M'\!:\!B$ (union that in the proof of $\Delta \vdash_{\mathsf{opt}} N\!:\!A$ if $N \neq \epsilon$).

From this we can derive a strengthened completeness theorem:

Theorem 6 (Completeness of optimal typing). *If* M *is SN then there exist* A^+ *and* Γ *such that* $\Gamma \vdash_{\mathsf{opt}} M\!:\!A^+$.

This raises the question of why we did not set our theory (say, the characterisation of strongly normalising terms) with optimal typing from the start. First, the optimal property is not preserved when going into sub-terms: if $\Gamma \vdash_{\mathsf{opt}} (\lambda x.M)N : A^+$, then it is not necessarily the case that $\Gamma \vdash_{\mathsf{opt}} N : B^+$ (it could be a type B that is not of the form B^+). Second the optimal property is not preserved by arbitrary reductions, as the use of subsumption for typing abstractions is necessary for Subject Reduction to hold.

Example 1

$$\lambda x.x((\lambda y.z)x) \longrightarrow_\beta \lambda x.xz$$

However, Subject Reduction does hold for $\hookrightarrow$:

Theorem 7 (Subject Reduction, optimal case)
Assume $\Gamma \vdash_{\mathsf{opt}} M{:}B$ and $M \hookrightarrow_N M'$.
If $N \neq \epsilon$ there exist Δ and A such that $\Delta \vdash N{:}A$, otherwise let $\Delta = ()$.
There exists Γ' such that $\Gamma' \vdash_{\mathsf{opt}} M'{:}B$ with $\Gamma = \Gamma' \cap \Delta$.

Proof. By investigating the proof of Theorem 1, noticing that, in Lemma 2:

- if the typing trees of $\Gamma, x : U \vdash^{n_1} M : V$ and $\Delta \vdash^{n_2} N : U$ do not use subsumption then neither does that of $\Gamma \cap \Delta \vdash^{n_1+n_2} M\{x := N\}{:}V$;
- the forgotten types in the typing trees of $\Gamma, x{:}U \vdash^{n_1} M{:}V$ and $\Delta \vdash^{n_2} N{:}U$ are those in $\Gamma \cap \Delta \vdash^{n_1+n_2} M\{x := N\}{:}V$.

Again, the multiset of forgotten types in the proof of $\Gamma \vdash_{\mathsf{opt}} M{:}B$ is that in the proof of $\Gamma' \vdash_{\mathsf{opt}} M'{:}B$ (union that in the proof of $\Delta \vdash_{\mathsf{opt}} N{:}A$ if $N \neq \epsilon$).

Remark 9. Notice the particular case of λI, where $\beta\pi$ reductions and expansions both preserve optimal typings and multisets of forgotten types, and therefore they preserve the degree of optimal typing trees.

4.2 Principal-Optimal Typing Trees

We now introduce the notion of principal typing, and for that we first define the commutativity and associativity of the intersection rule.

Definition 11 (AC of the intersection rule). *Let $\simeq$ be the smallest congruence on typing trees containing the two equations*

$$\frac{\Gamma \vdash M{:}A \quad \Delta \vdash M{:}B}{\Gamma \cap \Delta \vdash M{:}A \cap B} \simeq \frac{\Delta \vdash M{:}B \quad \Gamma \vdash M{:}A}{\Gamma \cap \Delta \vdash M{:}A \cap B}$$

$$\frac{\dfrac{\Gamma_1 \vdash M{:}A_1 \quad \Gamma_2 \vdash M{:}A_2}{\Gamma_1 \cap \Gamma_2 \vdash M{:}A_1 \cap A_2} \quad \Gamma_3 \vdash M{:}A_3}{\Gamma_1 \cap \Gamma_2 \cap \Gamma_3 \vdash M{:}A_1 \cap A_2 \cap A_3} \simeq \frac{\Gamma_1 \vdash M{:}A_1 \quad \dfrac{\Gamma_2 \vdash M{:}A_2 \quad \Gamma_3 \vdash M{:}A_3}{\Gamma_2 \cap \Gamma_3 \vdash M{:}A_2 \cap A_3}}{\Gamma_1 \cap \Gamma_2 \cap \Gamma_3 \vdash M{:}A_1 \cap A_2 \cap A_3}$$

Definition 12 (Principal typing)

- *A substitution σ mapping the atomic types $\tau_1, \ldots, \tau_n$ to the types $F_1, \ldots, F_n$ acts on types, contexts, judgements and typing trees, so we can write $A\sigma$, $\Gamma\sigma$, $\pi\sigma$,...*
- *We write $\pi \leq \pi'$ if there exists a substitution σ such that $\pi' \simeq \pi\sigma$. (In that case if π concludes $\Gamma \vdash M : A$ then π' must conclude $\Gamma\sigma \vdash M : A\sigma$).*
- *A typing tree π concluding $\Gamma \vdash_{\mathsf{opt}} M : A^+$ is said to be* principal *if for any typing tree π' concluding $\Gamma' \vdash_{\mathsf{opt}} M : A'^+$ we have $\pi \leq \pi'$.*

Typing trees produced by the proof of the completeness theorem are principal:

Theorem 8 (Principal typing always exists). *If M is SN then there exist F, Γ and a principal typing tree π concluding $\Gamma \vdash_{\mathsf{opt}} M : F$.*

Proof. The proof follows that of Theorems 4 and 6: first by induction on the longest reduction sequence starting from M then by induction on the size of M. The novelty resides in checking principality:

- If there exists M' such that $M \hookrightarrow_N M'$, we assume another optimal typing π of M and use Theorem 7 to get an optimal typing π' for M'. By principality, π' must be an instance of the principal typing tree we have recursively constructed for M' (and similarly for N if $N \neq \epsilon$). From this we deduce that π is an instance of the one we got by Subject Expansion.
- If not then M is of one of these forms:
  - $M = \lambda x.N$ or $M = [M_1, M_2]$, in which case we call upon the induction hypothesis,
  - $M = xM_1 \ldots M_n$, in which case the induction hypothesis provide principal $\Gamma_1 \vdash_{\mathsf{opt}} M_1 : A_1^+$, ..., $\Gamma_n \vdash_{\mathsf{opt}} M_n : A_n^+$, and principality is ensured by choosing a fresh atomic type τ in the type $A_1^+ \to \cdots \to A_n^+ \to \tau$ of x.

Remark 10. The shape of an optimal typing tree is unique but it is not syntax-directed, so we cannot use this unicity to have an algorithm to directly compute the typing tree (other than “executing” the term).

One can also notice that in λI, there is a direct link between the measure read off from a principal optimal typing and its degree.

Lemma 6 (Degree of λI's normal forms). *If π is a principal typing tree of $\Gamma \vdash^n_{\mathsf{opt}} M : A$, where M is a λI-term in $\beta\pi$-normal form, then $\delta(\pi) = n$. It is also the number of applications in the term M.*

Proof. Again, by inspecting the proof of completeness: every time we type $M = xM_1 \ldots M_n$, using $\Gamma_1 \vdash_{\mathsf{opt}} M_1 : A_1^+$, ..., $\Gamma_n \vdash_{\mathsf{opt}} M_n : A_n^+$, we add as many arrows by constructing the type $A_1^+ \to \cdots \to A_n^+ \to \tau$ as we use new occurrences of rule (App).

5 Complexity

In this section we derive two complexity results, one for each fragment of our calculus: Church-Klop's λI-calculus and the pure λ-calculus.

5.1 Complexity Result for Church-Klop's λI

In this section every term is assumed to be in the λI-fragment of the calculus. Remember that in that fragment, $\hookrightarrow_{\epsilon}$ is the same as $\longrightarrow_{\beta\pi}$.

We first identify a smaller reduction relation $\xhookrightarrow{\beta\mathsf{small}}$ (within λI) that will always decrease the measure of optimal trees exactly by one.

Definition 13 (Small-reduction). *Small-reduction, written* $\xhookrightarrow{\beta\mathit{small}}$*, is defined in Fig. 3.*

$$\frac{}{(\lambda x.M)\ N\ \overrightarrow{N_i} \xhookrightarrow{\beta\mathsf{small}} M\{x:=N\}\ \overrightarrow{N_i}}$$

$$\frac{N \xhookrightarrow{\beta\mathsf{small}} N' \qquad x \notin fv(M)}{(\lambda x.[M,x])\ N\ \overrightarrow{N_i} \xhookrightarrow{\beta\mathsf{small}} (\lambda x.[M,x])\ N'\ \overrightarrow{N_i}}$$

$$\frac{N \xhookrightarrow{\beta\mathsf{small}} N'}{x\ \overrightarrow{N_i}\ N\ \overrightarrow{M_j} \xhookrightarrow{\beta\mathsf{small}} x\ \overrightarrow{N_i}\ N'\ \overrightarrow{M_j}} \qquad \frac{M \xhookrightarrow{\beta\mathsf{small}} M'}{\lambda x.M \xhookrightarrow{\beta\mathsf{small}} \lambda x.M'}$$

$$\frac{M \xhookrightarrow{\beta\mathsf{small}} M'}{[M,N] \xhookrightarrow{\beta\mathsf{small}} [M',N]} \qquad \frac{N \xhookrightarrow{\beta\mathsf{small}} N'}{[M,N] \xhookrightarrow{\beta\mathsf{small}} [M,N']}$$

Fig. 3. Small-reduction

Remark 11. If a term can be reduced by $\longrightarrow_{\beta}$ and not by $\longrightarrow_{\pi}$ then it can be reduced by $\xhookrightarrow{\beta\mathsf{small}}$[2].

Lemma 7 (Small reduction decreases the measure by 1)
If $\Gamma \vdash^{n}_{\mathsf{opt}} M\!:\!A$ *and* $M \xhookrightarrow{\beta\mathit{small}} M'$ *then* $\Gamma \vdash^{n-1}_{\mathsf{opt}} M'\!:\!A$.

Proof. The typing tree is the one produced by the proof of Subject Reduction.

Checking that that tree is also optimal with measure $n-1$ is done by induction on M (or equivalently by induction on the derivation of $M \xhookrightarrow{\beta\mathsf{small}} M'$ then by induction on n for those rules that feature a series of n applications).

[2] We could call $\xhookrightarrow{\beta\mathsf{small}}$ a strategy, but it does not necessarily determine a unique redex to reduce.

The optimal property of typing is preserved in inductive steps, i.e. while going into the term M and until the base case of $\xhookrightarrow{\beta_{\mathsf{small}}}$ is found (the first rule):

- In the first rule of $\xhookrightarrow{\beta_{\mathsf{small}}}$, notice that only one application is removed only because we are in the λI-fragment.
- In the second rule, x has a type of the form B^-, i.e. of the form $C_1^+ \to \cdots \to C_s^+ \to \tau$ (with $s \geq n$), so the typing of N is optimal. We can then use the induction hypothesis to conclude.
- In the third rule, the type of x is a forgotten type, so by the optimal property is must be of the form A^+, and by construction N has the very same type. This makes its typing optimal and we can then use the induction hypothesis to conclude.
- In the fourth rule, the typing of M is optimal if that of $\lambda x.M$ is.
- In the fifth rule, the typing of M is optimal if that of $[M, N]$ is (the two terms have the same type in the same context).
- In the sixth rule, the type of N is a forgotten type, so by optimality is has to be of the form A^+, so again the typing of N is optimal.

Theorem 9 (Complexity result)
If $\Gamma \vdash^n_{\mathsf{opt}} M : A$ with a principal typing tree of degree n' then there exists a $\beta\pi$-normal form M' such that

$$M \longrightarrow^*_{\pi} (\longrightarrow_{\beta} \ \longrightarrow^*_{\pi})^{n-n'} M'$$

This reduction sequence from M to M' is of maximal length[3].

Proof. By induction on n. We reduce by $\xhookrightarrow{\beta_{\mathsf{small}}}$ and $\longrightarrow_{\pi}$ until hitting the normal form M', also typed by $\Gamma \vdash^{n'}_{\mathsf{opt}} M' : A$ with some principal typing tree of measure n'. By Lemma 6, n' is both the number of applications in M' and the degree of its principal typing tree. That degree is not changed by expansions, so it is also the degree of the principal typing tree of M which we started with.

5.2 Complexity Result for Pure λ-Calculus

In this section we derive a similar result for the pure λ-calculus. For this we reduce the problem of pure λ to that of λI (treated above).

In order to make the distinction very clear about what terms are in pure λ and what terms are in λI, we use two different notational styles: $t, u, v, \ldots$ for pure λ-terms and $T, U, V, \ldots$ for λI-terms.

We want to exhibit in pure λ-calculus the longest reduction sequences, and show that their lengths are exactly those that can be predicted in λI.

For the longest reduction sequences we simply use the perpetual strategy from [vRSSX99], shown in Fig. 4.

[3] As Subject reduction implies that any other reduction sequence has a length less than or equal to $n - n'$.

$$\frac{x \in fv(t) \text{ or } t' \text{ is a } \beta\text{-normal form}}{(\lambda x.t)\ t'\ \overrightarrow{t_j} \rightsquigarrow t\{x := t'\}\ \overrightarrow{t_j}} \qquad \frac{t' \rightsquigarrow t'' \qquad x \notin fv(t)}{(\lambda x.t)\ t'\ \overrightarrow{t_j} \rightsquigarrow (\lambda x.t)\ t''\ \overrightarrow{t_j}}$$

$$\frac{t \rightsquigarrow t'}{x\ \overrightarrow{t_j}\ t\ \overrightarrow{p_j} \rightsquigarrow x\ \overrightarrow{t_j}\ t'\ \overrightarrow{p_j}} \qquad \frac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'}$$

Fig. 4. A perpetual reduction strategy for λ

Remark 12. $\rightsquigarrow \subseteq \longrightarrow_\beta$
If t is not a β-normal form, then there is a λ-term t' such that $t \rightsquigarrow t'$.

Although we do not need it here, it is worth mentioning that $\rightsquigarrow$ defines a perpetual strategy w.r.t. β-reduction, i.e. if M is not β-strongly normalising and $M \rightsquigarrow M'$, then neither is M' [vRSSX99]. In that sense it can be seen as the worst strategy (the least efficient). We show here that it is the worst in a stronger sense: it maximises the lengths of reduction sequences. For that we show that the length of a reduction sequence produced by that strategy matches that of the longest reduction sequence in λI. This requires encoding the syntax of the pure λ-calculus into λI, as shown in Fig. 5 (from [Len05, Len06]).

$\mathsf{i}(x)$	$:=\ x$	$\mathsf{i}(\lambda x.M)$	$:=$	$\lambda x.\mathsf{i}(M)$	if $x \in fv(M)$
$\mathsf{i}(M\ N)$	$:=\ \mathsf{i}(M)\ \mathsf{i}(N)$	$\mathsf{i}(\lambda x.M)$	$:=$	$\lambda x.[\mathsf{i}(M), x]$	if $x \notin fv(M)$

Fig. 5. Encoding from λ to λI

Lemma 8 ([Len05, Len06]). *For any λ-terms t and u,*

- $fv(\mathit{i}(t)) = fv(t)$
- $\mathit{i}(t)\{x := \mathit{i}(u)\} = \mathit{i}(t\{x := u\})$

But this encoding will not allow the simulation of $\rightsquigarrow$ by $\longrightarrow_{\beta\pi}$ in λI. To allow the simulation we need to generalise the i-encoding into a larger encoding that needs to be non-deterministic (i.e. to be a relation rather than a function).

Definition 14 (Relation between λ & λI [Len05, Len06]). *The relation $\mathcal{G}$ between λ-terms & λI-terms is given by the rules of Fig. 6 and (non-deterministically) generalises the i encoding.*

$$\frac{}{((\lambda x.t)\ t'\ \overrightarrow{t_j})\ \mathcal{G}\ \mathsf{i}((\lambda x.t)\ t'\ \overrightarrow{t_j})} \qquad \frac{t'\ \mathcal{G}\ T' \qquad x \notin fv(t)}{((\lambda x.t)\ t'\ \overrightarrow{t_j})\ \mathcal{G}\ (\mathsf{i}(\lambda x.t)\ T'\ \overrightarrow{\mathsf{i}(t_j)})}$$

$$\frac{\forall j \qquad t_j\ \mathcal{G}\ T_j}{(x\ \overrightarrow{t_j})\ \mathcal{G}\ (x\ \overrightarrow{T_j})} \qquad \frac{t\ \mathcal{G}\ T \quad x \in fv(T)}{\lambda x.t\ \mathcal{G}\ \lambda x.T} \qquad \frac{t\ \mathcal{G}\ T \qquad N \text{ is a normal form for } \beta\pi}{t\ \mathcal{G}\ [T, N]}$$

Fig. 6. Relation between λ & λI

Lemma 9 ([Len05, Len06])

1. *If t is a β-normal form and $t\ \mathcal{G}\ T$, then T is a $\beta\pi$-normal form.*
2. *For any λ-term t, $t\ \mathcal{G}\ \mathsf{i}(t)$.*

In [Len06, KL07] it is shown that the perpetual strategy can be simulated in λI through the i-encoding:

Theorem 10 (Strong simulation of $\rightsquigarrow$ in λI [Len06, KL07])
If $t\ \mathcal{G}\ T$ and $t \rightsquigarrow t'$ then there exists T' in λI such that $t'\ \mathcal{G}\ T'$ and $T {\longrightarrow^{+}}_{\beta\pi} T'$.

By inspecting the proof of the simulation in [Len06, KL07], one notices that $T {\longrightarrow^{+}}_{\beta\pi} T'$ **is in fact** $T({\longrightarrow^{*}}_{\pi} \xhookrightarrow{\beta_{\mathsf{small}}} {\longrightarrow^{*}}_{\pi})T'$, which decreases the measure read off an optimal typing tree exactly by one.

Now given Lemma 9, this means that the perpetual strategy from [vRSSX99] generates, from a given term t, a reduction sequence to its normal form t' of the same length as a reduction sequence, in λI, from $\mathsf{i}(t)$ to its normal form T' (with $t'\ \mathcal{G}\ T'$). This length can be predicted in the measure read off an optimal typing tree for $\mathsf{i}(t)$; and it so happens that it is the same as the measure read off an optimal typing tree for t:

Lemma 10 (Preservation of optimal typing by i). *Let t be a pure λ-term. If $\Gamma \vdash^{n}_{\mathsf{opt}} t : A$ then $\Gamma \vdash^{n}_{\mathsf{opt}} \mathsf{i}(t) : A$. If the typing is principal then it remains principal and the degree is not changed.*

Proof. By induction on the derivation tree we prove that if $\Gamma \vdash^{n} t : A$ with no subsumption then $\Gamma \vdash^{n} \mathsf{i}(t) : A$ with no subsumption and with the same forgotten types.

Theorem 11 (Complexity result for λ)
If $\Gamma \vdash^{n}_{\mathsf{opt}} t : A$ with a principal typing tree of degree n' then there exists a β-normal form t' such that

$$t \longrightarrow_{\beta}^{n-n'} t'$$

This reduction sequence from t to t' is of maximal length[4].

[4] As Subject reduction implies that any other reduction sequence has a length less than or equal to $n - n'$.

6 Conclusion

We have defined a typing system for non-idempotent intersection types. We have shown that it characterises strongly normalising terms in a more natural way than idempotent intersection types do. With some reasonable restrictions on the derivation tree we have obtained results on the maximum number of β-reductions in a reduction sequence of a λ-term (with Klop's extension).

We noticed *a posteriori* that our technology is similar to that which can be found in e.g. [KW99, NM04]. One of the concerns of this line of research is how the process of type inference compares to that of normalisation, in terms of complexity *classes* (these two problems being parameterised by the size of terms and a notion of *rank* for types).

The present paper shows how such a technology can actually provide an exact equality, specific to each λ-term and its typing tree, between the number read off the tree and the length of the longest reduction sequence. Of course this only emphasises the fact that type inference is as hard as normalisation, but type inference as a process is not a concern of this paper.

Our non-idempotent intersection type system and our results can be lifted to other calculi featuring e.g. explicit substitutions, combinators, or algebraic constructors and destructors (to handle integers, products, sums,...).

Idempotent intersection types have been used to provide model-based proofs of strong normalisation for well-known typing systems (simple types, system F, system F_ω,...,). Such model constructions (I-filters [CS07], orthogonality) can also be done with non-idempotent intersection types with no increased difficulty, and with the extra advantage that the strong normalisation of terms in the models is much simpler to prove. This is our next paper.

References

[Abr93] Abramsky, S.: Computational interpretations of linear logic. Theoret. Comput. Sci. 111, 3–57 (1993)

[BBdH93] Benton, N., Bierman, G., de Paiva, V., Hyland, M.: A term calculus for intuitionistic linear logic. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 75–90. Springer, Heidelberg (1993)

[BCDC83] Barendregt, H., Coppo, M., Dezani-Ciancaglini, M.: A filter lambda model and the completeness of type assignment. J. of Symbolic Logic 48(4), 931–940 (1983)

[BEM10] Bucciarelli, A., Ehrhard, T., Manzonetto, G.: Categorical models for simply typed resource calculi. ENTCS 265, 213–230 (2010)

[BM03] Baillot, P., Mogbil, V.: Soft lambda-calculus: a language for polynomial time computation. CoRR, cs.LO/0312015 (2003)

[Bou03] Boudol, G.: On strong normalization in the intersection type discipline. In: Hofmann, M. (ed.) TLCA 2003. LNCS, vol. 2701, pp. 60–74. Springer, Heidelberg (2003)

[CD78] Coppo, M., Dezani-Ciancaglini, M.: A new type assignment for lambda-terms. Archiv für mathematische Logik und Grundlagenforschung 19, 139–156 (1978)

[CD80] Coppo, M., Dezani-Ciancaglini, M.: An extension of the basic functionality theory for the λ-calculus. Notre Dame J. of Formal Logic 21(4), 685–693 (1980)

[CS07] Coquand, T., Spiwack, A.: A proof of strong normalisation using domain theory. Logic. Methods Comput. Science 3(4) (2007)

[dC05] de Carvalho, D.: Intersection types for light affine lambda calculus. ENTCS 136, 133–152 (2005)

[dC09] de Carvalho, D.: Execution time of lambda-terms via denotational semantics and intersection types. CoRR, abs/0905.4251 (2009)

[DCHM00] Dezani-Ciancaglini, M., Honsell, F., Motohama, Y.: Compositional characterizations of lambda-terms using intersection types (extended abstract). In: Nielsen, M., Rovan, B. (eds.) MFCS 2000. LNCS, vol. 1893, p. 304. Springer, Heidelberg (2000)

[DCT07] Dezani-Ciancaglini, M., Tatsuta, M.: A Behavioural Model for Klop's Calculus. In: Corradini, F., Toffalori, C. (eds.) Logic, Model and Computer Science. ENTCS, vol. 169, pp. 19–32. Elsevier, Amsterdam (2007)

[ER03] Ehrhard, T., Regnier, L.: The differential lambda-calculus. Theoret. Comput. Sci. 309(1-3), 1–41 (2003)

[Ghi96] Ghilezan, S.: Strong normalization and typability with intersection types. Notre Dame J. Formal Loigc 37(1), 44–52 (1996)

[Gir87] Girard, J.-Y.: Linear logic. Theoret. Comput. Sci. 50(1), 1–101 (1987)

[GR07] Gaboardi, M., Ronchi Della Rocca, S.: A soft type assignment system for *lambda* -calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 253–267. Springer, Heidelberg (2007)

[How80] Howard, W.A.: The formulae-as-types notion of construction. In: Seldin, J.P., Hindley, J.R. (eds.) To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, pp. 479–490. Academic Press, London (1980), Reprint of a manuscript written 1969

[KL07] Kesner, D., Lengrand, S.: Resource operators for the λ-calculus. Inform. and Comput. 205, 419–473 (2007)

[Klo80] Klop, J.-W.: Combinatory Reduction Systems, volume 127 of Mathematical Centre Tracts. CWI, PhD Thesis (1980)

[KW99] Kfoury, A.J., Wells, J.B.: Principality and decidable type inference for finite-rank intersection types. In: Proc. of the 26th Annual ACM Symp. on Principles of Programming Languages (POPL 1999), pp. 161–174. ACM Press, New York (1999)

[Laf04] Lafont, Y.: Soft linear logic and polynomial time. Theoret. Comput. Sci. 318(1-2), 163–180 (2004)

[Lei86] Leivant, D.: Typing and computational properties of lambda expressions. Theoretical Computer Science 44(1), 51–68 (1986)

[Len05] Lengrand, S.: Induction principles as the foundation of the theory of normalisation: Concepts and techniques. Technical report, PPS laboratory, Université Paris 7 (March 2005), http://hal.ccsd.cnrs.fr/ccsd-00004358

[Len06] Lengrand, S.: Normalisation & Equivalence in Proof Theory & Type Theory. PhD thesis, Univ. Paris 7 & Univ. of St Andrews (2006)

[NM04] Neergaard, P.M., Mairson, H.G.: Types, potency, and idempotency: why nonlinearity and amnesia make a type system work. In: Okasaki, C., Fisher, K. (eds.) Proc. of the ACM International Conference on Functional Programming, pp. 138–149. ACM Press, New York (September 2004)

[Sør97] Sørensen, M.H.B.: Strong normalization from weak normalization in typed lambda-calculi. Inform. and Comput. 37, 35–71 (1997)

[Val01] Valentini, S.: An elementary proof of strong normalization for intersection types. Arch. Math. Log. 40(7), 475–488 (2001)

[vB92] van Bakel, S.: Complete restrictions of the intersection type discipline. Theoret. Comput. Sci. 102(1), 135–163 (1992)

[vRSSX99] van Raamsdonk, F., Severi, P., Sørensen, M.H.B., Xi, H.: Perpetual reductions in λ-calculus. Inform. and Comput. 149(2), 173–225 (1999)

[Xi97] Xi, H.: Weak and strong beta normalisations in typed lambda-calculi. In: de Groote, P. (ed.) TLCA 1997. LNCS, vol. 1210, pp. 390–404. Springer, Heidelberg (1997)

Realizability and Parametricity in Pure Type Systems

Jean-Philippe Bernardy[1] and Marc Lasson[2]

[1] Chalmers University of Technology and University of Gothenburg
[2] ENS Lyon, Université de Lyon, LIP (UMR 5668 CNRS ENS Lyon UCBL INRIA)

Abstract. We describe a systematic method to build a logic from any programming language described as a Pure Type System (PTS). The formulas of this logic express properties about programs. We define a parametricity theory about programs and a realizability theory for the logic. The logic is expressive enough to internalize both theories. Thanks to the PTS setting, we abstract most idiosyncrasies specific to particular type theories. This confers generality to the results, and reveals parallels between parametricity and realizability.

1 Introduction

During the past decades, a recurring goal among logicians was to give a computational interpretation of the reasoning behind mathematical proofs. In this paper we adopt the converse approach: we give a systematic way to build a logic from a programming language. The structure of the programming language is replicated at the level of the logic: the expressive power of the logic (e.g. the ability of expressing conjunctions) is directly conditioned by the constructions available in the programming language (e.g. presence of products).

We use the framework of Pure Type Systems (PTS) to represent both the starting programming language and the logic obtained by our construction. A PTS [2, 3] is a generalized λ-calculus where the syntax for terms and types are unified. Many systems can be expressed as PTSs, including the simply typed λ-calculus, Girard and Reynolds polymorphic λ-calculus (System F) and its extension System Fω, Coquand's Calculus of Constructions, as well as some exotic, and even inconsistent systems such as λU [8]. PTSs can model the functional core of many modern programming languages (Haskell, Objective Caml) and proof assistants (Coq [25], Agda [19], Epigram [17]). This unified framework provides meta-theoretical such as substitution lemmas, subject reduction and uniqueness of types.

In Sec. 3, we describe a transformation which maps any PTS P to a PTS P^2. The starting PTS P will be viewed as a programming language in which live *types* and *programs* and P^2 will be viewed as a proof system in which live *proofs* and *formulas*. The logic P^2 is expressive enough to state properties about the programs. It is therefore a setting of choice to develop a parametricity and a realizability theory.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 108–122, 2011.

Parametricity. Reynolds [23] originally developed the theory of parametricity to capture the meaning of types of his polymorphic λ-calculus (equivalent to Girard's System F). Each closed type can be interpreted as a predicate that all its inhabitants satisfy. Reynolds' approach to parametricity has proven to be a successful tool: applications range from program transformations to speeding up program testing [28, 7, 4].

Parametricity theory can be adapted to other λ-calculi, and for each calculus, parametricity predicates are expressed in a corresponding logic. For example, Abadi et al. [1] remark that the simply-typed lambda calculus corresponds to LCF [18]. For System F, predicates can be expressed in second order predicate logic, in one or another variant [1, 16, 29]. More recently, Bernardy et al. [5] have shown that parametricity conditions for a reflective PTS can be expressed in the PTS itself.

Realizability. The notion of realizability was first introduced by Kleene [10] in his seminal paper. The idea of relating programs and formulas, in order to study their constructive content, was then widely used in proof theory. For example, it provides tools for proving that an axiom is not derivable in a system (excluded middle in [11, 26]) or that intuitionistic systems satisfy the *existence property*[1] [9, 26]; see Van Oosten [27] for an historical account of realizability.

Originally, Kleene represented programs as integers in a theory of recursive functions. Later, this technique has been extended to other notions of programs like combinator algebra [24, 26] or terms of Gödel's System T [12, 26] in Kreisel's modified realizability. In this article, we generalize the latter approach by using an arbitrary pure type system as the language of programs.

Krivine [13] and Leivant [15] have used realizability to prove Girard's representation theorem[2] [8] and to build a general framework for extracting programs from proofs in second-order logic [14]. In this paper, we extend Krivine's methodology to languages with dependent types, like Paulin-Mohring [20, 21] did with the realizability theory behind the program extraction in the Coq proof assistant [25].

Contributions. Viewed as syntactical notions, realizability and parametricity bear a lot of similarities. Our aim was to understand through the generality of PTSs how they are related. Our main contributions are:

- The general construction of a logic from the programming language of its realizers with syntactic definitions of parametricity and realizability (Sec. 3).
- The proof that this construction is strongly normalizing if the starting programming language is (Thm. 2).
- A characterization of both realizability in terms of parametricity (Thm. 6) and parametricity in terms of realizability (Thm. 5).

[1] If $\forall x \exists y, \varphi(x, y)$ is a theorem, then there exists a program f such that $\forall x, \varphi(x, f(x))$.

[2] Functions definable in System F are exactly those provably total in second-order arithmetic.

2 The First Level

In this section, we recall basic definitions and theorems about pure types systems (PTSs). We refer the reader to [2] for a comprehensive introduction to PTSs. A PTS is defined by a specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where $\mathcal{S}$ is a set of *sorts*, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ a set of *axioms* and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ a set of *rules*, which determines the typing of product types. The typing judgement is written $\Gamma \vdash A : B$. The notation $\Gamma \vdash A : B : C$ is a shorthand for having both $\Gamma \vdash A : B$ and $\Gamma \vdash B : C$ simultaneously.

Example 1 (System F*).* The PTS F has the following specification:

$$\mathcal{S}_{\mathrm{F}} = \{\star, \square\} \qquad \mathcal{A}_{\mathrm{F}} = \{(\star, \square)\} \qquad \mathcal{R}_{\mathrm{F}} = \{(\star, \star, \star), (\square, \star, \star)\}\,.$$

It defines the λ-calculus with polymorphic types known as system F [8]. The rule $(\star, \star, \star)$ corresponds to the formation of arrow types (usually written $\sigma \rightarrow \tau$) and the rule $(\square, \star, \star)$ corresponds to quantification over types $(\forall \alpha, \tau)$.

Even though we use F as a running example throughout the article to illustrate our general definitions our results apply to any PTS.

Sort annotations. We sometimes decorate terms with *sort annotations.* They function as a syntactic reminder of the first component of the rule used to type a product. We divide the set of variables into disjoint infinite subsets $\mathcal{V} = \bigsqcup\{\mathcal{V}_s | s \in \mathcal{S}\}$ and we write x^s to indicate that a variable x belongs to $\mathcal{V}_s$. We also annotate applications $F\,a$ with the sort of the variable of the product type of F. Using this notation, the product rule and the application rule are written

$$\underset{\textsc{Product}\ (s_1, s_2, s_3) \in \mathcal{R}}{\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma, x^{s_1} : A \vdash B : s_2}{\Gamma \vdash (\Pi x^{s_1} : A.\, B) : s_3}} \qquad \underset{\textsc{Application}}{\dfrac{\Gamma \vdash F : (\Pi x^{s} : A.\, B) \qquad \Gamma \vdash a : A}{\Gamma \vdash (F\,a)_s : B[x \mapsto a]}}\,.$$

Since sort annotations can always be recovered by using the type derivation, we do not write them in our examples.

Example 2 (System F *terms).* In System F, we adopt the following convention: the letters x, y, z, ... range over $\mathcal{V}_\star$, and α, β, γ, ... over $\mathcal{V}_\square$. For instance, the identity program $\mathrm{Id} \equiv \lambda(\alpha : \star)(x : \alpha).x$ is of type $\mathrm{Unit} \equiv \Pi\alpha : \star.\alpha \rightarrow \alpha$. The Church numeral $0 \equiv \lambda(\alpha : \star)(f : \alpha \rightarrow \alpha)(x : \alpha).x$ has type $\mathrm{Nat} \equiv \Pi\alpha : \star.(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ and the successor function on Church numerals $\mathrm{Succ} \equiv \lambda(n : \mathrm{Nat})(\alpha : \star)(f : \alpha \rightarrow \alpha)(x : \alpha).f\,(n\,\alpha\,f\,x)$ is a program of type $\mathrm{Nat} \rightarrow \mathrm{Nat}$.

3 The Second Level

In this section we describe the logic to reason about the programs and types written in an arbitrary PTS P, as well as basic results concerning the consistency of the logic. This logic is also a PTS, which we name P^2. Because we carry out most of our development in P^2, judgments refer to that system unless the symbol $\vdash$ is subscripted with the name of a specific system.

Definition 1 (second-level system). *Given a PTS $P = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, we define $P^2 = (\mathcal{S}^2, \mathcal{A}^2, \mathcal{R}^2)$ by*

$$\begin{aligned}
\mathcal{S}^2 &= \mathcal{S} \cup \{\lceil s \rceil \mid s \in \mathcal{S}\} \\
\mathcal{A}^2 &= \mathcal{A} \cup \{(\lceil s_1 \rceil, \lceil s_2 \rceil) \mid (s_1, s_2) \in \mathcal{A}\} \\
\mathcal{R}^2 &= \mathcal{R} \cup \{(\lceil s_1 \rceil, \lceil s_2 \rceil, \lceil s_3 \rceil), (s_1, \lceil s_3 \rceil, \lceil s_3 \rceil) \mid (s_1, s_2, s_3) \in \mathcal{R}\} \\
&\qquad \cup \{(s_1, \lceil s_2 \rceil, \lceil s_2 \rceil) \mid (s_1, s_2) \in \mathcal{A}\}
\end{aligned}$$

Because we see P as a programming language and P^2 as a logic for reasoning about programs in P, we adopt the following terminology and conventions. We use the metasyntactic variables $s, s_1, s_2, \ldots$ to range over sorts in $\mathcal{S}$ and $t, t_1, t_2, \ldots$ to range over sorts in $\mathcal{S}^2$. We call *type* a term inhabiting a first-level sort in some context (we write $\Gamma \vdash A : s$ for a type A), *programs* are inhabitants of types ($\Gamma \vdash A : B : s$ for a program A of type B), *formulas* denote inhabitants of a lifted sort (written $\Gamma \vdash A : \lceil s \rceil$) and *proofs* are inhabitants of formulas ($\Gamma \vdash A : B : \lceil s \rceil$). We also say that types and programs are *first-level* terms, and formulas and proofs are *second-level* terms.

If s is a sort of P, then $\lceil s \rceil$ is the sort of formulas expressing properties of types of sort s. For each rule (s_1, s_2, s_3) in $\mathcal{R}$, $(\lceil s_1 \rceil, \lceil s_2 \rceil, \lceil s_3 \rceil)$ maps constructs of the programming language at the level of the logic, and $(s_1, \lceil s_3 \rceil, \lceil s_3 \rceil)$ allows to build the quantification of programs of sort s_1 in formulas of sort $\lceil s_3 \rceil$.

For each axiom (s_1, s_2) in $\mathcal{A}$, we add the rule $(s_1, \lceil s_2 \rceil, \lceil s_2 \rceil)$ in order to build the type of predicates of sort $\lceil s_2 \rceil$ parameterized by programs of sort s_1.

Example 3. The PTS F^2 has the following specification:

$$\begin{aligned}
\mathcal{S}^2_{\mathrm{F}} &= \{ \quad \star, \square, \lceil \star \rceil, \lceil \square \rceil \quad \} \\
\mathcal{A}^2_{\mathrm{F}} &= \{ \quad (\star, \square), (\lceil \star \rceil, \lceil \square \rceil) \quad \} \\
\mathcal{R}^2_{\mathrm{F}} &= \{ \ (\star, \star, \star), (\square, \star, \star), (\lceil \star \rceil, \lceil \star \rceil, \lceil \star \rceil), (\lceil \square \rceil, \lceil \star \rceil, \lceil \star \rceil) \\
&\qquad (\star, \lceil \square \rceil, \lceil \square \rceil), (\star, \lceil \star \rceil, \lceil \star \rceil), (\square, \lceil \star \rceil, \lceil \star \rceil) \quad \}.
\end{aligned}$$

We extend our variable-naming convention to $\mathcal{V}_{\lceil \star \rceil}$ and $\mathcal{V}_{\lceil \square \rceil}$ as follows: the variables h, h_1, h_2, ... range over $\mathcal{V}_{\lceil \star \rceil}$, and the variables X, Y, Z, ... range over $\mathcal{V}_{\lceil \square \rceil}$. The logic F^2 is a second-order logic with typed individuals (Wadler [29] gives another presentation of the same system). The sort $\star$ is the type of types and the only inhabitant of $\square$, while $\lceil \star \rceil$ is the sort of propositions. $\lceil \square \rceil$ is inhabited by the type of propositions ($\lceil \star \rceil$), the type of predicates ($\tau \to \lceil \star \rceil$), and in general the type of relations ($\tau_1 \to \cdots \to \tau_n \to \lceil \star \rceil$). The rules correspond to various type of quantifications as follows:

- $(\lceil \star \rceil, \lceil \star \rceil, \lceil \star \rceil)$ allows to build implication between formulas, written $P \to Q$.
- $(\star, \lceil \star \rceil, \lceil \star \rceil)$ allows to quantify over individuals (as in $\Pi x : \tau . P$).
- $(\square, \lceil \star \rceil, \lceil \star \rceil)$ allows to quantify over types (as in $\Pi \alpha : \star . P$).
- $(\star, \lceil \square \rceil, \lceil \square \rceil)$ is used to build types of predicates depending on programs.
- $(\lceil \square \rceil, \lceil \star \rceil, \lceil \star \rceil)$ allows to quantify over predicates (as in $\Pi X : \tau_1 \to \cdots \to \tau_n \to \lceil \star \rceil . P$).

In F^2, truth can be encoded by $\top \equiv \mathit{\Pi} X : \lceil\star\rceil . X \rightarrow X$ and is proved by $\mathrm{Obvious} \equiv \lambda(X : \lceil\star\rceil)(h : X).h$. The formula $x =_\tau y \equiv \mathit{\Pi} X : \tau \rightarrow \lceil\star\rceil . X\, x \rightarrow X\, y$ define the Leibniz equality at type τ. The term $\mathrm{Refl} \equiv \lambda(\alpha : \star)(x : \alpha)(X : \alpha \rightarrow \lceil\star\rceil)(h : X\, x).h$ is a proof of the reflexivity of equality $\mathit{\Pi}(\alpha : \star)(x : \alpha).x =_\alpha x$. And the induction principle over Church numerals is a formula $N \equiv \lambda x : \mathrm{Nat}\, .\mathit{\Pi} X : \mathrm{Nat} \rightarrow \lceil\star\rceil.(\mathit{\Pi} y : \mathrm{Nat}\, .X\, y \rightarrow X\, (\mathrm{Succ}\ y)) \rightarrow X\, 0 \rightarrow X\, x$.

3.1 Structure of P^2

Programs (or types) can never refer to proofs (nor formulas). In other words, a first-level term never contains a second-level term: it is typable in P. Formally:

Theorem 1 (separation). *For $s \in \mathcal{S}$, if $\mathit{\Gamma} \vdash A : B : s$ (resp. $\mathit{\Gamma} \vdash B : s$), then there exists a sub-context $\mathit{\Gamma}'$ of $\mathit{\Gamma}$ such that $\mathit{\Gamma}' \vdash_P A : B : s$ (resp. $\mathit{\Gamma}' \vdash_P B : s$).*

Proof. By induction on the structure of terms, and relying on the generation lemma [2, 5.2.13] and on the form of the rules in $\mathcal{R}^2$: assuming $(t_1, t_2, t_3) \in \mathcal{R}^2$ then $t_3 \in \mathcal{S} \Rightarrow (t_1 \in \mathcal{S} \wedge t_2 \in \mathcal{S})$ and $t_2 \in \mathcal{S} \Rightarrow (t_1 \in \mathcal{S} \wedge t_3 \in \mathcal{S})$.

Lifting. The major part of the paper is about transformations and relations between the first and the second level. The first and simplest transformation lifts terms from the first level to the second level, by substituting occurrences of a sort s by $\lceil s \rceil$ everywhere (see Fig. 1). The function is defined only on first-level terms, and is extended to contexts in the obvious way. In addition to substituting sorts, lifting performs renaming of a variable x in $\mathcal{V}_s$ to $\mathring{x}$ in $\mathcal{V}_{\lceil s \rceil}$.

Example 4. In F^2, the lifting of inhabited types gives rise to logical tautologies. For instance, $\lceil \mathrm{Unit} \rceil = \lceil \mathit{\Pi}\alpha : \star.\alpha \rightarrow \alpha \rceil = \mathit{\Pi} X : \lceil\star\rceil.X \rightarrow X = \top$, and $\lceil \mathrm{Nat} \rceil = \mathit{\Pi} X : \lceil\star\rceil.(X \rightarrow X) \rightarrow (X \rightarrow X)$.

Lemma 1 (lifting preserves typing)

$$\mathit{\Gamma} \vdash A : B : s \Rightarrow \lceil \mathit{\Gamma} \rceil \vdash \lceil A \rceil : \lceil B \rceil : \lceil s \rceil$$

$$\begin{array}{ll}
\lceil x \rceil & = \mathring{x} \\
\lceil s \rceil & = \lceil s \rceil \\
\lceil \mathit{\Pi} x : A.\, B \rceil & = \mathit{\Pi} \mathring{x} : \lceil A \rceil .\, \lceil B \rceil \\
\lceil \lambda x : A.\, b \rceil & = \lambda \mathring{x} : \lceil A \rceil .\, \lceil b \rceil \\
\lceil A\, B \rceil & = \lceil A \rceil\, \lceil B \rceil \\
\hline
\lceil <> \rceil & = <> \\
\lceil \mathit{\Gamma}, x : A \rceil & = \lceil \mathit{\Gamma} \rceil, \mathring{x} : \lceil A \rceil
\end{array}$$

$$\begin{array}{ll}
\lfloor x^{\lceil s \rceil} \rfloor & = \dot{x}^s \\
\lfloor \lceil s \rceil \rfloor & = s \\
\lfloor \mathit{\Pi} x^s : A.B \rfloor & = \lfloor B \rfloor \\
\lfloor \mathit{\Pi} x^{\lceil s \rceil} : A.B \rfloor & = \mathit{\Pi} \dot{x}^s : \lfloor A \rfloor . \lfloor B \rfloor \\
\lfloor \lambda x^s : A.B \rfloor & = \lfloor B \rfloor \\
\lfloor \lambda x^{\lceil s \rceil} : A.B \rfloor & = \lambda \dot{x}^s : \lfloor A \rfloor . \lfloor B \rfloor \\
\lfloor (A\, B)_s \rfloor & = \lfloor A \rfloor \\
\lfloor (A\, B)_{\lceil s \rceil} \rfloor & = \lfloor A \rfloor\, \lfloor B \rfloor \\
\hline
\lfloor <> \rfloor & = <> \\
\lfloor \mathit{\Gamma}, x^s : A \rfloor & = \lfloor \mathit{\Gamma} \rfloor \\
\lfloor \mathit{\Gamma}, x^{\lceil s \rceil} : A \rfloor & = \lfloor \mathit{\Gamma} \rfloor, \dot{x}^s : \lfloor A \rfloor .
\end{array}$$

Fig. 1. lifting (left) and projection (right)

Proof. A consequence of P^2 containing a copy of P with s mapped to $\lceil s \rceil$.

Lemma 2 (lifting preserves β-reduction)

$$A \longrightarrow_\beta B \Rightarrow \lceil A \rceil \longrightarrow_\beta \lceil B \rceil$$

Proof. By induction on the structure of A.

Projection. We define a projection from second-level terms into first-level terms, which maps second-level constructs into first-level constructs. The first-level subterms are removed, as well as the interactions between the first and second levels. The reader may worry that some variable bindings are removed, potentially leaving some occurrences unbound in the body of the transformed term. However, these variables are first level, and hence their occurrences are removed too (by the application case).

The function is defined only on second-level terms, and behaves differently when facing pure second level or interaction terms. In order to distinguish these cases, the projection takes sort-annotated terms as input. Like the lifting, the projection performs renaming of each variable x in $\mathcal{V}_{\lceil s \rceil}$ to $\dot{x}$ in $\mathcal{V}_s$. We postulate that this renaming cancels that of the lifting: we have $\mathring{\dot{x}} = x$.

Example 5 (projections in F^2)

$$\lfloor \top \rfloor = \text{Unit} \quad \lfloor \text{Obvious} \rfloor = \text{Id} \quad \lfloor \Pi(\alpha : \star)(x : \alpha).x =_\alpha x \rfloor = \text{Unit} \quad \lfloor N\,t \rfloor = \text{Nat}$$

Lemma 3 (projection is the left inverse of lifting). $\lfloor \lceil A \rceil \rfloor = A$

Proof. By induction on the structure of A.

Lemma 4 (projection preserves typing)

$$\Gamma \vdash A : B : \lceil s \rceil \Rightarrow \lfloor \Gamma \rfloor \vdash \lfloor A \rfloor : \lfloor B \rfloor : s$$

Proof. By induction on the derivation $\Gamma \vdash A : B$.

In contrast to lifting, which keeps a term intact, projection may remove parts of a term, in particular abstractions at the interaction level. Therefore, β-reduction steps may be removed by projection.

Lemma 5 (projection preserves or removes β-reduction)

If $A \longrightarrow_\beta B$, then either $\lfloor A \rfloor \longrightarrow_\beta \lfloor B \rfloor$ or $\lfloor A \rfloor = \lfloor B \rfloor$.

3.2 Strong Normalization

Theorem 2 (normalization). *If P is strongly normalizing, so is P^2.*

Proof. The proof is based on the observation that, if a term A is typable in P^2 and not normalizable, then at least either:

- one of the first-level subterms of A is not normalizable, or
- the first-level term $\lfloor A \rfloor$ is not normalizable.

And yet $\lfloor A \rfloor$ and the first-level subterms are typable in P (Thm. 1) which would contradict the strong normalization of P.

3.3 Parametricity

In this section we develop Reynolds-style [23] parametricity for P, in P^2. While parametricity theory is often defined for binary relations, we abstract from the arity and develop the theory for an arbitrary arity n, though we omit the index n when the arity of relations plays no role or is obvious from the context.

The definition of parametricity is done in two parts: first we define what it means for a n-tuple of programs $\overline{z}$ to satisfy the relation generated by a type T ($\overline{z} \in [\![T]\!]_n$); then we define the translation from a program z of type T to a proof $[\![z]\!]_n$ that a tuple $\overline{z}$ satisfies the relation.

The definition below uses $n+1$ renamings: one of them ($\mathring{\cdot}$) coincides with that of lifting, and the others map x respectively to $x_1, \ldots, x_n$. The tuple $\overline{A}$ denotes n terms A_i, where A_i is the term A where each free variable x is replaced by a fresh variable x_i.

Definition 2 (parametricity)

$$\begin{array}{ll}
\overline{C} \in [\![s]\!] & = \overline{C} \to \lceil s \rceil \\
\overline{C} \in [\![\Pi x : A.\, B]\!] & = \Pi \overline{x : A}.\, \Pi \mathring{x} : \overline{x} \in [\![A]\!].\, \overline{C\, x} \in [\![B]\!] \\
\overline{C} \in [\![T]\!] & = [\![T]\!]\, \overline{C} \text{ otherwise} \\
\hline
[\![x]\!] & = \mathring{x} \\
[\![\lambda x : A.\, B]\!] & = \lambda \overline{x : A}.\, \lambda \mathring{x} : \overline{x} \in [\![A]\!].\, [\![B]\!] \\
[\![A\, B]\!] & = [\![A]\!]\, \overline{B}\, [\![B]\!] \\
[\![T]\!] & = \lambda \overline{z : T}.\, \overline{z} \in [\![T]\!] \text{ otherwise} \\
\hline
[\![<>]\!] & = <> \\
[\![\Gamma, x : A]\!] & = [\![\Gamma]\!], \overline{x : A}, \mathring{x} : \overline{x} \in [\![A]\!]
\end{array}$$

Because the syntax of values and types are unified in a PTS, each of the definitions $\cdot \in [\![\cdot]\!]$ and $[\![\cdot]\!]$ must handle all constructions. In both cases, this is done by using a catch-all case (the last line) that refers to the other part of the definition.

Remark 1. *For arity* 0*, parametricity specializes to lifting* ($[\![A]\!]_0 = \lceil A \rceil$)*.*

Example 6. For instance, in F^2, we have

$$\begin{array}{l}
(f, g) \in [\![\Pi(\alpha : \star).\alpha \to \Pi(\beta : \star).\beta \to \alpha]\!] \equiv \Pi(\alpha_1\, \alpha_2 : \star)(X : \alpha_1 \to \alpha_2 \to \lceil \star \rceil) \\
\quad (\beta_1\, \beta_2 : \star)(Y : \beta_1 \to \beta_2 \to \lceil \star \rceil)(x_1 : \alpha_1)(x_2 : \alpha_2).X\, x_1\, x_2 \to \\
\qquad \Pi(y_1 : \beta_1)(y_2 : \beta_2).Y\, y_1\, y_2 \to X\, (f\, \alpha_1\, \beta_1\, x_1\, y_1)\, (g\, \alpha_2\, \beta_2\, x_2\, y_2).
\end{array}$$

Theorem 3 (abstraction). *If* $\Gamma \vdash A : B : s$*, then* $[\![\Gamma]\!] \vdash [\![A]\!] : (\overline{A} \in [\![B]\!]) : \lceil s \rceil$

Proof. The result is a consequence of the following lemmas which are proved by simultaneous induction on the typing derivation:

- $A \longrightarrow_\beta B \Rightarrow [\![A]\!] \longrightarrow^*_\beta [\![B]\!]$
- $\Gamma \vdash A : B \Rightarrow [\![\Gamma]\!] \vdash \overline{A : B}$
- $\Gamma \vdash B : s \Rightarrow [\![\Gamma]\!], \overline{z : B} \vdash \overline{z} \in [\![B]\!] : \lceil s \rceil$
- $\Gamma \vdash A : B : s \Rightarrow [\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \in [\![B]\!]$

A direct reading of the above result is as a typing judgement about translated terms (as for lemmas 1 and 4): if A has type B, then $[\![A]\!]$ has type $\overline{A} \in [\![B]\!]$. However, it can also be understood as an abstraction theorem for system P: if a program A has type B in Γ, then various interpretations of A ($\overline{A}$) in related environments ($[\![\Gamma]\!]$) are related, by the formula $\overline{A} \in [\![B]\!]$.

The system P^2 is a natural setting to express parametricity conditions for P. Indeed, the interaction rules of the form $(s, \lceil s' \rceil, \lceil s' \rceil)$ coming from axioms in P are needed to make the sort case valid; and the interaction rules $(s_1, \lceil s_3 \rceil, \lceil s_3 \rceil)$ are needed for the quantification over individuals in the product case.

3.4 Realizability

We develop here a Krivine-style [13] internalized realizability theory. Realizability bears similarities both to the projection and the parametricity transformations defined above.

Definition 3 (realizability)

$$\begin{array}{ll}
C \Vdash \lceil s \rceil & = C \to \lceil s \rceil \\
C \Vdash \Pi x^s : A.B & = \Pi x^s : A.C \Vdash B \\
C \Vdash \Pi x^{\lceil s \rceil} : A.B & = \Pi(\dot{x}^s : \lfloor A \rfloor)(x^{\lceil s \rceil} : \dot{x} \Vdash A).(C\,\dot{x}) \Vdash B \\
C \Vdash F & = \langle F \rangle\, C \text{ otherwise} \\
\hline
\langle x^{\lceil s \rceil} \rangle & = x^{\lceil s \rceil} \\
\langle \lambda x^s : A.B \rangle & = \lambda x^s : A.\langle B \rangle \\
\langle \lambda x^{\lceil s \rceil} : A.B \rangle & = \lambda(\dot{x}^s : \lfloor A \rfloor)(x^{\lceil s \rceil} : \dot{x} \Vdash A).\langle B \rangle \\
\langle (A\,B)_s \rangle & = (\langle A \rangle\, B)_s \\
\langle (A\,B)_{\lceil s \rceil} \rangle & = ((\langle A \rangle\, \lfloor B \rfloor)_s\, \langle B \rangle)_{\lceil s \rceil} \\
\langle T \rangle & = \lambda z^s : \lfloor T \rfloor .\, z \Vdash T \text{ otherwise} \\
\hline
\langle \Gamma, x^s : A \rangle & = \langle \Gamma \rangle, x^s : A \\
\langle \Gamma, x^{\lceil s \rceil} : A \rangle & = \langle \Gamma \rangle, \dot{x}^s : \lfloor A \rfloor, x^{\lceil s \rceil} : \dot{x} \Vdash A
\end{array}$$

Like the projection, the realizability transformation is applied on second-level constructs, and behaves differently depending on whether it treats interaction constructs or pure second-level ones. It is also similar to parametricity, as it is defined in two parts. In the first part we define what it means for a program C to realize a formula F ($C \Vdash F$); then we define the translation from a proof p to a proof $\langle p \rangle$ that the program $\lfloor p \rfloor$ satisfies the realizability predicate.

Theorem 4 (adequacy). *If* $\Gamma \vdash A : B : \lceil s \rceil$*, then* $\langle \Gamma \rangle \vdash \langle A \rangle : \lfloor A \rfloor \Vdash B : \lceil s \rceil$

Proof (idea). Similar in structure to the proof of the abstraction theorem.

Example 7. In F^2, the formula $y \Vdash N\,x$ unfolds to

$$\Pi(\alpha : \star)(X : \mathrm{Nat} \to \alpha \to \star)(f : \alpha \to \alpha).$$

$$(\Pi(n : \mathrm{Nat})(y : \alpha).X\,n\,y \to X\,(\mathrm{Succ}\ n)\,(f\,y)) \to \Pi(z : \alpha).X\,0\,y \to X\,x\,(y\,\alpha\,f\,z)$$

In F^2 this formula may be used to prove a representation theorem. We can prove that $\Sigma \vdash \Pi x\, y : \mathrm{Nat}\, .y \Vdash N\, x \Leftrightarrow x =_{\mathrm{Nat}} y \wedge N\, x$ where Σ is a set of extensionality axioms ($\wedge$ and $\Leftrightarrow$ are defined by usual second-order encodings). Let π be a proof of $\Pi x : \mathrm{Nat}\, .N\, x \rightarrow N\,(f\, x)$ then $\vdash \lfloor \pi \rfloor : \mathrm{Nat} \rightarrow \mathrm{Nat}$ and $\vdash \langle \pi \rangle : \lfloor \pi \rfloor \Vdash \Pi x : \mathrm{Nat}\, .N\, x \rightarrow N\,(f\, x)$ which unfold to $\vdash \langle \pi \rangle : \Pi x\, y : \mathrm{Nat}\, .y \Vdash Nx \rightarrow \lfloor \pi \rfloor y \Vdash N(fx)$. Let m be a term in closed normal form such that $\vdash m : \mathrm{Nat}$, we can prove $N\, m$ and therefore $m \Vdash N\, m$. We now have a proof (under Σ) that $\lfloor \pi \rfloor\, m \Vdash N\,(f\, m)$ and we conclude that $\lfloor \pi \rfloor\, m =_{\mathrm{Nat}} f\, m$. We have proved that the projection of any proof of $\Pi x : \mathrm{Nat}\, .N\, x \rightarrow N\,(f\, x)$ can be proved extensionally equal to f. See [29, 13, 15] for more details.

4 The Third Level

By casting both parametricity and realizability in the mold of PTSs, we are able to discern the connections between them. The connections already surface in the previous sections: the definitions of parametricity and realizability bear some resemblance, and the adequacy and abstraction theorems appear suspiciously similar. In this section we precisely spell out the connection: realizability and parametricity can be defined in terms of each other.

Theorem 5 (realizability increases arity of parametricity). *For any tuple terms $(B, \overline{C})$,*

$$\big(B, \overline{C}\big) \in [\![A]\!]_{n+1} = B \Vdash \big(\overline{C} \in [\![A]\!]_n\big) \qquad \textit{and} \qquad [\![A]\!]_{n+1} = \langle [\![A]\!]_n \rangle.$$

Proof. By induction on the structure of A.

As a corollary, n-ary parametricity is the composition of lifting and n realizability steps:

Corollary 1 (from realizability to parametricity)

$$\overline{C} \in [\![A]\!]_n = C_1 \Vdash C_2 \Vdash \cdots \Vdash C_n \Vdash \lceil A \rceil \qquad \textit{and} \qquad [\![A]\!]_n = \langle \cdots \langle \lceil A \rceil \rangle \cdots \rangle$$

(assuming right-associativity of $\Vdash$).

Proof. By induction on n. The base case uses $[\![A]\!]_0 = \lceil A \rceil$.

One may also wonder about the converse: is it possible to define realizability in terms of parametricity? We can answer by the affirmative, but we need a bigger system to do so. Indeed, we need to extend $[\![\cdot]\!]$ to work on second-level terms, and that is possible only if a third level is present in the system. To do so, we can iterate the construction used in Sec. 3 to build a logic for an arbitrary PTS.

Definition 4 (third-level system). *Given a PTS $P = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, we define $P^3 = \left(P^2\right)^2$, where the sort-lifting $\lceil \cdot \rceil$ used by both instances of the $\cdot^2$ transformation are the same.*

Remark 2. *Because the sort-lifting used by both instances of the* $\cdot^2$ *transformation are the same,* P^3 *contains only three copies of* P *(not four). In fact* $P^3 = (\mathcal{S}^3, \mathcal{A}^3, \mathcal{R}^3)$*, where*

$$\begin{aligned}
\mathcal{S}^3 &= \mathcal{S} \ \cup \lceil\mathcal{S}\rceil \cup \lceil\lceil\mathcal{S}\rceil\rceil \\
\mathcal{A}^3 &= \mathcal{A} \ \cup \lceil\mathcal{A}\rceil \cup \lceil\lceil\mathcal{A}\rceil\rceil \\
\mathcal{R}^3 &= \mathcal{R} \ \cup \lceil\mathcal{R}\rceil \cup \lceil\lceil\mathcal{R}\rceil\rceil \\
&\quad \cup \{(s_1, \lceil s_3\rceil, \lceil s_3\rceil), (\lceil s_1\rceil, \lceil\lceil s_3\rceil\rceil, \lceil\lceil s_3\rceil\rceil) \mid (s_1, s_2, s_3) \in \mathcal{R}\} \\
&\quad \cup \{(s_1, \lceil s_2\rceil, \lceil s_2\rceil), (\lceil s_1\rceil, \lceil\lceil s_2\rceil\rceil, \lceil\lceil s_2\rceil\rceil) \mid (s_1, s_2) \in \mathcal{A}\}
\end{aligned}$$

The $[\![\cdot]\!]$ transformation is extended second-level constructs in P^2, mapping them to third-level ones in P^3. The $\lfloor\cdot\rfloor$ transformation is be similarly extended, to map the third level constructs to the second level, in addition of mapping the second to the first one (only the first level is removed).

Given these extensions, we obtain that realizability is the composition of parametricity and projection.

Lemma 6. *If* A *is a first-level term, then*

$$A = \lfloor C \in [\![A]\!]_1 \rfloor \qquad \textit{and} \qquad A = \lfloor [\![A]\!]_1 \rfloor$$

Proof. By induction on the structure of A, using separation (Thm. 1).

Theorem 6 (from parametricity to realizability). *If* A *is a second-level term, then*

$$C \Vdash A = \lfloor \lceil C\rceil \in [\![A]\!]_1 \rfloor \qquad \textit{and} \qquad \langle A\rangle = \lfloor [\![A]\!]_1 \rfloor$$

Proof. By induction on the structure of A, using the above lemma.

5 Extensions

5.1 Inductive Definitions

Even though our development assumes pure type systems, with only axioms of the form (s_1, s_2), the theory easily accommodates the addition of inductive definitions.

For parametricity, the way to extend the theory is exposed by Bernardy et al. [5]. In brief: if for every inductive definition in the programming language there is a corresponding inductive definition in the logic, then the abstraction theorem holds. For instance, to the indexed inductive definition I corresponds $[\![I]\!]$, as defined below. (We write only one constructor c_p for concision, but the result applies to any number of constructors).

$$\begin{array}{l}
\textsf{data}\ I : \mathit{\Pi}(x_1 : A_1) \cdots (x_n : A_n).s\ \textsf{where} \\
\quad c_p : \mathit{\Pi}(x_1 : B_{p,1}) \cdots (x_{n_1} : B_{p,n_1}).I\, a_{p,1} \cdots a_{p,n}
\end{array}$$

$$\begin{array}{l}
\textsf{data}\ [\![I]\!] : \overline{I} \in [\![\mathit{\Pi}(x_1 : A_1) \cdots (x_n : A_n).s]\!]\ \textsf{where} \\
\quad [\![c_p]\!] : \overline{c_p} \in [\![\mathit{\Pi}(x_1 : B_{p,1}) \cdots (x_{n_1} : B_{p,n_1}).I\, a_{p,1} \cdots a_{p,n}]\!]
\end{array}$$

The result can be transported to realizability by following the correspondence developed in the previous section. By taking the composition of $[\![\cdot]\!]$ and $\lfloor\cdot\rfloor$ for the definition of realizability, and knowing how to extend $[\![\cdot]\!]$ to inductive types, it suffices to extend $\lfloor\cdot\rfloor$ as well (respecting typing: Lem. 4). The corresponding extension to realizability is compatible with the definition for a pure system (by Thm. 6). Adequacy is proved by the composition of abstraction and Lem. 4. The definition of $\lfloor\cdot\rfloor$ is straightforward: each component of the definition must be transformed by $\lfloor\cdot\rfloor$. That is, for any inductive definition in the logic, there must be another inductive definition in the programming language that realizes it. For instance, given the definition I given below, one must also have $\lfloor I\rfloor$. $\langle I\rangle$ is then given by $\langle I\rangle = \lfloor[\![I]\!]\rfloor$, but can also be expanded as below.

$$\begin{array}{l}\textsf{\textbf{data}}\ I : \Pi(x_1 : A_1)\cdots(x_n : A_n).\lceil s\rceil\ \textsf{\textbf{where}}\\ \quad c_p : \Pi(x_1 : B_{p,1})\cdots(x_{n_1} : B_{p,n_1}).I\,a_{p,1}\cdots a_{p,n}\end{array}$$

$$\begin{array}{l}\textsf{\textbf{data}}\ \lfloor I\rfloor : \lfloor\Pi(x_1 : A_1)\cdots(x_n : A_n).\lceil s\rceil\rfloor\ \textsf{\textbf{where}}\\ \quad \lfloor c_p\rfloor : \lfloor\Pi(x_1 : B_{p,1})\cdots(x_{n_1} : B_{p,n_1}).I\,a_{p,1}\cdots a_{p,n}\rfloor\end{array}$$

$$\begin{array}{l}\textsf{\textbf{data}}\ \langle I\rangle : \lfloor I\rfloor \Vdash (\Pi(x_1 : A_1)\cdots(x_n : A_n).\lceil s\rceil)\ \textsf{\textbf{where}}\\ \quad \langle c_p\rangle : \lfloor c_p\rfloor \Vdash (\Pi(x_1 : B_{p,1})\cdots(x_{n_1} : B_{p,n_1}).I\,a_{p,1}\cdots a_{p,n})\end{array}$$

We can use inductive types to encode usual logical connectives, and derive realizability for them.

Example 8 (conjunction). The encoding of conjunction in a sort $\lceil s\rceil$ is as follows:

$$\begin{array}{l}\textsf{\textbf{data}}\ \_\wedge\_ : \lceil s\rceil \to \lceil s\rceil \to \lceil s\rceil\ \textsf{\textbf{where}}\\ \quad \textsf{conj} : \Pi\,P\,Q : \lceil s\rceil.P \to Q \to P\wedge Q\end{array}$$

If we apply the projection operator to the conjunction we obtain the type of its realizers: the cartesian product in s.

$$\begin{array}{l}\textsf{\textbf{data}}\ \_\times\_ : s \to s \to s\ \textsf{\textbf{where}}\\ \quad (\_,\_) : \Pi\,\alpha\,\beta : s.\alpha \to \beta \to \alpha\times\beta\end{array}$$

Now we can apply our realizability construction to obtain a predicate telling what it means to realize a conjunction.

$$\begin{array}{l}\textsf{\textbf{data}}\ \langle\wedge\rangle : \Pi(\alpha : s).(\alpha \to \lceil s\rceil) \to\\ \qquad\qquad \Pi(\beta : s).(\beta \to \lceil s\rceil) \to\\ \qquad\qquad \alpha\times\beta \to s\ \textsf{\textbf{where}}\\ \quad \langle\textsf{conj}\rangle : \Pi(\alpha : s)(P : \alpha \to \lceil s\rceil)\\ \qquad\qquad (\beta : s)(Q : \beta \to \lceil s\rceil)(x : \alpha)(y : \beta).\\ \qquad\qquad P\,x \to Q\,y \to \langle\wedge\rangle\,\alpha\,P\,\beta\,Q\,(x,y)\end{array}$$

By definition, $t \Vdash P\wedge Q$ means $\langle\wedge\rangle\,\lfloor P\rfloor\,\langle P\rangle\,\lfloor Q\rfloor\,\langle Q\rangle\,t$. We have

$$t \Vdash P\wedge Q \Leftrightarrow (\pi_1\,t) \Vdash P \wedge (\pi_2\,t) \Vdash Q$$

where π_1 and π_2 are projections upon Cartesian product.

We could build the realizers of other logical constructs in the same way: we would obtain a sum-type for the disjunction, an empty type for falsity, and a box type for the existential quantifier. All the following properties (corresponding to the usual definition of the realizability predicate) would then be satisfied:

- $t \Vdash P \vee Q \Leftrightarrow \textsf{\textbf{case}}\, t\, \textsf{\textbf{with}}\, \iota_1\, x \to x \Vdash P \,|\, \iota_2\, x \to x \Vdash Q$.
- $t \Vdash \bot \Leftrightarrow \bot$ and $t \Vdash \neg P \Leftrightarrow \Pi(x : \lfloor P \rfloor).\neg(x \Vdash P)$
- $t \Vdash \exists x : A.P \Leftrightarrow \exists x : A.(\mathit{unbox}\, t) \Vdash P$

where **case**... **with**... is the destruction of the sum type, and *unbox* is the destructor of the box type.

5.2 Program Extraction and Computational Irrelevance

An application of the theory developed so far is the extraction of programs from proofs. Indeed, an implication of the adequacy theorem is that the program $\lfloor A \rfloor$, obtained by projection of a proof A of a formula B, corresponds to an implementation of B, viewed as a specification. One says that $\lfloor \cdot \rfloor$ implements program extraction.

For example, applying extraction to an expression involving vectors ($Vec : (A : \lceil \star \rceil) \to Nat \to \lceil \star \rceil$) yields a program over lists. This means that programs can be justified in the rich system P^2, and realized in the simple system P. Practical benefits include a reduction in memory usage: Brady et al. [6] measure an 80% reduction using a technique with similar goals.

While P^2 is already much more expressive than P, it is possible to further increase the expressive power of the system, while retaining the adequacy theorem, by allowing quantification of first-level terms by second-level terms.

Definition 5 **($P^{2'}$).** *Let $P = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, we define $P^{2'} = (\mathcal{S}^{2'}, \mathcal{A}^{2'}, \mathcal{R}^{2'})$*

$$\begin{aligned}
\mathcal{S}^{2'} &= \mathcal{S} \cup \{\lceil s \rceil \mid s \in \mathcal{S}\} \\
\mathcal{A}^{2'} &= \mathcal{A} \cup \{(\lceil s_1 \rceil, \lceil s_2 \rceil) \mid (s_1, s_2) \in \mathcal{A}\} \\
\mathcal{R}^{2'} &= \mathcal{R} \cup \{(\lceil s_1 \rceil, \lceil s_2 \rceil, \lceil s_3 \rceil), (s_1, \lceil s_3 \rceil, \lceil s_3 \rceil), (\lceil s_1 \rceil, s_3, s_3) \mid (s_1, s_2, s_3) \in \mathcal{R}\} \\
&\quad\ \cup \{(s_1, \lceil s_2 \rceil, \lceil s_2 \rceil), (\lceil s_1 \rceil, s_2, s_2) \mid (s_1, s_2) \in \mathcal{A}\}
\end{aligned}$$

The result is a symmetric system, with two copies of P. Within either side of the system, one can reason about terms belonging to the other side. Furthermore, either side has a computational interpretation where the terms of the other side are irrelevant. For the second level, this interpretation is given by $\lfloor \cdot \rfloor$.

Even though there is no separation between first and second level in $P^{2'}$, adequacy is preserved: the addition of rules of the form $(\lceil s_1 \rceil, s_2, s_3)$ only adds first level terms, which are removed by projection.

6 Related Work and Conclusion

Our work is based on Krivine-style realizability [13] and Reynolds-style parametricity [23], which have both spawned large bodies of work.

Logics for parametricity. Study of parametricity is typically semantic, including the seminal work of Reynolds [23]. There, the concern is to capture the polymorphic character of λ-calculi (typically System F) in a model.

Mairson [16] pioneered a different angle of study, where the expressions of the programming language are (syntactically) translated to formulas describing the program. That style has then been picked by various authors before us, including Abadi et al. [1], Plotkin and Abadi [22], Bernardy et al. [5].

Plotkin and Abadi [22] introduce a logic for parametricity, similar to F^2, but with several additions. The most important addition is that of a parametricity axiom. This addition allows to prove the initiality of Church-style encoding of types.

Wadler [29] defines essentially the same concepts as us, but in the special case of System F. He points out that realizability transforms unary parametricity into binary parametricity, but does not generalize to arbitrary arity. We find the $n = 0$ case particularly interesting, as it shows that parametricity can be constructed purely in terms of realizability and a trivial lifting to the second level. We additionally show that realizability can be obtained by composing realizability and projection, while Wadler only defines the realizability transformation as a separate construct.

The parametricity transformation and the abstraction theorem that we expose here are a modified version of [5]. The added benefits of the present version is that we handle finite PTSs, and we allow the target system to be different from the source. The possible separation of source and targets is already implicit in that paper though. The way we handle finite PTSs is by separating the treatment of types and programs.

Realizability. Our realizability construction can be understood as an extension of the work of Paulin-Mohring [20], providing a realizability interpretation for a variant of the Calculus of Construction. Paulin-Mohring [20] splits CC in two levels; one where $\star$ becomes *Prop* and one where it becomes *Spec*. Perhaps counter-intuitively, *Prop* lies in what we call the first level; and *Spec* lies in the second level. Indeed, *Prop* is removed from the realizers. The system is symmetric, as the one we expose in Sec. 5.2, in the sense that there is both a rule $(\mathit{Spec}, \mathit{Prop}, \mathit{Prop})$ and $(\mathit{Prop}, \mathit{Spec}, \mathit{Spec})$. In order to see that Paulin-Mohring's construction as a special case of ours, it is necessary to recognize a number of small differences:

1. The sort *Spec* is transformed into *Prop* in the realizability transformation, whereas we would keep *Spec*.
2. The sorts of the original system use a different set of names (*Data* and *Order*). Therefore the sort *Spec* is transformed into *Data* in the projection, whereas we would use *Prop*.
3. The types of *Spec* and *Prop* inhabit the same sort, namely *Type*.
4. There is elimination from *Spec* to *Prop*, breaking the computational irrelevance in that direction.

The first two differences are essentially renamings, and thus unimportant.

Connections. We are unaware of previous work showing the connection between realizability and parametricity, at least as clearly as we do. Wadler [29] comes close, giving a version of Thm. 5 specialized to System F, but not its converse, Thm. 6. Mairson [16] mentions that his work on parametricity is directly inspired by that of Leivant [15] on realizability, but does not formalize the parallels.

Conclusion. We have given an account of parametricity and realizability in the framework of PTSs. The result is very concise: the definitions occupy only a dozen of lines. By recognizing the parallels between the two, we are able to further shrink the number of primitive concepts.

Our work points the way towards the transportation of every parametricity theory into a corresponding realizability theory, and *vice versa*.

Acknowledgments. Thanks to Andreas Abel, Thorsten Altenkirch, Thierry Coquand, Peter Dybjer and Guilhem Moulin for helpful comments and discussions.

References

[1] Abadi, M., Cardelli, L., Curien, P.: Formal parametric polymorphism. In: Proc. of POPL 1993, pp. 157–170. ACM, New York (1993)
[2] Barendregt, H.P.: Lambda calculi with types. In: Handbook of Logic in Computer Science, vol. 2, pp. 117–309 (1992)
[3] Berardi, S.: Type Dependence and Constructive Mathematics. PhD thesis, Dipartimento di Informatica, Torino (1989)
[4] Bernardy, J.-P., Jansson, P., Claessen, K.: Testing polymorphic properties. In: Gordon, A. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 125–144. Springer, Heidelberg (2010)
[5] Bernardy, J.-P., Jansson, P., Paterson, R.: Parametricity and dependent types. In: Proc. of ICFP 2010, pp. 345–356. ACM, New York (2010)
[6] Brady, E., McBride, C., McKinna, J.: Inductive families need not store their indices. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 115–129. Springer, Heidelberg (2004)
[7] Gill, A., Launchbury, J., Peyton Jones, S.: A short cut to deforestation. In: Proc. of FPCA, pp. 223–232. ACM, New York (1993)
[8] Girard, J.-Y.: Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur. Thése d'état, Université de Paris 7 (1972)
[9] Harrop, R.: On disjunctions and existential statements in intuitionistic systems of logic. Mathematische Annalen 132(4), 347–361 (1956)
[10] Kleene, S.C.: On the interpretation of intuitionistic number theory. J. of Symbolic Logic 10(4), 109–124 (1945)
[11] Kleene, S.C.: Introduction to metamathematics. Wolters-Noordhoff (1971)
[12] Kreisel, G.: Interpretation of analysis by means of constructive functionals of finite types. In: Heyting, A. (ed.) Constructivity in mathematics, pp. 101–128 (1959)
[13] Krivine, J.-L.: Lambda-calcul types et modèles. Masson (1990)
[14] Krivine, J.-L., Parigot, M.: Programming with proofs. J. Inf. Process. Cybern. 26(3), 149–167 (1990)
[15] Leivant, D.: Contracting proofs to programs. Logic and Comp. Sci., pp. 279–327 (1990)
[16] Mairson, H.: Outline of a proof theory of parametricity. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 313–327. Springer, Heidelberg (1991)

[17] McBride, C., McKinna, J.: The view from the left. J. Funct. Program. 14(01), 69–111 (2004)

[18] Milner, R.: Logic for Computable Functions: description of a machine implementation. Artificial Intelligence (1972)

[19] Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers Tekniska Högskola (2007)

[20] Paulin-Mohring, C.: Extracting Fω's programs from proofs in the calculus of constructions. In: POPL 1989, pp. 89–104. ACM, New York (1989)

[21] Paulin-Mohring, C.: Extraction de programmes dans le Calcul des Constructions. PhD thesis, Université Paris 7 (1989)

[22] Plotkin, G., Abadi, M.: A logic for parametric polymorphism. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 361–375. Springer, Heidelberg (1993)

[23] Reynolds, J.C.: Types, abstraction and parametric polymorphism. Information processing 83(1), 513–523 (1983)

[24] Staples, J.: Combinator realizability of constructive finite type analysis. Cambridge Summer School in Mathematical Logic, pp. 253–273 (1973)

[25] The Coq development team. The Coq proof assistant (2010)

[26] Troelstra, A.: Realizability. In: Handbook of proof theory. Elsevier, Amsterdam (1998)

[27] Van Oosten, J.: Realizability: a historical essay. Mathematical Structures in Comp. Sci. 12(03), 239–263 (2002)

[28] Wadler, P.: Theorems for free. In: Proc. of FPCA 1989, pp. 347–359. ACM, New York (1989)

[29] Wadler, P.: The Girard–Reynolds isomorphism. Theor. Comp. Sci. 375(1–3), 201–226 (2007)

Sound Bisimulations for Higher-Order Distributed Process Calculus[*]

Adrien Piérard and Eijiro Sumii[**]

Tohoku University
{adrien,sumii}@kb.ecei.tohoku.ac.jp

Abstract. While distributed systems with transfer of processes have become pervasive, methods for reasoning about their behaviour are underdeveloped. In this paper we propose a bisimulation technique for proving behavioural equivalence of such systems modelled in the *higher-order π-calculus with passivation* (and restriction). Previous research for this calculus is limited to context bisimulations and normal bisimulations which are either impractical or unsound. In contrast, we provide a sound and useful definition of *environmental bisimulations*, with several non-trivial examples. Technically, a central point in our bisimulations is the clause for parallel composition, which must account for passivation of the spawned processes in the middle of their execution.

1 Introduction

1.1 Background

Higher-order distributed systems are ubiquitous in today's computing environment. To name but a few examples, companies like Dell and Hewlett-Packard sell products using virtual machine live migration [14,3], and Gmail users execute remote JavaScript code on local browsers. In this paper we call *higher-order* the ability to transfer processes, and *distribution* the possibility of location-dependent system behaviour. In spite of the *de facto* importance of such systems, they are hard to analyse because of their inherent complexity.

The π-calculus [8] and its dialects prevail as models of concurrency, and several variations of these calculi have been designed for distribution. First-order variations include the ambient calculus [1] and Dπ [2], while higher-order include more recent Homer [4] and Kell [15] calculi. In this paper, we focus on the higher-order π-calculus with *passivation* [7], a simple high-level construct to express distribution. It is an extension of the higher-order π-calculus [9] (with which the reader is assumed to be familiar) with *located processes* $a[P]$ and two additional transition rules: $a[P] \xrightarrow{\overline{a}\langle P\rangle} 0$ (PASSIV), and $a[P] \xrightarrow{\alpha} a[P']$ if $P \xrightarrow{\alpha} P'$ (TRANSP).

[*] Appendix with full proofs at http://www.kb.ecei.tohoku.ac.jp/~adrien/pubs/SoundAppendix.pdf

[**] This research is partially supported by KAKENHI 22300005, the Nakajima Foundation, and the Casio Science Promotion Foundation. The first author is partially supported by the Global COE Program CERIES.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 123–137, 2011.

The new syntax $a[P]$ reads as "process P located at a" where a is a name. Rule TRANSP specifies the transparency of locations, i.e. that a location has no impact on the transitions of the located process. Rule PASSIV indicates that a located process can be *passivated*, that is, be output to a channel of the same name as the location. Using passivation, various characteristics of distributed systems are expressible. For instance, failure of process P located at a can be modelled like $a[P] \mid a(X).\overline{fail} \rightarrow 0 \mid \overline{fail}$, and migration of process Q from location b to c like $b[P] \mid b(X).c[X] \rightarrow 0 \mid c[P]$.

One way to analyse the behaviour of systems is to compare implementations and specifications. Such comparison calls for satisfying notions of behavioural equivalence, such as *reduction-closed barbed equivalence* (and *congruence*) [5], written $\approx$ (and $\approx_c$ respectively) in this paper.

Unfortunately, these equivalences have succinct definitions that are not very practical as a proof technique, for they both include a condition that quantifies over arbitrary processes, like: if $P \approx Q$ then $\forall R.\ P \mid R \approx Q \mid R$. Therefore, more convenient definitions like *bisimulations*, for which membership implies behavioural equivalence, and which come with a co-inductive proof method, are sought after.

Still, the combination of both higher order and distribution has long been considered difficult. Recent research on higher-order process calculi led to defining sound *context bisimulations* [10] (often at the cost of appealing to Howe's method [6] for proving congruence) but those bisimulations suffer from their heavy use of universal quantification: suppose that $\nu\widetilde{c}.\overline{a}\langle M\rangle.P \;\mathcal{X}\; \nu\widetilde{d}.\overline{a}\langle N\rangle.Q$, where $\mathcal{X}$ is a context bisimulation; then it is roughly required that for any process R, we have $\nu\widetilde{c}.(P \mid R\{M/X\}) \;\mathcal{X}\; \nu\widetilde{d}.(Q \mid R\{N/X\})$. Not only must we consider the outputs M and N, but we must also handle interactions of arbitrary R with the continuation processes P and Q. Alas, this almost comes down to showing reduction-closed barbed equivalence! In the higher-order π-calculus, by means of encoding into a first-order calculus, normal bisimulations [10] coincide with (and are a practical alternative to) context bisimulations. Unfortunately, normal bisimulations have proved to be unsound in the presence of passivation (and restriction) [7]. While this result cast a doubt on whether sound normal bisimulations exist for higher-order distributed calculi, it did not affect the potential of environmental bisimulations [16,17,12,13] as a useful proof technique for behavioural equivalence in those calculi.

1.2 Our Contribution

To the best of our knowledge, there are not yet any useful sound bisimulations for higher-order distributed process calculi. In this paper we develop environmental (weak) bisimulations for the higher-order π-calculus with passivation, which (1) are sound with respect to reduction-closed barbed equivalence, (2) can actually be used to prove behavioural equivalence of non-trivial processes (with restrictions), and (3) can also be used to prove reduction-closed barbed *congruence* of processes (see Corollary 1). To prove reduction-closed barbed equivalence (and congruence), we find a new clause to guarantee preservation of bisimilarity by parallel composition of arbitrary processes. Unlike the corresponding clause in previous research [7,13], it can also handle the later removal (i.e. passivation) of these processes while keeping the bisimulation proofs tractable. Several examples are given, thereby supporting our claim of the first useful bisimulations for a higher-order distributed process calculus. Moreover, we define an up-to context variant of the environmental bisimulations that significantly lightens the burden of equivalence proofs, as utilised in the examples.

Overview of the bisimulation: We now outline the definition of our environmental bisimulations. (Generalities on environmental bisimulations can be found in [12].) We define an environmental bisimulation $\mathcal{X}$ as a set of quadruples $(r, \mathcal{E}, P, Q)$ where r is a set of names (i.e. channels and locations), $\mathcal{E}$ is a binary relation (called the *environment*) on terms, and P, Q are processes. The bisimulation is a game where the processes P and Q are compared to each other by an *attacker* (or *observer*) who knows and can use the terms in the environment $\mathcal{E}$ and the names in r. For readability, the membership $(r, \mathcal{E}, P, Q) \in \mathcal{X}$ is often written $P \, \mathcal{X}_{\mathcal{E};r} \, Q$, and should be understood as "processes P and Q are bisimilar, under the environment $\mathcal{E}$ and the known names r."

The environmental bisimilarity is co-inductively defined by several conditions concerning the tested processes and the knowledge. As usual with weak bisimulations, we require that an internal transition by one of the processes is matched by zero or more internal transitions by the other, and that the remnants are still bisimilar.

As usual with (more recent and less common) environmental bisimulations, we require that whenever a term M is output to a known channel, the other tested process can output another term N to the same channel, and that the residues are bisimilar under the environment extended with the pair (M, N). The extension of the environment stands for the growth of knowledge of the attacker of the bisimulation game who observed the outputs (M, N), although he cannot analyse them. This spells out like: for any $P \, \mathcal{X}_{\mathcal{E};r} \, Q$ and $a \in r$, if $P \xrightarrow{\nu\widetilde{c}.\overline{a}\langle M\rangle} P'$ for fresh $\widetilde{c}$, then $Q \overset{\nu\widetilde{d}.\overline{a}\langle N\rangle}{\Longrightarrow} Q'$ for fresh $\widetilde{d}$ and $P' \, \mathcal{X}_{\mathcal{E}\cup\{(M,N)\};r} \, Q'$.

Unsurprisingly, input must be doable on the same known channel by each process, and the continuations must still be bisimilar under the same environment since nothing is learnt by the context. However, we require that the input terms are generated from the *context closure* of the environment. Intuitively, this closure represents all the processes an attacker can build by combining what he has learnt from previous outputs. Roughly, we define it as:

$$(\mathcal{E}; r)^\star = \{(C[\widetilde{M}], C[\widetilde{N}]) \mid C \ context,\ fn(C) \subseteq r,\ \widetilde{M} \, \mathcal{E} \, \widetilde{N}\}$$

where $\widetilde{M}$ denotes a sequence $M_0, \ldots, M_n$, and $\widetilde{M}\mathcal{E}\widetilde{N}$ means that for all $0 \leq i \leq n$, $M_i \mathcal{E} N_i$. Therefore, the input clause looks like: for any $P \, \mathcal{X}_{\mathcal{E};r} \, Q$, $a \in r$ and $(M, N) \in (\mathcal{E}; r)^\star$, if $P \xrightarrow{a(M)} P'$, then $Q \overset{a(N)}{\Longrightarrow} Q'$ and $P' \, \mathcal{X}_{\mathcal{E};r} \, Q'$.

The set r of known names can be extended at will by the observer, provided that the new names are fresh: for any $P \, \mathcal{X}_{\mathcal{E};r} \, Q$ and n fresh, we have $P \, \mathcal{X}_{\mathcal{E};r\cup\{n\}} \, Q$.

Parallel composition: The last clause is crucial to the soundness and usefulness of environmental bisimulations for languages with passivation, and not as straightforward as the other clauses. The idea at its base is that not only may an observer run arbitrary processes R in parallel to the tested ones (as in reduction-closed barbed equivalence), but he may also run arbitrary processes M, N he assembled from previous observations. It is critical to ensure that bisimilarity (and hopefully equivalence) is preserved by such parallel composition, and that this property can be easily proved. As $(\mathcal{E}; r)^\star$ is this set of processes that can be assembled from previous observations, we would naively expect the appropriate clause to look like:

$$\text{For any } P\ \mathcal{X}_{\mathcal{E};r}\ Q \text{ and } (M,N) \in (\mathcal{E};r)^\star,\ \text{ we have } P \mid M\ \mathcal{X}_{\mathcal{E};r}\ Q \mid N$$

but this subsumes the already impractical clause of reduction-closed barbed equivalence which we want to get round. Previous research [7,13] uses a weaker condition:

$$\text{For any } P\ \mathcal{X}_{\mathcal{E};r}\ Q \text{ and } (M,N) \in \mathcal{E},\ \text{ we have } P \mid M\ \mathcal{X}_{\mathcal{E};r}\ Q \mid N$$

arguing that $(\mathcal{E};r)^\star$ can informally do no more observations than $\mathcal{E}$, but this clause is unsound in the presence of passivation. The reason behind the unsoundness is that, in our settings, not only can a context spawn new processes M, N, but it can also *remove* running processes it created by passivating them later on. For example, consider the following processes $P = \overline{a}\langle R\rangle.!R$ and $Q = \overline{a}\langle 0\rangle.!R$. Under the above weak condition, it would be easy to construct an environmental bisimulation that relates P and Q. However, a process $a(X).m[X]$ may distinguish them. Indeed, it may receive processes R and start running it in location m, or may receive process 0 and run a copy of R from $!R$. If R is a process doing several sequential actions (for example if $R = \mathit{lock}.\mathit{unlock}$) and is passivated *in the middle* of its execution, then the remaining processes after passivation would not be equivalent any more.

To account for this new situation, we decide to modify the condition on the provenance of process that can be spawned, drawing them from $\{(a[M], a[N]) \mid a \in r,\ (M,N) \in \mathcal{E}\}$, thus giving the clause:

$$\text{For any } P\ \mathcal{X}_{\mathcal{E};r}\ Q,\ a \in r \text{ and } (M,N) \in \mathcal{E},\ \text{ we have } P \mid a[M]\ \mathcal{X}_{\mathcal{E};r}\ Q \mid a[N].$$

The new condition allows for any running process that has been previously created by the observer to be passivated, that is, removed from the current test. This clause is much more tractable than the first one using $(\mathcal{E};r)^\star$ and, unlike the second one using only $\mathcal{E}$, leads to sound environmental bisimulations (albeit with a limitation; see Remark 1).

Example: With our environmental bisimulations, non-trivial equivalence of higher-order distributed processes can be shown, such as $P_0 = !a[e \mid \overline{e}]$ and $Q_0 = !a[e] \mid !a[\overline{e}]$, where e abbreviates $e(X).0$ and $\overline{e}$ is $\overline{e}\langle 0\rangle.0$. We explain here informally how we build a bisimulation $\mathcal{X}$ relating those processes.

$$\begin{aligned}\mathcal{X} = \{(r,\mathcal{E},P,Q) \mid{}& r \supseteq \{a,e\},\ \ \mathcal{E} = \{0,e,\overline{e},e \mid \overline{e}\} \times \{0,e,\overline{e}\},\\ & P \equiv P_0 \mid \textstyle\prod_{i=1}^n l_i[M_i],\ \ Q \equiv Q_0 \mid \prod_{i=1}^n l_i[N_i],\ \ n \geq 0,\\ & \widetilde{l} \in r,\ \ (\widetilde{M},\widetilde{N}) \in \mathcal{E}\}\end{aligned}$$

Since we want $P_0\ \mathcal{X}_{\mathcal{E};r}\ Q_0$, the spawning clause of the bisimulation requires that for any $(M_1,N_1) \in \mathcal{E}$ and $l_1 \in r$, we have $P_0 \mid l_1[M_1]\ \mathcal{X}_{\mathcal{E};r}\ Q_0 \mid l_1[N_1]$. Then, by repeatedly applying this clause, we obtain $(P_0 \mid \prod_{i=1}^n l_i[M_i])\ \mathcal{X}_{\mathcal{E};r}\ (Q_0 \mid \prod_{i=1}^n l_i[N_i])$. Since the observer can add fresh names at will, we require r to be a superset of the free names $\{a,e\}$ of P_0 and Q_0. Also, we have the intuition that the only possible outputs from P and Q are processes $e \mid \overline{e}$, e, $\overline{e}$, and 0. Thus, we set ahead $\mathcal{E}$ as the Cartesian product of $\{0,e,\overline{e},e \mid \overline{e}\}$ with $\{0,e,\overline{e}\}$, that is, the combination of expectable outputs. We emphasize that it is indeed reasonable to relate $\overline{e}$, e and $e \mid \overline{e}$ to 0, e and $\overline{e}$ in $\mathcal{E}$ for the observer cannot analyse the pairs: he can only use them along the tested processes P and Q which, by the design of environmental bisimulations, will make up for the differences.

$$\begin{tikzcd}
 & P_0 \mid a[\overline{\gamma}] \mid \prod_{i=1}^{n} l_i[M_i] & \text{(i)} \\
 & = P_0 \mid \prod_{i=1}^{n+1} l_i[M_i] \quad \text{for } l_{n+1} = a,\, M_{n+1} = \overline{\gamma} & \\
P_0 \mid \prod_{i=1}^{n} l_i[M_i] \arrow[uur, bend left, "\gamma \in \{e, \overline{e}\}"] \arrow[r, "\overline{l_n}\langle M_n \rangle"] \arrow[dr, bend right, "\alpha \in \{e, \overline{e}\}"'] & P_0 \mid \prod_{i=1}^{n-1} l_i[M_i] & \text{(ii)} \\
 & P_0 \mid \prod_{i=1}^{n-1} l_i[M_i] \mid l_n[M_n'] & \text{(iii)} \\
 & \equiv P_0 \mid a[e \mid \overline{e}] \mid \prod_{i=1}^{n-1} l_i[M_i] \mid l_n[M_n'] & \\
 & = P_0 \mid \prod_{i=1}^{n+1} l_i[M_i'] & \\
 & \text{for} \quad M_i' = M_i \ (0 \leq i \leq n-1),\ M_n \xrightarrow{\alpha} M_n', & \\
 & l_{n+1} = a,\ M_{n+1} = e \mid \overline{e}. &
\end{tikzcd}$$

$$\begin{tikzcd}
 & Q_0 \mid a[0] \mid \prod_{i=1}^{n} l_i[N_i] & \text{(i)} \\
 & = Q_0 \mid \prod_{i=1}^{n+1} l_i[N_i] \quad \text{for } l_{n+1} = a,\, N_{n+1} = 0 & \\
Q_0 \mid \prod_{i=1}^{n} l_i[N_i] \arrow[uur, bend left, "\gamma"] \arrow[r, "\overline{l_n}\langle N_n \rangle"] \arrow[dr, bend right, "\alpha"'] & Q_0 \mid \prod_{i=1}^{n-1} l_i[N_i] & \text{(ii)} \\
 & Q_0 \mid a[0] \mid \prod_{i=1}^{n} l_i[N_i] & \text{(iii)} \\
 & = Q_0 \mid \prod_{i=1}^{n+1} l_i[N_i] \quad \text{for } l_{n+1} = a,\, N_{n+1} = 0 &
\end{tikzcd}$$

Fig. 1. Simulation of observable transitions

Let us now observe the possible transitions from P and their corresponding transitions from Q by glossing over two pairs of trees, where related branches represent the correspondences. (Simulation in the other direction is similar and omitted for brevity.) First, let us consider the input and output actions as shown in Figure 1. (i) When P_0 does an input action e or an output action $\overline{e}$, it leaves behind a process $a[\overline{e}]$ or $a[e]$, respectively. Q_0 can also do the same action, leaving $a[0]$. Since both $(\overline{e}, 0)$ and $(e, 0)$ are in $\mathcal{E}$, we can add the leftover processes to the respective products $\prod$; (ii) output by passivation is trivial to match (without loss of generality, we only show the case $i = n$), and (iii) observable actions α of an M_n, leaving a residue M_n', are matched by one of Q_0's $a[\alpha]$, leaving $a[0]$. To pair with this $a[0]$, we replicate an $a[e \mid \overline{e}]$ from P_0, and then, as in (i), they add up to the products $\prod$.

In a similar way, we explain how τ transitions of P are matched by Q, with another pair of transitions trees described in Figure 2.

(1) When an $a[e \mid \overline{e}]$ from P_0 turns into $a[0]$, Q does not have to do any action, for we work with weak bisimulations. By replication, Q can produce a copy $a[e]$ (or alternatively $a[\overline{e}]$) from Q_0, and since $(0, e)$ is in $\mathcal{E}$, we can add the $a[0]$ and the copy $a[e]$ to the products $\prod$; (2) P can also make a reaction between two copies of $a[e \mid \overline{e}]$ in P_0, leaving behind $a[e]$ and $a[\overline{e}]$. As in (1), Q can draw two copies of $a[e]$ from Q_0, and each product can be enlarged by two elements; (3) it is also possible for $M_n = e \mid \overline{e}$ to do a τ transition, becoming $M_n' = 0$. It stands that $(M_n', N_n) \in \mathcal{E}$ and we are done; (4) very similarly, two processes M_n and M_{n-1} may react, becoming M_n' and M_{n-1}'. It stands also that (M_{n-1}', N_{n-1}) and (M_n', N_n) are in $\mathcal{E}$, so the resulting processes are still related; (5) it is possible for M_n to follow the transition $M_n \xrightarrow{\alpha} M_n'$ and react with a copy from P_0 which leaves behind $a[\alpha]$ (since $\overline{\alpha}$ has been consumed to conclude the

$$\begin{tikzcd}[row sep=small]
& P_0 \mid a[0] \mid \prod_{i=1}^{n} l_i[M_i] \qquad (1) \\
& = P_0 \mid \prod_{i=1}^{n+1} l_i[M_i] \ \text{for} \ l_{n+1} = a,\ M_{n+1} = 0 \\
& P_0 \mid a[e] \mid a[\overline{e}] \mid \prod_{i=1}^{n} l_i[M_i] \qquad (2) \\
& = P_0 \mid \prod_{i=1}^{n+2} l_i[M_i] \ \text{for} \ l_{n+1} = l_{n+2} = a,\ M_{n+1} = e,\ M_{n+2} = \overline{e} \\
& P_0 \mid \prod_{i=1}^{n} l_i[M'_i] \qquad (3) \\
& \text{for} \ M_i = M'_i \ (0 \le i \le n-1),\ M_n \xrightarrow{\tau} M'_n \\
P_0 \mid \prod_{i=1}^{n} l_i[M_i] \arrow[r] \arrow[uuuuuur, bend left] \arrow[uuuur, bend left] \arrow[uur, bend left] \arrow[ddr, bend right] \arrow[dddddr, bend right] \arrow[dddddddr, bend right] & P_0 \mid \prod_{i=1}^{n} l_i[M'_i] \qquad (4) \\
& \text{for} \ M_i = M'_i \ (0 \le i \le n-2),\ M_{n-1} \xrightarrow{e} M'_{n-1},\ M_n \xrightarrow{\overline{e}} M'_n \\
& P_0 \mid a[\alpha] \mid \prod_{i=1}^{n} l_i[M'_i] \ = \ P_0 \mid \prod_{i=1}^{n+1} l_i[M'_i] \qquad (5) \\
& \text{for} \ M'_i = M_i \ (0 \le i \le n-1),\ M_n \xrightarrow{\alpha} M'_n, \\
& \quad \alpha \in \{e, \overline{e}\},\ l_{n+1} = a,\ M_{n+1} = \alpha \\
& P_0 \mid a[\overline{e}] \mid \prod_{i=1}^{n-1} l_i[M_i] \ = \ P_0 \mid \prod_{i=1}^{n} l'_i[M'_i] \qquad (6) \\
& \text{for} \ l'_i = l_i,\ M'_i = M_i \ (0 \le i \le n-1),\ l'_n = a,\ M'_n = \overline{e} \\
& P_0 \mid \prod_{i=1}^{n-2} l_i[M_i] \mid l_{n-1}[M'_{n-1}] \qquad (7) \\
& \equiv P_0 \mid a[e \mid \overline{e}] \mid \prod_{i=1}^{n-2} l_i[M_i] \mid l_{n-1}[M'_{n-1}] \ = \ P_0 \mid \prod_{i=1}^{n} l'_i[M'_i] \\
& \text{for} \ M'_i = M_i \ (0 \le i \le n-2),\ M_{n-1} \xrightarrow{e} M'_{n-1}, \\
& \quad l'_i = l_i \ (0 \le i \le n-1),\ l'_n = a,\ M'_n = e \mid \overline{e}
\end{tikzcd}$$

$$\begin{tikzcd}[row sep=small]
& Q_0 \mid \prod_{i=1}^{n} l_i[N_i] \ \equiv \ Q_0 \mid a[e] \mid \prod_{i=1}^{n} l_i[N_i] \qquad (1) \\
& = Q_0 \mid \prod_{i=1}^{n+1} l_i[N_i] \ \text{for} \ l_{n+1} = a, N_{n+1} = e \\
& Q_0 \mid \prod_{i=1}^{n} l_i[N_i] \ \equiv \ Q_0 \mid a[e] \mid a[e] \mid \prod_{i=1}^{n} l_i[N_i] \qquad (2) \\
& = Q_0 \mid \prod_{i=1}^{n+2} l_i[N_i] \ \text{for} \ l_{n+1} = l_{n+2} = a,\ N_{n+1} = N_{n+2} = e \\
& Q_0 \mid \prod_{i=1}^{n} l_i[N_i] \qquad (3) \\
Q_0 \mid \prod_{i=1}^{n} l_i[N_i] \arrow[r, dashed] \arrow[uuuuur, dashed, bend left] \arrow[uuur, dashed, bend left] \arrow[ur, dashed, bend left] \arrow[dr, dashed, bend right] \arrow[dddr, bend right] \arrow[dddddr, bend right] & Q_0 \mid \prod_{i=1}^{n} l_i[N_i] \qquad (4) \\
& Q_0 \mid \prod_{i=1}^{n} l_i[N_i] \ \equiv \ Q_0 \mid a[e] \mid \prod_{i=1}^{n} l_i[N_i] \qquad (5) \\
& = Q_0 \mid \prod_{i=1}^{n+1} l_i[N_i] \ \text{for} \ l_{n+1} = a,\ N_{n+1} = e \\
& Q_0 \mid a[0] \mid \prod_{i=1}^{n-1} l_i[N_i] \ = \ Q_0 \mid \prod_{i=1}^{n} l'_i[N'_i] \qquad (6) \\
& \text{for} \ l'_i = l_i,\ N'_i = N_i \ (0 \le i \le n-1),\ l'_n = a,\ N'_n = 0 \\
& Q_0 \mid a[0] \mid \prod_{i=1}^{n-1} l_i[N_i] \ = \ Q_0 \mid \prod_{i=1}^{n} l'_i[N'_i] \qquad (7) \\
& \text{for} \ l'_i = l_i,\ N'_i = N_i \ (0 \le i \le n-1),\ l'_n = a,\ N'_n = 0
\end{tikzcd}$$

Fig. 2. Simulation of internal transitions (dotted lines mean zero transitions)

reaction). Again, it stands that M'_n and N_n are related by $\mathcal{E}$, and that we can draw an $a[e]$ from Q_0 to pair it with the residue M'_n in the products $\prod$; (6) also, a copy $a[e \mid \overline{e}]$ from P_0 may passivate an $l_i[M_i]$, provided $l_i = e$, and leave a residue $a[\overline{e}]$. Q can do the same passivation using Q_0's $a[e]$, and leave $a[0]$. As it happens that $(\overline{e}, 0)$ is in $\mathcal{E}$, the residues can be added to the products too; (7) finally, the process $l_n[M_n]$, if $l_n = e$, may be passivated by M_{n-1}, reducing the size of P's product. Q can passivate $l_n[N_n]$ too, using a copy $a[e]$ from P_0, which becomes $a[0]$ after the reaction. Q's product too is shorter, but we need to add the $a[0]$ to it. To do so, we draw a copy $a[e \mid \overline{e}]$ from P_0, and since $(e \mid \overline{e}, 0)$ is in $\mathcal{E}$, $a[e \mid \overline{e}]$ and $a[0]$ are merged into their respective product.

This ends the sketch of the proof that $\mathcal{X}$ is an environmental bisimulation, and therefore that $!a[e \mid \overline{e}]$ and $!a[e] \mid a[\overline{e}]$ are behaviourally equivalent.

1.3 Overview of the Paper

The rest of this paper is structured as follows. In Section 2 we describe the higher-order π-calculus with passivation. In Section 3 we formalize our environmental bisimulations. In Section 4 we give some examples of bisimilar processes. In Section 5, we bring up some future work to conclude our paper.

2 Higher-Order π-Calculus with Passivation

We introduce a slight variation of the higher-order π-calculus with passivation [7]—HOπP for short—through its syntax and a labelled transitions system.

2.1 Syntax

The syntax of our HOπP processes P, Q is given by the following grammar, very similar to that of Lenglet *et al.* [7] (the higher-order π-calculus extended with located processes and their passivation):

$$\begin{aligned} P, Q &::= 0 \mid a(X).P \mid \overline{a}\langle M\rangle.P \mid (P \mid P) \mid a[P] \mid \nu a.P \mid\ !P \mid \mathit{run}(M) \\ M, N &::= X \mid `P \end{aligned}$$

X ranges over the set of variables, and a over the set of names which can be used for both locations and channels. $a[P]$ denotes the process P running in location a. To define a general up-to context technique (Definition 2, see also Section 5), we distinguish terms M, N from processes P, Q and adopt explicit syntax for processes as terms ‘P and their execution $\mathit{run}(M)$.

2.2 Labelled Transitions System

We define n, fn, bn and fv to be the functions that return respectively the set of names, free names, bound names and free variables of a process or an action. We abbreviate a (possibly empty) sequence $x_0, x_1, \ldots, x_n$ as $\widetilde{x}$ for any meta-variable x. The transition semantics of HOπP is given by the following labelled transition system, which is based on that of the higher-order π-calculus (omitting symmetric rules PAR-R and REACT-R):

$$\frac{}{a(X).P \xrightarrow{a(M)} P\{M/X\}}\ \textsc{Ho-in} \qquad \frac{}{\overline{a}\langle M\rangle.P \xrightarrow{\overline{a}\langle M\rangle} P}\ \textsc{Ho-out}$$

$$\frac{P_1 \xrightarrow{\alpha} P_1' \quad bn(\alpha)\cap fn(P_2)=\emptyset}{P_1 \mid P_2 \xrightarrow{\alpha} P_1' \mid P_2}\ \textsc{Par-L} \qquad \frac{!P \mid P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}\ \textsc{Rep}$$

$$\frac{P_1 \xrightarrow{(\nu\widetilde{b}).\overline{a}\langle M\rangle} P_1' \quad P_2 \xrightarrow{a(M)} P_2' \quad \{\widetilde{b}\}\cap fn(P_2)=\emptyset}{P_1 \mid P_2 \xrightarrow{\tau} \nu\widetilde{b}.(P_1' \mid P_2')}\ \textsc{React-L}$$

$$\frac{P \xrightarrow{\alpha} P' \quad a\notin n(\alpha)}{\nu a.P \xrightarrow{\alpha} \nu a.P'}\ \textsc{Guard} \qquad \frac{P \xrightarrow{(\nu\widetilde{b}).\overline{a}\langle M\rangle} P' \quad c\neq a \quad c\in fn(M)\setminus\{\widetilde{b}\}}{\nu c.P \xrightarrow{\nu(\widetilde{b},c).\overline{a}\langle M\rangle} P'}\ \textsc{Extr}$$

extended with the following three rules:

$$\frac{P \xrightarrow{\alpha} P'}{a[P] \xrightarrow{\alpha} a[P']}\ \textsc{Transp} \qquad \frac{}{a[P] \xrightarrow{\overline{a}\langle `P\rangle} 0}\ \textsc{Passiv} \qquad \frac{}{\mathit{run}(`P) \xrightarrow{\tau} P}\ \textsc{Run}$$

Assuming again knowledge of the standard higher-order π-calculus [9,11], we only explain below the three added rules that are not part of it. The `Transp` rule expresses the *transparency* of locations, the fact that transitions can happen below a location and be observed outside its boundary. The `Passiv` rule illustrates that, at any time, a process running under a location can be passivated (stopped and turned into a term) and sent along the channel corresponding to the location's name. Quotation of the process output reminds us that higher-order communications transport terms. Finally, the `Run` rule shows how, at the cost of an internal transition, a process term be instantiated. As usual with small-steps semantics, transition does not progress for undefined cases (such as $\mathit{run}(X)$) or when the assumptions are not satisfied.

Henceforth, we shall write $\overline{a}.P$ to mean $\overline{a}\langle `0\rangle.P$ and $a.P$ for $a(X).P$ if $X \notin fv(P)$. We shall also write $\equiv$ for the structural congruence, whose definition is standard (see the appendix, Definition A.1).

3 Environmental Bisimulations of HOπP

Given the higher-order nature of the language, and in order to get round the universal quantification issue of context bisimulations, we would like observations (terms) to be stored and reusable for further testing. To this end, let us define an *environmental relation* $\mathcal{X}$ as a set of elements $(r, \mathcal{E}, P, Q)$ where r is a finite set of names, $\mathcal{E}$ is a binary relation (with finitely many free names) on variable-closed terms (i.e. terms with no free variables), and P and Q are variable-closed processes.

We generally write $x \oplus S$ to express the set union $\{x\} \cup S$. We also use graphically convenient notation $P\ \mathcal{X}_{\mathcal{E};r}\ Q$ to mean $(r, \mathcal{E}, P, Q) \in \mathcal{X}$ and define the *term context closure* $(\mathcal{E}; r)^\star = \mathcal{E} \cup \{(`P, `Q) \mid (P, Q) \in (\mathcal{E}; r)^\circ\}$ with the *process context closure* $(\mathcal{E}; r)^\circ = \{(C[\widetilde{M}], C[\widetilde{N}]) \mid \widetilde{M}\mathcal{E}\widetilde{N},\ C \text{ context},\ bn(C) \cap fn(\mathcal{E}, r) = \emptyset,\ fn(C) \subseteq r\}$, where a context is a process with zero or more *holes* for terms. Note the distinction of terms $`P$, $`Q$ from processes P, Q. We point out that $(\emptyset; r)^\star$ is the identity on terms with free names in r, that $(\mathcal{E};r)^\star$ includes $\mathcal{E}$ by definition, and that the context closure operations are monotonic on $\mathcal{E}$ (and r). Therefore, for any $\mathcal{E}$ and r, the set $(\mathcal{E};r)^\star$ includes the identity $(\emptyset;r)^\star$ too. Also, we use the notations $\mathcal{S}.1$ and $\mathcal{S}.2$ to denote the first and second projections of a relation (i.e. set of pairs) S. Finally, we define weak transitions $\Rightarrow$ as the reflexive, transitive closure of $\xrightarrow{\tau}$, and $\overset{\alpha}{\Rightarrow}$ as $\Rightarrow \xrightarrow{\alpha} \Rightarrow$ for $\alpha \neq \tau$ (and define $\overset{\tau}{\Rightarrow}$ as $\Rightarrow$).

We can now define environmental bisimulations formally:

Definition 1. *An environmental relation $\mathcal{X}$ is an environmental bisimulation if $P\ \mathcal{X}_{\mathcal{E};r}\ Q$ implies:*

1. *if $P \xrightarrow{\tau} P'$, then $\exists Q'.\ Q \Rightarrow Q'$ and $P'\ \mathcal{X}_{\mathcal{E};r}\ Q'$,*
2. *if $P \xrightarrow{a(M)} P'$ with $a \in r$, and if $(M,N) \in (\mathcal{E};r)^\star$, then $\exists Q'.\ Q \overset{a(N)}{\Longrightarrow} Q'$ and $P'\ \mathcal{X}_{\mathcal{E};r}\ Q'$,*
3. *if $P \xrightarrow{\nu\widetilde{b}.\overline{a}\langle M\rangle} P'$ with $a \in r$ and $\widetilde{b} \notin fn(r,\mathcal{E}.1)$, then $\exists Q', N.\ Q \overset{\nu\widetilde{c}.\overline{a}\langle N\rangle}{\Longrightarrow} Q'$ with $\widetilde{c} \notin fn(r,\mathcal{E}.2)$ and $P'\ \mathcal{X}_{(M,N)\oplus\mathcal{E};r}\ Q'$,*
4. *for any $(`P_1, `Q_1) \in \mathcal{E}$ and $a \in r$, we have $P \mid a[P_1]\ \mathcal{X}_{\mathcal{E};r}\ Q \mid a[Q_1]$,*
5. *for any $n \notin fn(\mathcal{E},P,Q)$, we have $P\ \mathcal{X}_{\mathcal{E};n\oplus r}\ Q$, and*
6. *the converse of 1, 2 and 3 on Q's transitions.*

Modulo the symmetry resulting from clause 6, clause 1 is usual; clause 2 enforces bisimilarity to be preserved by any input that can be built from the knowledge, hence the use of the context closure; clause 3 enlarges the knowledge of the observer with the leaked out terms. Clause 4 allows the observer to spawn (and immediately run) terms concurrently to the tested processes, while clause 5 shows that he can also create fresh names at will.

A few points related to the handling of free names are worth mentioning: as the set of free names in $\mathcal{E}$ is finite, clause 5 can always be applied; therefore, the attacker can add arbitrary fresh names to the set r of known names so as to use them in terms M and N in clause 2. Fresh $\widetilde{b}$ and $\widetilde{c}$ in clause 3 also exist thanks to the finiteness of free names in $\mathcal{E}$ and r.

We define environmental bisimilarity $\sim$ as the union of all environmental bisimulations, and it holds that it is itself an environmental bisimulation (all the conditions above are monotone on $\mathcal{X}$). Therefore, $P \sim_{\mathcal{E};r} Q$ if and only if $P\ \mathcal{X}_{\mathcal{E};r}\ Q$ for some environmental bisimulation $\mathcal{X}$. We do particularly care about the situation where $\mathcal{E} = \emptyset$ and $r = fn(P,Q)$. It corresponds to the equivalence of two processes when the observer knows all of their free names (and thus can do all observations), but has not yet learnt any output pair.

For improving the practicality of our bisimulation proof method, let us devise an up-to context technique [11, p. 86]: for an environmental relation $\mathcal{X}$, we write $P\ \mathcal{X}^\star_{\mathcal{E};r}\ Q$ if $P \equiv \nu\widetilde{c}.(P_0 \mid P_1)$, $Q \equiv \nu\widetilde{d}.(Q_0 \mid Q_1)$, $P_0\ \mathcal{X}_{\mathcal{E}';r'}\ Q_0$, $(P_1,Q_1) \in (\mathcal{E}';r')^\circ$, $\mathcal{E} \subseteq (\mathcal{E}';r')^\star$, $r \subseteq r'$, and $\{\widetilde{c}\} \cap fn(r,\mathcal{E}.1) = \{\widetilde{d}\} \cap fn(r,\mathcal{E}.2) = \emptyset$. As a matter of fact, this is actually an up-to context and up-to environment and up-to restriction and up-to structural congruence technique, but because of the clumsiness of this appellation we will restrain ourselves to "up-to context" to preserve clarity. To roughly explain the convenience behind this notation and its (long) name: (1) "up-to context" states that we can take any (P_1, Q_1) from the (process) context closure $(\mathcal{E}'; r')^\circ$ of the environment $\mathcal{E}'$ (with free names in r') and execute them in parallel with processes P_0 and Q_0 related by $\mathcal{X}_{\mathcal{E}';r'}$; similarly, we allow environments $\mathcal{E}$ with terms that are not in $\mathcal{E}'$ itself but are in the (term) context closure $(\mathcal{E}'; r')^\star$; (2) "up-to environment" states that, when proving the bisimulation clauses, we please ourselves with environments $\mathcal{E}'$ that are *larger* than the $\mathcal{E}$ requested by Definition 1; (3) "up-to restriction" states that we also content ourselves with tested processes P, Q with extra restrictions $\nu\widetilde{c}$ and $\nu\widetilde{d}$ (i.e. less observable names); (4) finally, "up-to structural congruence" states that we identify all processes that are structurally congruent to $\nu\widetilde{c}.(P_0 \mid P_1)$ and $\nu\widetilde{d}.(Q_0 \mid Q_1)$.

Using this notation, we define environmental bisimulations up-to context as follows:

Definition 2. *An environmental relation $\mathcal{X}$ is an environmental bisimulation up-to context if $P \;\mathcal{X}_{\mathcal{E};r}\; Q$ implies:*

1. *if $P \xrightarrow{\tau} P'$, then $\exists Q'.\; Q \Rightarrow Q'$ and $P' \;\mathcal{X}^\star_{\mathcal{E};r}\; Q'$,*
2. *if $P \xrightarrow{a(M)} P'$ with $a \in r$, and if $(M, N) \in (\mathcal{E}; r)^\star$, then $\exists Q'.\; Q \overset{a(N)}{\Longrightarrow} Q'$ and $P' \;\mathcal{X}^\star_{\mathcal{E};r}\; Q'$,*
3. *if $P \xrightarrow{\nu\widetilde{b}.\overline{a}\langle M\rangle} P'$ with $a \in r$ and $\widetilde{b} \notin fn(r, \mathcal{E}.1)$, then $\exists Q', N.\; Q \overset{\nu\widetilde{c}.\overline{a}\langle N\rangle}{\Longrightarrow} Q'$ with $\widetilde{c} \notin fn(r, \mathcal{E}.2)$ and $P' \;\mathcal{X}^\star_{(M,N)\oplus\mathcal{E};r}\; Q'$,*
4. *for any $(`P_1, `Q_1) \in \mathcal{E}$ and $a \in r$, we have $P \mid a[P_1] \;\mathcal{X}^\star_{\mathcal{E};r}\; Q \mid a[Q_1]$,*
5. *for any $n \notin fn(\mathcal{E}, P, Q)$, we have $P \;\mathcal{X}_{\mathcal{E};n\oplus r}\; Q$, and*
6. *the converse of 1, 2 and 3 on Q's transitions.*

The conditions on each clause (except 5, which is unchanged for the sake of technical convenience) are weaker than that of the standard environmental bisimulations, as we require in the positive instances bisimilarity modulo a context, not just bisimilarity itself. It is important to remark that, unlike in [12] but as in [13], we do not need a specific context to avoid stating a tautology in clause 4; indeed, we spawn terms $(`P_1, `Q_1) \in \mathcal{E}$ immediately as processes P_1 and Q_1, while the context closure can only use the terms under an explicit *run* operator.

We prove the soundness (under some condition; see Remark 1) of environmental bisimulations as follows. Full proofs are found in the appendix, Section B but are nonetheless sketched below.

Lemma 1 (Input lemma). *If $(P_1, Q_1) \in (\mathcal{E}; r)^\circ$ and $P_1 \xrightarrow{a(M)} P_1'$ then $\forall N.\exists Q_1'.\; Q_1 \xrightarrow{a(N)} Q_1'$ and $(P_1', Q_1') \in ((M, N)\oplus\mathcal{E}; r)^\circ$.*

Lemma 2 (Output lemma). *If $(P_1, Q_1) \in (\mathcal{E}; r)^\circ$, $\{\widetilde{b}\} \cap fn(\mathcal{E}, r) = \emptyset$ and $P_1 \xrightarrow{\nu\widetilde{b}.\overline{a}\langle M\rangle} P_1'$ then $\exists Q_1', N.\; Q_1 \xrightarrow{\nu\widetilde{b}.\overline{a}\langle N\rangle} Q_1'$, $(P_1', Q_1') \in (\mathcal{E}; \widetilde{b}\oplus r)^\circ$ and $(M, N) \in (\mathcal{E}; \widetilde{b}\oplus r)^\star$.*

Definition 3 (Run-erasure). *We write $P \leq Q$ if P can be obtained by (possibly repeatedly) replacing zero or more subprocesses $run(`R)$ of Q with R, and write $P \;\mathcal{Y}^{-}_{\mathcal{E};r}\; Q$ for $P \leq \mathcal{Y}^\star_{\leq\mathcal{E}\geq;r} \geq Q$.*

Definition 4 (Simple environment). *A process is called* simple *if none of its subprocesses has the form $\nu a.P$ or $a(X).P$ with $X \in fv(P)$. An environment is called simple if all the processes in it are simple. An environmental relation is called simple if all of its environments are simple (note that the tested processes may still be non-simple).*

Lemma 3 (Reaction lemma). *For any simple environmental bisimulation up-to context $\mathcal{Y}$, if $P \ \mathcal{Y}^{-}_{\mathcal{E};r} \ Q$ and $P \xrightarrow{\tau} P'$, then there is a Q' such that $Q \overset{\tau}{\Rightarrow} Q'$ and $P' \ \mathcal{Y}^{-}_{\mathcal{E};r} \ Q'$.*

Proof sketch. Lemma 1 (resp. 2) is proven by straightforward induction on the transition derivation of $P_1 \xrightarrow{a(M)} P_1'$ (resp. $P_1 \xrightarrow{\nu\widetilde{b}.\overline{a}\langle M\rangle} P_1'$). Lemma 3 is proven last, as it uses the other two lemmas (for the internal communication case).

Lemma 4 (Soundness of up-to context). *Simple bisimilarity up-to context is included in bisimilarity.*

Proof sketch. By checking that $\{(r, \mathcal{E}, P, Q) \mid P \ \mathcal{Y}^{-}_{\mathcal{E};r} \ Q\}$ is included in $\sim$, where $\mathcal{Y}$ is the simple environmental bisimilarity up-to context. In particular, we use Lemma 1 for clause 2, Lemma 2 for clause 3, and Lemma 3 for clause 1 of the environmental bisimulation.

Our definitions of reduction-closed barbed equivalence $\approx$ and congruence $\approx_c$ are standard and omitted for brevity; see the appendix, Definition B.2 and B.3

Theorem 1 (Barbed equivalence from environmental bisimulation) *If $P \ \mathcal{Y}^{-}_{\emptyset;fn(P,Q)} \ Q$ for a simple environmental bisimulation up-to context $\mathcal{Y}$, then $P \approx Q$.*

Proof sketch. By verifying that each clause of the definition of $\approx$ is implied by membership of $\mathcal{Y}^{-}$, using Lemma 4 for the parallel composition clause.

Corollary 1 (Barbed congruence from environmental bisimulation) *If $\overline{a}\langle \text{‘}P\rangle \ \mathcal{Y}^{-}_{\emptyset;a\oplus fn(P,Q)} \ \overline{a}\langle \text{‘}Q\rangle$ for a simple environmental bisimulation up-to context $\mathcal{Y}$, then $P \approx_c Q$.*

We recall that, in context bisimulations, showing the equivalence of $\overline{a}\langle \text{‘}P\rangle$ and $\overline{a}\langle \text{‘}Q\rangle$ almost amounts to testing the equivalence of P and Q in every context. However, with environmental bisimulations, only the location context in clause 4 of the bisimulation has to be considered.

Remark 1. The extra condition "simple" is needed because of a technical difficulty in the proof of Lemma 3: when an input process $a(X).P$ is spawned under location b in parallel with an output context $\nu c.\overline{a}\langle M\rangle.Q$ (with $c \in fn(M)$), they can make the transition $b[a(X).P \mid \nu c.\overline{a}\langle M\rangle.Q] \xrightarrow{\tau} b[\nu c.(P\{M/X\} \mid Q)]$, where the restriction operator νc appears *inside* the location b (and therefore can be passivated together with the processes); however, our spawning clause only gives us $b[a(X).P] \mid \nu c.\overline{a}\langle M\rangle.Q \xrightarrow{\tau} \nu c.(b[P\{M/X\}] \mid Q)$ and does not cover the above case. Further investigation is required to overcome this difficulty (although we have not yet found a concrete counterexample of soundness, we conjecture some modification to the bisimulation clauses would be necessary). Note that, even if the environments are simple, the tested processes do not always have to be simple, as in Example 4 and 5. Moreover, thanks to up-to context, even the output terms (including passivated processes) can be non-simple.

4 Examples

Here, we give some examples of HOπP processes whose behavioural equivalence is proven with the help of our environmental bisimulations. In each example, we prove the equivalence by exhibiting a relation $\mathcal{X}$ containing the two processes we consider, and by showing that it is indeed a bisimulation up-to context (and environment, restriction and structural congruence). We write $P \mid \ldots \mid P$ for a finite, possibly null, product of the process P.

Example 1. $e \mid !a[e] \mid !a[0] \approx !a[e] \mid !a[0]$. (This example comes from [7].)

Proof. Take $\mathcal{X} = \{(r,\ \emptyset,\ e \mid P,\ P) \mid r \supseteq \{a, e\}\} \cup \{(r, \emptyset, P, P) \mid r \supseteq \{a, e\}\}$ where $P = !a[e] \mid !a[0]$. It is immediate to verify that whenever $P \xrightarrow{\alpha} P'$, we have $P' \equiv P$, and therefore that transition $e \mid P \xrightarrow{\alpha} e \mid P' \equiv e \mid P$ can be matched by $P \xrightarrow{\alpha} P' \equiv P$ and conversely. Also, for $e \mid P \xrightarrow{e} P$, we have that $P \xrightarrow{e} !a[e] \mid a[0] \mid !a[0] \equiv P$ and we are done since $(r, \emptyset, P, P) \in \mathcal{X}$. Moreover, the set r must contain the free names of P, and to satisfy clause 5 about adding fresh names, bigger r's must be allowed too. The passivations of $a[e]$ and $a[0]$ can be matched by syntactically equal actions with the pairs of output terms $(‘e, ‘e)$ and $(‘0, ‘0)$ included in the identity, which in turn is included in the context closure $(\emptyset; r)^\star$. Finally clause 4 of the bisimulation is vacuously satisfied because the environment is empty. We therefore have $e \mid !a[e] \mid !a[0] \approx !a[e] \mid !a[0]$ from the soundness of environmental bisimulation up-to context.

Example 2. $!\overline{a} \mid !e \ \approx !a[e]$.

Proof sketch. Take $\mathcal{X} = \{(r,\ \mathcal{E},\ P,\ Q) \mid r \supseteq \{a, e, l_1, \ldots, l_n\} \mid \mathcal{E} = \{(‘0, ‘e)\},\ n \geq 0,$ $P = !\overline{a} \mid !e \mid \prod_{i=1}^{n} l_i[0],\ \ Q = !a[e] \mid \prod_{i=1}^{n} l_i[e] \mid a[0] \mid \ldots \mid a[0]\}$. See the appendix, Example C.1 for the rest of the proof.

Example 3. $!a[e] \,|\, !b[\overline{e}] \approx !a[b[e \,|\, \overline{e}]]$. This example shows the equivalence proof of more complicated processes with nested locations.

Proof sketch. Take:

$$\begin{aligned}
\mathcal{X} = \{(r,\ \mathcal{E},\ P,\ Q) \mid\ & r \ \ \supseteq \{a, e, b, l_1, \ldots, l_n\},\\
& P_0 = !a[e] \mid !b[\overline{e}],\ \ Q_0 = !a[b[e \mid \overline{e}]],\\
& P \ \ = P_0 \mid \textstyle\prod_{i=1}^{n} l_i[P_i] \mid b[0] \mid \ldots \mid b[0],\\
& Q \ \ = Q_0 \mid \textstyle\prod_{i=1}^{n} l_i[Q_i],\\
& (‘\widetilde{P}, ‘\widetilde{Q}) \in \mathcal{E},\ n \geq 0\},\\
\mathcal{E}\ \ = \{(‘x, ‘y) \mid\ & x \in \{0, e, \overline{e}\},\ y \equiv\in \{0, e, \overline{e}, (e \mid \overline{e}), b[0], b[e], b[\overline{e}], b[e \mid \overline{e}]\}\}\,.
\end{aligned}$$

See the appendix, Example C.2 for the rest of the proof.

Example 4. $c(X).\mathit{run}(X) \approx \nu f.(f[c(X).\mathit{run}(X)] \mid !f(Y).f[\mathit{run}(Y)])$. The latter process models a system where a process $c(X).\mathit{run}(X)$ runs in location f, and executes any process P it has received. In parallel is a process $f(Y).f[\mathit{run}(Y)]$ which can passivate $f[P]$ and respawn the process P under the same location f. Informally, this models a system which can restart a computer and resume its computation after a failure.

Proof. Take $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ where:

$$\begin{aligned}
\mathcal{X}_1 &= \{(r,\ \emptyset,\ c(X).\mathit{run}(X),\ \nu f.(f[c(X).\mathit{run}(X)] \mid !f(Y).f[\mathit{run}(Y)])) \mid r \supseteq \{c\}\},\\
\mathcal{X}_2 &= \{(r,\ \emptyset,\ P,\ Q) \mid r\ \supseteq c \oplus \mathit{fn}(R), \qquad S = \mathit{run}(‘\mathit{run}(\ldots ‘\mathit{run}(‘R)\ldots)),\\
&\qquad\qquad P \in \{\mathit{run}(‘R), R\},\ \ Q = \nu \widetilde{f}.(f[S] \mid !f(Y).[\mathit{run}(Y)])\}.
\end{aligned}$$

As usual, we require that r contains at least the free name c of the tested processes. All outputs belong to $(\emptyset; r)^\star$ since they come from a process R drawn from $(\emptyset; r)^\star$, and therefore, we content ourselves with an empty environment $\emptyset$. Also, by the emptiness of the environment, clause 4 of environmental bisimulations is vacuously satisfied.

Verification of transitions of elements of $\mathcal{X}_1$, i.e. inputs of some $‘R$ (with $(‘R, ‘R) \in (\emptyset; r)^\star$) from c, is immediate and leads to checking elements of $\mathcal{X}_2$. For elements of $\mathcal{X}_2$, we observe that $P = \mathit{run}(‘R)$ can do one τ transition to become R, while Q can do an internal transition passivating the process $\mathit{run}(‘R)$ running in f and place it inside $f[\mathit{run}(‘\ )]$, again and again. Q can also do τ transitions that consume all the $\mathit{run}(‘\ )$'s until it becomes R. Whenever P (resp. Q) makes an observable transition, Q (resp. P) can consume the $\mathit{run}(‘\ )$'s and weakly do the same action as they exhibit the same process. We observe that all transitions preserve membership in $\mathcal{X}_2$ (thus in $\mathcal{X}$), and therefore we have that $\mathcal{X}$ is an environmental bisimulation up-to context, which proves the behavioural equivalence of the original processes $c(X).\mathit{run}(X)$ and $c(X).\nu f.(f[c(X).\mathit{run}(X)] \mid !f(Y).f[\mathit{run}(Y)])$.

Example 5. $c(X).\mathit{run}(X) \approx c(X).\nu a.(\overline{a}\langle X\rangle \mid !\nu f.(f[a(X).\mathit{run}(X)] \mid f(Y).\overline{a}\langle Y\rangle))$. This example is a variation of Example 4 modelling a system where computation is resumed on another computer after a failure.

Proof. Take $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3$ where:

$$\begin{aligned}
\mathcal{X}_1 &= \{(r,\ \emptyset,\ c(X).\mathit{run}(X),\ c(X).\nu a.(\overline{a}\langle X\rangle \mid F)) \mid r \supseteq \{c\}\},\\
\mathcal{X}_2 &= \{(r,\ \emptyset,\ P_1,\ \nu a.(F \mid R_1 \mid R_2 \mid \overline{a}\langle ‘P_2\rangle)) \mid\\
&\qquad r \supseteq \{c\} \oplus \mathit{fn}(P),\ \ P_1, P_2 \in \{\mathit{run}(‘P), P\},\ \ R_1 = \overline{a}\langle N_1\rangle \mid \ldots \mid \overline{a}\langle N_n\rangle,\\
&\qquad R_2 = \nu l_1.(l_1[Q_1] \mid l_1(Y).\overline{a}\langle Y\rangle) \mid \ldots \mid \nu l_m.(l_m[Q_m] \mid l_m(Y).\overline{a}\langle Y\rangle),\\
&\qquad N_1, \ldots, N_n, ‘Q_1, \ldots, ‘Q_m = ‘\mathit{run}(‘\mathit{run}(\ldots ‘\mathit{run}(‘a(X).\mathit{run}(X))\ldots)),\ n \geq 0\},\\
\mathcal{X}_3 &= \{(r,\ \emptyset,\ P_1,\ \nu a.(F \mid R_1 \mid R_2 \mid \nu l.(l[P_2] \mid l(Y).\overline{a}\langle Y\rangle)))) \mid\\
&\qquad r \supseteq \{c\} \oplus \mathit{fn}(P),\ \ P_1, P_2 \in \{\mathit{run}(‘P), P\},\ \ R_1 = \overline{a}\langle N_1\rangle \mid \ldots \mid \overline{a}\langle N_n\rangle,\\
&\qquad R_2 = \nu l_1.(l_1[Q_1] \mid l_1(Y).\overline{a}\langle Y\rangle) \mid \ldots \mid \nu l_m.(l_m[Q_m] \mid l_m(Y).\overline{a}\langle Y\rangle),\\
&\qquad N_1, \ldots, N_n, ‘Q_1, \ldots, ‘Q_m = ‘\mathit{run}(‘\mathit{run}(\ldots ‘\mathit{run}(‘a(X).\mathit{run}(X))\ldots)),\ n \geq 0\},\\
F\ &= !\nu f.(f[a(X).\mathit{run}(X)] \mid f(Y).\overline{a}\langle Y\rangle).
\end{aligned}$$

The set of names r and the environment share the same fate as those of Example 4 for identical reasons. For ease, we write *lhs* and *rhs* to conveniently denote each of the tested processes.

Verification of the bisimulation clauses of $\mathcal{X}_1$ is immediate and leads to a member $(r, \emptyset, \mathit{run}(‘P), \nu a.(\overline{a}\langle ‘P\rangle \mid F))$ of $\mathcal{X}_2$ for some $‘P$ with $(‘P, ‘P) \in (\emptyset; r)^\star$. For $\mathcal{X}_2$, *lhs* can do an internal action (consuming its outer $\mathit{run}(‘\ )$) that *rhs* does not have to follow since we work with weak bisimulations, and the results is still in $\mathcal{X}_2$; conversely, internal actions of *rhs* do not have to be matched. Some of those transitions that *rhs* can do are reactions between replications from F. All those transitions creates elements of either R_1 or R_2 that can do nothing but internal actions and can be ignored further in the proof thanks to the weakness of our bisimulations.

Whenever *lhs* does an observable action α, that is, when $P_1 = P \xrightarrow{\alpha} P'$, *rhs* must do a reaction between $\overline{a}\langle`P_2\rangle$ and F, giving $\nu l.(l[P_2] \,|\, l(Y).\overline{a}\langle Y\rangle) \overset{\alpha}{\Rightarrow} \nu l.(l[P'] \,|\, l(Y).\overline{a}\langle Y\rangle)$ which satisfies $\mathcal{X}_3$'s definition. Moreover, all transitions of P_1 or P_2 in $\mathcal{X}_3$ can be matched by the other, hence preserving the membership in $\mathcal{X}_3$. Finally, a subprocess $\nu l.(l[P_2] \mid l(Y).\overline{a}\langle Y\rangle)$ of *rhs* of $\mathcal{X}_3$ can do a τ transition to $\overline{a}\langle`P_2\rangle$ and the residues belong back to $\mathcal{X}_2$.

This concludes the proof of behavioural equivalence of the original processes $c(X).\mathit{run}(X)$ and $c(X).\nu a.(\overline{a}\langle X\rangle.!\nu f.(f[a(X).\mathit{run}(X)] \mid f(Y).f[\mathit{run}(Y)]))$.

5 Discussion and Future Work

In the original higher-order π-calculus with passivation described by Lenglet *et al.* [7], terms are identified with processes: its syntax is just $P ::= 0 \mid X \mid a(X).P \mid \overline{a}\langle P\rangle.P \mid (P \,|\, P) \mid a[P] \mid \nu a.P \mid !P$. We conjecture that it is also possible to develop sound environmental bisimulations (and up-to context, etc.) for this version of HOπP, as we [12] did for the standard higher-order π-calculus. However we chose not to cover directly the original higher-order π-calculus with passivation, for two reasons: (1) the proof method of [12] which relies on guarded processes and a factorisation trick using the spawning clause of the bisimulation is inadequate in the presence of locations; (2) there is a very strong constraint in clause 4 of up-to context in [12, Definition E.1 (Appendix)] (the context has no hole for terms from $\mathcal{E}$). By distinguishing processes from terms, not only is our up-to context method much more general, but our proofs are also direct and technically simple. Although one might argue that the presence of the *run* operator is a burden, by using Definition 3, one could devise an "up-to *run*" technique and abstract $\mathit{run}(\ldots `\mathit{run}(`P))$ as P, making equivalence proofs easier to write and understand.

As described in Remark 1, removing the limitation on the environments is left for future work. We also plan to apply environmental bisimulations to (a substantial subset of) the Kell calculus so that we can provide a practical alternative to context bisimulations in a more expressive higher-order distributed process calculus. In the Kell calculus, locations are not transparent: one discriminates messages on the grounds of their origins (i.e. from a location above, below, or from the same level). For example, consider the (simplified) Kell processes $P = \overline{a}\langle M\rangle.!b[\overline{a}]$ and $Q = \overline{a}\langle N\rangle.!b[\overline{a}]$ where $M = \overline{a}$ and $N = 0$. They seem bisimilar assuming environmental bisimulations naively like those in this paper: intuitively, both P and Q can output (respectively M and N) to channel a, and their continuations are identical; passivation of spawned $l[M]$ and $l[N]$ for known location l would be immediately matched; finally, the output to channel a under l, turning P's spawned $l[M]$ into $l[0]$, could be matched by an output to a under b by Q's replicated $b[\overline{a}]$. However, M and N behave differently when observed from the same level (or below), for example as in $l[M \mid a(Y).\overline{ok}]$ and $l[N \mid a(Y).\overline{ok}]$ even under the presence of $!b[\overline{a}]$. More concretely, the context $[\cdot]_1 \,|\, a(X).c[X \,|\, a(Y).\overline{ok}]$ distinguishes P and Q, showing the unsoundness of such naive definition. This suggests that, to define sound environmental bisimulations in Kell-like calculi with non-transparent locations, we should require a stronger condition such as bisimilarity of M and N in the output clause. Developments on this idea are in progress.

References

1. Cardelli, L., Gordon, A.D.: Mobile ambients. In: Nivat, M. (ed.) FOSSACS 1998. LNCS, vol. 1378, pp. 140–155. Springer, Heidelberg (1998)
2. Hennessy, M., Riely, J.: Resource access control in systems of mobile agents. Information and Computation 173, 82–120 (2002)
3. Hewlett-Packard: Live migration across data centers and disaster tolerant virtualization architecture with HP storageworks cluster extension and Microsoft Hyper-V, http://h20195.www2.hp.com/V2/GetPDF.aspx/4AA2-6905ENW.pdf
4. Hildebrandt, T., Godskesen, J.C., Bundgaard, M.: Bisimulation congruences for Homer: a calculus of higher-order mobile embedded resources. Technical Report TR-2004-52, IT University of Copenhagen (2004)
5. Honda, K., Yoshida, N.: On reduction-based process semantics. Theoretical Computer Science 151(2), 437–486 (1995)
6. Howe, D.J.: Proving congruence of bisimulation in functional programming languages (1996)
7. Lenglet, S., Schmitt, A., Stefani, J.-B.: Normal bisimulations in calculi with passivation. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 257–271. Springer, Heidelberg (2009)
8. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press, Cambridge (1999)
9. Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD thesis, University of Edinburgh (1992)
10. Sangiorgi, D.: Bisimulation for higher-order process calculi. Information and Computation 131, 141–178 (1996)
11. Sangiorgi, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
12. Sangiorgi, D., Kobayashi, N., Sumii, E.: Environmental bisimulations for higher-order languages. In: Proceedings of the Twenty-Second Annual IEEE Symposium on Logic in Computer Science, pp. 293–302 (2007)
13. Sato, N., Sumii, E.: The higher-order, call-by-value applied pi-calculus. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 311–326. Springer, Heidelberg (2009)
14. Schmidt, D., Dhawan, P.: Live migration with Xen virtualization software, http://www.dell.com/downloads/global/power/ps2q06-20050322-Schmidt-OE.pdf
15. Schmitt, A., Stefani, J.-B.: The kell calculus: A family of higher-order distributed process calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)
16. Sumii, E., Pierce, B.C.: A bisimulation for dynamic sealing. Theoretical Computer Science 375(1-3), 169–192 (2007); Extended abstract appeared in Proceedings of 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 161–172 (2004)
17. Sumii, E., Pierce, B.C.: A bisimulation for type abstraction and recursion. Journal of the ACM 54, 1–43 (2007); Extended abstract appeared in Proceedings of 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 63–74 (2005)

Deriving Labels and Bisimilarity for Concurrent Constraint Programming[*]

Andrés Aristizábal[1], Filippo Bonchi[2], Catuscia Palamidessi[1], Luis Pino[1], and Frank Valencia[1]

[1] Comète, LIX, Laboratoire de l'École Polytechnique associé à l'INRIA
[2] CNRS - Laboratoire de l'Informatique du Parallélisme, ENS Lyon

Abstract. Concurrent constraint programming (ccp) is a well-established model for concurrency. Bisimilarity is one of the central reasoning techniques in concurrency. The standard definition of bisimilarity, however, is not completely satisfactory for ccp since it yields an equivalence that is too fine grained. By building upon recent foundational investigations, we introduce a labelled transition semantics and a novel notion of bisimilarity that is fully abstract w.r.t. the typical observational equivalence in ccp.

Introduction

Concurrency is concerned with systems of multiple computing agents, usually called *processes*, that interact with each other. *Process calculi* treat processes much like the λ-calculus treats computable functions. They provide a language in which processes are represented by terms, and computational steps are represented as transitions between them. These formalisms are equipped with equivalence relations that determine what processes are deemed indistinguishable. *Bisimilarity* is one of the main representative of these. It captures our intuitive notion of process equivalence; two processes are equivalent if they can match each other's moves. Furthermore, it provides an elegant co-inductive proof technique based on the notion of bisimulation.

Concurrent Constraint Programming (ccp) [26] is a well-established formalism that combines the traditional algebraic and operational view of process calculi with a declarative one based upon first-order logic. In ccp, processes interact by *adding* (or *telling*) and *asking* information (namely, constraints) in a medium (the *store*). Ccp is parametric in a *constraint system* indicating interdependencies (entailment) between constraints and providing for the specification of data types and other rich structures. The above features have recently attracted a renewed attention as witnessed by the works [23,9,5,4] on calculi exhibiting data-types, logic assertions as well as tell and ask operations.

There have been few attempts to define a notion of bisimilarity for ccp. The ones we are aware of are those in [26] and [19] upon which we build. These equivalences are not completely satisfactory: We shall see that the first one may tell apart processes with identical behaviour, while the second quantifies over all possible inputs from the environment, and hence it is not clear whether it can lead to a feasible proof technique.

[*] This work has been partially supported by the project ANR-09-BLAN-0169-01 PANDA and by the INRIA DRI Equipe Associée FORCES.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 138–152, 2011.

The goal of this paper is to define a notion of bisimilarity for ccp which will allow to benefit of the feasible proof and verification techniques typically associated with bisimilarity. Furthermore, we aim at studying the relationship between this equivalence and other existing semantic notions for ccp. In particular, its elegant denotational characterization based on closure operators [27] and the connection with logic [19].

Labels and Bisimilarity from Reductions. Bisimilarity relies on *labelled transitions*: each evolution step of a system is tagged by some information aimed at capturing the possible interactions of a process with the environment. Nowadays process calculi tend to adopt reduction semantics based on *unlabelled transitions* and *barbed congruence* [21]. The main drawback of this approach is that to verify barbed congruences it is often necessary to analyze the behaviour of processes under every context.

This scenario has motivated a novel stream of research [29,18,11,28,7,25,13,6] aimed at defining techniques for "deriving labels and bisimilarity" from unlabeled reduction semantics. The main intuition is that labels should represent the "minimal contexts allowing a process to reduce". The *theory of reactive systems* by Leifer and Milner [18] provides a formal characterization (by means of a categorical construction) of such "minimal contexts" and it focuses on the bisimilarity over transition systems labeled as: $P \xrightarrow{C} P'$ iff $C[P] \longrightarrow P'$ and C is the minimal context allowing such reduction.

In [7,6], it is argued that the above bisimilarity is often too fine grained and an alternative, coarser, notion of bisimilarity is provided. Intuitively, in the bisimulation game, each move (transition) $P \xrightarrow{C} P'$, has to be matched it with a move $C[Q] \longrightarrow Q'$.

Labels and Bisimilarity for ccp. The operational semantics of ccp is expressed by reductions between configurations of the form $\langle P, d\rangle \longrightarrow \langle P', d'\rangle$ meaning that the process P with store d may reduce to P' with store d'. From this semantics we shall derive a labeled transition system for ccp by exploiting the intuition of [29,18]. The transition $\langle P, d\rangle \xrightarrow{e} \langle P', d'\rangle$ means that e is a "minimal constraint" (from the environment) that needs to be added to d to reduce from $\langle P, d\rangle$ into $\langle P', d'\rangle$.

Similar ideas were already proposed in [26] but, the recent developments in [6] enlighten the way for obtaining a fully abstract equivalence. Indeed, the standard notion of bisimilarity defined on our labeled semantics can be seen as an instance of the one proposed in [18]. As for the bisimilarity in [26], it is too fine grained, i.e., it separates processes which are indistinguishable. Instead, the notion of bisimulation from [6] (instantiated to the case of ccp) is fully abstract with respect to the standard observational equivalence given in [27]. Our work can therefore be also regarded as a compelling application of the theory of reactive systems.

Contributions. We provide a labelled transition semantics and a novel notion of labelled bisimilarity for ccp by building upon the work in [26,6]. We also establish a strong correspondence with existing ccp notions by providing a fully-abstract characterization of a standard observable behaviour for *infinite* ccp processes: *The limits of fair computations*. From [27] this implies a fully-abstract correspondence with the closure operator denotational semantics of ccp. Therefore, this work provides ccp with a new co-inductive proof technique, coherent with the existing ones, for reasoning about process equivalence.

Missing proofs and additional examples are in [1].

1 Background

In this section we recall the syntax, the operational semantics and the observational equivalence of concurrent constraint programming (ccp). We begin with the notion of constraint system. We presuppose some basic knowledge of domain theory (see [2]).

1.1 Constraint Systems

The ccp model is parametric in a *constraint system* specifying the structure and interdependencies of the information that processes can ask and tell. Following [27,10], we regard a constraint system as a complete algebraic lattice structure in which the ordering $\sqsubseteq$ is the reverse of an entailment relation ($c \sqsubseteq d$ means that d contains "more information" than c, hence c can be derived from d). The top element false represents inconsistency, the bottom element true is the empty constraint, and the *least upper bound* (lub) $\sqcup$ represents the join of information.

Definition 1. *A* constraint system **C** *is a complete algebraic lattice* $(Con, Con_0, \sqsubseteq, \sqcup, true, false)$ *where Con (the set of constraints) is a partially ordered set w.r.t.* $\sqsubseteq$, Con_0 *is the subset of finite elements of Con,* $\sqcup$ *is the lub operation, and true, false are the least and greatest elements of Con, respectively.*

Recall that **C** is a *complete lattice* iff every subset of Con has a least upper bound in Con. An element $c \in Con$ is *finite* iff for any directed subset D of Con, $c \sqsubseteq \bigsqcup D$ implies $c \sqsubseteq d$ for some $d \in D$. **C** is *algebraic* iff each element $c \in Con$ is the least upper bound of the finite elements below c.

In order to model *hiding* of local variables and *parameter passing*, in [27] the notion of constraint system is enriched with *cylindrification operators* and *diagonal elements*, concepts borrowed from the theory of cylindric algebras (see [14]).

Let us consider a (denumerable) set of variables Var with typical elements $x, y, z, \ldots$ Define $\exists_{Var}$ as the family of operators $\exists_{Var} = \{\exists_x \mid x \in Var\}$ (*cylindric operators*) and D_{Var} as the set $D_{Var} = \{d_{xy} \mid x, y \in Var\}$ (*diagonal elements*).

A *cylindric constraint system* over a set of variables Var is a constraint system whose support set $Con \supseteq D_{Var}$ is closed under the cylindric operators $\exists_{Var}$ and quotiented by Axioms C1 $-$ C4, and whose ordering $\sqsubseteq$ satisfies Axioms C5 $-$ C7 :

$\mathbf{C1}.\ \exists_x \exists_y c = \exists_y \exists_x c$ $\quad\mathbf{C2}.\ d_{xx} = true$

$\mathbf{C3}.\ \mathit{if}\ z \neq x, y\ \mathit{then}\ d_{xy} = \exists_z (d_{xz} \sqcup d_{zy})$ $\quad\mathbf{C4}.\ \exists_x (c \sqcup \exists_x d) = \exists_x c \sqcup \exists_x d$

$\mathbf{C5}.\ \exists_x c \sqsubseteq c$ $\quad\mathbf{C6}.\ \mathit{if}\ c \sqsubseteq d\ \mathit{then}\ \exists_x c \sqsubseteq \exists_x d$ $\quad\mathbf{C7}.\ \mathit{if}\ x \neq y\ \mathit{then}\ c \sqsubseteq d_{xy} \sqcup \exists_x (c \sqcup d_{xy})$

where c, c_i, d indicate finite constraints, and $\exists_x c \sqcup d$ stands for $(\exists_x c) \sqcup d$. For our purposes, it is enough to think the operator $\exists_x$ as *existential quantifier* and the constraint d_{xy} as the equality $x = y$.

We assume notions of *free variable* and of *substitution* that satisfy the following conditions, where $c[y/x]$ is the constraint obtained by substituting x by y in c and $\mathit{fv}(c)$ is the set of free variables of c: (1) if $y \notin \mathit{fv}(c)$ then $(c[y/x])[x/y] = c$; (2) $(c \sqcup d)[y/x] = c[y/x] \sqcup d[y/x]$; (3) $x \notin \mathit{fv}(c[y/x])$; (4) $\mathit{fv}(c \sqcup d) = \mathit{fv}(c) \cup \mathit{fv}(d)$.

We now define the cylindric constraint system that will be used in all the examples.

Example 1 (The $\mathcal{S}$ Constraint System). Let $S = (\omega + 1, 0, \infty, =, <, \mathit{succ})$ be a first-order structure whose domain of interpretation is $\omega + 1 \stackrel{\mathtt{def}}{=} \omega \cup \{\infty\}$, i.e., the natural numbers extended with a top element ∞. The constant symbols 0 and ∞ are interpreted as zero and infinity, respectively. The symbols $=$, $<$ and succ are all binary predicates on $\omega + 1$. The symbol $=$ is interpreted as the identity relation. The symbol $<$ is interpreted as the set of pairs (n, m) s.t., $n \in \omega$, $m \in \omega + 1$ and n strictly smaller than m. The symbol succ is interpreted as the set of pairs (n, m) s.t., $n, m \in \omega$ and $m = n + 1$.

Let *Var* be an infinite set of variables. Let $\mathcal{L}$ be the logic whose formulae ϕ are: $\phi ::= t \mid \phi_1 \wedge \phi_2 \mid \exists_x \phi$ and $t ::= e_1 = e_2 \mid e_1 < e_2 \mid \mathit{succ}(e_1, e_2)$ where e_1 and e_2 are either 0 or ∞ or variables in Var. Note that formulas like $x = n$ or $x < n$ (for $n = 1, 2, \ldots$) do not belong to $\mathcal{L}$. A useful abbreviation to express them is $\mathit{succ}^n(x, y) \stackrel{\mathtt{def}}{=} \exists y_0 \ldots \exists y_n (\bigwedge_{0<i\leq n} \mathit{succ}(y_{i-1}, y_i) \wedge x = y_0 \wedge y = y_n)$. We use $x = n$ as shorthand for $\mathit{succ}^n(0, x)$ and $x < n$ as shorthand for $\exists_y (x < y \wedge y = n)$.

A variable assignment is a function $\mu : \mathit{Var} \longrightarrow \omega + 1$. We use $\mathcal{A}$ to denote the set of all assignments; $\mathcal{P}(X)$ to denote the powerset of a set X, $\emptyset$ the empty set and $\cap$ the intersection of sets. We use $\mathcal{M}(\phi)$ to denote the set of all assignments that *satisfy* the formula ϕ, where the definition of *satisfaction* is as expected.

We can now introduce a *constraint system* as follows: the set of constraints is $\mathcal{P}(\mathcal{A})$, and define $c \sqsubseteq d$ iff $c \supseteq d$. The constraint *false* is $\emptyset$, while *true* is $\mathcal{A}$. Given two constraints c and d, $c \sqcup d$ is the intersection $c \cap d$. By abusing the notation, we will often use a formula ϕ to denote the corresponding constraint, i.e., the set of all assignments satisfying ϕ. E.g. we use $1 < x \sqsubseteq 5 < x$ to mean $\mathcal{M}(1 < x) \sqsubseteq \mathcal{M}(5 < x)$.

From this structure, let us now define the *cylindric constraint system* $\mathcal{S}$ as follows. We say that an assignment μ' is *an x-variant* of μ if $\forall y \neq x$, $\mu(y) = \mu'(y)$. Given $x \in \mathit{Var}$ and $c \in \mathcal{P}(\mathcal{A})$, the constraint $\exists_x c$ is the set of assignments μ such that exists $\mu' \in c$ that is an x-variant of μ. The diagonal element d_{xy} is $x = y$. □

We make an assumption that will be pivotal in Section 3. Given a partial order $(C, \sqsubseteq)$, we say that c is strictly smaller than d (written $c \sqsubset d$) if $c \sqsubseteq d$ and $c \neq d$. We say that $(C, \sqsubseteq)$ is *well-founded* if there exists no infinite descending chains $\cdots \sqsubset c_n \sqsubset \cdots \sqsubset c_1 \sqsubset c_0$. For a set $A \subseteq C$, we say that an element $m \in A$ is *minimal* in A if for all $a \in A$, $a \not\sqsubset m$. We shall use $\mathit{min}(A)$ to denote the set of all minimal elements of A. Well-founded order and minimal elements are related by the following result.

Lemma 1. *Let $(C, \sqsubseteq)$ be a well-founded order and $A \subseteq C$. If $a \in A$, then $\exists m \in \mathit{min}(A)$ s.t., $m \sqsubseteq a$.*

In spite of its being a reasonable assumption, well-foundedness of $(\mathit{Con}, \sqsubseteq)$ is not usually required in the standard theory of ccp. We require it because the above lemma is fundamental for proving the completeness of labeled semantics (Lemma 5).

1.2 Syntax

Concurrent constraint programming (ccp) was proposed in [30] and then refined in [26,27]. We restrict ourselves to the summation-free fragment of ccp. The distinctive confluent nature of this fragment is necessary for showing that our notion of bisimilarity coincides with the observational equivalence for infinite ccp processes given in [27].

Definition 2. *Assume a cylindric constraint system* $\mathbf{C} = (Con, Con_0, \sqsubseteq, \sqcup, true, false)$ *over a set of variables* Var. *The ccp processes are given by the following syntax:*

$$P, Q \ldots \ ::= \ \mathbf{tell}(c) \mid \mathbf{ask}(c) \rightarrow P \mid P \parallel Q \mid \exists_x P \mid p(\boldsymbol{z})$$

where $c \in Con_0, x \in Var, \boldsymbol{z} \in Var^*$. *We use Proc to denote the set of all processes.*

Finite processes. Intuitively, the tell process $\mathbf{tell}(c)$ adds c to the global store. The addition is performed regardless the generation of inconsistent information. The ask process $\mathbf{ask}(c) \rightarrow P$ may execute P if c is entailed from the information in the store. The process $P \parallel Q$ stands for the *parallel execution* of P and Q; $\exists_x$ is a *hiding operator*, namely it indicates that in $\exists_x P$ the variable x is *local* to P. The occurrences of x in $\exists_x P$ are said to be bound. The bound variables of P, $bv(P)$, are those with a bound occurrence in P, and its free variables, $fv(P)$, are those with an unbound occurrence.

Infinite processes. To specify infinite behaviour, ccp provides parametric process definitions. A process $p(\boldsymbol{z})$ is said to be a *procedure call* with identifier p and actual parameters $\boldsymbol{z}$. We presuppose that for each procedure call $p(z_1 \ldots z_m)$ there exists a unique *procedure definition* possibly *recursive*, of the form $p(x_1 \ldots x_m) \stackrel{\texttt{def}}{=} P$ where $fv(P) \subseteq \{x_1, \ldots, x_m\}$. Furthermore we require recursion to be *guarded*: I.e., each procedure call within P must occur within an ask process. The behaviour of $p(z_1 \ldots z_m)$ is that of $P[z_1 \ldots z_m / x_1 \ldots x_m]$, i.e., P with each x_i replaced with z_i (applying α-conversion to avoid clashes). We shall use D to denote the set of all process definitions.

Although we have not defined yet the semantics of processes, we find it instructive to illustrate the above operators with the following example. Recall that we shall use $\mathcal{S}$ in Ex. 1 as the underlying constraint system in all examples.

Example 2. Consider the following (family of) process definitions.

$$up_n(x) \stackrel{\texttt{def}}{=} \exists_y (\mathbf{tell}(y = n) \parallel \mathbf{ask}\ (y = n) \ \rightarrow \ \ up(x, y))$$

$$up(x, y) \stackrel{\texttt{def}}{=} \exists_{y'} (\mathbf{tell}(y < x \wedge succ^2(y, y')) \parallel \mathbf{ask}(y < x \wedge succ^2(y, y')) \rightarrow up(x, y'))$$

Intuitively, $up_n(x)$, where n is a natural number, specifies that x should be greater than any natural number (i.e., $x = \infty$ since $x \in \omega + 1$) by telling (adding to the global store) the constraints $y_{i+1} = y_i + 2$ and $y_i < x$ for some $y_0, y_1, \ldots$ with $y_0 = n$. The process $up_0(x) \parallel \mathbf{ask}(42 < x) \rightarrow \mathbf{tell}(z = 0)$, can set $z = 0$ when it infers from the global store that $42 < x$. (This inference is only possible after the 22^{nd} call to *up*.) □

1.3 Reduction Semantics

To describe the evolution of processes, we extend the syntax by introducing a process $\mathbf{stop}$ representing successful termination, and a process $\exists_x^e P$ representing the evolution of a process of the form $\exists_x P$, where e is the local information (*local store*) produced during this evolution. The process $\exists_x P$ can be seen as a particular case of $\exists_x^e P$: it represents the situation in which the local store is empty. Namely, $\exists_x P = \exists_x^{true} P$.

Table 1. Reduction semantics for ccp (The symmetric Rule for R3 is omitted)

R1	$\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{stop}, d \sqcup c\rangle$	R2	$\dfrac{c \sqsubseteq d}{\langle \mathbf{ask}\ (c) \ \rightarrow\ P, d\rangle \longrightarrow \langle P, d\rangle}$
R3	$\dfrac{\langle P, d\rangle \longrightarrow \langle P', d'\rangle}{\langle P \parallel Q, d\rangle \longrightarrow \langle P' \parallel Q, d'\rangle}$	R4	$\dfrac{\langle P, e \sqcup \exists_x d\rangle \longrightarrow \langle P', e' \sqcup \exists_x d\rangle}{\langle \exists_x^e P, d\rangle \longrightarrow \langle \exists_x^{e'} P', d \sqcup \exists_x e'\rangle}$
R5	$\dfrac{\langle P[\boldsymbol{z}/\boldsymbol{x}], d\rangle \longrightarrow \gamma'}{\langle p(\boldsymbol{z}), d\rangle \longrightarrow \gamma'}$ where $p(\boldsymbol{x}) \overset{\text{def}}{=} P$ is a process definition in D		

A configuration is a pair $\langle P, d\rangle$ representing the state of a system; d is a constraint representing the global store, and P is a process in the extended syntax. We use *Conf* with typical elements $\gamma, \gamma', \ldots$ to denote the set of configurations. The operational model of ccp can be described formally in the SOS style by means of the transition relation between configurations $\longrightarrow\ \subseteq \mathit{Conf} \times \mathit{Conf}$ defined in Table 1.

Rules R1-R3 and R5 are easily seen to realize the above process intuitions. Rule R4 is somewhat more involved. Here, we show an instructive example of its use.

Example 3. We have the below reduction of $P = \exists_x^e(\mathbf{ask}\ (y > 1) \ \rightarrow\ Q)$ where the local store is $e = x < 1$, and the global store $d' = d \sqcup \alpha$ with $d = y > x$, $\alpha = x > 1$.

$$\begin{array}{ll} \text{R2} & \dfrac{(y > 1) \sqsubseteq e \sqcup \exists_x d'}{\langle \mathbf{ask}\ (y > 1) \ \rightarrow\ Q, e \sqcup \exists_x d'\rangle \longrightarrow \langle Q, e \sqcup \exists_x d'\rangle} \\ \text{R4} & \overline{\qquad\langle P, d'\rangle \longrightarrow \langle \exists_x^e Q, d' \sqcup \exists_x e\rangle\qquad} \end{array}$$

Note that the x in d' is hidden, by using existential quantification in the reduction obtained by Rule R2. This expresses that the x in d' is different from the one bound by the local process. Otherwise an inconsistency would be generated (i.e., $(e \sqcup d') = \mathit{false}$). Rule R2 applies since $(y > 1) \sqsubseteq e \sqcup \exists_x d'$. Note that the free x in $e \sqcup \exists_x d'$ is hidden in the global store to indicate that is different from the global x. □

1.4 Observational Equivalence

The notion of fairness is central to the definition of observational equivalence for ccp. To define fair computations, we introduce the notions of *enabled* and *active* processes, following [12]. Observe that any transition is generated either by a process $\mathbf{tell}(c)$ or by a process $\mathbf{ask}\ (c) \ \rightarrow\ Q$. We say that a process P is *active* in a transition $t = \gamma \longrightarrow \gamma'$ if it generates such transition; i.e if there exist a derivation of t where R1 or R2 are used to produce a transition of the form $\langle P, d\rangle \longrightarrow \gamma''$. Moreover, we say that a process P is *enabled* in a configuration γ if there exists γ' such that P is active in $\gamma \longrightarrow \gamma'$.

Definition 3. *A computation $\gamma_0 \longrightarrow \gamma_1 \longrightarrow \gamma_2 \longrightarrow \ldots$ is said to be* fair *if for each process enabled in some γ_i there exists $j \geq i$ such that the process is active in γ_j.*

Note that a finite fair computation is guaranteed to be *maximal*, namely no outgoing transitions are possible from its last configuration.

The standard notion of observables for ccp are the *results* computed by a process for a given initial store. The result of a computation is defined as the least upper bound of all the stores occurring in the computation, which, due to the monotonic properties of ccp, form an increasing chain. More formally, given a finite or infinite computation ξ of the form $\langle Q_0, d_0\rangle \longrightarrow \langle Q_1, d_1\rangle \longrightarrow \langle Q_2, d_2\rangle \longrightarrow \ldots$ the result of ξ, denoted by $\mathit{Result}(\xi)$, is the constraint $\bigsqcup_i d_i$. Note that for a finite computation the result coincides with the store of the last configuration.

The following theorem from [27] states that all the fair computations of a configuration have the same result (due to fact that summation-free ccp is confluent).

Theorem 1 (from [27]). *Let γ be a configuration and let ξ_1 and ξ_2 be two computations of γ. If ξ_1 and ξ_2 are fair, then $\mathit{Result}(\xi_1) = \mathit{Result}(\xi_2)$.*

This allows us to set $\mathit{Result}(\gamma) \stackrel{\texttt{def}}{=} \mathit{Result}(\xi)$ for any fair computation ξ of γ.

Definition 4. (Observational equivalence) *Let $\mathcal{O} : \mathit{Proc} \rightarrow \mathit{Con}_0 \rightarrow \mathit{Con}$ be given by $\mathcal{O}(P)(d) = \mathit{Result}(\langle P, d\rangle)$. We say that P and Q are observational equivalent, written $P \sim_o Q$, iff $\mathcal{O}(P) = \mathcal{O}(Q)$.*

Example 4. Consider the processes $P = \mathit{up}_0(x) \parallel \mathit{up}_1(y)$ and $Q = \exists_z(\mathbf{tell}(z = 0) \parallel \mathbf{ask}(z = 0) \rightarrow \mathit{fairup}(x, y, z))$ with up_0 and up_1 as in Ex. 2 and $\mathit{fairup}(x, y, z) \stackrel{\texttt{def}}{=}$

$$\exists_{z'}(\mathbf{tell}(z < x \wedge \mathit{succ}(z, z')) \parallel \mathbf{ask}\ ((z < x) \wedge \mathit{succ}(z, z')) \rightarrow \mathit{fairup}(y, x, z')))$$

Let $s(\gamma)$ denote the store in the configuration γ. For every infinite computation $\xi : \langle P, \mathit{true}\rangle = \gamma_0 \longrightarrow \gamma_1 \longrightarrow \ldots$ with $(1 < y) \not\sqsubseteq s(\gamma_i)$ for each $i \geq 0$, ξ is not fair and $\mathit{Result}(\xi) = (x = \infty)$. In contrast, every infinite computation $\xi : \langle Q, \mathit{true}\rangle = \gamma_0 \longrightarrow \gamma_1 \longrightarrow \ldots$ is fair and $\mathit{Result}(\xi) = (x = \infty \wedge y = \infty)$. Nevertheless, under our fair observations, P and Q are indistinguishable, i.e., $\mathcal{O}(P) = \mathcal{O}(Q)$. □

2 Saturated Bisimilarity for ccp

We introduce a notion of bisimilarity in terms of (unlabelled) reductions and *barbs* and we prove that this equivalence is fully abstract w.r.t. observational equivalence.

2.1 Saturated Barbed Bisimilarity

Barbed equivalences have been introduced in [21] for CCS, and have become the standard behavioural equivalences for formalisms equipped with unlabeled reduction semantics. Intuitively, *barbs* are basic observations (predicates) on the states of a system.

The choice of the "right" barbs is a crucial step in the barbed approach, and it is usually not a trivial task. For example, in synchronous languages like CCS or π-calculus both the inputs and the outputs are considered as barbs, (see e.g. [21,20]), while in the asynchronous variants only the outputs (see e.g. [3]). Even several works (e.g. [24,15]) have proposed abstract criteria for defining "good" barbs.

We shall take as barbs all the finite constraints in Con_0. This choice allows us to introduce a barbed equivalence (Def. 7) that coincides with the standard observational equivalence (Def. 4). It is worth to note that in $\sim_o$, the observables are all the constraints in Con and not just the finite ones.

We say that $\gamma = \langle P, d\rangle$ *satisfies* the barb c, written $\gamma \downarrow_c$, iff $c \sqsubseteq d$; γ *weakly satisfies* the barb c, written $\gamma \Downarrow_c$, iff $\gamma \longrightarrow^* \gamma'$ and $\gamma' \downarrow_c$[1].

Definition 5. *(Barbed bisimilarity) A barbed bisimulation is a symmetric relation $\mathcal{R}$ on configurations such that whenever $(\gamma_1, \gamma_2) \in \mathcal{R}$:*

(i) if $\gamma_1 \downarrow_c$ then $\gamma_2 \downarrow_c$,
(ii) if $\gamma_1 \longrightarrow \gamma_1'$ then there exists γ_2' such that $\gamma_2 \longrightarrow \gamma_2'$ and $(\gamma_1', \gamma_2') \in \mathcal{R}$.

We say that γ_1 and γ_2 are barbed bisimilar, written $\gamma_1 \dot{\sim}_b \gamma_2$, if there exists a barbed bisimulation $\mathcal{R}$ s.t. $(\gamma_1, \gamma_2) \in \mathcal{R}$. We write $P \dot{\sim}_b Q$ iff $\langle P, true\rangle \dot{\sim}_b \langle Q, true\rangle$.

Congruence characterization. One can verify that $\dot{\sim}_b$ is an equivalence. However, it is not a *congruence*; i.e., it is not preserved under arbitrary contexts. A context C is a term with a hole $[-]$ s.t., replacing it with a process P yields a process term $C[P]$. E.g., $C = \mathbf{tell}(c) \parallel [-]$ and $C[\mathbf{tell}(d)] = \mathbf{tell}(c) \parallel \mathbf{tell}(d)$.

Example 5. Let us consider the context $C = \mathbf{tell}(a) \parallel [-]$ and the processes $P = \mathbf{ask}\ (b) \rightarrow \mathbf{tell}(d)$ and $Q = \mathbf{ask}\ (c) \rightarrow \mathbf{tell}(d)$ with $a, b, c, d \neq true, b \sqsubseteq a$ and $c \not\sqsubseteq a$. We have $\langle P, true\rangle \dot{\sim}_b \langle Q, true\rangle$ because both configurations cannot move and they only satisfy the barb $true$. But $\langle C[P], true\rangle \not\dot{\sim}_b \langle C[Q], true\rangle$, because the former can perform three transitions (in sequence), while the latter only one. □

An elegant solution to modify bisimilarity for obtaining a congruence has been introduced in [22] for the case of weak bisimilarity in CCS. This work has inspired the introduction of *saturated bisimilarity* [7] (and its extension to the barbed approach [6]). The basic idea is simple: saturated bisimulations are closed w.r.t. all the possible contexts of the language. In the case of ccp, it is enough to require that bisimulations are *upward closed* as in condition (iii) below.

Definition 6. *(Saturated barbed bisimilarity). A saturated barbed bisimulation is a symmetric relation $\mathcal{R}$ on configurations such that whenever $(\gamma_1, \gamma_2) \in \mathcal{R}$ with $\gamma_1 = \langle P, d\rangle$ and $\gamma_2 = \langle Q, e\rangle$:*

(i) if $\gamma_1 \downarrow_c$ then $\gamma_2 \downarrow_c$,
(ii) if $\gamma_1 \longrightarrow \gamma_1'$ then there exists γ_2' such that $\gamma_2 \longrightarrow \gamma_2'$ and $(\gamma_1', \gamma_2') \in \mathcal{R}$,
(iii) for every $a \in Con_0$, $(\langle P, d \sqcup a\rangle, \langle Q, e \sqcup a\rangle) \in \mathcal{R}$.

We say that γ_1 and γ_2 are saturated barbed bisimilar, written $\gamma_1 \dot{\sim}_{sb} \gamma_2$, if there exists a saturated barbed bisimulation $\mathcal{R}$ s.t. $(\gamma_1, \gamma_2) \in \mathcal{R}$. We write $P \dot{\sim}_{sb} Q$ iff $\langle P, true\rangle \dot{\sim}_{sb} \langle Q, true\rangle$.

[1] As usual, $\longrightarrow^*$ denotes the reflexive and transitive closure of $\longrightarrow$.

Definition 7. *(Weak saturated barbed bisimilarity). Weak saturated barbed bisimilarity ($\dot{\approx}_{sb}$) is obtained from Def. 6 by replacing $\longrightarrow$ with $\longrightarrow^*$ and $\downarrow_c$ with $\Downarrow_c$.*

Since $\dot{\sim}_{sb}$ is itself a saturated barbed bisimulation, it is obvious that it is upward closed. This fact also guarantees that it is a congruence w.r.t. all the contexts of ccp: a context C can modify the behaviour of a configuration γ only by adding constraints to its store. The same holds for $\dot{\approx}_{sb}$.

2.2 Correspondence with Observational Equivalence

We now show that $\dot{\approx}_{sb}$ coincides with the observational equivalence $\sim_o$. From [27] it follows that $\dot{\approx}_{sb}$ coincides with the standard denotational semantics for ccp.

First, we recall some basic facts from domain theory central to our proof. Two (possibly infinite) chains $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \ldots$ and $e_0 \sqsubseteq e_1 \sqsubseteq \cdots \sqsubseteq e_n \sqsubseteq \ldots$ are said to be *cofinal* if for all d_i there exists an e_j such that $d_i \sqsubseteq e_j$ and, viceversa, for all e_i there exists a d_j such that $e_i \sqsubseteq d_j$.

Lemma 2. *Let $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \ldots$ and $e_0 \sqsubseteq e_1 \sqsubseteq \cdots \sqsubseteq e_n \sqsubseteq \ldots$ be two chains. (1) If they are cofinal, then they have the same limit, i.e., $\bigsqcup d_i = \bigsqcup e_i$. (2) If the elements of the chains are* finite *and $\bigsqcup d_i = \bigsqcup e_i$, then the two chains are cofinal.*

In the proof, we will show that the stores of any pairs of *fair* computations of equivalent processes form pairs of cofinal chains. First, the following result relates weak barbs and fair computations.

Lemma 3. *Let $\langle P_0, d_0\rangle \longrightarrow \langle P_1, d_1\rangle \longrightarrow \ldots \longrightarrow \langle P_n, d_n\rangle \longrightarrow \ldots$ be a (possibly infinite) fair computation. If $\langle P_0, d_0\rangle \Downarrow_c$ then there exist a store d_i (in the above computation) such that $c \sqsubseteq d_i$.*

Theorem 2. *$P \sim_o Q$ if and only if $P \dot{\approx}_{sb} Q$.*

Proof. The proof proceeds as follows:

- *From $\dot{\approx}_{sb}$ to $\sim_o$.* Suppose that $\langle P, \mathit{true}\rangle \ \dot{\approx}_{sb} \ \langle Q, \mathit{true}\rangle$ and take a finite input $b \in \mathit{Con}_0$. Let

$$\langle P, b\rangle \longrightarrow \langle P_0, d_0\rangle \longrightarrow \langle P_1, d_1\rangle \longrightarrow \ldots \longrightarrow \langle P_n, d_n\rangle \longrightarrow \ldots$$
$$\langle Q, b\rangle \longrightarrow \langle Q_0, e_0\rangle \longrightarrow \langle Q_1, e_1\rangle \longrightarrow \ldots \longrightarrow \langle Q_n, e_n\rangle \longrightarrow \ldots$$

  be two fair computations. Since $\dot{\approx}_{sb}$ is upward closed, $\langle P, b\rangle \ \dot{\approx}_{sb} \ \langle Q, b\rangle$ and thus, for all d_i, $\langle Q, b\rangle \Downarrow_{d_i}$. By Lemma 3, it follows that there exists an e_j (in the above computation) such that $d_i \sqsubseteq e_j$. Analogously, for all e_i there exists a d_j such that $e_i \sqsubseteq d_j$. Then the two chains are cofinal and by Lemma 2.1, it holds that $\bigsqcup d_i = \bigsqcup e_i$, that means $\mathcal{O}(P)(b) = \mathcal{O}(Q)(b)$.
- From $\sim_o$ to $\dot{\approx}_{sb}$. Suppose that $P \sim_o Q$. We first show that for all $b \in \mathit{Con}_0$, $\langle P, b\rangle$ and $\langle Q, b\rangle$ satisfy the same weak barbs. Let

$$\langle P, b\rangle \longrightarrow \langle P_0, d_0\rangle \longrightarrow \langle P_1, d_1\rangle \longrightarrow \ldots \longrightarrow \langle P_n, d_n\rangle \longrightarrow \ldots$$
$$\langle Q, b\rangle \longrightarrow \langle Q_0, e_0\rangle \longrightarrow \langle Q_1, e_1\rangle \longrightarrow \ldots \longrightarrow \langle Q_n, e_n\rangle \longrightarrow \ldots$$

  be two (possibly infinite) fair computations. Since $P \sim_o Q$, then $\bigsqcup d_i = \bigsqcup e_i$. Since all the stores of computations are finite constraints (only finite constraints can be added to the store), then by Lemma 2.2, it holds that for all d_i there exists an e_j such that $d_i \sqsubseteq e_j$. Now suppose that $\langle P, b\rangle \Downarrow_c$. By Lemma 3, it holds that there exists a d_i (in the above computation) such that $c \sqsubseteq d_i$. Thus $c \sqsubseteq d_i \sqsubseteq e_j$ that means $\langle Q, b\rangle \Downarrow_c$.

With this observation it is easy to prove that

$$\mathcal{R} = \{(\gamma_1, \gamma_2) \mid \exists b \text{ s.t. } \langle P, b\rangle \longrightarrow^* \gamma_1,\ \langle Q, b\rangle \longrightarrow^* \gamma_2\}$$

is a weak saturated barbed bisimulation (Def. 7). Take $(\gamma_1, \gamma_2) \in \mathcal{R}$.

If $\gamma_1 \Downarrow_c$ then $\langle P, b\rangle \Downarrow_c$ and, by the above observation, $\langle Q, b\rangle \Downarrow_c$. Since ccp is confluent, also $\gamma_2 \Downarrow_c$.

The fact that $\mathcal{R}$ is closed under $\longrightarrow^*$ is evident from the definition of $\mathcal{R}$. While for proving that $\mathcal{R}$ is upward-closed take $\gamma_1 = \langle P', d'\rangle$ and $\gamma_2 = \langle Q', e'\rangle$. It is easy to see that for all $a \in \mathit{Con}_0$, $\langle P, b \sqcup a\rangle \longrightarrow^* \langle P', d' \sqcup a\rangle$ and $\langle Q, b \sqcup a\rangle \longrightarrow^* \langle Q', e' \sqcup a\rangle$. Thus, by definition of $\mathcal{R}$, $(\langle P', d' \sqcup a\rangle, \langle Q', e' \sqcup a\rangle) \in \mathcal{R}$. □

3 Labeled Semantics

Although $\dot{\sim}_{sb}$ is fully abstract, it is at some extent unsatisfactory because of the upward-closure (namely, the quantification over all possible $a \in \mathit{Con}_0$ in condition (iii)) of Def. 6. We shall deal with this by refining the notion of transition by adding to it a label that carries additional information about the constraints that cause the reduction.

Labelled Transitions. Intuitively, we will use transitions of the form

$$\langle P, d\rangle \xrightarrow{\alpha} \langle P', d'\rangle$$

where label α represents a *minimal* information (from the environment) that needs to be added to the store d to evolve from $\langle P, d\rangle$ into $\langle P', d'\rangle$, i.e., $\langle P, d \sqcup \alpha\rangle \longrightarrow \langle P', d'\rangle$. From a more abstract perspective, our labeled semantic accords with the proposal of [29,18] of looking at "labels as the minimal contexts allowing a reduction". In our setting we take as contexts only the constraints that can be added to the store.

The Rules. The labelled transition $\longrightarrow \subseteq \mathit{Conf} \times \mathit{Con}_0 \times \mathit{Conf}$ is defined by the rules in Table 3. We shall only explain rules LR2 and LR4 as the other rules are easily seen to realize the above intuition and follow closely the corresponding ones in Table 1.

The rule LR2 says that $\langle \mathbf{ask}\ (c) \rightarrow P, d\rangle$ can evolve to $\langle P, d \sqcup \alpha\rangle$ if the environment provides a minimal constraint α that added to the store d entails c, i.e., $\alpha \in \min\{a \in \mathit{Con}_0 \mid c \sqsubseteq d \sqcup a\ \}$. Note that assuming that $(\mathit{Con}, \sqsubseteq)$ is well-founded (Sec. 1.1) is necessary to guarantee that α exists whenever $\{a \in \mathit{Con}_0 \mid c \sqsubseteq d \sqcup a\ \}$ is not empty.

To give an intuition about LR4, it may be convenient to first explain why a naive adaptation of the analogous reduction rule R4 in Table 1 would not work. One may be tempted to define the rule for the local case, by analogy to the labelled local rules in other process calculi (e.g., the π-calculus) and R4, as follows:

$$(*)\ \frac{\langle P, e \sqcup \exists_x d\rangle \xrightarrow{\alpha} \langle Q, e' \sqcup \exists_x d\rangle}{\langle \exists_x^e P, d\rangle \xrightarrow{\alpha} \langle \exists_x^{e'} Q, d \sqcup \exists_x e'\rangle} \quad \text{where } x \notin \mathit{fv}(\alpha)$$

Table 2. Labelled Transitions (The symmetric Rule for LR3 is omitted)

$$\text{LR1}\quad \langle \mathbf{tell}(c), d\rangle \xrightarrow{\mathit{true}} \langle \mathbf{stop}, d \sqcup c\rangle$$

$$\text{LR2}\quad \frac{\alpha \in \min\{a \in \mathit{Con}_0 \mid c \sqsubseteq d \sqcup a\}}{\langle \mathbf{ask}\ (c)\ \rightarrow\ P, d\rangle \xrightarrow{\alpha} \langle P, d \sqcup \alpha\rangle} \qquad \text{LR3}\quad \frac{\langle P, d\rangle \xrightarrow{\alpha} \langle P', d'\rangle}{\langle P \parallel Q, d\rangle \xrightarrow{\alpha} \langle P' \parallel Q, d'\rangle}$$

$$\text{LR4}\ \frac{\langle P[z/x], e[z/x] \sqcup d\rangle \xrightarrow{\alpha} \langle P', e' \sqcup d \sqcup \alpha\rangle}{\langle \exists_x^e P, d\rangle \xrightarrow{\alpha} \langle \exists_x^{e'[x/z]} P'[x/z], \exists_x(e'[x/z]) \sqcup d \sqcup \alpha\rangle}\ x \notin \mathit{fv}(e'), z \notin \mathit{fv}(P) \cup \mathit{fv}(e \sqcup d \sqcup \alpha)$$

$$\text{LR5}\quad \frac{\langle P[\boldsymbol{z}/\boldsymbol{x}], d\rangle \xrightarrow{\alpha} \gamma'}{\langle p(\boldsymbol{z}), d\rangle \xrightarrow{\alpha} \gamma'} \quad \text{where } p(\boldsymbol{x}) \overset{\mathtt{def}}{=} P \text{ is a process definition in } D$$

This rule however is not "complete" (in the sense of Lemma 5 below) as it does not derive all the transitions we wish to have.

Example 6. Let P as in Ex. 3, i.e., $P = \exists_x^{x<1}(\mathbf{ask}\ (y > 1)\ \rightarrow\ Q)$ and $d = y > x$. Note that $\alpha = x > 1$ is a minimal constraint that added to d enables a reduction from P. In Ex. 3 we obtained the transition: $\langle P, d \sqcup \alpha\rangle \longrightarrow \langle \exists_x^{x<1} Q, d \sqcup \alpha \sqcup \exists_x(x < 1)\rangle$ Thus, we would like to have a transition from $\langle P, d\rangle$ labelled with α. But such a transition cannot be derived with Rule (*) above since $x \in \mathit{fv}(\alpha)$. □

Now, besides the side condition, another related problem with Rule (*) arises from the existential quantification $\exists_x d$ in the antecedent transition $\langle P, e \sqcup \exists_x d\rangle \xrightarrow{\alpha} \langle Q, e' \sqcup \exists_x d\rangle$. This quantification hides the effect of d on x and thus is not possible to identify the x in α with the x in d. The information from the environment α needs to be added to the global store d, hence the occurrences of x in both d and α must be identified. Notice that dropping the existential quantification of x in d in the antecedent transition does identify the occurrences of x in d with those in α but also with those in the local store e thus possibly generating variable clashes.

The rule LR4 in Table 2 solves the above-mentioned issues by using in the antecedent derivation a fresh variable z that acts as a substitute for the free occurrences of x in P and its local store e. (Recall that $T[z/x]$ represents T with x replaced with z). This way we identify with z the free occurrences of x in P and e and avoid clashes with those in α and d. E.g., for the process defined in the Ex.6, using LR4 (and LR2) one can derive

$$\frac{\langle \mathbf{ask}\ (y > 1)\ \rightarrow\ Q[z/x], z < 1 \sqcup y > x\rangle \xrightarrow{x>1} \langle Q[z/x], z < 1 \sqcup y > x \sqcup x > 1\rangle}{\langle \exists_x^{x<1}(\mathbf{ask}\ (y > 1)\ \rightarrow\ Q), y > x\rangle \xrightarrow{x>1} \langle \exists_x^{x<1} Q, \exists_x(x < 1) \sqcup y > x \sqcup x > 1\rangle}$$

The labeled semantics is *sound* and *complete* w.r.t. the unlabeled one. Soundness states that $\langle P, d\rangle \xrightarrow{\alpha} \langle P', d'\rangle$ corresponds to our intuition that if α is added to d, P can reach $\langle P', d'\rangle$. Completeness states that if we add a to (the store in) $\langle P, d\rangle$ and reduce to $\langle P', d'\rangle$, it exists a minimal information $\alpha \sqsubseteq a$ such that $\langle P, d\rangle \xrightarrow{\alpha} \langle P', d''\rangle$ with $d'' \sqsubseteq d'$.

Lemma 4. *(Soundness). If $\langle P, d\rangle \xrightarrow{\alpha} \langle P', d'\rangle$ then $\langle P, d \sqcup \alpha\rangle \longrightarrow \langle P', d'\rangle$.*

Lemma 5. *(Completeness). If $\langle P, d \sqcup a\rangle \longrightarrow \langle P', d'\rangle$ then $\exists \alpha, b$ s.t. $\langle P, d\rangle \xrightarrow{\alpha} \langle P', d''\rangle$ and $\alpha \sqcup b = a$, $d'' \sqcup b = d'$.*

Corollary 1. *$\langle P, d\rangle \xrightarrow{true} \langle P', d'\rangle$ if and only if $\langle P, d\rangle \longrightarrow \langle P', d'\rangle$.*

By virtue of the above, we will write $\longrightarrow$ to mean $\xrightarrow{true}$.

4 Strong and Weak Bisimilarity

Having defined our labelled transitions for ccp, we now proceed to define an equivalence that characterizes $\dot{\sim}_{sb}$ without the upward closure condition.

When defining bisimilarity over a labeled transition system, barbs are not usually needed because they can be somehow inferred by the labels of the transitions. For example in CCS, $P \downarrow_a$ iff $P \xrightarrow{a}$. The case of ccp is different: barbs cannot be removed from the definition of bisimilarity because they cannot be inferred by the transitions. In order to remove barbs from ccp, we could have inserted labels showing the store of processes (as in [26]) but this would have betrayed the philosophy of "labels as minimal constraints". Then, we have to define bisimilarity as follows.

Definition 8. *(Syntactic bisimilarity). A syntactic bisimulation is a symmetric relation $\mathcal{R}$ on configurations such that whenever $(\gamma_1, \gamma_2) \in \mathcal{R}$:*

(i) if $\gamma_1 \downarrow_c$ then $\gamma_2 \downarrow_c$,
(ii) if $\gamma_1 \xrightarrow{\alpha} \gamma_1'$ then $\exists \gamma_2'$ such that $\gamma_2 \xrightarrow{\alpha} \gamma_2'$ and $(\gamma_1', \gamma_2') \in \mathcal{R}$.

We say that γ_1 and γ_2 are syntactically bisimilar, written $\gamma_1 \sim_S \gamma_2$, if there exists a syntactic bisimulation $\mathcal{R}$ such that $(\gamma_1, \gamma_2) \in \mathcal{R}$.

We called the above bisimilarity "syntactic", because it does not take into account the "real meaning" of the labels. This equivalence coincides with the one in [26] (apart from the fact that in the latter, barbs are implicitly observed by the transitions) and, from a more general point of view can be seen as an instance of bisimilarity in [18] (by identifying contexts with constraints). In [7], it is argued that the equivalence in [18] is often over-discriminating. This is also the case of ccp, as illustrated in the following.

Example 7. Let $P = \mathbf{ask}\ (x < 10) \rightarrow \mathbf{tell}(y = 0)$ and $Q = \mathbf{ask}\ (x < 5) \rightarrow \mathbf{tell}(y = 0)$. The configurations $\gamma_1 = \langle P \parallel Q, true\rangle$ and $\gamma_2 = \langle P \parallel P, true\rangle$ are not equivalent according to $\sim_S$. Indeed $\gamma_1 \xrightarrow{x<10} \gamma_1' \xrightarrow{x<5} \gamma_1''$, while γ_2 after performing $\gamma_2 \xrightarrow{x<10} \gamma_2'$ can only perform $\gamma_2' \xrightarrow{true} \gamma_2''$. However $\gamma_1 \dot{\sim}_{sb} \gamma_2$. □

To obtain coarser equivalence (coinciding with $\dot{\sim}_{sb}$), we define the following.

Definition 9. *(Strong bisimilarity). A strong bisimulation is a symmetric relation $\mathcal{R}$ on configurations such that whenever $(\gamma_1, \gamma_2) \in \mathcal{R}$ with $\gamma_1 = \langle P, d\rangle$ and $\gamma_2 = \langle Q, e\rangle$:*

(i) if $\gamma_1 \downarrow_c$ *then* $\gamma_2 \downarrow_c$*,*
(ii) if $\gamma_1 \xrightarrow{\alpha} \gamma_1'$ *then* $\exists \gamma_2'$ *s.t.* $\langle Q, e \sqcup \alpha \rangle \longrightarrow \gamma_2'$ *and* $(\gamma_1', \gamma_2') \in \mathcal{R}$*.*

We say that γ_1 *and* γ_2 *are strongly bisimilar, written* $\gamma_1 \dot{\sim} \gamma_2$*, if there exists a strong bisimulation* $\mathcal{R}$ *such that* $(\gamma_1, \gamma_2) \in \mathcal{R}$*.*

To give some intuition about the above definition, let us recall that in $\langle P, d \rangle \xrightarrow{\alpha} \gamma'$ the label α represents *minimal* information from the environment that needs to be added to the store d to evolve from $\langle P, d \rangle$ into γ'. We do not require the transitions from $\langle Q, e \rangle$ to match α. Instead (ii) requires something weaker: If α is added to the store e, it should be possible to reduce into some γ'' that it is in bisimulation with γ'. This condition is weaker because α may not be a minimal information allowing a transition from $\langle Q, e \rangle$ into a γ'' in the bisimulation, as shown in the previous example.

Definition 10. *(Weak bisimilarity). A weak bisimulation is a symmetric relation* $\mathcal{R}$ *on configurations such that whenever* $(\gamma_1, \gamma_2) \in \mathcal{R}$ *with* $\gamma_1 = \langle P, d \rangle$ *and* $\gamma_2 = \langle Q, e \rangle$ *:*

(i) if $\gamma_1 \downarrow_c$ *then* $\gamma_2 \Downarrow_c$*,*
(ii) if $\gamma_1 \xrightarrow{\alpha} \gamma_1'$ *then* $\exists \gamma_2'$ *s.t.* $\langle Q, e \sqcup \alpha \rangle \longrightarrow^* \gamma_2'$ *and* $(\gamma_1', \gamma_2') \in \mathcal{R}$*.*

We say that γ_1 *and* γ_2 *are weakly bisimilar, written* $\gamma_1 \dot{\approx} \gamma_2$*, if there exists a weak bisimulation* $\mathcal{R}$ *such that* $(\gamma_1, \gamma_2) \in \mathcal{R}$*.*

Example 8. We can show that $\mathbf{tell}(\mathit{true}) \dot{\approx} \mathbf{ask}(c) \rightarrow \mathbf{tell}(d)$ when $d \sqsubseteq c$. Intuitively, this corresponds to the fact that the implication $c \Rightarrow d$ is equivalent to true when c entails d. Let us take $\gamma_1 = \langle \mathbf{tell}(\mathit{true}), \mathit{true} \rangle$ and $\gamma_2 = \langle \mathbf{ask}(c) \rightarrow \mathbf{tell}(d), \mathit{true} \rangle$. Their labeled transition systems are the following: $\gamma_1 \xrightarrow{\mathit{true}} \langle \mathbf{stop}, \mathit{true} \rangle$ and $\gamma_2 \xrightarrow{c} \langle \mathbf{tell}(d), c \rangle \xrightarrow{\mathit{true}} \langle \mathbf{stop}, c \rangle$. It is now easy to see that the symmetric closure of the relation $\mathcal{R}$ given below is a weak bisimulation.

$$\mathcal{R} = \{(\gamma_2, \gamma_1),\ (\gamma_2, \langle \mathbf{stop}, \mathit{true} \rangle),\ (\langle \mathbf{tell}(d), c \rangle, \langle \mathbf{stop}, c \rangle),\ (\langle \mathbf{stop}, c \rangle, \langle \mathbf{stop}, c \rangle)\} \ \square$$

The following theorem states that strong and weak bisimilarity coincide, resp., with $\dot{\sim}_{sb}$ and $\dot{\approx}_{sb}$. Hence γ_1 and γ_2 in the above example are also in $\dot{\approx}_{sb}$ (and, by Thm 2, also in $\sim_o$). It is worth noticing that any saturated barbed bisimulation (Def. 7) relating γ_1 and γ_2 is infinite in dimension, since it has to relate $\langle \mathbf{tell}(\mathit{true}), a \rangle$ and $\langle \mathbf{ask}(c) \rightarrow \mathbf{tell}(d), a \rangle$ for all constraints $a \in \mathit{Con}_0$. Instead, the relation $\mathcal{R}$ above is finite and it represents (by virtue of the following theorem) a proof also for $\gamma_1 \dot{\approx}_{sb} \gamma_2$.

Theorem 3. $\dot{\sim}_{sb} = \dot{\sim}$ *and* $\dot{\approx}_{sb} = \dot{\approx}$*.*

5 Conclusions, Related and Future Work

In this paper we introduced labeled semantics and bisimilarity for ccp. Our equivalence characterizes the observational semantics introduced in [27] based on limits of infinite computations, by means of a co-inductive definition. It follows from [27] that our bisimilarity coincides with the equivalence induced by the standard closure operators semantics of ccp. Therefore, our weak bisimulation approach represents a novel sound and complete proof technique for observational equivalence in ccp.

Our work is also interesting for the research programme on "labels derivation". Our labeled semantics can be regarded as an instance of the one introduced at an abstract level in [18]. Syntactical bisimulation (Def. 8) as an instance of the one in [18], while strong and weak bisimulations (Def. 9 and Def. 10) as instances of those in [6]. Furthermore, syntactical bisimulation intuitively coincides with the one in [26], while saturated barbed bisimulation (Def. 6) with the one in [19]. Recall that syntactical bisimilarity is too fine grained, while saturated barbed bisimulation requires the relation to be upward closed (and thus, infinite in dimension). Our weak bisimulation instead is fully abstract and avoid the upward closure. Summarizing, the framework in [6] provides us an abstract approach for deriving a novel interesting notion of bisimulation.

It is worth noticing that the restriction to the summation-free fragment is only needed for proving the coincidence with [27]. The theorem in Section 2.1 still holds in the presence of summation. Analogously, we could extend all the definitions to infinite constraints without invalidating these theorems.

Some recent works [9,17,16] have defined bisimilarity for novel languages featuring the interaction paradigms of both ccp and the π-calculus. In these works, bisimilarity is defined starting from transition systems whose labels represent communications in the style of the π-calculus. Instead we employ barbs on a purely unlabeled semantics. Preliminary attempts have shown that defining a correspondence with our semantics is not trivial. We left this for an extended version of the paper.

As shown e.g. in [19] there are strong connections between ccp processes and logic formulae. As future work we would like to investigate whether our present results can be adapted to provide a novel characterization of logic equivalence in terms of bisimilarity. Preliminary results show that at least the propositional fragment, without negation, can be characterized in terms of bisimilarity.

Finally, we are implementing a checker for our equivalence by employing [8].

References

1. Extended version. Technical report, `http://www.lix.polytechnique.fr/ luis.pino/files/FOSSACS11-extended.pdf`
2. Abramsky, S., Jung, A.: Domain theory. In: Handbook of Logic in Computer Science, pp. 1–168. Clarendon Press, Oxford (1994)
3. Amadio, R.M., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous pi-calculus. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 147–162. Springer, Heidelberg (1996)
4. Bartoletti, M., Zunino, R.: A calculus of contracting processes. In: LICS, pp. 332–341. IEEE Computer Society, Los Alamitos (2010)
5. Bengtson, J., Johansson, M., Parrow, J., Victor, B.: Psi-calculi: Mobile processes, nominal data, and logic. In: LICS, pp. 39–48 (2009)
6. Bonchi, F., Gadducci, F., Monreale, G.V.: Reactive systems, barbed semantics, and the mobile ambients. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 272–287. Springer, Heidelberg (2009)

7. Bonchi, F., König, B., Montanari, U.: Saturated semantics for reactive systems. In: LICS, pp. 69–80 (2006)
8. Bonchi, F., Montanari, U.: Minimization algorithm for symbolic bisimilarity. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 267–284. Springer, Heidelberg (2009)
9. Buscemi, M.G., Montanari, U.: Open bisimulation for the concurrent constraint pi-calculus. In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 254–268. Springer, Heidelberg (2008)
10. de Boer, F.S., Pierro, A.D., Palamidessi, C.: Nondeterminism and infinite computations in constraint programming. Theor. Comput. Sci. 151(1), 37–78 (1995)
11. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 151–166. Springer, Heidelberg (2004)
12. Falaschi, M., Gabbrielli, M., Marriott, K., Palamidessi, C.: Confluence in concurrent constraint programming. Theor. Comput. Sci. 183(2), 281–315 (1997)
13. Di Gianantonio, P., Honsell, F., Lenisa, M.: Rpo, second-order contexts, and lambda-calculus. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 334–349. Springer, Heidelberg (2008)
14. Henkin, J.M.L., Tarski, A.: Cylindric Algebras (Part I). North-Holland, Amsterdam (1971)
15. Honda, K., Yoshida, N.: On reduction-based process semantics. Theor. Comput. Sci. 151(2), 437–486 (1995)
16. Johansson, M., Bengtson, J., Parrow, J., Victor, B.: Weak equivalences in psi-calculi. In: LICS, pp. 322–331 (2010)
17. Johansson, M., Victor, B., Parrow, J.: A fully abstract symbolic semantics for psi-calculi. CoRR, abs/1002.2867 (2010)
18. Leifer, J.J., Milner, R.: Deriving bisimulation congruences for reactive systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 243–258. Springer, Heidelberg (2000)
19. Mendler, N.P., Panangaden, P., Scott, P.J., Seely, R.A.G.: A logical view of concurrent constraint programming. Nord. J. Comput. 2(2), 181–220 (1995)
20. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge University Press, Cambridge (1999)
21. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992)
22. Montanari, U., Sassone, V.: Dynamic congruence vs. progressing bisimulation for ccs. FI 16(1), 171–199 (1992)
23. Palamidessi, C., Saraswat, V.A., Valencia, F.D., Victor, B.: On the expressiveness of linearity vs persistence in the asychronous pi-calculus. In: LICS, pp. 59–68 (2006)
24. Rathke, J., Sassone, V., Sobociński, P.: Semantic barbs and biorthogonality. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 302–316. Springer, Heidelberg (2007)
25. Rathke, J., Sobocinski, P.: Deconstructing behavioural theories of mobility. In: IFIP TCS, vol. 273, pp. 507–520. Springer, Heidelberg (2008)
26. Saraswat, V.A., Rinard, M.C.: Concurrent constraint programming. In: POPL, pp. 232–245 (1990)
27. Saraswat, V.A., Rinard, M.C., Panangaden, P.: Semantic foundations of concurrent constraint programming. In: POPL, pp. 333–352 (1991)
28. Sassone, V., Sobocinski, P.: Reactive systems over cospans. In: LICS, pp. 311–320 (2005)
29. Sewell, P.: From rewrite rules to bisimulation congruences. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 269–284. Springer, Heidelberg (1998)
30. Saraswat, V.A.: Concurrent Constraint Programming. PhD thesis, Carnegie-Mellon University (1989)

Ordinal Theory for Expressiveness of Well Structured Transition Systems

Remi Bonnet[1], Alain Finkel[1], Serge Haddad[1,⋆], and Fernando Rosa-Velardo[2,⋆⋆]

[1] Ecole Normale Supérieure de Cachan, LSV, CNRS UMR 8643, Cachan, France
{remi.bonnet,alain.finkel,serge.haddad}@lsv.ens-cachan.fr
[2] Sistemas Informáticos y Computación, Universidad Complutense de Madrid
fernandorosa@sip.ucm.es

Abstract. To the best of our knowledge, we characterize for the first time the importance of resources (counters, channels, alphabets) when measuring expressiveness of WSTS. We establish, for usual classes of wpos, the equivalence between the existence of order reflections (non-monotonic order embeddings) and the simulations with respect to coverability languages. We show that the non-existence of order reflections can be proved by the computation of order types. This allows us to solve some open problems and to unify the existing proofs of the WSTS classification.

1 Introduction

WSTS. Infinite-state systems appear in a lot of models and applications: stack automata, counter systems, Petri nets or VASSs, reset/transfer Petri nets, fifo (lossy) channel systems, parameterized systems. Among these infinite-state systems, a part of them, called Well-Structured Transition Systems (WSTS) [8] enjoys two nice properties: there is a well partial ordering (wpo) on the set of states and the transition relation is monotone with respect to this wpo.

The theory of WSTS has been successfully applied for the verification of safety properties of numerous infinite-state models like Lossy Channel Systems, extensions of Petri Nets like reset/transfer and Affine Well Nets [9], or broadcast protocols. Most of the positive results are based on the decidability of the coverability problem (whether an upward closed set of states is reachable from the initial state) for WSTS, under natural effectiveness hypotheses. The reachability problem, on the contrary, is undecidable even for the class of Petri nets extended with reset or transfer transitions.

Expressiveness. Well Structured Languages [10] were introduced as a measure of the expressiveness of subclasses of WSTS. More precisely, the language of an instance of a model is defined as the class of *finite* words accepted by it, with

[⋆] Authors partially supported by the Agence Nationale de la Recherche, AVERISS (grant ANR-06-SETIN-001) and AVERILES (grant ANR-05-RNTL-002).

[⋆⋆] Author partially supported by the MEC Spanish project DESAFIOS10 TIN2009-14599-C03-01, and the CAM program PROMETIDOS S2009/TIC-1465.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 153–167, 2011.

coverability as accepting condition, that is, generated by traces that reach a state which is bigger than a given final state. Convincing arguments show that the class of coverability languages is the right one. For instance, though reachability languages are more precise than coverability languages, the class of reachability languages is RE for almost all Petri Nets extensions containing Reset Petri Nets or Transfer Petri Nets.

The expressive power of WSTS comes from two natural sources: from the structure of the state space and from the semantics of the transition relation. These two notions were often extremely interwined in the proofs. We propose ourselves to separate them in order to have a formal and generalizable method.

The study of the state space is related to the relevance of resources: A natural question when confronted to an extension of a model is whether the additional resources actually yield an increase in expressiveness. For example, if we look at Timed Automata, clocks are a strict resource: Timed Automata with k clocks are less expressive that Timed Automata with $k + 1$ clocks [4]. Surprisingly, no similar results exist for well-known models like Petri Nets (with respect to the number of places) or Lossy Channel Systems (with respect to the number of channels, or number of symbols in the alphabet) except in some particular recent works [7].

Ordinal theory for partial orders. Ordinals are a well-known representation of well-founded total orders. Thanks to de Jongh, Parikh, Schmidt ([11], [17]) and others, this representation has been extended to well partial orders. We are mainly interested in the order type of a wpo, which can be understood as the "size" of the order. The order types of the union, product, and finite words have been computed since de Jongh and Parikh. Recently, Weiermann [18] has completed this view by computing the order type for multisets.

Contribution. First, we introduce order reflections, a variation of order embeddings that are allowed to be non-monotonic. We define a notion of witnessing, that reflects the ability of a WSTS to recognize a wpo through a coverability language. We establish the equivalence between the existence of order reflections and the simulations with respect to coverability languages, modulo the ability of the WSTS classes to witness their own state space.

Second, we show how to use results from the theory of ordinals, and more precisely the properties of maximal order types, studied by de Jongh and Parikh [11] and Schmidt [17] to easily prove the absence of reflections.

Last, we study Lossy Channel Systems and extensions of Petri Nets. We show that most of known classes of WSTS are self-witnessing. This allows us to unify and simplify the existing proofs regarding the classification of WSTS, also solving the open problem [15] of the relative expressiveness of two Petri Nets extensions called ν-Petri Nets and Data Nets, also yielding that the number of unbounded places for these Petri Nets extensions and the size of the alphabet for Lossy Channel Systems are relevant resources when considering their expressiveness.

Related work. Coverability languages have been used to discriminate the expressive power of several WSTS, like Lossy Channel Systems or several monotonic extensions of Petri Nets. In [10] several pumping lemmas are proved to discriminate between extensions of Petri Nets. In [1,2] the expressive power of Petri Nets is proved to be strictly below that of Affine Well Nets, and Affine Well Nets are proved to be strictly less expressive than Lossy Channel Systems. Similar results are obtained in [15], though some significant problems are left open, like the distinction between ν-Petri Nets [14] and Data Nets [13] that we solve here.

Outline. The rest of the paper is organized as follows. In Section 2 we introduce wpos, WSTS and ordinals. Then in Section 3 we develop the study of reflections and its links with expressiveness of WSTS. Afterwards in Section 4 we apply our result to the classical models of Petri Nets and Lossy Channel Systems. Section 5 presents the extension of our results applicable to more recent models of WSTS. Finally we conclude and give perspectives to this work in Section 6.

For lack of space, some proofs have been omitted. We refer the interested reader to [6] that contains the appendices with all proofs.

2 Preliminaries and WSTS

Well Orders. $(X, \leq_X)$ is a *quasi-order* (qo) if $\leq_X$ is a reflexive and transitive binary relation on X. For a qo we write $x <_X y$ iff $x \leq_X y$ and $y \not\leq_X x$. A *partial order* (po) is an antisymmetric quasi-order. Given any qo $(X, \leq_X)$, the quotient set $X/\equiv_{\leq_X}$ is a po where $x \equiv_{\leq_X} y$ is defined by $x \leq_X y \wedge y \leq_X x$. Hence, in all the paper, we will suppose that $(X, \leq_X)$ is a po.

The *downward closure* of a subset $A \subseteq X$ is defined as $\downarrow A = \{x \in X \mid \exists x' \in A,\ x \leq x'\}$. A subset A is *downward closed* iff $\downarrow A = A$. A po $(X, \leq_X)$ is a *well partial order* (wpo) if for every infinite sequence $x_0, x_1, \ldots \in X$ there are i and j with $i < j$ such that $x_i \leq x_j$. Equivalently, a po is a wpo when there are no strictly decreasing (for inclusion) sequences of downward closed sets.

We will shorten $(X, \leq_X)$ to X when the underlying order is obvious. Similarly, $\leq$ will be used instead of $\leq_X$ when X can be deduced from the context.

If X and Y are wpos, their cartesian product, denoted $X \times Y$ is well ordered by $(x, y) \leq_{X \times Y} (x', y') \iff x \leq_X x' \wedge y \leq_Y y'$. Their disjoint union, denoted $X \uplus Y$ is well ordered by:

$$z \leq_{X \uplus Y} z' \iff \begin{cases} z, z' \in X \\ z \leq_X z' \end{cases} \text{or} \quad \begin{cases} z, z' \in Y \\ z \leq_Y z' \end{cases}$$

A po $(X, \leq)$ is *total* (or *linear*) if for any $x, x' \in X$ either $x \leq x'$ or $x' \leq x$. If $(X_i, \leq_i)$ are total po for $i \in \mathbb{N}$ we can define the (irreflexive) total order $<_{lex}$ in $\bigcup_k X_1 \times ... \times X_k$ by $(x_1, ..., x_p) <_{lex} (x'_1, ..., x'_q)$ iff there is $i \in \{1, ..., min(p, q)\}$ such that $x_j = x'_j$ for $j < i$ and $x_i <_i x'_i$ or $(x_1, ..., x_p) = (x'_1, ..., x'_p)$ and $q > p$. Then $\leq_{lex}$ given by $x \leq_{lex} x'$ iff $x = x'$ or $x <_{lex} x'$ is a total order.

Functions. Given a partial function (shortly: function) $f : X \to Y$, the *domain* of f is defined by $dom(f) = \{x \in X \mid \exists y \in Y,\ f(x) = y\}$ and its *range* by $range(f) = \{y \in Y \mid \exists x \in X,\ f(x) = y\}$. A function f is *surjective* if $range(f) = Y$ and it is *total* if $dom(f) = X$. Total functions are called *mappings*. A mapping f is *injective* if for all x, x', $f(x) = f(x') \implies x = x'$. Finally, let us consider a mapping f: if X and Y are ordered, f is *increasing* (resp. *strictly increasing*) if $x \leq_X y \implies f(x) \leq_Y f(y)$ (resp. if $x <_X y \implies f(x) <_Y f(y)$); f is an *order embedding* (shortly: embedding) if $f(x) \leq_Y f(x') \iff x \leq_X x'$. A bijective order embedding is called an *order isomorphism* (shortly: isomorphism).

Multisets. Given a set X, we denote by $X^{\oplus}$ the set of finite multisets of X, that is, the set of mappings $m : X \to \mathbb{N}$ with a finite support $sup(m) = \{x \in X \mid m(x) \neq 0\}$. We use the set-like notation $\{|...|\}$ for multisets when convenient, with $\{|x^n|\}$ describing the multiset containing x n times. We use $+$ and $-$ for multiset operations. If X is a wpo then so is $X^{\oplus}$ ordered by $\leq_{\oplus}$ defined by $\{|x_1, \ldots, x_n|\} \leq_{\oplus} \{|x'_1, \ldots, x'_m|\}$ if there is an injection $h : \{1, \ldots, n\} \to \{1, \ldots, m\}$ such that $x_i \leq_X x'_{h(i)}$ for each $i \in \{1, \ldots, n\}$.

Words. Given a set X, any $u = x_1 \cdots x_n$ with $n \geq 0$ and $x_i \in X$, for all i, is a finite word on X. We denote by X^* the set of finite words on X. If $n = 0$ then u is the empty word, which is denoted by ϵ. A language L on X is a subset of X^*. Given L and L' two languages on X^*, we define the language $LL' = \{uv \mid u \in L, v \in L'\}$. If X is a wpo then so is X^* ordered by $\leq_{X^*}$ which is defined as follows: $x_1 \ldots x_n \leq_{X^*} x'_1 \ldots x'_m$ if there is a strictly increasing mapping $h : \{1, \ldots, n\} \to \{1, \ldots, m\}$ such that $x_i \leq_X x'_{h(i)}$ for each $i \in \{1, \ldots, n\}$.

Ordinals below ϵ_0. In this paper, we shall work with set theoretical ordinals. Let us recall a few properties of these objects. The class of ordinals is totally ordered by inclusion, and each ordinal α is equal to the set of ordinals $\{\beta \mid \beta < \alpha\}$ below it. Every total well order $(X, \leq_X)$ is isomorphic to a unique ordinal $ot(X, \leq_X)$, called the *order type* of X.

In the context of ordinals, we define $0 = \emptyset$, $n = \{0, ..., n-1\}$ and $\omega = \mathbb{N}$, ordered by the usual order. Moreover, given α and α' ordinals, we define $\alpha + \alpha'$ as the order type of $(\{0\} \times \alpha) \cup (\{1\} \times \alpha')$ ordered by $\leq_{lex}$. In the same way, $\alpha * \alpha'$ is defined as the order type of $\alpha' \times \alpha$ ordered by $\leq_{lex}$. Note that these operations are not commutative: we have $1 + \omega = \omega \neq \omega + 1$. This definition of $+$ and $*$ coincides with the usual operations on $\mathbb{N}$ for ordinals below ω and we have $\alpha + \overset{k}{\cdots} + \alpha = \alpha * k$. Exponentiation can be similarly defined, but for simplicity of presentation, we let this definition outside this short introduction to ordinals. Note that the most important properties of exponentiation can be obtained from the ordering on Cantor's Normal Forms (CNF) that we develop below.

In this paper, we will work with ordinals below ϵ_0, that is, those that can be bounded by a tower $\omega^{\omega^{\cdot^{\cdot^{\cdot^{\omega}}}}}$. These can be represented by the hierarchy of ordinals in CNF that is recursively given by the following rules:

$C_0 = \{0\}$.
$C_{n+1} = \left\{\omega^{\alpha_1} + \cdots + \omega^{\alpha_p} \mid p \in \mathbb{N},\ \alpha_1, \ldots, \alpha_p \in C_n \text{ and } \alpha_1 \geq \cdots \geq \alpha_p \right\}$ ordered by:

$$\omega^{\alpha_1} + \cdots + \omega^{\alpha_p} \leq \omega^{\alpha'_1} + \cdots + \omega^{\alpha'_q} \iff (\alpha_1, \ldots, \alpha_p) \leq_{lex} (\alpha'_1, \ldots, \alpha'_q)$$

Each ordinal below ϵ_0 has a unique CNF. If $\alpha = \omega^{\beta_1} + \cdots + \omega^{\beta_n}$, we denote by *Cantor*$(\alpha)$ the multiset $\{|\beta_1, \ldots, \beta_n|\}$.

WSTS. A *Labelled Transition System* (LTS) is a tuple $\mathcal{S} = \langle X, \Sigma, \rightarrow\rangle$ where X is the set of states, Σ is the labelling alphabet and $\rightarrow \subseteq X \times (\Sigma \cup \{\epsilon\}) \times X$ is the transition relation. We write $x \xrightarrow{a} x'$ to say that $(x, a, x') \in \rightarrow$. This relation is extended for $u \in \Sigma^*$ by $x \xrightarrow{u} x' \iff x \xrightarrow{a_1} x_1 ... x_{k-1} \xrightarrow{a_k} x'$ and $u = a_1 a_2 \cdots a_k$ (note that some a_i's can be ϵ). A *Well Structured Transition System* (shortly a WSTS) is a tuple $\mathcal{S} = (X, \Sigma, \rightarrow, \leq)$, where $(X, \Sigma, \rightarrow)$ is an lts, and $\leq$ is a wpo on X, satisfying the following monotonicity condition: for all $x_1, x_2, x'_1 \in X, u \in \Sigma^*$, $x_1 \leq x'_1$, $x_1 \xrightarrow{u} x_2$ implies the existence of $x'_2 \in X$ such that $x'_1 \xrightarrow{u} x'_2$ and $x_2 \leq x'_2$. For a class $\mathbf{X}$ of wpos, we will denote by $\mathit{WSTS}_{\mathbf{X}}$ the class of WSTS with state space in $\mathbf{X}$, or just WSTS_X for $\mathit{WSTS}_{\{X\}}$.

Coverability and Reachability Languages. Trace languages, reachability languages and coverability languages are natural candidates for measuring the expressive power of classes of WSTS. Given a WSTS $\mathcal{S}$ and two states x_0 and x_f, the reachability language is $L_R(\mathcal{S}, x_0, x_f) = \{u \in \Sigma^* \mid x_0 \xrightarrow{u} x_f\}$ while the coverability language is $L(\mathcal{S}, x_0, x_f) = \{u \in \Sigma^* \mid x_0 \xrightarrow{u} x,\ x \geq x_f\}$. Let us remark that all trace languages are coverability languages in taking $x_f = \perp$ where $\perp$ is the least element of X.

The class of reachability languages is the set of recursively enumerable languages for almost all Petri nets extensions containing reset Petri nets or transfer Petri nets. Thus such a criterium does not discriminates sufficiently . One could consider infinite coverability languages. A sensible accepting condition in this case could be repeated coverability, that is, the capacity of covering a given marking infinitely often, in the style of Büchi automata. However, analogously to what happens with reachability, repeated coverability is generally undecidable, which makes ω-languages a bad candidate to study the relative expressive power of WSTS. In conclusion, we will use the class of coverability languages, as in [10,1,2,15].

For two classes of WSTS, $\mathbf{S}_1$ and $\mathbf{S}_2$, we write $\mathbf{S}_1 \preceq \mathbf{S}_2$ whenever for every language $L(\mathcal{S}_1, x_1, x'_1)$ with $\mathcal{S}_1 \in \mathbf{S}_1$, and x_1, x'_1 two states of $\mathcal{S}_1$, there exists another system $\mathcal{S}_2 \in \mathbf{S}_2$ and two states x_2, x'_2 of $\mathcal{S}_2$ such that $L(\mathcal{S}_2, x_2, x'_2) = L(\mathcal{S}_1, x_1, x'_1)$. When $\mathbf{S}_1 \preceq \mathbf{S}_2$ and $\mathbf{S}_2 \preceq \mathbf{S}_1$, one denotes the equivalence of classes by $\mathbf{S}_1 \simeq \mathbf{S}_2$. We write $\mathbf{S}_1 \prec \mathbf{S}_2$ for $\mathbf{S}_1 \preceq \mathbf{S}_2$ and $\mathbf{S}_2 \not\preceq \mathbf{S}_1$.

The Lossy semantics. The *lossy* semantics $\mathcal{S}_l$ of a WSTS $\mathcal{S}$ with space X is the original system $\mathcal{S}$ completed by all ϵ-transitions $x \xrightarrow{\epsilon} y$, for all $x, y \in X$ such that $y < x$. We observe that $\mathcal{S}_l$ satisfies the monotonicity condition, hence $\mathcal{S}_l$ is still a WSTS; and moreover, due to the lossy semantics, one has: for all $x_1, x_2 \in X, u \in \Sigma^*$, $x_1 \xrightarrow{u} x_2$ implies $x_1 \xrightarrow{u} x'_2$ for all $x'_2 \leq x_2$. For any x_0, x_f, we have: $L(\mathcal{S}, x_0, x_f) = L(\mathcal{S}_l, x_0, x_f)$.

3 A Method for Comparing WSTS

In this section we propose a method to compare the expressiveness of WSTS mainly based on their state space. We will prove some results that will provide us with tools to establish strict relations between classes of WSTS.

3.1 A New Tool: Order Reflections

Definition 1. *Let $(X, \leq_X)$ and $(Y, \leq_Y)$ be two partially ordered sets. A mapping $\varphi : X \to Y$ is an* order reflection *(shortly: reflection) if $\varphi(x) \leq_Y \varphi(x')$ implies $x \leq_X x'$ for all $x, x' \in X$.*

We will write $X \sqsubseteq Y$ if there is an embedding from X to Y and $X \sqsubseteq_{refl} Y$ if there is a reflection from X to Y. We will use $\not\sqsubseteq$ and $\not\sqsubseteq_{refl}$ for their negation and $\sqsubset$ and $\sqsubset_{refl}$ for their antisymmetric version (i.e. $X \sqsubset Y \iff X \sqsubseteq Y \wedge Y \not\sqsubseteq X$). Here are some basic properties of reflections we will use throughout the paper: for any set X, any injective mapping to $(X, =)$ is a reflection; every reflection is injective; the composition of two reflections is a reflection (so $\sqsubseteq_{refl}$ is a qo).

Furthermore, if φ is an embedding from X to Y then X is isomorphic to $\varphi(X)$ and hence can be identified to it. Clearly, existence of embeddings are a stronger requirement than the existence of reflections. In particular, it can be the case that a wpo X cannot be embedded in another wpo Y, even if there are reflections from X to Y, as implied by the following result.

Proposition 1. *The following properties hold:*

- $\mathbb{N}^k \sqsubseteq_{refl} \mathbb{N}^{\oplus}$*, for any $k > 0$.*
- $\mathbb{N}^k \not\sqsubseteq \mathbb{N}^{\oplus}$ *for any $k \geq 3$ (but $\mathbb{N}^2 \sqsubseteq \mathbb{N}^{\oplus}$).*

3.2 Expressiveness of WSTS and Order Reflections

Reflections are more appropriate than embeddings for the comparison of WSTS. In particular, the existence of a reflection implies the relation between the corresponding classes of WSTS.

Theorem 1. *Let X and Y be two wpo. We have:*

$$X \sqsubseteq_{refl} Y \implies \mathit{WSTS}_X \preceq \mathit{WSTS}_Y$$

This is easily shown by taking a WSTS of state space X, looking at its lossy equivalent through the order reflection, and realizing this is another WSTS which recognizes the same language. The detailed proof is in the appendix of [6].

We would like to obtain the converse of the previous result: $X \not\sqsubseteq_{refl} Y \implies \mathit{WSTS}_X \not\preceq \mathit{WSTS}_Y$. First, we only present this result for "simple" state spaces. The case of more complex state spaces will be handled in later sections.

Given an alphabet $\Sigma = \{a_1, ..., a_k\}$, we define $\overline{\Sigma}$ by $\overline{\Sigma} = \{\overline{a_1}, \cdots, \overline{a_k}\}$ where $\overline{a_i}$'s are fresh symbols (i.e. $\Sigma \cap \overline{\Sigma} = \emptyset$). This notation is extended to words by $\overline{u} = \overline{a_1} \cdots \overline{a_k}$ for $u = a_1 \cdots a_k \in \Sigma^*$. In the same way, given $L \subseteq \Sigma^*$, we have $\overline{L} = \{\overline{u} \mid u \in L\} \subseteq \overline{\Sigma}^*$.

Definition 2. *Let X be a wpo and Σ a finite alphabet. A surjective partial function from Σ^* to X is called a Σ-representation of X. Given a Σ-representation γ of X, we define $L_\gamma = \{u\overline{v} \mid u, v \in dom(\gamma) \text{ and } \gamma(v) \leq \gamma(u)\}$. A language $L \in (\Sigma \cup \bar{\Sigma})^*$ is a γ-witness (shortly: witness) of X if $L \cap dom(\gamma)\overline{dom(\gamma)} = L_\gamma$.*

In particular, L_γ is a witness of X for any Σ-representation γ of X. Intuitively, given a witness L of X, the fact that a WSTS can recognize L *witnesses* that the WSTS can represent the structure of X: it is capable of accepting all words starting with some u (representing some state $\gamma(u)$), followed by some v that represents $\gamma(v) \leq \gamma(u)$. Witness languages are useful in proving strict relations between classes of WSTS:

Theorem 2. *Let L be a witness of X. If $X \not\sqsubseteq_{refl} Y$ then there are no $y_0, y_f \in Y$ and no $\mathcal{S} \in WSTS_Y$ such that $L = L(\mathcal{S}, y_0, y_f)$.*

Proof. Assume by contradiction that L is a covering language of a WSTS $\mathcal{S}$ whose state space is Y with y_0 and y_f as initial and final states, respectively. For each $x \in X$, let us take $u_x \in \Sigma^*$ such that $\gamma(u_x) = x$. The word $u_x\overline{u_x}$ is recognized by $\mathcal{S}$, hence we can find y_x and y'_x such that $y_0 \xrightarrow{u_x} y_x \xrightarrow{\overline{u_x}} y'_x \geq y_f$.

We define $\varphi(x) = y_x$. Let us see that φ is an order reflection from X to Y, thus reaching a contradiction. Assume that $\varphi(x) \leq \varphi(x')$. Since $\mathcal{S}$ is a WSTS any sequence fireable from $\varphi(x)$ is also fireable from $\varphi(x')$ and the state reached by this subsequence is greater or equal than the one reached from $\varphi(x)$. Hence, the state reached after $u_{x'}\overline{u_x}$ is bigger than the one reached after $u_x\overline{u_x}$, which means that $u_{x'}\overline{u_x} \in L \cap dom(\gamma)\overline{dom(\gamma)}$, implying $x \leq x'$, so that φ is an order reflection.

The simple state spaces we mentioned before, will be the ones produced by the following grammar:

$$\begin{array}{rll} \Gamma ::= & Q & \text{(finite set with equality)} \\ \mid & \mathbb{N} & \text{(naturals with the standard order)} \\ \mid & \Sigma^* & \text{(words on a finite set with the order defined in Section 2)} \\ \mid & \Gamma \times \Gamma & \text{(cartesian product with the order defined in Section 2)} \end{array}$$

As $\mathbb{N}$ is isomorphic to Σ^* when Σ is a singleton, any set produced by Γ is isomorphic to a set $Q \times \Sigma_1^* \times \cdots \times \Sigma_k^*$ where Q and each Σ_i are finite sets.

Proposition 2. *Let X be a set produced by the grammar Γ. Then, there is a witness of X that is recognized by a WSTS of state space X.*

When a WSTS can recognize a witness of its own state space the following holds:

Proposition 3. *Let X be a wpo produced by Γ and Y any wpo. Then,*

$$X \sqsubseteq_{refl} Y \iff WSTS_X \preceq WSTS_Y$$

Proof. The direction from left to right is given by Theorem 1. For the converse, let us prove that $X \not\sqsubseteq_{refl} Y \Rightarrow WSTS_X \not\preceq WSTS_Y$. We can find a witness L of X recognized by a WSTS of state space X (Prop. 2). By Theorem 2, this language can not be recognized by a WSTS of state space Y, hence the result.

3.3 Self-witnessing WSTS Classes

The reason we were able to build our equivalence between the existence of a reflection from X to Y and $WSTS_X \preceq WSTS_Y$ for any wpo X produced by Γ was Prop. 2. However, we conjecture that for any state space X that embeds $\mathbb{N}^{\oplus}$, there is no WSTS of state space X that can recognize a witness of X. This prompts us to define a new notion:

Definition 3. *Let* $\mathbf{X}$ *be a class of wpos and* $\mathbf{S}$ *a class of WSTS whose state spaces are included in* $\mathbf{X}$*.* $(\mathbf{X}, \mathbf{S})$ *is self-witnessing if, for all* $X \in \mathbf{X}$*, there exists* $\mathcal{S} \in \mathbf{S}$ *that recognizes a witness of* X*.*

We will shorten $(\mathbf{X}, \mathbf{S})$ as $\mathbf{S}$ when the state space is not explicitly needed. We extend the relation $\sqsubseteq_{refl}$ to classes of wpo by $\mathbf{X} \sqsubseteq_{refl} \mathbf{X}'$ if for any $X \in \mathbf{X}$, there exists $X' \in \mathbf{X}'$ such that $X \sqsubseteq_{refl} X'$.

Proposition 4. *Let* $(\mathbf{X}, \mathbf{S})$ *be a self-witnessing WSTS class and* $\mathbf{S}'$ *a WSTS class using state spaces inside* $\mathbf{X}'$*. Then,* $\mathbf{S} \preceq \mathbf{S}' \implies \mathbf{X} \sqsubseteq_{refl} \mathbf{X}'$*.*
Moreover, if $\mathbf{S}' = WSTS_{\mathbf{X}'}$*,* $\mathbf{S} \preceq \mathbf{S}' \iff \mathbf{X} \sqsubseteq_{refl} \mathbf{X}'$*.*

Proof. Let us show the first implication. Let $X \in \mathbf{X}$. Since $(\mathbf{X}, \mathbf{S})$ is self-witnessing, there is $\mathcal{S} \in \mathbf{S}$ that recognizes L, a witness of X. Because $\mathbf{S} \preceq \mathbf{S}'$, there is $\mathcal{S}' \in \mathbf{S}'$ recognizing L. $\mathcal{S}'$ has state space $X' \in \mathbf{X}'$, and by Theorem 2, $X \sqsubseteq_{refl} X'$.

For the second implication, for any $X \in \mathbf{X}$, we have $X' \in \mathbf{X}'$ such that $X \sqsubseteq_{refl} X'$. Because of Theorem 1, $WSTS_X \preceq WSTS_{X'}$. Hence, $WSTS_{\mathbf{X}} \preceq WSTS_{\mathbf{X}'}$.

We will see in sections 4 and 5 that many usual classes of WSTS, even those outside the algebra Γ, are self-witnessing.

3.4 How to Prove the Non-existence of Reflections?

Because of Prop. 3 and Prop. 4, the non existence of reflections will be a powerful tool to prove strict relations between WSTS. We provide here a simple way from order theory. Let us recall that a *linearization* of a po $\leq_X$ is a linear order $\leq'_X$ on X such that $x \leq_X y \implies x \leq'_X y$. A linearization of a wpo is a well total order, hence isomorphic to an ordinal. We extend the definition of order types to non-total wpos:

Definition 4. *Let* $(X, \leq_X)$ *be a wpo. The* maximal order type *(shortly: order type) of* $(X, \leq_X)$ *is* $ot(X, \leq_X) = sup \; \{ot(X, \leq'_X) \mid \leq'_X$ *linearization of* $\leq_X\}$*.*

The existence of the *sup* comes from ordinal theory. de Jongh and Parikh [11] even show that this *sup* is actually attained. Let $Down(X)$ be the set of downward closed subsets of X. Then, another well-known characterization of the maximal order type is the following (proofs of propositions 5 and 6 are in the appendix of [6]):

Proposition 5. $ot(X)+1 = sup\ \{\alpha \mid \exists f : \alpha \to Down(X),\ f \textit{ strictly increasing}\}$

This leads us to the proposition that we use to separate many classes of WSTS:

Proposition 6. *[18] Let X and Y be two wpos. $X \sqsubseteq_{refl} Y \implies ot(X) \leq ot(Y)$.*

The order types of the usual state spaces used for WSTS are known. We will recall some classic results on these order types, but we need the following definitions of addition and multiplication on ordinals to be able to characterize the order types of $X \uplus Y$ and $X \times Y$. Remember (Section 2) that an ordinal α below ε_0 is uniquely determined by $Cantor(\alpha)$, hence the validity of the following definition.

Definition 5. *(Hessenberg 1906, [11]) The natural addition, denoted $\oplus$, and the natural multiplication, denoted $\otimes$, are defined by:*

$$Cantor(\alpha \oplus \alpha') = Cantor(\alpha) + Cantor(\alpha')$$
$$Cantor(\alpha \otimes \alpha') = \{\!|\beta \oplus \beta' \mid \beta \in Cantor(\alpha), \beta' \in Cantor(\alpha')|\!\}$$

We already know that the order type of a finite set (with any order) is its cardinality and that the order type of $\mathbb{N}$ is ω. De Jongh and Parikh [11], and Schmidt [17] have shown a way to compose order types with the disjoint union, the cartesian product, and the Higman ordering. A more recent and difficult result, by Weiermann [18], provides us with the order type of multisets. These results are summed up here:

Proposition 7. *([11], [17], [18])*

- $ot(X \uplus Y) = ot(X) \oplus ot(Y)$
- $ot(X \times Y) = ot(X) \otimes ot(Y)$
- $ot(X^*) = \begin{cases} \omega^{\omega^{ot(X)-1}} & \textit{if } X \textit{ finite} \\ \omega^{\omega^{ot(X)}} & \textit{otherwise (for } ot(X) < \epsilon_0) \end{cases}$
- $ot(X^{\oplus}) = \omega^{ot(X)} \quad \textit{for } ot(X) < \epsilon_0$

Formulas exist even for $ot(X) \geq \epsilon_0$. We refer the interested reader to [11] and [18] for the complete formulas. With these general results we can obtain many strict relations between wpo.

Corollary 1. *The following strict relations hold for any $k > 0$:*

(1) $\mathbb{N}^k \sqsubset_{refl} \mathbb{N}^{k+1}$
(2) $(\mathbb{N}^k)^{\oplus} \sqsubset_{refl} (\mathbb{N}^{k+1})^{\oplus}$
(3) $(\mathbb{N}^k)^* \sqsubset_{refl} (\mathbb{N}^{k+1})^*$
(4) $\mathbb{N}^k \sqsubset_{refl} \mathbb{N}^{\oplus}$
(5) $\mathbb{N}^k \sqsubset_{refl} \Sigma^*$ *(for $|\Sigma| > 1$)*

Proof. The non-strict relations in (1), (2) and (3) are clear, and for (4) this is Prop. 1. For (5), $\varphi(n_1, \ldots, n_k) = a^{n_1}b \ldots ba^{n_k}$ is a reflection. Strictness follows from Prop. 6 and the following order types, obtained according to the previous results: $ot(\mathbb{N}^k) = \omega^k$, $ot((\mathbb{N}^k)^{\oplus}) = \omega^{\omega^k}$, $ot((\mathbb{N}^k)^*) = \omega^{\omega^{\omega^k}}$, and $ot(\Sigma^*) = \omega^{\omega^{|\Sigma|-1}}$.

4 Vector Addition Systems and Lossy Channel Systems

The state spaces described by Prop. 3 are exactly those of Petri Nets and Lossy Channel Systems. We will look more closely at these systems to see the implication of this theorem regarding their expressiveness.

4.1 Vector Addition Systems and Petri Nets

We work with *Vector Addition Systems with States* (VASS), which are equivalent to Petri nets. A VASS of dimension k is a tuple $(Q, T, \delta, \Sigma, \lambda)$, where Q is a finite (and non-empty) set of control sates, T is a finite set of transitions, $\delta : T \to Q \times \mathbb{Z}^k \times Q$, Σ is the finite labelling alphabet, and $\lambda : T \to \Sigma \cup \{\epsilon\}$ is the mapping which labels transitions. Transition t is enabled in (p, x) if $\delta(t) = (p, y, q)$ for some $q \in Q$ and some $y \in \mathbb{Z}^k$ with $x \geq -y$, in which case t can occur, reaching state $(q, x + y)$. VASS are WSTS by taking $(p, x) \leq (q, y)$ iff $p = q$ and $x \leq y$. The transition relation $\to$ of the WSTS associated with the VASS is defined by: $((p, x), a, (q, x + y)) \in \to$ if there is a transition $t \in T$ which is enabled in (p, x) such that $\delta(t) = (p, y, q)$ and $\lambda(t) = a$.

Let us denote by VASS_k the class of VASS with dimension k. Notice that the state space of any VASS with dimension k is in $\mathbf{X}_k = \{Q \times \mathbb{N}^k \mid Q \text{ finite}\}$. Then we have the following:

Theorem 3. *For any $k > 0$, $\mathit{VASS}_k \not\preceq \mathit{WSTS}_{\mathbf{X}_{k-1}}$.*

Proof. We remark that the WSTS defined in the proof of Prop. 2 is actually a lossy VASS when $X = Q \times \mathbb{N}^k$. This induces that we can take the non-lossy version of this VASS, which is still a WSTS. Hence, VASS_k is self-witnessing, and therefore so is $\mathit{WSTS}_{\mathbf{X}_k}$. Since $\mathbb{N}^k \not\sqsubseteq_{refl} Q \times \mathbb{N}^{k-1}$ for all finite Q (indeed, $ot(\mathbb{N}^k) = \omega^k \not\leq \omega^{k-1} * |Q| = ot(Q \times \mathbb{N}^{k-1})$), we have $\mathbf{X}_k \not\sqsubseteq_{refl} \mathbf{X}_{k-1}$ and by Prop. 4 we conclude.

We remark that even the class of lossy VASS with dimension k is not included in the class of WSTS with state space in $\mathbf{X}_{k-1}$. Moreover, if we consider *Affine Well Nets* (*AWN*) (an extension of Petri nets with whole-place operations like transfers or resets), and denote by AWN_k the class of AWN with k unbounded places (therefeore, with state space in $\mathbf{X}_k$), we can obtain from the previous result the following simple consequences.

Corollary 2. *$\mathit{VASS}_k \prec \mathit{VASS}_{k+1} \not\preceq \mathit{AWN}_k$ for all $k \geq 0$.*

4.2 Lossy Channel Systems

Let Op denote any vector of k operations on a (fifo) channel such that for every $i \in \{1, \ldots, k\}$, $Op(i)$ is either a send operation $!a$ on channel i, a receive operation $?a$ from channel i ($a \in A$), a test for emptyness $\epsilon?$ on channel i or a null operation *nop*. Let us denote OP_k the set of operations Op.

A *Lossy Channel System* (LCS)[1] with k channels is a tuple $(Q, A, T, \delta, \Sigma, \lambda)$ where Q is a finite (and non-empty) set of states, A is the finite set of messages, T is a finite set of transitions, $\delta : T \to Q \times OP_k \times Q$, Σ is the labelling alphabet and $\lambda : T \to \Sigma \cup \{\epsilon\}$ is the mapping which labels transitions. The set of configurations is $Q \times (A^*)^k$.

For (non lossy) channel systems, transition t is enabled in $(p, u_1, \ldots, u_k)$ if $\delta(t) = (p, Op, q)$ for some $q \in Q$ and some $Op \in OP_k$, and for all $i \in \{1, \ldots, k\}$, if $Op(i) = nop$ then $u_i = u'_i$, if $Op(i) = \epsilon?$ then $u_i = u'_i = \epsilon$, if $Op(i) = !a$ then $u'_i = u_i a$ and if $Op(i) = ?a$ then $u_i = a u'_i$, in which case t can occur, reaching state $(q, u'_1, \ldots, u'_k)$.

The semantics of LCS is given as the lossy version of the previous semantics, when considering the canonic order in $Q \times (A^*)^k$ for which LCS are WSTS.

If Σ_p is defined by $\Sigma_p = \{\alpha_1, ..., \alpha_p\}$ where α_i's are constant symbols, we define $LCS(k, p)$ as the subclass of LCS with k channels and set of messages Σ_p. We have:

Theorem 4. $LCS(k, p) \prec LCS(k + 1, p) \prec LCS(1, p + 1)$

Proof. $LCS(k, p) \preceq LCS(k+1, p)$ clearly holds. The proof that $LCS(k+1, p) \preceq LCS(1, p + 1)$ is based on the well-known fact that one can simulate the $k + 1$ channels by inserting a new symbol k times as delimiters. A proof is available in the appendix of [6]. For the strictness, we remark again that the WSTS introduced in the proof of Prop. 2 is actually a LCS, that is, given a state space $X = Q \times (\Sigma_p^*)^k$, we can find $\mathcal{S}$ in $LCS(k, p)$ and a witness L of X such that $\mathcal{S}$ recognizes L. This implies that $LCS(k, p)$ is self-witnessing. For all k and p, $ot(Q \times (\Sigma_p^*)^k) = \omega^{\omega^{p-1} * k} * |Q|$. This implies that $(\Sigma_p^*)^{k+1} \not\sqsubseteq_{refl} Q \times (\Sigma_p^*)^k$ and $\Sigma_{p+1}^* \not\sqsubseteq_{refl} Q \times (\Sigma_p^*)^k$ for all Q. To conclude we only need to apply proposition 4.

Moreover, in [2] the authors prove that $AWN \prec LCS$. We can easily get back this result:

Proposition 8. $LCS(1, 2) \not\preceq AWN$.

Proof. As in the previous result, we remark that $LCS(1, 2)$ and AWN are self-witnessing. Thus, we only need to apply Prop. 4, considering that for any $k > 0$, $\Sigma_2^* \not\sqsubseteq_{refl} \mathbb{N}^k$ (Cor. 1).

This result is tight: $LCS(0, p) \simeq FA$ (Finite Automata), $LCS(k, 1) \simeq VASS_k$.

5 Petri Nets Extensions with Data

Many extensions of Petri nets with data have been defined in the literature to gain expressive power for better modeling capabilities. Data Nets (DN) [13] are a monotonic extension of Petri nets in which tokens are taken from a linearly

[1] This definition is a slight variation of the usal one in order to uniformise presentation of VASS and LCS without effect on their expressive power.

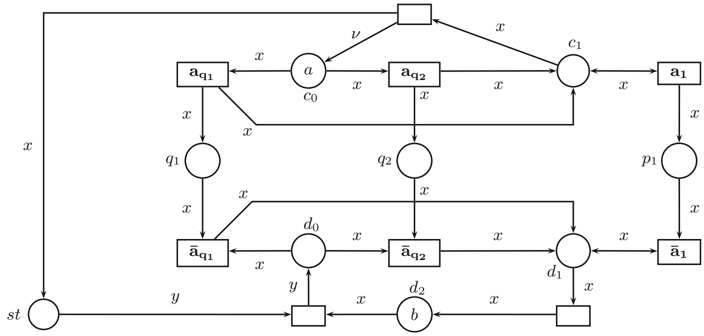


Fig. 1. Net in ν-PN_1 recognizing a witness of $(Q \times \mathbb{N})^{\oplus}$ with $|Q| = 2$

ordered and dense domain, and transitions can perform whole place operations like transfers, resets or broadcasts. A similar model, in which tokens can only be compared with equality, is that of ν-Petri Nets (ν-*PN*) [14]. The relative expressive power of *DN* and ν-*PN* has been an open problem since [15]. In this section we prove that ν-$PN \prec DN$. We work with the subclass of *DN* without whole place operations, called *Petri Data Net* (*PDN*), since $DN \simeq PDN$ [2].

Now we briefly define ν-*PN*. The definition of *PDN* is in the appendix of [6]. We consider an infinite set *Id* of names, a set *Var* of variables and a subset of special variables $\Upsilon \subset Var$ for fresh name creation. A ν-*PN* is a tuple $N = (P, T, F, \Sigma, \lambda)$, where P and T are finite disjoint sets, $F : (P \times T) \cup (T \times P) \to Var^{\oplus}$, Σ is the finite labelling alphabet, and $\lambda : T \to (\Sigma \cup \{\epsilon\})$ labels transitions.

A *marking* is a mapping $M : P \to Id^{\oplus}$. A *mode* is an injection $\sigma : Var(t) \to Id$. A transition t can be fired with mode σ for a marking M if for all $p \in P$, $\sigma(F(p,t)) \subseteq M(p)$ and for every $\nu \in \Upsilon$, $\sigma(\nu) \notin M(p)$ for all p. In that case we have $M \overset{\lambda(t)}{\to} M'$, where $M'(p) = (M(p) - \sigma(F(p,t))) + \sigma(F(t,p))$ for all $p \in P$.

Markings can be identified up to renaming of names. Thus, markings of a ν-*PN* with k places can be represented as elements in $(\mathbb{N}^k)^{\oplus}$, each tuple representing the occurrences in each place of one name [16]. E.g., if $P = \{p_1, p_2\}$ and M is such that $M(p_1) = \{\!|a, a, b|\!\}$ and $M(p_2) = \{\!|b|\!\}$, then we can represent M as $\{\!|(2,0),(1,1)|\!\}$.

The i-th place of a ν-*PN* is *bounded* if every tuple $(n_1, ..., n_k)$ in every reachable marking satisfies $n_i \leq b$, for some $b \geq 0$. Therefore, a bounded place may contain arbitrarily many names, provided each of them appears a bounded number of times.

Let us denote by ν-PN_k the class of ν-*PN* with k unbounded places. If a net in ν-PN_k has m places bounded by some $b \geq 0$, then we can use as state space $(Q \times \mathbb{N}^k)^{\oplus}$ with $Q = \{0, ..., b\}^m$ (finite and non-empty). Thus, the state space of nets in ν-PN_k is in $\mathbf{X}_k^{\oplus} = \{(Q \times \mathbb{N}^k)^{\oplus} \mid Q \text{ finite}\}$. Analogously, the class PDN_k of *PDN* with k unbounded places has $\mathbf{X}_k^{*} = \{(Q \times \mathbb{N}^k)^{*} \mid Q \text{ finite}\}$ as set of state spaces. Moreover, we take $\mathbf{X}^{\oplus} = \{(\mathbb{N}^k)^{\oplus} \mid k > 0\}$ and $\mathbf{X}^{*} = \{(\mathbb{N}^k)^{*} \mid k > 0\}$.

Proposition 9. *For every $k \geq 0$, ν-PN_k and PDN_k are self-witnessing.*

Proof. The proof for PDN_k is in the long version [6]. Let us see it for ν-PN_k. Let $(Q \times \mathbb{N}^k)^{\oplus} \in \mathbf{X}_k^{\oplus}$. We consider an alphabet $\Sigma = \{a_q \mid q \in Q\} \cup \{a_1, ..., a_k\}$ and we define $\gamma : \Sigma^* \to (Q \times \mathbb{N}^k)^{\oplus}$ by

$$\gamma(a_{q_1} a_1^{n_1^1} ... a_k^{n_1^k} .... a_{q_l} a_1^{n_l^1} ... a_k^{n_l^k}) = \{\!|(q_1, n_1^1, ..., n_1^k), ..., (q_l, n_l^1, ..., n_l^k)|\!\}$$

Let us build N in ν-PN_k such that $L(N) \cap dom(\gamma)\overline{dom(\gamma)} = L_\gamma$. Assume $Q = \{q_1, ..., q_r\}$. Fig. 1 shows the case with $k = 1$ and $r = 2$.

The only unbounded places of N are $p_1, ..., p_k$ (hence $N \in \nu$-PN_k). We consider $q_1, ..., q_r$ as places, a place st that stores all the names that have been used (once each name, hence bounded), and places $c_0, c_1, ..., c_k$ containing one name in mutual exclusion. When the name is in c_0 it is non-deterministically copied in some q (action labelled by a_q), and moved to c_1. For every, $1 \leq i \leq k$, when the name is in c_i it can be copied arbitrarily often to p_i (labelled by a_i). At any time, this name can be transferred to c_{i+1} when $i < k$ or to st for $i = k$ (labelled by ϵ). In the last case a fresh name is put in c_0 (thanks to $\nu \in \Upsilon$).

The second phase is analogous, with control places $d_0, d_1, ..., d_{k+1}$, marked in mutual exclusion with names taken from st. At any point, the name in d_{k+1} can be removed, and one name moved from st to d_0 (labelled by ϵ). That name must appear in some q. Thus, for each q we have a transition that removes the name from d_0 and q and puts it in d_1 (labelled by $\bar{a}_q$). For each $1 \leq i \leq k$, the name in d_i can be removed zero or more times from p_i (labelled by $\bar{a}_i$). At any point, the name is transferred from d_i to d_{i+1} (labelled by ϵ).

The initial and final marking is that with a name in c_0 and another name in d_{k+1} (and empty elsewhere). It holds that $L(N) \cap dom(\gamma)\overline{dom(\gamma)} = L_\gamma$, so we conclude.

Notice that since ν-PN_k and PDN_k are self-witnessing for every $k \geq 0$, so are ν-PN and PDN.

Proposition 10. *$\mathbf{X}_1^* \not\subseteq_{refl} \mathbf{X}^{\oplus}$, $\mathbf{X}_{k+1}^{\oplus} \not\subseteq_{refl} \mathbf{X}_k^{\oplus}$ and $\mathbf{X}_{k+1}^* \not\subseteq_{refl} \mathbf{X}_k^*$ for all k.*

Proof. $\mathbf{X}_1^* \not\subseteq_{refl} \mathbf{X}^{\oplus}$ holds because $ot(\mathbb{N}^*) = \omega^{\omega^\omega} \not\leq \omega^{\omega^k} = ot((\mathbb{N}^k)^{\oplus})$, so that $\mathbb{N}^* \not\subseteq_{refl} (\mathbb{N}^k)^{\oplus}$ for all k. The others are obtained similarly, considering that $ot((Q \times \mathbb{N}^k)^{\oplus}) = \omega^{\omega^k * |Q|}$ and $ot((Q \times \mathbb{N}^k)^*) = \omega^{\omega^{\omega^k * |Q|}}$.

Corollary 3. *ν-$PN \prec PDN$. Moreover, $PDN_1 \not\preceq \nu$-PN.*

Proof. ν-$PN \preceq PDN$ is from [15]. $PDN_1 \not\preceq \nu$-PN is a consequence of Prop. 4, considering that both classes are self-witnessing, and that $\mathbf{X}_1^* \not\subseteq_{refl} \mathbf{X}^{\oplus}$.

We can even be more precise in the hierarchy of Petri Nets extensions.

Proposition 11. *For any $k \geq 0$, ν-$PN_k \prec \nu$-PN_{k+1} and $PDN_k \prec PDN_{k+1}$.*

Proof. Clearly ν-$PN_k \preceq \nu$-PN_{k+1} and $PDN_k \preceq PDN_{k+1}$ for any $k \geq 0$. For the converses, again we can apply Prop. 4, considering that all the classes considered are self-witnessing and that $\mathbf{X}_{k+1}^{\oplus} \not\subseteq_{refl} \mathbf{X}_k^{\oplus}$ and $\mathbf{X}_{k+1}^* \not\subseteq_{refl} \mathbf{X}_k^*$ hold.

Finally, we can strengthen the result $AWN \prec \nu$-PN proved in [15] in a very straightforward way.

Proposition 12. *ν-PN*$_1 \not\preceq$ *AWN*

Proof. Both *AWN* and *ν-PN*$_1$ are self-witnessing, and $\mathbf{X}_1^{\oplus} \not\sqsubseteq_{refl} \{\mathbb{N}^k \mid k > 0\}$ because $\mathbb{N}^{\oplus} \not\sqsubseteq_{refl} \mathbb{N}^k$ for all k (indeed, $ot(\mathbb{N}^{\oplus}) = \omega^{\omega} \not\leq \omega^k = ot(\mathbb{N}^k)$). By Prop. 4 we conclude.

Again, the previous result is tight. Indeed, a *ν-PN* with no unbounded places can be simulated by a Petri net, so that *ν-PN*$_0 \simeq$ *VASS*.

6 Conclusion and Perspectives

To show a strict hierarchy of WSTS classes, we have proposed a generic method based on two principles: the ability of WSTS to recognize some specific witness languages linked to their state space, and the use of order theory to show the absence of order reflections from one wpo to another. This allowed us to unify some existing results, while also solving open problems. We summarize the current picture on expressiveness of WSTS below w.r.t number of resources and type of resources. On the other hand, showing equivalence between WSTS classes is a problem deeply linked to the semantics of the models, and hence that remains to be solved on a case-by-case basis.

Quantitative results. (All results are new.)

For every $k \in \mathbb{N}$ $VASS_k \prec VASS_{k+1} \not\preceq AWN_k$

For every $k, p \in \mathbb{N}$ $LCS(k,p) \prec LCS(k+1,p) \prec LCS(1,p+1)$

For every $k \in \mathbb{N}$ *ν-PN*$_k \prec$ *ν-PN*$_{k+1}$ and $PDN_k \prec PDN_{k+1}$

Qualitative results. (New results are *ν-PN* $\prec$ *DN* and *PDN* $\simeq$ *TdPN*)

VASS $\prec \mathcal{M} \prec$ *DN* $\simeq$ *PDN* $\simeq$ *TdPN*

where $\mathcal{M}$ is either *ν-PN* or *LCS*

TdPN [3] are Timed Petri nets and we have proved the related result in a companion report [5].

An interesting case that remains open is the relative expressiveness of *LCS* and *ν-PN*. Their state space are quite distinct but their order type are the same for some values of their parameters. We conjecture that there is no reflection from one to the other, but such a proof would require more than order type analysis.

As all the models that we have studied in this paper use a state space whose order type is bounded by ϵ_0, it is tempting to look at WSTS that would use a greater state space. It is known that the Kruskal ordering has an order type greater than ϵ_0 [17], even for unlabelled binary trees. However, studies of WSTS based on trees have been quite scarce [12]. We believe some interesting problems might lie in this direction.

Acknowledgements. We would like to thank the anonymous reviewers for their numerous comments and suggestions, that allowed us to greatly improve the quality of this paper.

References

1. Abdulla, P.A., Delzanno, G., Van Begin, L.: Comparing the Expressive Power of Well-Structured Transition Systems. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 99–114. Springer, Heidelberg (2007)
2. Abdulla, P.A., Delzanno, G., Van Begin, L.: A Language-Based Comparison of Extensions of Petri Nets with and without Whole-Place Operations. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 71–82. Springer, Heidelberg (2009)
3. Abdulla, P.A., Nylen, A.: Timed Petri Nets and BQOs. In: Colom, J.-M., Koutny, M. (eds.) ICATPN 2001. LNCS, vol. 2075, pp. 53–70. Springer, Heidelberg (2001)
4. Alur, R., Courcoubetis, C., Henzinger, T.A.: The Observational Power of Clocks. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 162–177. Springer, Heidelberg (1994)
5. Finkel, A., Bonnet, R., Haddad, S., Rosa-Velardo, F.: Comparing Petri Data Nets and Timed Petri Nets. LSV Research Report 10-23 (2010)
6. Finkel, A., Bonnet, R., Haddad, S., Rosa-Velardo, F.: Ordinal Theory for Expresiveness of Well-Structured Transition Systems. LSV Research Report 11-01 (2011)
7. Figueira, D., Figueira, S., Schmitz, S., Schnoebelen, P.: Ackermann and Primitive-Recursive Bounds with Dickson's Lemma. CoRR abs/1007.2989 (2010)
8. Finkel, A.: A generalization of the procedure of karp and miller to well structured transition systems. In: Ottmann, T. (ed.) ICALP 1987. LNCS, vol. 267, pp. 499–508. Springer, Heidelberg (1987)
9. Finkel, A., McKenzie, P., Picaronny, C.: A well-structured framework for analysing petri net extensions. Information and Computation 195(1-2), 1–29 (2004)
10. Geeraerts, G., Raskin, J., Van Begin, L.: Well-structured languages. Acta Informatica 44, 249–288 (2007)
11. de Jongh, D.H.J., Parikh, R.: Well partial orderings and hierarchies. Indagationes Mathematicae (Proceedings) 80, 195–207 (1977)
12. Kouchnarenko, O., Schnoebelen, P.: A Formal Framework for the Analysis of Recursive-Parallel Programs. In: Malyshkin, V.E. (ed.) PaCT 1997. LNCS, vol. 1277, pp. 45–59. Springer, Heidelberg (1997)
13. Lazic, R., Newcomb, T.C., Ouaknine, J., Roscoe, A.W., Worrell, J.: Nets with Tokens Which Carry Data. Fund. Informaticae 88(3), 251–274 (2008)
14. Rosa-Velardo, F., de Frutos-Escrig, D.: Name creation vs. replication in Petri Net systems. Fund. Informaticae 88(3), 329–356 (2008)
15. Rosa-Velardo, F., Delzanno, G.: Language-Based Comparison of Petri Nets with black tokens, pure names and ordered data. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 524–535. Springer, Heidelberg (2010)
16. Rosa-Velardo, F., de Frutos-Escrig, D.: Forward Analysis for Petri Nets with Name Creation. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 185–205. Springer, Heidelberg (2010)
17. Schmidt, D.: Well-partial orderings and their maximal order types. Fakultät für Mathematik der Ruprecht-Karls-Universität Heidelberg. Habilitationsschrift (1979)
18. Weiermann, A.: A Computation of the Maximal Order Type of the Term Ordering on Finite Multisets. In: Ambos-Spies, K., Löwe, B., Merkle, W. (eds.) CiE 2009. LNCS, vol. 5635, pp. 488–498. Springer, Heidelberg (2009)

Alternation Elimination for Automata over Nested Words*

Christian Dax and Felix Klaedtke

Computer Science Department, ETH Zurich, Switzerland

Abstract. This paper presents constructions for translating alternating automata into nondeterministic nested-word automata (NWAs). With these alternation-elimination constructions at hand, we straightforwardly obtain translations from various temporal logics over nested words from the literature like CaRet and μNWTL, and extensions thereof to NWAs, which correct, simplify, improve, and generalize the previously given translations. Our alternation-elimination constructions are instances of an alternation-elimination scheme for automata that operate over the tree unfolding of graphs. We obtain these instances by providing constructions for complementing restricted classes of automata with respect to the graphs given by nested words. The scheme generalizes our alternation-elimination scheme for word automata and the presented complementation constructions generalize existing complementation constructions for word automata.

1 Introduction

The regular nested-word languages [6] (a.k.a. visibly pushdown languages [5]) extend the classical regular languages by adding a hierarchical structure to words. Such hierarchical structures in linear sequences occur often and naturally. For instance, an XML document is a linear sequence of characters, where the opening and closing tags structure the document hierarchically. Another example from system verification are the traces of imperative programs, where the hierarchical structure is given by the calls and returns of subprograms. Many automata-theoretic methods for reasoning about regular languages carry over to regular nested-word languages. Instead of word automata one uses nested-word automata (NWAs) [6] or equivalently visibly pushdown automata [5], a restricted class of pushdown automata, where the input symbols determine when the pushdown automaton can push or pop symbols from its stack. For instance, model checking regular nested-word properties of recursive state machines, which can model control flows of imperative programs [3,4], and of Boolean programs [7], which are widely used as abstractions in software model checking, can be carried out in an automata-theoretic setting, similar to finite-state model checking [23]. That is, the traces of a recursive state machine or a Boolean program are described by an NWA and the negation of the specification, which is given as

* This work was partially supported by the Swiss National Science Foundation (SNF).

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 168–183, 2011.

a formula in a temporal logic over nested words like CaRet [4], NWTL [2], and μNWTL [10], is translated into a language-equivalent NWA. It is then checked whether the intersection of the automata's languages is empty.

In this paper, we view a nested word as a graph with *linear* and *hierarchical* edges. The nodes of the graph are the positions of the nested word. A linear edge connects two neighboring positions and a hierarchical edge connects every call with its matching return position. We present constructions for translating alternating automata that take as input the graphs of nested words into NWAs. These constructions are of immediate relevance for translating temporal logics over nested words like CaRet, NWTL, and μNWTL and extensions thereof to language-equivalent NWAs. A temporal-logic formula is first translated into such an alternating automaton and from this alternating automaton one obtains an NWA by applying such an alternation-elimination construction. Translations of declarative specification languages into alternating automata are usually rather direct and easy to establish due to the rich combinatorial structure of alternating automata. Translating an alternating automaton into a nondeterministic automaton is a purely combinatorial problem. Hence, using alternating automata as an intermediate step is a mathematically elegant way to formalize such translations and to establish their correctness.

We obtain the alternation-elimination constructions for automata that describe nested-word languages from a construction scheme, which we previously presented for word automata [11] and which we generalize in this paper to automata that operate over the tree unfolding of graphs. In a nutshell, the construction scheme shows that the problem of translating an alternating automaton into a nondeterministic automaton reduces to the problem of complementing an existential automaton, i.e., an automaton that nondeterministically inspects only a single branch in the tree unfolding of the given input graph. To obtain the instances of the construction scheme for nested words, we also provide complementation constructions for restricted classes of existential automata, namely, automata that operate over graphs that represent nested words.

The main benefit of our approach for translating temporal logics over nested words to NWAs is its simplicity and modularity compared to state-of-the-art approaches. By our scheme, complicated translations are divided into smaller independent parts. Moreover, ingredients of the presented constructions are based on existing well established and thoroughly optimized constructions and techniques for nondeterministic word automata, which we generalize to automata that operate over the tree unfolding of the graphs given by nested words. First, we extend our complementation constructions for classes of nondeterministic two-way co-Büchi word automata [11] to classes of existential co-Büchi automata, where the inputs are the graphs of nested words. Our new constructions take the non-local transitions, which stem from the hierarchical structure of nested words, of an existential automaton into account. Intuitively, in such transitions, the read-only head of the automaton jumps from a call directly to the corresponding return or vice versa. Second, in the presented alternation-elimination constructions for alternating parity automata, where the inputs are the graphs of nested words, we also use and generalize techniques and constructions from [13, 14, 22, 19] for word automata. Finally, as a by product, we obtain a complementation construction for NWAs along the lines of the construction in [13] for complementing nondeterministic Büchi word automata.

We see our contributions as follows. First, based on a general alternation-elimination scheme for automata that operate over the tree unfolding of graphs and several complementation constructions, we provide alternation-elimination constructions for the class of automata that take the graphs of nested words as input with the Büchi and the parity acceptance conditions. Second, we modularize, simplify, and correct existing translations from temporal logics over nested words to NWAs. Third, with the presented complementation constructions we illustrate that various constructions for word automata generalize with some modifications to constructions for automata that describe nested-word languages.

We proceed as follows. In Section 2, we recapitulate basic definitions and define alternating automata. In Section 3, we present our general alternation-elimination scheme. In Section 4, we present complementation constructions for restricted classes of existential automata with respect to nested-word languages. Furthermore, we instantiate our scheme with these constructions. Finally, in Section 5, we sketch applications of these instances. In particular, we present our translations of various temporal logics over nested words into language-equivalent NWAs. Omitted proof details can be found in the full version of the paper, which is publicly available from the authors' web pages.

2 Preliminaries

In this section, we fix the notation and terminology that we use in the remainder of the text.

Propositional Logic. We denote the set of *positive Boolean formulas* over the set P of propositions by $\mathsf{Bool}^+(P)$, i.e., $\mathsf{Bool}^+(P)$ consists of the formulas that are inductively built from the Boolean constants $\mathtt{tt}$ and $\mathtt{ff}$, the propositions in P, and the connectives $\vee$ and $\wedge$. For $M \subseteq P$ and $b \in \mathsf{Bool}^+(P)$, we write $M \models b$ iff b evaluates to true when assigning true to the propositions in M and false to the propositions in $P \setminus M$. Moreover, we write $M \mathrel{|\!\!\models} b$ if M is a *minimal model* of b, i.e., $M \models b$ and there is no $p \in M$ such that $M \setminus \{p\} \models b$.

Words and Trees. We denote the set of finite words over the alphabet Σ by Σ^*, the set of infinite words over Σ by Σ^ω, and the empty word by ε. The length of a word w is written as $|w|$, where $|w| = \omega$ when w is an infinite word. For a word w, w_i denotes the symbol of w at position $i < |w|$. We write $v \preceq w$ if v is a prefix of the word w.

A (Σ-labeled) *tree* is a function $t : T \to \Sigma$, where $T \subseteq \mathbb{N}^*$ satisfies the conditions: (i) T is prefix-closed (i.e., $v \in T$ and $u \preceq v$ imply $u \in T$) and (ii) if $vi \in T$ and $i > 0$ then $v(i-1) \in T$. The elements in T are called the *nodes* of t and the empty word ε is called the *root* of t. A node $vi \in T$ with $i \in \mathbb{N}$ is called a *child* of the node $v \in T$. A *branch* in t is a word $\pi \in \mathbb{N}^* \cup \mathbb{N}^\omega$ such that either

Table 1. Types of acceptance conditions

type	finite description α, acceptance condition A
	$\alpha = F \subseteq Q$
Büchi	$A := \{\pi \in Q^\omega \mid \inf(\pi) \cap F \neq \emptyset\}$
co-Büchi	$A := \{\pi \in Q^\omega \mid \inf(\pi) \cap F = \emptyset\}$
	$\alpha = \{F_0, \dots, F_{2k-1}\} \subseteq 2^Q$, where $F_0 \subseteq F_1 \subseteq \dots \subseteq F_{2k-1}$
parity	$A := \big\{\pi \in Q^\omega \bigm\vert \min\{i \mid F_i \cap \inf(\pi) \neq \emptyset\} \text{ is even}\big\}$
co-parity	$A := \big\{\pi \in Q^\omega \bigm\vert \min\{i \mid F_i \cap \inf(\pi) \neq \emptyset\} \text{ is odd}\big\}$
	$\alpha = \{(B_1, C_1), \dots, (B_k, C_k)\} \subseteq 2^Q \times 2^Q$
Rabin	$A := \bigcup_i \{\pi \in Q^\omega \mid \inf(\pi) \cap B_i \neq \emptyset \text{ and } \inf(\pi) \cap C_i = \emptyset\}$
Streett	$A := \bigcap_i \{\pi \in Q^\omega \mid \inf(\pi) \cap B_i = \emptyset \text{ or } \inf(\pi) \cap C_i \neq \emptyset\}$

$\pi \in T$ and π does not have any children, or π is infinite and every finite prefix of π is in T. We write $t(\pi)$ for the word $t(\varepsilon)t(\pi_0)t(\pi_0\pi_1)\dots t(\pi_0\pi_1\dots\pi_{n-1}) \in \Sigma^*$ if π is a finite word of length n and $t(\varepsilon)t(\pi_0)t(\pi_0\pi_1)\dots \in \Sigma^\omega$ if π is infinite.

Alternating Automata. In the following, we define alternating automata, where the inputs are graphs. Such an automaton is essentially an alternating tree automaton that operates over the tree unfolding of the given input.[1] We obtain the classical automata models for words and trees when viewing words and trees in a rather straightforward way as graphs of the following form and restricting the inputs to the respective class of graphs.

Let D be a nonempty finite set. We call the elements in D *directions.* A D-*skeleton* is a directed, edge-labeled, and pointed graph $\big(V, (E_d)_{d \in D}, v_I\big)$, where V is a set of vertices, the relation $E_d \subseteq V \times V$ describes the edges with label $d \in D$, and $v_I \in V$ is the source. We denote the set of labels of the outgoing edges of the vertex $v \in V$ by $\ell(v)$. For an alphabet Σ and a set $\mathcal{S}$ of D-skeletons, the set of *input graphs* $\Sigma^{\mathcal{S}}$ is the set of pairs (S, λ) with $S \in \mathcal{S}$ and $\lambda : V \to \Sigma$, where V is the set of vertices of S.

Let $\mathcal{S}$ be a nonempty set of D-skeletons. An *alternating* $\mathcal{S}$-*automaton* is a tuple $\mathcal{A} = \big(Q, \Sigma, (\delta_{D'})_{D' \subseteq D}, q_I, A\big)$, where Q is a finite set of states, Σ is a nonempty finite alphabet, $\delta_{D'} : Q \times \Sigma \to \mathsf{Bool}^+(Q \times D')$ is the transition function for the directions $D' \subseteq D$, $q_I \in Q$ is the initial state, and $A \subseteq Q^\omega$ is the acceptance condition. The acceptance condition A is usually specified in a certain finite way—the *type* of an acceptance condition. Commonly used types of acceptance conditions are listed in Table 1, where $\inf(\pi)$ denotes the set of states that occur infinitely often in $\pi \in Q^\omega$ and the integer k is the *index* of the automaton. If A is specified by the type τ, we say that $\mathcal{A}$ is an alternating τ $\mathcal{S}$-automaton. Moreover, if the type of the acceptance condition is clear from the context, we

[1] The reasons for having graphs as inputs is that it allows us to establish a broadly applicable alternation-elimination scheme (Section 3). In particular, we can use this automata model with the alternation-elimination scheme for translating temporal logics over nested words into NWAs (Section 5) by viewing nested words as graphs, where we restrict the inputs to that class of graphs (Section 4).

just give the finite description α instead of A. For instance, an alternating Büchi $\mathcal{S}$-automaton is given as a tuple $\big(Q, \Sigma, (\delta_{D'})_{D' \subseteq D}, q_I, \alpha\big)$ with $\alpha \subseteq Q$.

Let $\mathcal{A} = \big(Q, \Sigma, (\delta_{D'})_{D' \subseteq D}, q_I, A\big)$ be an alternating $\mathcal{S}$-automaton and $G \in \Sigma^{\mathcal{S}}$ with $G = (S, \lambda)$ and $S = \big(V, (E_d)_{d \in D}, v_I\big)$. A *run* of $\mathcal{A}$ on G is a tree $r : R \to V \times Q$ with some $R \subseteq \mathbb{N}^*$ such that $r(\varepsilon) = (v_I, q_I)$ and for each node $x \in R$ with $r(x) = (v, p)$, we have $M \models \delta_{\ell(v)}(p, \lambda(v))$, where

$$M := \big\{(q, d) \in Q \times D' \,\big|\, x \text{ has a child } y \text{ with } r(y) = (v', q) \text{ and } (v, v') \in E_d\big\} .$$

Roughly speaking, $\mathcal{A}$ starts scanning an input graph from the skeleton's initial vertex, where $\mathcal{A}$ is in its initial state. The label (v, p) of the node x in the run is the current configuration of $\mathcal{A}$. That is, $\mathcal{A}$ is currently in the state p and the read-only head is at the position v in the input graph. The transition $\delta_{D'}(p, \lambda(v))$ specifies a constraint that has to be fulfilled by the automaton's successor states, where D' is the set of labels $\ell(v)$ in which the read-only head can move at the current position. An infinite branch π in a run r with $r(\pi) = (v_0, q_0)(v_1, q_1)\dots$ is *accepting* if $q_0 q_1 \dots \in A$. The run r is *accepting* if every infinite branch in r is accepting. The *language* of $\mathcal{A}$ is the set $L(\mathcal{A}) := \{G \in \Sigma^{\mathcal{S}} \mid$ there is an accepting run of $\mathcal{A}$ on $G\}$.

We call an alternating $\mathcal{S}$-automaton $\mathcal{A} = \big(Q, \Sigma, (\delta_{D'})_{D' \subseteq D}, q_I, A\big)$ *existential* if $\delta_{D'}$ returns a disjunction for all inputs, for all $D' \subseteq D$. Note that a run r of an existential automaton consists of a single branch π. To increase readability, we call $r(\pi)$ also a run. Existential automata are closely related to nondeterministic automata in the sense that an existential automaton also nondeterministically chooses its successor state in a run with respect to the current configuration and its transition function. However, an existential automaton only inspects a single path of the input graph, since together with the chosen successor state it picks a single direction in which it moves its read-only head.

3 Alternation-Elimination Scheme

In this section, we generalize our alternation-elimination scheme for word automata, which we presented in [11], to automata that operate over graphs.

3.1 Reduction to Complementation

The scheme only applies to automata with an acceptance condition for which so-called memoryless runs are sufficient. Formally, for an alternating $\mathcal{S}$-automaton $\mathcal{A}$, we require that $L(\mathcal{A}) = M(\mathcal{A})$, where the set $M(\mathcal{A})$ is defined as follows. A run $r : R \to V \times Q$ of the alternating $\mathcal{S}$-automaton $\mathcal{A} = \big(Q, \Sigma, (\delta_{D'})_{D' \subseteq D}, q_I, A\big)$ on $(S, \lambda) \in \Sigma^{\mathcal{S}}$ with $S = \big(V, (E_d)_{d \in D}, v_I\big)$ is *memoryless* if equally labeled nodes have isomorphic subtrees, i.e., for all $x, y \in R$ and $z \in \mathbb{N}^*$, if $r(x) = r(y)$ then $xz \in R$ iff $yz \in R$ and whenever $xz \in R$ then $r(xz) = r(yz)$. We define $M(\mathcal{A}) := \{G \in \Sigma^{\mathcal{S}} \mid$ there is an accepting memoryless run of $\mathcal{A}$ on $G\}$. Obviously, $L(\mathcal{A}) \supseteq M(\mathcal{A})$. For an alternating $\mathcal{S}$-automata $\mathcal{A}$ with the Büchi, co-Büchi, parity, or Rabin acceptance condition, it is well known that the converse

$L(\mathcal{A}) \subseteq M(\mathcal{A})$ also holds. However, if $\mathcal{A}$ is, e.g., an alternating $\mathcal{S}$-automata with the Streett acceptance condition, then $L(\mathcal{A}) \subseteq M(\mathcal{A})$ does not hold in general.

Since the children of equally labeled nodes in a memoryless run $r : R \to V \times Q$ are also equally labeled, we can represent a memoryless run by the function $\sigma^r : V \times Q \to 2^{Q \times D}$, where

$$\sigma^r(v,q) := \big\{(q',d) \in Q \times D \,\big|\, \text{there are nodes } x, y \in R \text{ such that } y \text{ is a child of } x,\; r(x) = (v,q),\; r(y) = (v',q'),\text{ and } (v,v') \in E_d\big\}\,.$$

By "currying" the function σ^r, we obtain the function $\lambda^r : V \to \Gamma$, where Γ is the set of functions from Q to $2^{Q \times D}$. We represent the run r as the input graph $G^r := (S, \lambda^r) \in \Gamma^{\mathcal{S}}$. We point out that the graph representation of the run has the same skeleton S as the skeleton of the given input graph G.

We now define an existential $\mathcal{S}$-automaton $\mathcal{R}$ that scans input graphs in $(\Sigma \times \Gamma)^{\mathcal{S}}$, i.e., input graphs of $\mathcal{A}$ that are annotated with information about the configurations of the runs of $\mathcal{A}$. $\mathcal{R}$ refutes whenever the annotations correspond to an accepting memoryless run of $\mathcal{A}$ on $\mathcal{A}$'s input graph. Formally, $\mathcal{R}$ is the existential $\mathcal{S}$-automaton $\big(Q, \Sigma \times \Gamma, (\eta_{D'})_{D' \subseteq D}, q_I, Q^\omega \setminus A\big)$, where its transition function $\eta_{D'} : Q \times (\Sigma \times \Gamma) \to \mathsf{Bool}^+(Q \times D')$ for $D' \subseteq D$ is defined as

$$\eta_{D'}\big(q,(a,g)\big) := \begin{cases} \bigvee_{(p,d) \in g(q)} (p,d) & \text{if } g(q) \models \delta_{D'}(q,a), \\ \mathsf{tt} & \text{otherwise.} \end{cases}$$

Intuitively, $\mathcal{R}$ works as follows. It uses its nondeterminism to inspect a path in the skeleton of the input graph. There are two cases in which $\mathcal{R}$ accepts the given input graph. (1) The annotations on the inspected path do not correspond to a branch in a memoryless run of $\mathcal{A}$. (2) The annotations yield an infinite sequence of states that is not accepting for $\mathcal{A}$, i.e., the sequence is not in A.

The formal statement about $\mathcal{R}$'s language is given in Lemma 1 below, where we use the following notation. Let $G = (S, \lambda)$ be an input graph in $(\Sigma \times \Gamma)^{\mathcal{S}}$. $G_{\restriction \Sigma}$ denotes the input graph in $\Sigma^{\mathcal{S}}$ by projecting G's labeling to the first component, i.e., $G_{\restriction \Sigma} := (S, \lambda_{\restriction \Sigma})$ with $\lambda_{\restriction \Sigma}(v) := a$ for $\lambda(v) = (a,g)$. Analogously, $G_{\restriction \Gamma}$ denotes the input graph in $\Gamma^{\mathcal{S}}$ with the skeleton S and the labeling $\lambda_{\restriction \Gamma}(v) := g$.

Lemma 1. *For any input graph $G \in (\Sigma \times \Gamma)^{\mathcal{S}}$, it holds*

$G \notin L(\mathcal{R})$ *iff* *there is an accepting memoryless run r of $\mathcal{A}$ on $G_{\restriction \Sigma}$ such that $G_{\restriction \Gamma}$ and G^r are isomorphic.*

The following theorem allows us to reduce the problem of constructing for $\mathcal{A}$ a language-equivalent nondeterministic automaton to the problem of complementing $\mathcal{R}$. Note that from an existential automaton that accepts the complement of $\mathcal{R}$, we easily obtain a nondeterministic automaton that accepts $L(\mathcal{A})$ by projecting the alphabet $\Sigma \times \Gamma$ to Σ. The benefit of this reduction is that it only requires a complementation construction for existential automata. In Section 4, we give such complementation constructions for specific automata classes.

Theorem 1. *If $L(\mathcal{A}) = M(\mathcal{A})$ then $L(\mathcal{A}) = \{G_{\restriction \Sigma} \mid G \notin L(\mathcal{R})\}$.*

3.2 Inherited Properties

In the following, we show that the existential $\mathcal{S}$-automaton $\mathcal{R}$ from Section 3.1 inherits properties from the alternating $\mathcal{S}$-automaton $\mathcal{A}$. We exploit these properties in our complementation constructions in Section 4.

Let $\mathcal{A} = \big(Q, \Sigma, (\delta_{D'})_{D' \subseteq D}, A\big)$ be an alternating $\mathcal{S}$-automaton and let $W \subseteq D$. The automaton $\mathcal{A}$ is *W-way* if $\delta_{D'}(q,a) \in \mathsf{Bool}^+(Q \times (D' \cap W))$, for all $D' \subseteq D$, $q \in Q$, and $a \in \Sigma$. Intuitively, $\mathcal{A}$ moves its read-only head only along edges in the input graph that are labeled by directions in W. A weaker condition on the allowed movements of the automaton's read-only head is the following. Intuitively, the automaton $\mathcal{A}$ is eventually W-way when it eventually moves its read-only head only along edges that are labeled by directions in W. Formally, this condition is defined as follows. Let $G \in \Sigma^{\mathcal{S}}$ be an input graph with $G = (S, \lambda)$ and $S = \big(V, (E_d)_{d \in D}, v_I\big)$. We define $\Pi_G(\mathcal{A})$ as the set of words $(q_0, v_0)(q_1, v_1) \ldots \in (Q \times V)^\omega$ with $(q_0, v_0) = (q_I, v_I)$ and for all $i \in \mathbb{N}$, there is some $d \in \ell(v_i)$ and a minimal model M of $\delta_{\ell(v_i)}(q_i, \lambda(v_i))$ such that $(q_{i+1}, d) \in M$ and $(v_i, v_{i+1}) \in E_d$. The automaton $\mathcal{A}$ is *eventually W-way* if for every input graph $G \in \Sigma^{\mathcal{S}}$ and every word $(q_0, v_0)(q_1, v_1) \ldots \in \Pi_G(\mathcal{A})$, there is an index $n \in \mathbb{N}$ such that for all $i \geq n$, we have $(v_i, v_{i+1}) \in E_d$, for some $d \in W$.

The following definition of *weak* automata generalizes the standard definition [13, 17], where the automata's acceptance condition is a Büchi acceptance condition. Let $\mathcal{A}$ be the alternating $\mathcal{S}$-automaton $\big(Q, \Sigma, (\delta_{D'})_{D' \subseteq D}, q_I, A\big)$. We call a state set $S \subseteq Q$ *accepting* if $\inf(r(\pi)) \subseteq S$ implies $r(\pi) \in A$, for each run r and each infinite branch π in r. Analogously, we call S *rejecting* if $\inf(r(\pi)) \subseteq S$ implies $r(\pi) \notin A$, for each run r and each infinite branch π in r. The automaton $\mathcal{A}$ is *weak* if there is a partition $Q_1, \ldots, Q_n$ of Q such that (i) each Q_i is either accepting or rejecting and (ii) there is a partial order $\preceq$ on the Q_is such that for every $p \in Q_i$, $q \in Q_j$, $a \in \Sigma$, $D' \subseteq D$, and $d \in D'$, if (q, d) occurs in $\delta_{D'}(p, a)$ then $Q_j \preceq Q_i$. The automaton $\mathcal{A}$ is *very weak* if each Q_i is a singleton. The intuition of weakness is that each infinite branch of a run of a weak automaton that gets trapped in one of the Q_is is accepting iff Q_i is accepting.

Lemma 2. *Let $\mathcal{R}$ be the existential $\mathcal{S}$-automaton as defined in Section 3.1 for the $\mathcal{S}$-automaton $\mathcal{A}$. Moreover, let $W \subseteq D$. The following properties hold.*

(i) If $\mathcal{A}$ is (eventually) W-way then $\mathcal{R}$ is (eventually) W-way.

(ii) If $\mathcal{A}$ is (very) weak then $\mathcal{R}$ is (very) weak.

4 Instances for Automata over Nested Words

In this section, we present alternation-elimination constructions for several classes of automata that take as input the graphs of nested words. We obtain these constructions from our alternation-elimination scheme by providing complementation constructions for existential automata.

4.1 Automata over Nested Words

Nested words [5,6] are linear sequences equipped with a hierarchical structure. In this paper, we impose this structure by tagging letters with brackets.[2] More formally, a *nested word* over Σ is a word over the tagged alphabet $\hat{\Sigma} := \Sigma_{int} \cup \Sigma_{call} \cup \Sigma_{ret}$, where the sets $\Sigma_{int} := \Sigma$, $\Sigma_{call} := \{\langle a \mid a \in \Sigma\}$, and $\Sigma_{ret} := \{a\rangle \mid a \in \Sigma\}$ are pairwise disjoint. A position $i \in \mathbb{N}$ in a nested word $w \in \hat{\Sigma}^\omega$ with $w_i \in \Sigma_{int}$ is an *internal position*. Similarly, if $w_i \in \Sigma_{call}$ then i is a *call position* and if $w_i \in \Sigma_{ret}$ then i is a *return position*. Observe that with the attached brackets $\langle$ and $\rangle$ to the letters in Σ, we implicitly group words into subwords. This grouping can be nested. However, not every bracket at a position in a nested word needs to have a matching bracket. The call and return positions in a nested word without matching brackets are called *pending*.

Intuitively speaking, a *nested-word (Büchi) automaton* [5,6], NWA for short, $\mathcal{N}$ is a nondeterministic pushdown automaton[3] that pushes a stack symbol when reading a letter in Σ_{call}, pops a stack symbol when reading a letter in Σ_{ret} (in case it is not the bottom stack symbol), and does not use its stack when reading a letter in Σ_{int}. The NWA $\mathcal{N}$ accepts a word in $\hat{\Sigma}^\omega$ if there is run on that word that visits infinitely often an accepting state. We denote the set of nested words for which there is an accepting run of $\mathcal{N}$ by $L(\mathcal{N})$.

In the following, we view nested words as input graphs, where the hierarchical structure is made explicit by adding to each position the edges that point to its successor and predecessor positions. Formally, these input graphs with their skeletons are defined as follows. Let D be the set $\{-2,-1,0,1,2\}$ and let $\mathcal{S}$ be the set of D-skeletons $S = \big(V, (\curvearrowright_d)_{d\in D}, v_I\big)$, where $V = \mathbb{N}$, $v_I = 0$, and the edge relations are as follows: $\curvearrowright_0$ is the identity relation over $\mathbb{N}$, $\curvearrowright_1$ is the successor relation over $\mathbb{N}$, and $\curvearrowright_2$ is a *matching jump relation*. That is, for all $i, j \in \mathbb{N}$, the relation $\curvearrowright_2$ satisfies the conditions (1) if $i \curvearrowright_2 j$ then $i < j$, (2) $|\{k \mid i \curvearrowright_2 k\}| \leq 1$ and $|\{k \mid k \curvearrowright_2 j\}| \leq 1$, and (3) if $i \curvearrowright_2 j$ then there are no $i', j' \in \mathbb{N}$ with $i' \curvearrowright_2 j'$ and $i < i' \leq j < j'$. The relations $\curvearrowright_{-1}$ and $\curvearrowright_{-2}$ are the inverses of $\curvearrowright_1$ and $\curvearrowright_2$, respectively. For a nested word $w \in \hat{\Sigma}^\omega$, the input graph G_w makes the matching jump relation, which is implicitly given by w, explicit. That is, the D-skeleton $S = \big(\mathbb{N}, (\curvearrowright_d)_{d\in D}, 0\big) \in \mathcal{S}$ and the labeling $\lambda : \mathbb{N} \to \hat{\Sigma}$ of the input graph G_w fulfill the following conditions: (a) For all $i \in \mathbb{N}$, it holds $\lambda(i) = w_i$. (b) For all $i, j \in \mathbb{N}$, if $i \curvearrowright_2 j$ then $\lambda(i) \in \Sigma_{call}$ and $\lambda(j) \in \Sigma_{ret}$. (c) Pending call and return positions do not cross, i.e., for all $k \in \mathbb{N}$ with $\lambda(k) \in \Sigma_{call}$, if there is no $k' \in \mathbb{N}$ with $k \curvearrowright_2 k'$ then for all $j > k$ with $\lambda(j) \in \Sigma_{ret}$, there is some $i \in \mathbb{N}$ with $i \curvearrowright_2 j$. (d) The pending positions do not cross with the matching jump

[2] In [6], nested words are differently defined by not leaving the hierarchical structure implicit by tagging letters with brackets but by making it explicit with a so-called *matching relation* $\rightsquigarrow \subseteq (\{-\infty\} \cup \mathbb{N}) \times (\mathbb{N} \cup \{+\infty\})$. Both definitions are equivalent in the sense that there is a straightforward bijection between them [6].

[3] We point out that the stack in the definition in [6] of nested-word automata is implicit. Due to space limitations, we omit the precise definition of nested-word automata.

relation $\curvearrowright_2$, i.e., for all $k \in \mathbb{N}$ with $\lambda(k) \in \Sigma_{call} \cup \Sigma_{ret}$, if there is no $k' \in \mathbb{N}$ with $k \curvearrowright_2 k'$ or $k' \curvearrowright_2 k$ then there are no $i, j \in \mathbb{N}$ with $i \curvearrowright_2 j$ and $i < k < j$.

The following theorem shows that alternating automata are expressive enough to describe the class of nested-word languages recognizable by NWAs.

Theorem 2. *For every NWA $\mathcal{N}$, there is an alternating Büchi $\mathcal{S}$-automaton $\mathcal{A}$ such that for every nested word $w \in \hat{\Sigma}^\omega$, we have $w \in L(\mathcal{N})$ iff $G_w \in L(\mathcal{A})$. Furthermore, $\mathcal{A}$ is $\{1,2\}$-way and has $\mathcal{O}(n^2 s)$ states, where n is the number of states of $\mathcal{N}$ and s is the number of $\mathcal{N}$'s stack symbols.*

This result might be surprising since an NWA processes a nested word sequentially and has a stack to store additional information at call positions, which it can use later at the corresponding matching return positions. An alternating automaton does not have a stack. However, instead each node in the input graph of a nested-word explicitly carries the information whether it is a pending or non-pending position. Moreover, for a non-pending position, the matching return or call position, respectively, is also explicitly given to the alternating automaton.

The reason for the blowup in the alternating automaton's state space is that the alternating automaton splits the computation at each non-pending call position, which must synchronize at the corresponding return position. This synchronization is implemented by guessing and causes a blow-up of the factor $\mathcal{O}(ns)$ in the state space. We omit the details of this transformation construction since it is similar to a construction in [10] for so-called jumping automata, which are very similar to our alternating automata when restricting their inputs to the graph representation of nested words.

4.2 Complementing Existential co-Büchi Automata

In this subsection, we present a complementation construction that translates an eventually $\{1,2\}$-way existential co-Büchi $\mathcal{S}$-automaton $\mathcal{A}$ into an NWA $\mathcal{N}$ with $L(\mathcal{N}) = \{w \in \hat{\Sigma}^\omega \mid G_w \notin L(\mathcal{A})\}$. We also optimize this construction for more restricted automata classes. Recall that we immediately obtain translations of alternating Büchi automata over the graph representation of nested words to NWAs by instantiating our alternating-elimination scheme with these complementation constructions. The complementation constructions utilize the following lemma that characterizes the graph representations of nested words that are not accepted by the eventually $\{1,2\}$-way existential co-Büchi $\mathcal{S}$-automaton $\mathcal{A}$.

In the following, we abbreviate *existential co-Büchi $\mathcal{S}$-automaton* by the acronym ECA and assume that $\mathcal{A} = \big(Q, \hat{\Sigma}, (\delta_{D'})_{D' \subseteq D}, q_I, F\big)$. Furthermore, for $D' \subseteq D$, $\delta^d_{D'}(P, a)$ denotes the set of states that can be reached from a state in $P \subseteq Q$ by reading the letter $a \in \hat{\Sigma}$ and following a d-labeled edge, i.e., $\delta^d_{D'}(P, a) := \bigcup_{p \in P} \{q \mid \text{the proposition } (q, d) \text{ occurs in } \delta_{D'}(p, a)\}$.

Lemma 3. *For the eventually $\{1,2\}$-way ECA $\mathcal{A}$ and a nested word $w \in \hat{\Sigma}^\omega$, we have $G_w \notin L(\mathcal{A})$ iff there are words $R \in (2^Q)^\omega$ and $S \in (2^{Q \setminus F})^\omega$ that fulfill the following conditions, where $(\curvearrowright_d)_{d \in D}$ is the family of edge relations of the D-skeleton of G_w:*

(1) $q_I \in R_0$.
(2) *For all* $i, j \in \mathbb{N}$ *and* $d \in D$ *with* $i \curvearrowright_d j$, *we have* $\delta^d_{\ell(i)}(R_i, w_i) \subseteq R_j$.
(3) *For all* $i \in \mathbb{N}$ *and* $q \in R_i$, *we have* $\emptyset \not\models \delta_{\ell(i)}(q, w_i)$.
(4) $S_0 = R_0 \setminus F$.
(5) *For all* $i, j \in \mathbb{N}$ *and* $d \in D$ *with* $d > 0$ *and* $i \curvearrowright_d j$, *we have* $\delta^d_{\ell(i)}(S_i, w_i) \setminus F \subseteq S_j$.
(6) *There are infinitely many* $n \in \mathbb{N}$ *such that* $S_n = \emptyset$, $S_{n+1} = R_{n+1} \setminus F$, *and for all* $i, j \in \mathbb{N}$ *with* $i \curvearrowright_2 j$ *and* $i \leq n$, *we have* $j \leq n$.

The conditions (1) and (2) ensure that the word R contains all the runs (h_0, q_0) $(h_1, q_1) \ldots$ of the existential automaton $\mathcal{A}$ on the given input graph, i.e., $q_i \in R_{h_i}$, for all $i \in \mathbb{N}$. The conditions (3) to (6) on the words R and S ensure that all the runs are rejecting. Recall that an input graph is rejected if it is not accepted by a finite run and every infinite run visits a state in F infinitely often. Condition (3) ensures that there is no finite accepting run. All the infinite runs are rejecting if the word R can be split into infinitely nonempty segments such that each run of the existential automaton that starts at the beginning of a segment will visit a state in F before reaching the end of the segment. The conditions (4) to (6) on the word S ensure the existence of such a splitting. In particular, the ns from condition (6) mark the end positions of the segments in the splitting.

We remark that we have given in [11] a similar characterization for word automata. The main differences to nested words are as follows. First, the conditions (2) and (5) additionally take the non-local moves of the existential automaton between matched call and return positions into account. Second, condition (6) additionally requires that a segment in the splitting of the word R must not end between a call and its matching return position. Without this additional requirement there might be runs that pass the end of a segment with a non-local move without visiting a state in F.

We now turn to the construction of the NWA, which generalizes our construction in [11] for complementing the word language of an eventually $\{1\}$-way ECA, which in turn on is based on the breakpoint construction [16]. The additional constraints for the non-local moves in the conditions (2), (5), and (6) are handled by using the stack of the NWA. In particular, whenever the NWA is at a non-pending call position, it guesses its configuration at the matching return position and pushes it on the stack. When reaching the matching return position, it checks the correctness of the guess by popping an element from the stack. Furthermore, the NWA uses the stack to recognize whether it has processed a matched call while not having reached its matching return position yet by using a bit that is pushed on the stack at non-pending calls and popped from the stack at their matching returns.

Theorem 3. *For an eventually* $\{1, 2\}$*-way ECA* $\mathcal{A}$ *with* n *states, there is an NWA* $\mathcal{N}$ *with* $\mathcal{O}(2^{4n})$ *states,* $\mathcal{O}(2^{4n})$ *stack symbols, and* $L(\mathcal{N}) = \{w \in \hat{\Sigma}^\omega \mid G_w \notin L(\mathcal{A})\}$.

In the following, we optimize our complementation construction for restricted classes of eventually $\{1, 2\}$-way ECAs. When $\mathcal{A}$ is also very weak, we can characterize the graph representations of nested words that are not accepted by $\mathcal{A}$

by similar conditions as those given in Lemma 3. However, the existence of the word S together with the conditions (4) and (5) are not required anymore and condition (6) is replaced by the following condition:

> There is no $q \in Q \setminus F$ and no $h \in \mathbb{N}^\omega$ such that $q \in R_{h_0}$ and for all $j \in \mathbb{N}$, there is a direction $d \in \{1,2\}$ such that $h_j \curvearrowright_d h_{j+1}$ and $q \in \delta^d_{\ell(h_j)}(q, w_{h_j})$. (6')

Intuitively, condition (6') requires that no run of the existential automaton gets trapped in a state in $Q \setminus F$.

We exploit this specialized characterization to optimize our complementation construction from Theorem 3. Intuitively, the NWA checks that no run of the existential automaton $\mathcal{A}$ gets trapped in a state in $Q \setminus F$. Again, the construction is similar to a construction in [11] for complementing the word language of very weak, eventually $\{1\}$-way ECAs. However, a subtle difference is that if a run does not get trapped in a state in $Q \setminus F$ between a non-pending call position and the corresponding return position then we additionally must ensure that the run also does not get trapped along the hierarchical edges that directly connect call positions with their matching return positions.

Theorem 4. *For a very weak, eventually $\{1,2\}$-way ECA $\mathcal{A}$ with n states, there is an NWA $\mathcal{N}$ with $\mathcal{O}(2^{2^n}n)$ states, $\mathcal{O}(2^{2^n}n)$ stack symbols, and $L(\mathcal{N}) = \{w \in \hat{\Sigma}^\omega \mid G_w \notin L(\mathcal{A})\}$.*

Finally, we consider the case where $\mathcal{A}$ is $\{1,2\}$-way for which we can simplify condition (2), since the automaton moves its read-only head only forward:

> For all $i, j \in \mathbb{N}$ and $d \in D$ with $d > 0$ and $i \curvearrowright_d j$, we have $\delta^d_{\ell(i)}(R_i, w_i) \subseteq R_j$. (2')

We directly obtain the following two theorems as special cases of the Theorems 3 and 4, respectively. In a nutshell, we reduce the state space of the NWA by removing the state components that are used to check the consistency of the transitions that move the read-only head along the backward edges $\{-1,-2\}$.

Theorem 5. *For a $\{1,2\}$-way ECA $\mathcal{A}$ with n states, there is an NWA $\mathcal{N}$ with $\mathcal{O}(2^{2n})$ states, $\mathcal{O}(2^{2n})$ stack symbols, and $L(\mathcal{N}) = \{w \in \hat{\Sigma}^\omega \mid G_w \notin L(\mathcal{A})\}$.*

Theorem 6. *For a very weak, $\{1,2\}$-way ECA $\mathcal{A}$ with n states, there is an NWA $\mathcal{N}$ with $\mathcal{O}(2^n n)$ states, $\mathcal{O}(2^n n)$ stack symbols, and $L(\mathcal{N}) = \{w \in \hat{\Sigma}^\omega \mid G_w \notin L(\mathcal{A})\}$.*

4.3 Alternation Elimination for Parity Automata

In this subsection, we present constructions that translate an alternating parity $\mathcal{S}$-automaton $\mathcal{A}$, APA for short from now on, into an NWA $\mathcal{N}$ with $L(\mathcal{N}) = \{w \in \hat{\Sigma}^\omega \mid G_w \in L(\mathcal{A})\}$.

Our first alternating-elimination construction assumes that the given APA $\mathcal{A}$ is eventually $\{1,2\}$-way. The construction comprises two steps: We first translate

$\mathcal{A}$ into an alternating Büchi automaton $\mathcal{A}'$ from which we then obtain in a second construction step the NWA $\mathcal{N}$. In the second construction step, we use an optimized variant of the alternating-elimination construction based on the complementation construction from Theorem 3 that exploits the fact that the runs of $\mathcal{A}'$ have some special form. We remark that both construction steps use and generalize techniques from [13,14] for complementing nondeterministic automata over infinite words.

Theorem 7. *For an eventually $\{1,2\}$-way APA $\mathcal{A}$ with index k and n states, there is an NWA $\mathcal{N}$ with $2^{\mathcal{O}(nk\log n)}$ states, $2^{\mathcal{O}(nk\log n)}$ stack symbols, and $L(\mathcal{N}) = \{w \in \hat{\Sigma}^\omega \mid G_w \in L(\mathcal{A})\}$.*

By some additional work, we obtain the more general alternation-elimination construction for APAs, where we do not require that the given APA is eventually $\{1,2\}$-way. Recall that by our alternation-elimination scheme, it suffices to give a construction for complementing existential parity automata over nested words. The first ingredient of that complementation construction is a generalization of Shepherdson's translation [19,21] of 2-way nondeterministic finite word automata to deterministic ones that are 1-way. This generalization is obtained with only minor modifications and translates $\{-2,-1,0,1,2\}$-way existential automata to existential $\{1,2\}$-way automata. The second ingredient is a complementation construction for existential $\{1,2\}$-way automata, which we easily obtain from Theorem 7 by dualizing [18] the transition function of the given automaton and its acceptance condition, i.e., we swap the Boolean connectives ($\wedge$ and $\vee$) and the Boolean constants (tt and ff) in the automaton's transitions, and we complement its acceptance condition, which can be easily done by incrementing the parities of the states by 1. By instantiating our alternation-elimination scheme with a combination—along the same lines as in [20,22]—of these two ingredients we obtain the following result.

Corollary 1. *For an APA $\mathcal{A}$ with index k and n states, there is an NWA $\mathcal{N}$ with $2^{\mathcal{O}((nk)^2)}$ states, $2^{\mathcal{O}((nk)^2)}$ stack symbols, and $L(\mathcal{N}) = \{w \in \hat{\Sigma}^\omega \mid G_w \in L(\mathcal{A})\}$.*

5 Applications and Concluding Remarks

A first and immediate application of our alternation-elimination constructions is a construction for complementing NWAs: For a given NWA $\mathcal{N}$, we first construct by Theorem 2 a $\{1,2\}$-way alternating Büchi automaton $\mathcal{A}$. We complement $\mathcal{A}$'s language by dualizing $\mathcal{A}$ [18]. Note that $\mathcal{A}$'s Büchi acceptance condition becomes a co-Büchi acceptance condition in the dualized automaton. Since a co-Büchi acceptance condition can be written as a parity acceptance condition with index 2, we can apply Theorem 7 to the dualized automaton and obtain an NWA $\bar{\mathcal{N}}$ with $L(\bar{\mathcal{N}}) = \hat{\Sigma}^\omega \setminus L(\mathcal{N})$. The NWA $\bar{\mathcal{N}}$ has $2^{\mathcal{O}(n^2 s\log ns)}$ states and stack symbols, where n is the number of states of $\mathcal{N}$ and s is the number of $\mathcal{N}$'s stack symbols.

This construction generalizes the complementation construction in [13] from nondeterministic Büchi word automata to NWAs. However, observe that we obtain a worse upper bound, namely, $2^{\mathcal{O}(n^2 s\log ns)}$ instead of $2^{\mathcal{O}(n\log n)}$. One reason is that the construction for NWAs has to take the stack into account. Another reason is that we first translate the NWA $\mathcal{A}$ by Theorem 2 into an alternating automaton that does not have a stack but takes the graph representation of nested words as inputs. This translation causes a blowup of the factor $\mathcal{O}(ns)$ in the automaton's state space. It is also worth pointing out that our complementation construction based on alternating automata does not match the best known upper bound $2^{\mathcal{O}(n^2)}$ for complementing NWAs [6]. This better upper bound is achieved by splitting the complementation construction into two separate constructions, which are later combined by a simple product construction. Only one of these two constructions involves a complementation construction, where only nondeterministic Büchi word automata need to be complemented. It remains open whether our complementation construction based on Theorem 7 can be optimized so that it matches or improves the upper bound $2^{\mathcal{O}(n^2)}$.

Our second and main application area of the presented alternation-elimination constructions is the translation of temporal logics over nested words into NWAs for effectively solving the satisfiability problem and the model-checking problem for recursive state machines and Boolean programs. From the constructions in Section 4, we straightforwardly obtain such translations, which we sketch in the following and which improve, extend, and correct previously presented translations. Overall, our translations together with our results in [11] and [12] demonstrate that complementation constructions for restricted classes of nondeterministic automata are at the core in translating temporal logics into nondeterministic automata; thus they are also at the core in the automata-theoretic approach to model checking and satisfiability checking.

In [10], Bozzelli introduces the temporal logic μNWTL, which extends the linear-time μ-calculus [8] by next modalities for the calls and returns in nested words. μNWTL has the same expressive power as NWAs. Our alternation-elimination scheme allow us to modularize and optimize Bozzelli's monolithic translation to NWAs. Similar to Bozzelli, we first translate a μNWTL formula into an alternating parity automaton (alternating jump automaton in Bozzelli's paper, respectively) with k parities, where k is the alternation depth of the given μNWTL formula. The size of the automaton is linear in the formula length. We then apply Corollary 1 to obtain an NWA. The size of the resulting NWA is $2^{\mathcal{O}((nk)^2)}$, where n is the size of the alternating parity automaton. For formulas that do not refer to the past or only in a restricted way such that the alternating parity automaton is eventually $\{1, 2\}$-way, we can use Theorem 7 to reduce this upper bound to $2^{\mathcal{O}(nk \log n)}$.

In [2, 4], the respective authors introduce the temporal logics CaRet and NWTL, which extend the classical linear-time temporal logic LTL. The extensions consist of new modalities that take the hierarchical structure of nested words into account. In other words, the new modalities allow one to express properties along the different paths in a nested word. NWTL subsumes CaRet and is first-order complete. For both these logics, the authors of the respective papers also provide translations into NWAs. Their translations are direct, i.e., they do not use alternating automata as an intermediate step. Although the techniques used in such direct translations are rather standard, they are complex and their correctness proofs are cumbersome. As a matter of fact, the translation in [2] is flawed.[4]

Instead of directly constructing the NWA from a CaRet or an NWTL formula, we utilize our alternation-elimination scheme. In more detail, we first translate the given formula into an alternating automaton with a Büchi acceptance condition. As for LTL, the translation for CaRet and NWTL into alternating automata is straightforward and linear in the formula length, since each temporal operators in CaRet and also NWTL only allows us to specify a property along a single path in the graph representation of nested words. Moreover, the obtained automaton is eventually $\{1, 2\}$-way and very weak. Then, by instantiating the alternation-elimination scheme with the complementation constructions from Theorem 4, we obtain from such an alternating automaton an NWA.

The benefits of this translation is as follows. Its correctness is easier to establish. The difficult part is the alternation-elimination construction. However, by our scheme its correctness proof boils down of proving the correctness of a complementation construction for *existential* automata. Moreover, we can handle the future-only fragment of CaRet and NWTL more efficiently by using the specialized instance of our alternation-elimination scheme that we obtain from Theorem 6. Finally, our translation can easily be adapted to extensions of NWTL and other temporal logics. Similar to LTL and word automata [24], NWTL and thus also CaRet are strictly less expressive than NWAs.[5] For LTL, several extensions and variants have been proposed to overcome this limitation. Among them are Wolper's ETL [24] and the industrial-strength logic PSL [1]. Similar extensions are possible for NWTL to increase its expressiveness. For instance, we can extend NWTL with the PSL-specific temporal operators that allow one to use (semi-extended) regular expressions. With our alternation-elimination scheme, we obtain a translation to NWAs with only minor modifications. Namely, the translation into alternating automata is standard, see e.g., [9,12]. Furthermore, since the alternating automata are not necessarily very weak any more, we use the complementation construction from Theorem 3 instead of the more specialized one from Theorem 4 to instantiate the alternation-elimination scheme. However, it is open whether and which PSL-like extensions [15] of NWTL are capable of expressing all NWA-recognizable languages.

[4] A counterexample is given by the NWTL formula $\Diamond^a \mathsf{ff}$, where $\Diamond^a$ is the "abstract" version of classical eventually modality $\Diamond$ in LTL. The constructed NWA accepts the nested word $\big(\langle\emptyset \quad \emptyset \quad \emptyset\rangle\big)^\omega$, which is not a model of the formula $\Diamond^a \mathsf{ff}$ since $\Diamond^a \mathsf{ff}$ is unsatisfiable. More generally speaking, the constructions in [2] disregards unfoldings of least fixpoint formulas along the jumps from call to return positions. It should be possible to correct their tableaux-based construction by using the technique for ensuring condition (6) of Lemma 3 in our automaton construction from Theorem 3.

[5] The language of nested words in which every position is internal and the proposition p holds at every even position witnesses that NWTL cannot express all nested-word regular languages.

References

1. IEEE standard for property specification language (PSL). IEEE Std 1850TM (October 2005)
2. Alur, R., Arenas, M., Barceló, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. Log. Methods Comput. Sci. 4(4) (2008)
3. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T.W., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Progr. Lang. Syst. 27(4), 786–818 (2005)
4. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)
5. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: ACM Symposium on Theory of Computing (STOC), pp. 202–211. ACM Press, New York (2004)
6. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM 56(3), 1–43 (2009)
7. Ball, T., Rajamani, S.K.: Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research (2000)
8. Banieqbal, B., Barringer, H.: Temporal logic with fixed points. In: Banieqbal, B., Pnueli, A., Barringer, H. (eds.) Temporal Logic in Specification. LNCS, vol. 398, pp. 62–74. Springer, Heidelberg (1989)
9. Ben-David, S., Bloem, R., Fisman, D., Griesmayer, A., Pill, I., Ruah, S.: Automata construction algorithms optimized for PSL. Technical report, The Prosyd Project (2005), http://www.prosyd.org
10. Bozzelli, L.: Alternating automata and a temporal fixpoint calculus for visibly pushdown languages. In: Caires, L., Li, L. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 476–491. Springer, Heidelberg (2007)
11. Dax, C., Klaedtke, F.: Alternation elimination by complementation. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 214–229. Springer, Heidelberg (2008)
12. Dax, C., Klaedtke, F., Lange, M.: On regular temporal logics with past. Acta Inform. 47(4), 251–277 (2010)
13. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Trans. Comput. Log. 2(3), 408–429 (2001)
14. Kupferman, O., Vardi, M.Y.: Complementation constructions for nondeterministic automata on infinite words. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 206–221. Springer, Heidelberg (2005)
15. Lange, M.: Linear time logics around PSL: Complexity, expressiveness, and a little bit of succinctness. In: Caires, L., Li, L. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 90–104. Springer, Heidelberg (2007)
16. Miyano, S., Hayashi, T.: Alternating finite automata on ω-words. Theoret. Comput. Sci. 32(3), 321–330 (1984)
17. Muller, D., Saoudi, A., Schupp, P.: Alternating automata, the weak monadic theory of trees and its complexity. Theoret. Comput. Sci. 97(2), 233–244 (1992)
18. Muller, D., Schupp, P.: Alternating automata on infinite trees. Theoret. Comput. Sci. 54(2–3), 267–276 (1987)
19. Shepherdson, J.C.: The reduction of two-way automata to one-way automata. IBM Journal of Research and Development 3(2), 198–200 (1959)
20. Vardi, M.Y.: A temporal fixpoint calculus. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 250–259. ACM Press, New York (1988)

21. Vardi, M.Y.: A note on the reduction of two-way automata to one-way automata. Inform. Process. Lett. 30(5), 261–264 (1989)
22. Vardi, M.Y.: Reasoning about the past with two-way automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 628–641. Springer, Heidelberg (1998)
23. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Symposium on Logic in Computer Science (LICS), pp. 332–344. IEEE Computer Society, Los Alamitos (1986)
24. Wolper, P.: Temporal logic can be more expressive. Information and Control 56 (1-2), 72–99 (1983)

Co-Büching Them All

Udi Boker and Orna Kupferman

School of Computer Science and Engineering
Hebrew University, Israel
{udiboker,orna}@cs.huji.ac.il

Abstract. We solve the open problems of translating, when possible, all common classes of nondeterministic word automata to deterministic and nondeterministic co-Büchi word automata. The handled classes include Büchi, parity, Rabin, Streett and Muller automata. The translations follow a unified approach and are all asymptotically tight.

The problem of translating Büchi automata to equivalent co-Büchi automata was solved in [2], leaving open the problems of translating automata with richer acceptance conditions. For these classes, one cannot easily extend or use the construction in [2]. In particular, going via an intermediate Büchi automaton is not optimal and might involve a blow-up exponentially higher than the known lower bound. Other known translations are also not optimal and involve a doubly exponential blow-up.

We describe direct, simple, and asymptotically tight constructions, involving a $2^{\Theta(n)}$ blow-up. The constructions are variants of the subset construction, and allow for symbolic implementations. Beyond the theoretical importance of the results, the new constructions have various applications, among which is an improved algorithm for translating, when possible, LTL formulas to deterministic Büchi word automata.

1 Introduction

Finite *automata on infinite objects* are widely used in formal verification and synthesis of nonterminating systems. The automata-theoretic approach to verification reduces questions about systems and their specifications to automata-theoretic problems like language containment and emptiness [10,18]. Recent industrial-strength specification-languages such as Sugar, ForSpec and PSL 1.01 include regular expressions and/or automata, making automata-theory even more essential and popular [1].

There are various classes of automata, characterized by their branching mode and acceptance condition. Each class has its advantages, disadvantages, and common usages. Accordingly, an important challenge in the the study of automata on infinite objects is to provide algorithms for translating between the different classes. For most translations, our community was able to come up with satisfactory solutions, in the sense that the state blow-up involved in the algorithm is proved to be unavoidable. Yet, for some translations there is still a significant gap between the best known algorithm and the corresponding lower bound.

Among these open problems are the translations of nondeterministic automata to equivalent deterministic and nondeterministic co-Büchi automata (NCW and DCW),

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 184–198, 2011.

when possible.[1] In [2], we introduced the *augmented subset construction* and used it for translating a nondeterministic Büchi automaton (NBW) to NCW and DCW, when possible. We left open the problems of translating automata with richer acceptance conditions (parity, Rabin, Streett and Muller) to co-Büchi automata. For these classes, one cannot easily extend or use the construction in [2], and the gap between the lower and upper bounds is still significant (for some of the classes it is even exponential). In this paper, we solve these problems and study the translation of nondeterministic parity (NPW), Streett (NSW), Rabin (NRW), and Muller (NMW) word automata to NCW and to DCW.

A straightforward approach is to translate an automaton of the richer classes via an intermediate NBW. This approach, however, is not optimal. For example, starting with an NSW with n states and index k, the intermediate NBW has $n2^k$ states, thus the NCW would have $n2^{k+n2^k}$ states, making the dependency in k doubly-exponential. Note that the exponential blow-up in the translation of NSW or NMW to NBW cannot be avoided [15]. A different approach is to translate the original automaton, for example an NRW, to an equivalent DPW, which can then be translated to an equivalent DCW over the same structure [5]. However, translating an NRW to an equivalent DPW might be doubly exponential [4], with no matching lower bound, even for the problem of translating to a DCW, let alone translating to NCW.

Thus, the approaches that go via intermediate automata are far from optimal, and our goal is to find a direct translation of these stronger classes of automata to NCW and DCW. We first show that for NSW, an equivalent NCW can be defined on top of the augmented subset construction (the product of the original automaton with its subset construction). The definition of the corresponding co-Büchi acceptance condition is more involved in this case than in the case of translating an NBW, but the blow-up stays the same. Thus, even though NSW are exponentially more succinct than NBW, their translation to NCW is of exactly the same state complexity as is the one for NBW! This immediately provides an $n2^n$ upper bound for the translation of NSW to NCW. As in the case of translating an NBW, we can further determinize the resulting augmented subset construction, getting a 3^n upper bound for the translation of NSW to DCW. Both bounds are asymptotically tight, having matching lower bounds by the special cases of translating NBW to NCW [2] and NCW to DCW [3]. The above good news apply also to the parity and the generalized-Büchi acceptance conditions, as they are special cases of the Streett condition.

For NRW and NMW, the situation is more complicated. Unfortunately, an equivalent NCW cannot in general be defined on top of the augmented subset construction. Moreover, even though the results on NSW imply a translation of NRW[1] (that is, a nondeterministic Rabin automaton with a single pair) to NCW, one cannot hope to proceed via a decomposition of an NRW with index k to k NRW[1]s. Indeed, the underlying NRW[1]s may not be NCW-realizable, even when the NRW is, and the same for NMWs. We show that still, the NCW can be defined on top of k copies of the augmented subset construction, giving rise to a $kn2^n$ upper bound for the translation to NCW. Moreover, we show that when translating to an equivalent DCW, the k copies

[1] The co-Büchi condition is weaker than the Büchi acceptance condition, and not all ω-regular languages are NCW-recognizable, hence the “when possible”.

can be determinized separately, while connected in a round-robin fashion, which gives rise to a $k3^n$ blow-up. As with the other cases, the blow-up involved in the translations is asymptotically tight. The state blow-up involved in the various translations is summarized in Table 1 of the Section 6.

Beyond the theoretical challenge in tightening the gaps, and the fact they are related to other gaps in our knowledge [6], these translations have immediate important applications in formal methods. The interest in the co-Büchi condition follows from its simplicity and its duality to the Büchi acceptance condition. The interest in the stronger acceptance conditions follows from their richness and succinctness. In particular, standard translations of LTL to automata go via intermediate generalized Büchi automata, which are then being translated to Büchi automata. For some algorithms, it is possible to give up the last step and work directly with the generalized Büchi automaton [8]. It follows from our results that the same can be done with the algorithm of translating LTL formulas to NCW and DCW. By the duality of the co-Büchi and Büchi conditions, one can construct a DBW for ψ by dualizing the DCW for $\neg\psi$. Thus, since the translation of LTL to NSW may be exponentially more succinct than a translation to NBW, our construction suggests the best known translation of LTL to DBW, when exists.

An important and useful property of our constructions is the fact they have only a one-sided error when applied to automata whose language is not NCW-recognizable. Thus, given an automaton $\mathcal{A}$, the NCW $\mathcal{C}$ and the DCW $\mathcal{D}$ we construct are always such that $L(\mathcal{A}) \subseteq L(\mathcal{C}) = L(\mathcal{D})$, while $L(\mathcal{A}) = L(\mathcal{C}) = L(\mathcal{D})$ in case $\mathcal{A}$ is NCW-recognizable. Likewise, given an LTL formula ψ, the DBW $\mathcal{D}_\psi$ we construct is always such that $L(\mathcal{D}_\psi) \subseteq L(\psi)$, while $L(\mathcal{D}_\psi) = L(\psi)$ in case ψ is DBW-recognizable. As specified in Section 5, this enables us to extend the scope of the applications also to specifications that are not NCW-realizable.

2 Preliminaries

Given an alphabet Σ, a *word* over Σ is a (possibly infinite) sequence $w = w_1 \cdot w_2 \cdots$ of letters in Σ. For two words, x and y, we use $x \preceq y$ to indicate that x is a prefix of y and $x \prec y$ to indicate that x is a strict prefix of y. An *automaton* is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$, 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. We define several acceptance conditions below. Intuitively, $\delta(q, \sigma)$ is the set of states that $\mathcal{A}$ may move into when it is in the state q and it reads the letter σ. The automaton $\mathcal{A}$ may have several initial states and the transition function may specify many possible transitions for each state and letter, and hence we say that $\mathcal{A}$ is *nondeterministic*. In the case where $|Q_0| = 1$ and for every $q \in Q$ and $\sigma \in \Sigma$, we have that $|\delta(q, \sigma)| \leq 1$, we say that $\mathcal{A}$ is *deterministic*. The transition function extends to sets of states and to finite words in the expected way, thus for a set of states S and a finite word x, $\delta(S, x)$ is the set of states that $\mathcal{A}$ may move into when it is in a state in S and it reads x. Formally, $\delta(S, \epsilon) = S$ and $\delta(S, w \cdot \sigma) = \bigcup_{q \in \delta(S,w)} \delta(q, \sigma)$. We abbreviate $\delta(Q_0, x)$ by $\delta(x)$, thus $\delta(x)$ is the set of states that $\mathcal{A}$ may visit after reading x. For an automaton $\mathcal{A}$ and a state q of $\mathcal{A}$, we denote by $\mathcal{A}^q$ the automaton that is identical to $\mathcal{A}$, except for having $\{q\}$ as its set of initial states. An automaton without an acceptance condition is called a *semi-automaton*.

A run $r = r_0, r_1, \cdots$ of $\mathcal{A}$ on $w = w_1 \cdot w_2 \cdots \in \Sigma^\omega$ is an infinite sequence of states such that $r_0 \in Q_0$, and for every $i \geq 0$, we have that $r_{i+1} \in \delta(r_i, w_{i+1})$. Note that while a deterministic automaton has at most a single run on an input word, a nondeterministic automaton may have several runs on an input word. We sometimes refer to r as a word in Q^ω or as a function from the set of prefixes of w to the states of $\mathcal{A}$. Accordingly, we use $r(x)$ to denote the state that r visits after reading the prefix x.

Acceptance is defined with respect to the set $inf(r)$ of states that the run r visits infinitely often. Formally, $inf(r) = \{q \in Q \mid \text{ for infinitely many } i \in \mathbb{N}, \text{ we have } r_i = q\}$. As Q is finite, it is guaranteed that $inf(r) \neq \emptyset$. The run r is *accepting* iff the set $inf(r)$ satisfies the acceptance condition α.

Several acceptance conditions are studied in the literature. We consider here six:

- *Büchi*, where $\alpha \subseteq Q$, and r is accepting iff $inf(r) \cap \alpha \neq \emptyset$.
- *co-Büchi*, where $\alpha \subseteq Q$, and r is accepting iff $inf(r) \subseteq \alpha$. Note that the definition we use is less standard than the $inf(r) \cap \alpha = \emptyset$ definition; clearly, $inf(r) \subseteq \alpha$ iff $inf(r) \cap (Q \setminus \alpha) = \emptyset$, thus the definitions are equivalent. We chose to go with this variant as it better conveys the intuition that, as with the Büchi condition, a visit in α is a “good event”.
- *parity*, where $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_{2k}\}$ with $\alpha_1 \subset \alpha_2 \subset \cdots \subset \alpha_{2k} = Q$, and r is accepting if the minimal index i for which $inf(r) \cap \alpha_i \neq \emptyset$ is even.
- *Rabin*, where $\alpha = \{\langle\alpha_1, \beta_1\rangle, \langle\alpha_2, \beta_2\rangle, \ldots, \langle\alpha_k, \beta_k\rangle\}$, with $\alpha_i, \beta_i \subseteq Q$ and r is accepting iff for some $1 \leq i \leq k$, we have that $inf(r) \cap \alpha_i \neq \emptyset$ and $inf(r) \cap \beta_i = \emptyset$.
- *Streett*, where $\alpha = \{\langle\beta_1, \alpha_1\rangle, \langle\beta_2, \alpha_2\rangle, \ldots, \langle\beta_k, \alpha_k\rangle\}$, with $\beta_i, \alpha_i \subseteq Q$ and r is accepting iff for all $1 \leq i \leq k$, we have that $inf(r) \cap \beta_i = \emptyset$ or $inf(r) \cap \alpha_i \neq \emptyset$.
- *Muller*, where $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$, with $\alpha_i \subseteq Q$ and r is accepting iff for some $1 \leq i \leq k$, we have that $inf(r) = \alpha_i$.

The number of sets in the parity and Muller acceptance conditions or pairs in the Rabin and Streett acceptance conditions is called the *index* of the automaton. An automaton accepts a word if it has an accepting run on it. The language of an automaton $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of words that $\mathcal{A}$ accepts. We also say that $\mathcal{A}$ *recognizes* the language $L(\mathcal{A})$. For two automata $\mathcal{A}$ and $\mathcal{A}'$, we say that $\mathcal{A}$ and $\mathcal{A}'$ are *equivalent* if $L(\mathcal{A}) = L(\mathcal{A}')$.

We denote the different classes of automata by three letter acronyms in $\{\text{D}, \text{N}\} \times \{\text{B}, \text{C}, \text{P}, \text{R}, \text{S}, \text{M}\} \times \{\text{W}\}$. The first letter stands for the branching mode of the automaton (deterministic or nondeterministic); the second letter stands for the acceptance-condition type (Büchi, co-Büchi, parity, Rabin, Streett, or Muller); and the third letter indicates that the automaton runs on words. We say that a language L is γ*-recognizable* or γ*-realizable* if L can be recognized by an automaton in the class γ.

Different classes of automata have different expressive power. In particular, while NBWs recognize all ω-regular languages [12], DBWs are strictly less expressive than NBWs, and so are DCWs [11]. In fact, a language L is in DBW iff its complement is in DCW. Indeed, by viewing a DBW as a DCW and switching between accepting and non-accepting states, we get an automaton for the complementing language, and vice versa. The expressiveness superiority of the nondeterministic model over the deterministic one does not apply to the co-Büchi acceptance condition. There, every NCW has an equivalent DCW [13]. As for parity, Rabin, Streett and Muller automata, both the deterministic and nondeterministic models recognize all ω-regular languages [17].

Our constructions for translating the various automata to co-Büchi automata will use the *augmented subset construction* [2], which is the product of an automaton with its subset construction.

Definition 1 (Augmented subset construction). *[2] Let $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0 \rangle$ be a semi-automaton. The* augmented subset construction *$\mathcal{A}'$ of $\mathcal{A}$ is the product of $\mathcal{A}$ with its subset construction. Formally, $\mathcal{A}' = \langle \Sigma, Q', \delta', Q'_0 \rangle$, where*

- *$Q' = Q \times 2^Q$. That is, the states of $\mathcal{A}'$ are all the pairs $\langle q, E \rangle$ where $q \in Q$ and $E \subseteq Q$.*
- *For all $\langle q, E \rangle \in Q'$ and $\sigma \in \Sigma$, we have $\delta'(\langle q, E \rangle, \sigma) = \delta(q, \sigma) \times \{\delta(E, \sigma)\}$. That is, $\mathcal{A}'$ nondeterministically follows $\mathcal{A}$ on its Q-component and deterministically follows the subset construction of $\mathcal{A}$ on its 2^Q-component.*
- *$Q'_0 = Q_0 \times \{Q_0\}$.*

3 Translating to NCW

In this section we study the translation, when possible, of NPWs, NRWs, NSWs, and NMWs to NCWs. Since the Büchi acceptance condition is a special case of these stronger conditions, the $2^{\Omega(n)}$ lower bound from [2] applies, and the challenge is to come up with matching upper bounds. While nondeterministic Rabin, Streett, and Muller automata are not more expressive than nondeterministic Büchi automata, they are more succinct: translating an NRW, NSW, and NMW with n states and index k to an NBW, results in an NBW with $O(nk)$, $O(n2^k)$, and $O(n^2k)$ states, respectively [15,16]. Note that an NPW is a special case of both an NSW and an NRW.

A first attempt to translate NRWs, NSWs, and NMWs to NCWs is to go via intermediate NBWs, which can be translated to NCWs by the augmented subset construction [2]. By the blow-ups above, however, this results in NCWs that are far from optimal. A second attempt is to apply the augmented subset construction directly on the input automaton, and check the possibility of defining on top of it a suitable co-Büchi acceptance condition.

It is not hard to see that this second attempt does not work for all automata. Consider for example the Rabin acceptance condition. Note that the augmented subset construction does not alter a deterministic automaton. Also, DRWs are not DCW-type [7] (that is, there is a DRW $\mathcal{A}$ whose language is DCW-recognizable, but still no DCW equivalent to $\mathcal{A}$ can be defined on top of the structure of $\mathcal{A}$). It follows that there are NRWs whose language is NCW-recognizable, but still no NCW recognizing them can be defined on top of the automaton obtained by applying the augmented subset construction on them (see Theorem 2 for a concrete example).

With this in mind, this section is a collection of good news. First, we show in Subsection 3.1 that NSWs (and NPWs) can be translated to NCWs on top of the augmented subset construction. Second, while this is not valid for NRWs and NMWs, we show in Subsection 3.2 that they can be translated to NCWs on top of a union of copies of the augmented subset construction. Moreover, the translation of the obtained NCWs to equivalent DCWs does not involve an additional exponential blow-up (see Section 4).

We first provide some basic lemmata from [2]. We start with a property relating states of a DCW (in fact, any deterministic automaton) that are reachable via words that lead to the same state in the subset construction of an equivalent nondeterministic automaton.

Lemma 1. *[2] Consider a nondeterministic automaton $\mathcal{A}$ with a transition function $\delta_{\mathcal{A}}$ and a DCW $\mathcal{D}$ with a transition function $\delta_{\mathcal{D}}$ such that $L(\mathcal{A}) = L(\mathcal{D})$. Let d_1 and d_2 be states of $\mathcal{D}$ such that there are two finite words x_1 and x_2 such that $\delta_{\mathcal{D}}(x_1) = d_1$, $\delta_{\mathcal{D}}(x_2) = d_2$, and $\delta_{\mathcal{A}}(x_1) = \delta_{\mathcal{A}}(x_2)$. Then, $L(\mathcal{D}^{d_1}) = L(\mathcal{D}^{d_2})$.*

For automata on finite words, if two states of the automaton have the same language, they can be merged without changing the language of the automaton. While this is not the case for automata on infinite words, the lemma below enables us to do take advantage of such states.

Lemma 2. *[2] Consider a DCW $\mathcal{D} = \langle \Sigma, D, \delta, D_0, \alpha \rangle$. Let d_1 and d_2 be states in D such that $L(\mathcal{D}^{d_1}) = L(\mathcal{D}^{d_2})$. For all finite words u and v, if $\delta(d_1, u) = d_1$ and $\delta(d_2, v) = d_2$ then for all words $w \in (u + v)^*$ and states $d \in \delta(d_1, w) \cup \delta(d_2, w)$, we have $L(\mathcal{D}^d) = L(\mathcal{D}^{d_1})$.*

The next lemma takes further advantage of DCW states recognizing the same language.

Lemma 3. *[2] Let $\mathcal{D} = \langle \Sigma, D, \delta, D_0, \alpha \rangle$ be a DCW. Consider a state $d \in D$. For all nonempty finite words v and u, if $(v^* \cdot u^+)^\omega \subseteq L(\mathcal{D}^d)$ and for all words $w \in (v + u)^*$ and states $d' \in \delta(d, w)$, we have $L(\mathcal{D}^{d'}) = L(\mathcal{D}^d)$, then $v^\omega \in L(\mathcal{D}^d)$.*

3.1 From NSW to NCW

The translation of an NSW to an NCW, when exists, can be done on top of the augmented subset construction, generalizing the acceptance condition used for translating an NBW to an NCW.

In the translation of an NBW to an NCW, we start with an NBW $\mathcal{B}$ and define a state $\langle b, E \rangle$ of the augmented subset construction to be co-Büchi accepting if there is some path p in $\mathcal{B}$, taking $\langle b, E \rangle$ back to itself via a Büchi accepting state. The correctness of the construction follows from the fact that an NCW-recognizable language is closed under pumping such cycles. Thus, if $\mathcal{B}$ accepts a word that includes a subword along which p is read, then $\mathcal{B}$ also accepts words obtained by pumping the subword along which p is read. In turns out that this intuition is valid also when we start with an NSW $\mathcal{S}$: a state $\langle s, E \rangle$ of the augmented subset construction is co-Büchi accepting if there is some path p in $\mathcal{S}$, taking $\langle s, E \rangle$ back to itself, such that p visits α_i or avoid β_i for every pair i in the Streett acceptance condition. This guarantees that pumping p infinitely often results in a run that satisfies the Streett condition, which in turn implies that an NCW-recongnizable language is closed under such pumping.

We formalize and prove this idea below.

Theorem 1. *For every NSW $\mathcal{S}$ with n states that is NCW-recognizable, there is an equivalent NCW $\mathcal{C}$ with at most $n2^n$ states.*

Proof. Let $\mathcal{S} = \langle \Sigma, S, \delta_{\mathcal{S}}, S_0, \langle \beta_1, \alpha_1 \rangle, \ldots \langle \beta_k, \alpha_k \rangle \rangle$. We define the NCW $\mathcal{C} = \langle \Sigma, C, \delta_{\mathcal{C}}, C_0, \alpha_{\mathcal{C}} \rangle$ as the augmented subset construction of $\mathcal{S}$ with the following acceptance condition: a state is a member of $\alpha_{\mathcal{C}}$ if it is reachable from itself along a path whose projection on S visits α_i or avoids β_i for every $1 \leq i \leq k$.

Formally, $\langle s, E \rangle \in \alpha_{\mathcal{C}}$ if there is a finite word $z = z_1 z_2 \cdots z_m$ of length m and a sequence of $m+1$ states $\langle s_0, E_0 \rangle \ldots \langle s_m, E_m \rangle$ such that $\langle s_0, E_0 \rangle = \langle s_m, E_m \rangle = \langle s, E \rangle$, and for all $0 \leq l < m$ we have $\langle s_{l+1}, E_{l+1} \rangle \in \delta_C(\langle s_l, E_l \rangle, z_{l+1})$, and for every $1 \leq i \leq k$, either there is $0 \leq l < m$ such that $s_l \in \alpha_i$ or $s_l \notin \beta_i$ for all $0 \leq l < m$. We refer to z as the *witness* for $\langle s, E \rangle$. Note that z may be the empty word.

We prove the equivalence of $\mathcal{S}$ and $\mathcal{C}$. Note that the 2^S-component of C proceeds in a deterministic manner. Therefore, each run r of $\mathcal{S}$ induces a single run of $\mathcal{C}$ (the run in which the S-component follows r). Likewise, each run r of $\mathcal{C}$ induces a single run of $\mathcal{S}$, obtained by projecting r on its S-component.

We first prove that $L(\mathcal{S}) \subseteq L(\mathcal{C})$. Note that this direction is always valid, even if $\mathcal{S}$ is not NCW-recognizable. Consider a word $w \in L(\mathcal{S})$. Let r be an accepting run of $\mathcal{S}$ on w. We prove that the run r' induced by r is accepting. Let $J \subseteq \{1, \ldots, k\}$ denote the set of indices of acceptance-pairs whose β-element is visited infinitely often by r. That is, $J = \{j \mid \beta_j \cap inf(r) \neq \emptyset\}$. Consider a state $\langle s, E \rangle \in inf(r')$. We prove that $\langle s, E \rangle \in \alpha_{\mathcal{C}}$. Since $\langle s, E \rangle$ appears infinitely often in r' and r is accepting, it follows that there are two (not necessarily adjacent) occurrences of $\langle s, E \rangle$, between which r visits α_j for all $j \in J$ and avoids β_i for all $i \notin J$. Hence, we have the required witness for $\langle s, E \rangle$, and we are done.

We now prove that $L(\mathcal{C}) \subseteq L(\mathcal{S})$. Consider a word $w \in L(\mathcal{C})$, and let r be an accepting run of $\mathcal{C}$ on w. Let $J \subseteq \{1, \ldots, k\}$ denote the set of indices of acceptance-pairs whose β-element is visited infinitely often by r. That is, $J = \{j \mid (\beta_j \times 2^S) \cap inf(r) \neq \emptyset\}$. If J is empty then the projection of r on its $\mathcal{S}$-component is accepting, and we are done. Otherwise, we proceed as follows. For every $j \in J$, let $\langle s_j, E_j \rangle$ be a state in $(\beta_j \times 2^S) \cap inf(r)$.

By the definition of J, all the states $\langle s_j, E_j \rangle$, with $j \in J$, are visited infinitely often in r, whereas states whose $\mathcal{S}$-component is in β_i, for $i \notin J$, are visited only finitely often in r. Accordingly, the states $\langle s_j, E_j \rangle$, with $j \in J$, are strongly connected via a path that does not visit β_i, for $i \notin J$. In addition, for every $\langle s_j, E_j \rangle$, with $j \in J$, there is a witness z_j for the membership of $\langle s_j, E_j \rangle$ in $\alpha_{\mathcal{C}}$, going from $\langle s_j, E_j \rangle$ back to itself via α_j and either avoiding β_i or visiting α_i, for every $1 \leq i \leq k$. Let $\langle s, E \rangle$ be one of these $\langle s_j, E_j \rangle$ states, and let x be a prefix of w such that $r(x) = \langle s, E \rangle$. Then, there is a finite word z along which there is a path from $\langle s, E \rangle$ back to itself, visiting all α_j for $j \in J$ and either avoiding β_i or visiting α_i for every $1 \leq i \leq k$. Therefore, $x \cdot z^\omega \in L(\mathcal{S})$.

Recall that the language of $\mathcal{S}$ is NCW-recognizable. Let $\mathcal{D} = \langle \Sigma, D, \delta_D, D_0, \alpha_{\mathcal{D}} \rangle$ be a DCW equivalent to $\mathcal{S}$. Since $L(\mathcal{S}) = L(\mathcal{D})$ and $x \cdot z^\omega \in L(\mathcal{S})$, it follows that the run ρ of $\mathcal{D}$ on $x \cdot z^\omega$ is accepting. Since D is finite, there are two indices, l and m, such that $l < m$, $\rho(x \cdot z^l) = \rho(x \cdot z^m)$, and for all prefixes y of $x \cdot z^\omega$ such that $x \cdot z^l \preceq y$, we have $\rho(y) \in \alpha_{\mathcal{D}}$. Let q be the state of $\mathcal{D}$ such that $q = \rho(x \cdot z^l)$.

Consider the run η of $\mathcal{D}$ on w. Since r visits $\langle s, E\rangle$ infinitely often and D is finite, there must be a state $d \in D$ and infinitely many prefixes $p_1, p_2, \ldots$ of w such that for all $i \geq 1$, we have $r(p_i) = \langle s, E\rangle$ and $\eta(p_i) = d$. We claim that the states q and d of $\mathcal{D}$ satisfy the conditions of Lemma 1 with x_0 being p_1 and x_1 being $x \cdot z^l$. Indeed, $\delta_{\mathcal{D}}(p_1) = d$, $\delta_{\mathcal{D}}(x \cdot z^l) = q$, and $\delta_{\mathcal{S}}(p_1) = \delta_{\mathcal{S}}(x \cdot z^l) = E$. For the latter equivalence, recall that $\delta_{\mathcal{S}}(x) = E$ and $\delta_{\mathcal{S}}(E, z) = E$. Hence, by Lemma 1, we have that $L(\mathcal{D}^q) = L(\mathcal{D}^d)$.

Recall the sequence of prefixes $p_1, p_2, \ldots$. For all $i \geq 1$, let $p_{i+1} = p_i \cdot t_i$. We now claim that for all $i \geq 1$, the state d satisfies the conditions of Lemma 3 with u being z^{m-l} and v being t_i. The second condition is satisfied by Lemma 2. For the first condition, consider a word $w' \in (v^* \cdot u^+)^\omega$. We prove that $w' \in L(\mathcal{D}^d)$. Recall that there is a run of $\mathcal{S}^s$ on v that goes back to s while avoiding β_i for all $i \notin J$ and there is a run of $\mathcal{S}^s$ on u that goes back to s while visiting α_j for all $j \in J$ and either visiting α_i or avoiding β_i for all $i \notin J$. (Informally, u "fixes" all the problems of v, by visiting α_j for every β_j that v might visit.) Recall also that for the word p_1, we have that $r(p_1) = \langle s, E\rangle$ and $\eta(p_1) = d$. Hence, $p_1 \cdot w' \in L(\mathcal{S})$. Since $L(\mathcal{S}) = L(\mathcal{D})$, we have that $p_1 \cdot w' \in L(\mathcal{S})$. Therefore, $w' \in L(\mathcal{D}^d)$.

Thus, by Lemma 3, for all $i \geq 1$ we have that $t_i^\omega \in L(\mathcal{D}^d)$. Since $\delta_{\mathcal{D}}(d, t_i) = d$, it follows that all the states that $\mathcal{D}$ visits when it reads t_i from d are in $\alpha_{\mathcal{D}}$. Note that $w = p_1 \cdot t_1 \cdot t_2 \cdots$. Hence, since $\delta_{\mathcal{D}}(p_1) = d$, the run of $\mathcal{D}$ on w is accepting, thus $w \in L(\mathcal{D})$. Since $L(\mathcal{D}) = L(\mathcal{S})$, it follows that $w \in L(\mathcal{S})$, and we are done. □

Two common special cases of the Streett acceptance condition are the parity and the *generalized Büchi* acceptance conditions. In a generalized Büchi automaton with states Q, the acceptance condition is $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ with $\alpha_i \subseteq Q$, and a run r is accepting if $inf(r) \cap \alpha_i \neq \emptyset$ for all $1 \leq i \leq k$. Theorem 1 implies that an NCW-recognizable nondeterministic parity or generalized Büchi automaton with n states can be translated to an NCW with $n2^n$ states, which can be defined on top of the augmented subset construction.

3.2 From NRW and NMW to NCW

In this section we study the translation of NRWs and NMWs to NCWs, when exists. Unfortunately, for these automata classes we cannot define an equivalent NCW on top of the augmented subset construction. Intuitively, the key idea of Subsection 3.1, which is based on the ability to pump paths that satisfy the acceptance condition, is not valid in the Rabin and the Muller acceptance conditions, as in these conditions, visiting some "bad" states infinitely often need not be compensated by visiting some "good" ones infinitely often. We formalize this in the example below, which consists of the fact that DRWs are not DCW-type [7].

Theorem 2. *There is an NRW and an NMW that are NCW-recognizable but an equivalent NCW for them cannot be defined on top of the augmented subset construction.*

Proof. Consider the NRW $\mathcal{A}$ appearing in Figure 1. The language of $\mathcal{A}$ consists of all words over the alphabet $\{0, 1\}$ that either have finitely many 0's or have finitely many 1's. This language is clearly NCW-recognizable, as it is the union of two NCW-recognizable languages. Since $\mathcal{A}$ is deterministic and the augmented subset construction does not alter a deterministic automaton, it suffices to show that there is no co-Büchi acceptance condition α' that we can define on the structure of $\mathcal{A}$ and get an equivalent language. Indeed, α' may either be $\emptyset$, $\{q_0\}$, $\{q_1\}$, or $\{q_0, q_1\}$, none of which provides the language of $\mathcal{A}$. Since every NRW has an equivalent NMW over the same structure, the above result also applies to the NMW case. □

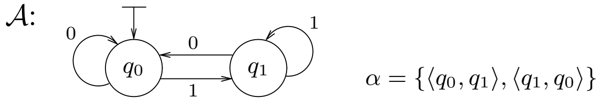

$\mathcal{A}$:
0
1
0
q_0
q_1
1
$\alpha = \{\langle q_0, q_1\rangle, \langle q_1, q_0\rangle\}$

Fig. 1. The NRW $\mathcal{A}$, having no equivalent NCW on top of its augmented subset construction

Consider an NRW or an NMW $\mathcal{A}$ with index k. Our approach for translating $\mathcal{A}$ to an NCW is to decompose it to k NSWs over the same structure, and apply the augmented subset construction on each of the components. Note that the components may not be NCW-realizable even when $\mathcal{A}$ is, thus, we should carefully analyze the proof of Theorem 1 and prove that the approach is valid.

We now formalize and prove the above approach. We start with the decomposition of an NRW or an NMW with index k into k NSWs over the same structure.

Lemma 4. *Every NRW or NMW $\mathcal{A}$ with index k is equivalent to the union of k NSWs over the same structure as $\mathcal{A}$.*

Proof. An NRW $\mathcal{A}$ with states A and index k is the union of k NRWs with index 1 over the same structure as $\mathcal{A}$. Since a single-indexed Rabin acceptance condition $\{\langle\alpha_1, \beta_1\rangle\}$ is equivalent to the Streett acceptance condition $\{\langle\alpha_1, \emptyset\rangle, \langle A, \beta_1\rangle\}$, we are done.

An NMW $\mathcal{A}$ with states A and index k is the union of k NMWs with index 1 over the same structure as $\mathcal{A}$. Since a single-indexed Muller acceptance condition $\{\alpha_1\}$ is equivalent to the Streett acceptance condition $\{\langle A \setminus \alpha_1, \emptyset\rangle\} \cup \bigcup_{s\in\alpha_1} \{\langle A, \{s\}\rangle\}$, we are done. □

Next we show that a union of k NSWs can be translated to a single NSW over their union.

Lemma 5. *Consider k NSWs, $\mathcal{S}_1, \ldots, \mathcal{S}_k$, over the same structure. There is an NSW $\mathcal{S}$ over the disjoint union of their structures, such that $L(\mathcal{S}) = \bigcup_{i=1}^{k} L(\mathcal{S}_i)$.*

Proof. We obtain the Streett acceptance condition of $\mathcal{S}$ by taking the union of the Streett acceptance conditions of the NSWs $\mathcal{S}_1, \ldots, \mathcal{S}_k$. Note that while the underlying NSWs are interpreted disjunctively (that is, in order for a word to be accepted by the union, there should be an accepting run on it in some $\mathcal{S}_i$), the pairs in the Streett condition are interpreted conjunctively (that is, in order for a run to be accepting, it has to satisfy the constraints by all the pairs in the Streett condition). We prove that still $L(\mathcal{S}) =$

$\bigcup_{i=1}^{k} L(\mathcal{S}_i)$. First, if a run r of $\mathcal{S}$ is an accepting run of an underlying NSW $\mathcal{S}_i$, then the acceptance conditions of the other underlying NSWs are vacuously satisfied. Hence, if a word is accepted by $\mathcal{S}_i$ for some $1 \leq i \leq k$, then $\mathcal{S}$ accepts it too. For the other direction, if a word w is accepted in $\mathcal{S}$, then its accepting run in $\mathcal{S}$ is also an accepting run of one of the underlying NSWs, thus w is in $\bigcup_{i=1}^{k} L(\mathcal{S}_i)$. □

Finally, we combine the translation to Streett automata with the augmented subset construction and get the required upper bound for NRW and NMW.

Theorem 3. *For every NCW-recognizable NRW or NMW with n states and index k, there is an equivalent NCW $\mathcal{C}$ with at most $kn2^n$ states.*

Proof. Consider an NRW or an NMW $\mathcal{A}$ with n states and index k. By Lemmas 4 and 5, there is an NSW $\mathcal{S}$ whose structure consists of k copies of the structure of $\mathcal{A}$ such that $L(\mathcal{S}) = L(\mathcal{A})$. Let $\mathcal{C}$ be the NCW equivalent to $\mathcal{S}$, defined over the augmented subset construction of $\mathcal{S}$, as described in Theorem 1. Note that $\mathcal{S}$ has nk states, thus a naive application of the augmented subset construction on it results in an NCW with $kn2^{kn}$ states. The key observation, which implies that we get an NCW with only $kn2^n$ states, is that applying the augmented subset construction on $\mathcal{S}$, the deterministic component of all the underlying NCWs is the same, and it coincides with the subset construction applied to $\mathcal{A}$. To see this, assume that $\mathcal{A} = \langle \Sigma, A, A_0, \delta, \alpha \rangle$. Then, $\mathcal{S} = \langle \Sigma, A \times \{1, \ldots, k\}, A_0 \times \{1, \ldots, k\}, \delta', \alpha' \rangle$, where for all $a \in A, 1 \leq j \leq k$, and $\sigma \in \Sigma$, we have that $\delta'(\langle a, j \rangle, \sigma) = \delta(a, \sigma) \times \{j\}$. Applying the augmented subset construction, we get the product of $\mathcal{S}$ and its subset construction, where the latter has a state for every reachable subset of S. That is, a subset $G' \subseteq S$ is a state of the subset construction if there is a finite word u for which $\delta'(u) = G'$. Since for all $a \in A, 1 \leq j \leq k$, and $\sigma \in \Sigma$, we have that $\delta'(\langle a, j \rangle, \sigma) = \delta(a, \sigma) \times \{j\}$, it follows that G' is of the form $G \times \{j\}$ for all $1 \leq j \leq k$ and some $G \subseteq A$. Hence, there are up to $2^{|A|} = 2^n$ states in the subset construction of $\mathcal{S}$. Thus, when we apply the augmented subset construction on $\mathcal{S}$, we end up with an NCW with only $kn2^n$ states, and we are done. □

4 Translating to DCW

In a first sight, the constructions of Section 3, which translate a nondeterministic word automaton to an NCW, are not useful for translating it to a DCW, as the determinization of an NCW to a DCW has an exponential state blow-up. Yet, we show that the special structure of the constructed NCW allows to determinize it without an additional exponential blow-up. The key to our construction is the observation that the augmented subset construction is transparent to additional applications of the subset construction. Indeed, applying the subset construction on an NCW $\mathcal{C}$ with state space $B \times 2^B$, one ends up in a deterministic automaton with state space $\{\{\langle q, E \rangle \mid q \in E\} : E \subseteq B\}$, which is isomorphic to 2^B.

The standard breakpoint construction [13] uses the subset construction as an intermediate layer in translating an NCW with state space C to a DCW with state space 3^C. Thus, the observation above suggests that applying it on our special NCW $\mathcal{C}$ would not involve an additional exponential blow-up on top of the one involved in going from some automaton $\mathcal{A}$ to $\mathcal{C}$. As we show in Theorem 4 below, this is indeed the case.

Starting with an NSW, the determinization of the corresponding NCW is straightforward, following [13]'s construction. However, when starting with an NRW or an NMW, the k different parts of the corresponding NCW (see Theorem 3) might cause a doubly-exponential blowup. Fortunately, we can avoid it by determinizing each of the k parts separately and connecting them in a round-robin fashion. We refer to the construction in Theorem 4 as the *breakpoint construction.*

Theorem 4. *For every DCW-recognizable NPW, NSW, NRW, or NMW $\mathcal{A}$ with n states there is an equivalent DCW $\mathcal{D}$ with $O(3^n)$ states.*

Proof. We start with the case $\mathcal{A}$ is an NSW. The DCW $\mathcal{D}$ follows all the runs of the NCW $\mathcal{C}$ constructed in Theorem 1. Let $\alpha_{\mathcal{C}} \subseteq A \times 2^A$ be the acceptance condition of $\mathcal{C}$. The DCW $\mathcal{D}$ accepts a word if some run of $\mathcal{C}$ remains in $\alpha_{\mathcal{C}}$ from some position.[2] At each state, $\mathcal{D}$ keeps the corresponding subset of the states of $\mathcal{C}$, and it updates it deterministically whenever an input letter is read. In order to check that some run of $\mathcal{C}$ remains in $\alpha_{\mathcal{C}}$ from some position, the DCW $\mathcal{D}$ keeps track of runs that do not leave $\alpha_{\mathcal{C}}$. The key observation in [13] is that keeping track of such runs can be done by maintaining the subset of states that belong to these runs.

Formally, let $\mathcal{A} = \langle \Sigma, A, \delta_{\mathcal{A}}, A_0, \alpha_{\mathcal{A}} \rangle$. We define a function $f : 2^A \to 2^A$ by $f(E) = \{a \mid \langle a, E \rangle \in \alpha_{\mathcal{C}}\}$. Thus, when the subset component of $\mathcal{D}$ is in state E, it should continue and check the membership in $\alpha_{\mathcal{C}}$ only for states in $f(E)$. We define the DCW $\mathcal{D} = \langle \Sigma, D, \delta_{\mathcal{D}}, D_0, \alpha_{\mathcal{D}} \rangle$ as follows.

- $D = \{\langle S, O \rangle \mid S \subseteq A$ and $O \subseteq S \cap f(S)\}$.
- For all $\langle S, O \rangle \in D$ and $\sigma \in \Sigma$, the transition function is defined as follows.
  - If $O \neq \emptyset$, then $\delta_D(\langle S, O \rangle, \sigma) = \{\langle \delta_{\mathcal{A}}(S, \sigma), \delta_{\mathcal{A}}(O, \sigma) \cap f(S) \rangle\}$.
  - If $O = \emptyset$, then $\delta_D(\langle S, O \rangle, \sigma) = \{\langle \delta_{\mathcal{A}}(S, \sigma), \delta_{\mathcal{A}}(S, \sigma) \cap f(S) \rangle\}$.
- $D_0 = \{\langle A_0, \emptyset \rangle\}$.
- $\alpha_{\mathcal{D}} = \{\langle S, O \rangle \mid O \neq \emptyset\}$.

Thus, the run of $\mathcal{D}$ on a word w has to visit states in $2^A \times \{\emptyset\}$ only finitely often, which holds iff some run of $\mathcal{C}$ on w eventually always visits $\alpha_{\mathcal{C}}$. Since each state of D corresponds to a function from A to the set { "in $S \cap O$", "in $S \setminus O$", "not in S"}, its number of states is at most $3^{|A|}$.

We proceed to the case $\mathcal{A}$ is an NRW or an NMW. Here, by Theorem 3, $\mathcal{A}$ has an equivalent NCW $\mathcal{C}$ with $kn2^n$ states. The NCW $\mathcal{C}$ is obtained by applying the augmented subset construction on k copies of $\mathcal{A}$, and thus has k unconnected components, $\mathcal{C}_1, \ldots, \mathcal{C}_k$ that are identical up to their acceptance conditions $\alpha_{\mathcal{C}_1}, \ldots, \alpha_{\mathcal{C}_k}$.

Since the k components of $\mathcal{C}$ all have the same $A \times 2^A$ structure, applying the standard subset construction on $\mathcal{C}$, one ends up with a deterministic automaton that is isomorphic to 2^A. Applying the standard breakpoint construction on $\mathcal{C}$, we could thus hope to obtain a deterministic automaton with only $3^{|A|}$ states. This construction, however, has to consider the different acceptance conditions α_i, maintaining in each state not only a pair $\langle S, O \rangle$, but a tuple $\langle S, O_1, \ldots, O_k \rangle$, where each $O_i \subseteq S$ corresponds to

[2] Readers familiar with the construction of [13] may find it easier to view the construction here as one that dualizes a translation of universal co-Büchi automata to deterministic Büchi automata, going through universal Büchi word automata – these constructed by dualizing Theorem 1.

the standard breakpoint construction with respect to α_i. Such a construction, however, involves a k^n blow-up.

We circumvent this blow-up by determinizing each of the $\mathcal{C}_i$'s separately and connecting the resulting $\mathcal{D}_i$'s in a round-robin fashion, moving from $\mathcal{D}_i$ to $\mathcal{D}_{i \pmod k)+1}$ when the set O, which maintains the set of states in paths in which $\mathcal{D}_i$ avoids α_i, becomes empty. Now, there is $1 \leq i \leq k$ such that $\mathcal{C}_i$ has a run that eventually gets stuck in α_i iff there is $1 \leq i \leq k$ such that in the round-robin construction, the run gets stuck in a copy that corresponds to $\mathcal{D}_i$ in states with $O \neq \emptyset$.

Formally, for every $1 \leq i \leq k$, we define a function $f_i : 2^A \rightarrow 2^A$ by $f_i(E) = \{a \mid \langle a, E\rangle \in \alpha_{\mathcal{C}_i}\}$. We define the DCW $\mathcal{D} = \langle \Sigma, D, \delta_{\mathcal{D}}, D_0, \alpha_{\mathcal{D}}\rangle$ as follows.

- $D = \{\langle S, O, i\rangle \mid S \subseteq A, O \subseteq S \cap f_i(S)$, and $i \in \{1, \dots k\}\}$.
- For all $\langle S, O, i\rangle \in D$ and $\sigma \in \Sigma$, the transition function is defined as follows.
  - If $O \neq \emptyset$, then $\delta_D(\langle S, O, i\rangle, \sigma) = \{\langle S', O', i'\rangle\}$, where $S' = \delta_{\mathcal{A}}(S, \sigma)$, $O' = \delta_{\mathcal{A}}(O, \sigma) \cap f_i(S)$ and $i' = i \pmod k + 1$ if $O' = \emptyset$ and i otherwise.
  - If $O = \emptyset$, then $\delta_D(\langle S, O, i\rangle, \sigma) = \{\langle S', O', i'\rangle\}$, where $S' = \delta_{\mathcal{A}}(S, \sigma)$, $O' = \delta_{\mathcal{A}}(S, \sigma) \cap f_i(S)$ and $i' = i \pmod k + 1$ if $O' = \emptyset$ and i otherwise.
- $D_0 = \{\langle A_0$ of $\mathcal{C}_1, \emptyset\rangle\}$.
- $\alpha_{\mathcal{D}} = \{\langle S, O, i\rangle \mid O \neq \emptyset\}$.

A run of $\mathcal{D}$ is accepting if it gets stuck in one of the sets of accepting states. Since the different parts of $\mathcal{C}$ are unconnected, we have that a run of $\mathcal{C}$ is accepting iff it gets stuck in the accepting states of one of the $\mathcal{C}_i$'s. Hence, a word is accepted by $\mathcal{C}$ iff it is accepted by $\mathcal{D}$, and we are done.

□

By [3], one cannot avoid the 3^n state blow-up for translating an NCW to a DCW. Since this lower bound clearly holds also for the stronger conditions, we can conclude with the following.

Theorem 5. *The tight bound for the state blow-up in the translation, when possible, of NPW, NSW, NRW and NMW to an equivalent DCW is* $\Theta(3^n)$.

5 Applications

The translations of nondeterministic automata to NCW and DCW are useful in various applications, mainly in procedures that currently involve determinization. The idea is to either use an NCW instead of a deterministic Büchi or parity automaton, or to use a DBW instead of a deterministic parity automaton. We elaborated on these applications in [2], where the starting point was NBWs. In this section we show that the starting point for the applications can be automata with richer acceptance conditions, and that starting with the richer acceptance conditions (and hence, with automata that may be exponentially more succinct!), involves no extra cost.

In addition, all the applications described in [2] that involve a translation of LTL formulas to NCWs, DCWs or DBWs, can now use an intermediate automaton of the richer classes rather than an NBW. Here too, this can lead to an exponential saving. Indeed, the exponential succinctness of NSW with respect to NBW [15] is proved using languages that can be described by LTL formulas of polynomial length. It follows that there are LTL formulas whose translation to NSW would be exponentially more succinct than their translation to NBW. Moreover, in practice, tools that translate LTL to NBW go through intermediate generalized-Büchi automata, which are a special case of NSW. Our results suggest that in the applications described below, one need not blow-up the state space by going all the way to an NBW.

We first note two important features of the translations. The first feature is the fact that the constructions in Theorems 1, 3, and 4 are based on the subset construction, have a simple state space, are amenable to optimizations, and can be implemented symbolically [14]. The second feature has to do with the one-sided error of the construction, when applied to automata that are not NCW-recognizable: Theorems 1, 3 and 4 guarantee that if the given automaton is NCW-recognizable, then the constructions result in equivalent automata. As stated below, if this is not the case, then the constructions have only a one-sided error.

Lemma 6. *For an automaton $\mathcal{A}$, let $\mathcal{C}$ be the NCW obtained by the translations of Theorems 1 and 3, and let $\mathcal{D}$ be the DCW obtained from $\mathcal{A}$ by applying the breakpoint construction of Theorem 4. Then, $L(\mathcal{A}) \subseteq L(\mathcal{C}) = L(\mathcal{D})$.*

Proof. It is easy to see that the proof of the $L(\mathcal{A}) \subseteq L(\mathcal{C})$ direction in Theorems 1 and 3, as well as the equivalence of $\mathcal{C}$ and $\mathcal{D}$ in Theorem 4, do not rely on the assumption that $\mathcal{A}$ is NCW-recognizable. □

Below we list the main applications. More details can be found in [2] (the description of the problems is the same, except that there the input or intermediate automata are NBWs, whereas here we can handle, at the same complexity, all other acceptance conditions).

- Deciding whether a given automaton (NSW, NPW, NRW, or NMW) is NCW-recognizable.
- Deciding whether a given LTL formula is NCW- or DBW-recognizable.
- Translating an LTL formula to a DBW: For an LTL formula ψ, let $L(\psi)$ denote the set of computations satisfying ψ. Then, the following is an easy corollary of the duality between DBW and DCW.

  Lemma 7. *Consider an LTL formula ψ that is DBW-recognizable. Let $\mathcal{A}_{\neg\psi}$ be a nondeterministic automaton accepting $L(\neg\psi)$, and let $\mathcal{D}_{\psi}$ be the DBW obtained by dualizing the breakpoint construction of $\mathcal{A}_{\neg\psi}$. Then, $L(\mathcal{D}_{\psi}) = L(\psi)$.*

  Note that one need not translate the LTL formula to an NBW, and can instead translate it to a nondeterministic generalized Büchi or even to a Streett automaton, which are more succinct.
- Translating LTL formula to the alternation-free μ-calculus.

Using the one-sided error. The one-sided error of the constructions suggest applications also for specifications that are not NCW-recognizable. The translation to DBW, for example, can be used in a decision procedure for CTL* even when the path formulas are not DBW-recognizable.

We demonstrate below how the one-sided error can be used for solving LTL synthesis. Given an arbitrary LTL formula ψ, let $\mathcal{D}_\psi$ be the DBW constructed as in Lemma 7. Lemma 6 implies that $L(\mathcal{D}_\psi) \subseteq L(\psi)$. The polarity of the error (that is, $\mathcal{D}_\psi$ underapproximates ψ) is the helpful one. If we get a transducer that realizes $\mathcal{D}_\psi$, we know that it also realizes ψ, and we are done. Moreover, as suggested in [9], in case $\mathcal{D}_\psi$ is unrealizable, we can check, again using an approximating DBW, whether $\neg\psi$ is realizable for the environment. Only if both ψ is unrealizable for the system and $\neg\psi$ is unrealizable for the environment, we need precise realizability. Note that then, we can also conclude that ψ is not in DBW.

6 Discussion

The simplicity of the co-Büchi condition and its duality to the Büchi condition makes it an interesting theoretical object. Its many recent applications in practice motivate further study of it. Translating automata of rich acceptance conditions to co-Büchi automata is useful in formal verification and synthesis, yet the state blow-up that such translations involve was a long-standing open problem. We solved the problem, and provided asymptotically tight constructions for translating all common automata classes to nondeterministic and deterministic co-Büchi automata.

All the constructions are extensions of the augmented subset construction and breakpoint construction, which are in turn an extension of the basic subset construction. In particular, the set of accepting states is induced by simple reachability queries in the graph of the automaton. Hence, the constructed automata have a simple state space and are amenable to optimizations and to symbolic implementations.

The state blow-up involved in the various translations is summarized in Table 1.

Table 1. The state blow-up involved in the translation, when possible, of a word automaton with n states and index k to an equivalent NCW and DCW

From $\diagdown$ To	NCW	DCW
NBW, NPW, NSW	$n2^n$	3^n
NRW, NMW	$kn2^n$	$k3^n$

Since the lower bounds for the translations are known for the special case of the origin automaton being an NBW, this is a "good news" paper, providing matching upper bounds. The new translations are significantly, in some cases exponentially, better than known translations. In particular, they show that the exponential blow-ups in the translation of NSW to NBW and of NBW to NCW are not additive. This is quite rare in the theory of automata on infinite words. The good news is carried over to the applications of the translations. In particular, our results suggest that one need not go via intermediate NBWs in the translation of LTL formulas to DBWs, and that working instead with intermediate NSWs can result in DBWs that are exponentially smaller.

Acknowledgments

We thank an anonymous reviewer for a delicate observation on the proof of Theorem 4.

References

1. Accellera. Accellera organization inc. (2006), http://www.accellera.org
2. Boker, U., Kupferman, O.: Co-ing Büchi made tight and helpful. In: Proc. 24th IEEE Symp. on Logic in Computer Science, pp. 245–254 (2009)
3. Boker, U., Kupferman, O., Rosenberg, A.: Alternation removal in büchi automata. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 76–87. Springer, Heidelberg (2010)
4. Cai, Y., Zhang, T., Luo, H.: An improved lower bound for the complementation of Rabin automata. In: Proc. 24th IEEE Symp. on Logic in Computer Science, pp. 167–176 (2009)
5. Krishnan, S.C., Puri, A., Brayton, R.K.: Deterministic ω-automata vis-a-vis deterministic Büchi automata. In: Du, D.-Z., Zhang, X.-S. (eds.) ISAAC 1994. LNCS, vol. 834, pp. 378–386. Springer, Heidelberg (1994)
6. Kupferman, O.: Tightening the exchange rates between automata. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 7–22. Springer, Heidelberg (2007)
7. Kupferman, O., Morgenstern, G., Murano, A.: Typeness for ω-regular automata. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 324–338. Springer, Heidelberg (2004)
8. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006)
9. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proc. 46th IEEE Symp. on Foundations of Computer Science, pp. 531–540 (2005)
10. Kurshan, R.P.: Computer Aided Verification of Coordinating Processes. Princeton Univ. Press, Princeton (1994)
11. Landweber, L.H.: Decision problems for ω–automata. Mathematical Systems Theory 3, 376–384 (1969)
12. McNaughton, R.: Testing and generating infinite sequences by a finite automaton. Information and Control 9, 521–530 (1966)
13. Miyano, S., Hayashi, T.: Alternating finite automata on ω-words. Theoretical Computer Science 32, 321–330 (1984)
14. Morgenstern, A., Schneider, K.: From LTL to symbolically represented deterministic automata. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 279–293. Springer, Heidelberg (2008)
15. Safra, S., Vardi, M.Y.: On ω-automata and temporal logic. In: Proc. 21st ACM Symp. on Theory of Computing, pp. 127–137 (1989)
16. Seidl, H., Niwiński, D.: On distributive fixed-point expressions. Theoretical Informatics and Applications 33(4-5), 427–446 (1999)
17. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, pp. 133–191 (1990)
18. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)

Minimizing Deterministic Lattice Automata

Shulamit Halamish and Orna Kupferman

Hebrew University, School of Engineering and Computer Science, Jerusalem 91904, Israel
{lamit,orna}@cs.huji.ac.il

Abstract. Traditional automata accept or reject their input, and are therefore Boolean. In contrast, weighted automata map each word to a value from a semiring over a large domain. The special case of *lattice automata*, in which the semiring is a finite lattice, has interesting theoretical properties as well as applications in formal methods. A *minimal deterministic automaton* captures the combinatoric nature and complexity of a formal language. Deterministic automata are used in run-time monitoring, pattern recognition, and modeling systems. Thus, the minimization problem for deterministic automata is of great interest, both theoretically and in practice.

For traditional automata on finite words, a minimization algorithm, based on the Myhill-Nerode right congruence on the set of words, generates in polynomial time a canonical minimal deterministic automaton. A polynomial algorithm is known also for weighted automata over the tropical semiring. For general deterministic weighted automata, the problem of minimization is open. In this paper we study minimization of lattice automata. We show that it is impossible to define a right congruence in the context of lattices, and that no canonical minimal automaton exists. Consequently, the minimization problem is much more complicated, and we prove that it is NP-complete. As good news, we show that while right congruence fails already for finite lattices that are fully ordered, for this setting we are able to combine a finite number of right congruences and generate a minimal deterministic automaton in polynomial time.

1 Introduction

Automata theory is one of the longest established areas in Computer Science. Standard applications of automata theory include pattern matching, syntax analysis, and formal verification. In recent years, novel applications of automata-theoretic concepts have emerged from numerous sciences, like biology, physics, cognitive sciences, control, and linguistics. These novel applications require significant advances in fundamental aspects of automata theory [2]. One such advance is a transition from a Boolean to a multi-valued setting: while traditional automata accept or reject their input, and are therefore Boolean, novel applications, for example in speech recognition and image processing [18], are based on weighted automata, which map an input word to a value from a semiring over a large domain [7].

Focusing on applications in formal verification, the multi-valued setting arises directly in *quantitative verification* [10], and indirectly in applications like *abstraction methods*, in which it is useful to allow the abstract system to have unknown assignments to atomic propositions and transitions [9], *query checking* [5], which can be reduced to

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 199–213, 2011.

model checking over multi-valued systems, and verification of systems from *inconsistent viewpoints* [11], in which the value of the atomic propositions is the composition of their values in the different viewpoints.

Recall that in the multi-valued setting, the automata map words to a value from a semiring over a large domain. A *distributive finite lattice* is a special case of a semiring. A lattice $\langle A, \leq \rangle$ is a partially ordered set in which every two elements $a, b \in A$ have a least upper bound (a join b) and a greatest lower bound (a meet b). In many of the applications of the multi-valued setting described above, the values are taken from finite lattices. For example (see Figure 2), in the abstraction application, researchers use the lattice $\mathcal{L}_3$ of three fully ordered values [3], as well as its generalization to $\mathcal{L}_n$ [6]. In query checking, the lattice elements are sets of formulas, ordered by the inclusion order [4]. When reasoning about inconsistent viewpoints, each viewpoint is Boolean, and their composition gives rise to products of the Boolean lattice, as in $\mathcal{L}_{2,2}$ [8]. Finally, when specifying prioritized properties of system, one uses lattices in order to specify the priorities [1].

In [13], the authors study lattice automata, their theoretical properties, and decision problems for them. In a nondeterministic lattice automaton on finite words (LNFA, for short), each transition is associated with a *transition value*, which is a lattice element (intuitively indicating the truth of the statement "the transition exists"), and each state is associated with an *initial value* and an *acceptance value*, indicating the truth of the statements "the state is initial/accepting", respectively. Each run r of an LNFA $\mathcal{A}$ has a value, which is the *meet* of the values of all the components of r: the initial value of the first state, the transition value of all the transitions taken along r, and the acceptance value of the last state. The value of a word w is then the *join* of the values of all the runs of $\mathcal{A}$ on w. Accordingly, an LNFA over an alphabet Σ and lattice $\mathcal{L}$ induces an $\mathcal{L}$-language $L : \Sigma^* \to \mathcal{L}$. Note that traditional finite automata (NFAs) can be viewed as a special case of LNFAs over the lattice $\mathcal{L}_2$. In a deterministic lattice automaton on finite words (LDFA, for short), at most one state has an initial value that is not $\perp$ (the least lattice element), and for every state q and letter σ, at most one state q' is such that the value of the transition from q on σ to q' is not $\perp$. Thus, an LDFA $\mathcal{A}$ has at most one run (whose value is not $\perp$) on each input word, and the value of this run is the value of the word in the language of $\mathcal{A}$.

For example, the LDFA $\mathcal{A}$ in Figure 1 below is over the alphabet $\Sigma = \{0, 1, 2\}$ and the lattice $\mathcal{L} = \langle 2^{\{a,b,c,d\}}, \subseteq \rangle$. All states have acceptance value $\{a, b, c, d\}$, and this is also the initial value of the single initial state. The $\mathcal{L}$-language of $\mathcal{A}$ is $L : \Sigma^* \to \mathcal{L}$ such that $L(\epsilon) = \{a, b, c, d\}$, $L(0) = \{c, d\}$, $L(0 \cdot 0) = \{d\}$, $L(1) = \{a, b\}$, $L(1 \cdot 0) = \{a\}$, $L(2) = \{c, d\}$, $L(2 \cdot 0) = \{c\}$, and $L(x) = \emptyset$ for all other $x \in \Sigma^*$.

A *minimal deterministic automaton* captures the combinatoric nature and complexity of a formal language. Deterministic automata are used in run-time monitoring, pattern recognition, and modeling systems. Thus, the minimization problem for deterministic automata is of great interest, both theoretically and in practice. For traditional automata on finite words, a minimization algorithm, based on the Myhill-Nerode right congruence on the set of words, generates in polynomial time a canonical minimal deterministic automaton [20,21]. In more detail, given a regular language L over Σ, then the relation $\sim_L \subseteq \Sigma^* \times \Sigma^*$, where $x \sim_L y$ iff for all $z \in \Sigma^*$ we have that $x \cdot z \in L$ iff

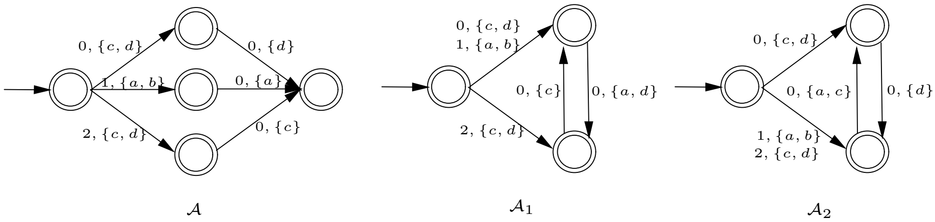


Fig. 1. An LDFA with two different minimal LDFAs

$y \cdot z \in L$, is an equivalence relation, its equivalence classes correspond to the states of a minimal automaton for $\mathcal{A}$, and they also uniquely induce the transitions of such an automaton. Further, given a deterministic automaton for L, it is possible to use the relation $\sim_L$ in order to minimize it in polynomial time.

A polynomial algorithm is known also for deterministic weighted automata over the *tropical semiring* [18]. In such automata, each transition has a value in $\mathbb{R}$, each state has an initial and acceptance values in $\mathbb{R}$, and the value of a run is the sum of the values of its components. Unlike the case of DFAs, in the case of weighted automata there may be several different minimal automata. They all, however, have the same graph topology, and only differ by the values assigned to the transitions and states. In other words, there is a canonical minimal topology, but no canonical minimal automaton. For semirings that are not the tropical semiring, and in particular, for lattice automata, the minimization problem is open.

In this work we study the minimization problem for lattice automata. An indication that the problem is not easy is the fact that in the latticed setting, the "canonical topology" property does not hold. To see this, consider again the LDFA $\mathcal{A}$ in Figure 1. Note that an automaton for $L(\mathcal{A})$ must have at least three states. Indeed, if it has at most two then the word $w = 000$ would not get the value $\perp$, whereas $L(000) = \perp$. Hence, the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ presented on its right are two minimal automata for L. Their topologies differ by the transition leaving the initial state with the letter 1[1].

The absence of the "canonical topology" property suggests that efforts to construct the minimal LDFA by means of a right congruence are hopeless. The main difficulty in minimizing LDFAs is that the "configuration" of an automaton consists not only of the current state, but also of the run that leads to the state, and the value accumulated reaching it[2]. Attempts to somehow allow the definition of the right congruence to refer to lattice values (as is the case in the successful minimization of weighted automata over the tropical semiring [18]) do not succeed. To see this, consider even a simpler

[1] Note that the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are not simple, in the sense that the transitions are associated with values from the lattice that are not $\perp$ or $\top$. The special case of simple lattice automata, where the value of a run is determined only by the value associated with the last state of the run is simpler, and has been solved in the context of fuzzy automata [17,22]. We will get back to it in Section 2.2.

[2] It is interesting to note that also in the context of deterministic Büchi automata, there is no single minimal topology for a minimal automaton. As with LDFAs, this has to do with the fact that the outcome of a run depends on its on-going behavior, rather than its last state only.

model of LDFA, in which all acceptance values are $\top$ (the greatest value in the lattice, and thus, $L(x \cdot z) \leq L(x)$ for all $x, z \in \Sigma^*$). A natural candidate for a right congruence for an $\mathcal{L}$-language L is the relation $\sim_L \subseteq \Sigma^* \times \Sigma^*$ such that $x \sim_L x'$ iff for all $z \in \Sigma^*$ there exists $l \in \mathcal{L}$ such that $L(x \cdot z) = L(x) \wedge l$ and $L(x' \cdot z) = L(x') \wedge l$. Unfortunately, the relation is not even transitive. For example, for the language L of the LDFA $\mathcal{A}$ in Figure 1, we have $0 \sim_L 1$ and $1 \sim_L 2$, but $0 \not\sim_L 2$.

We formalize these discouraging indications by showing that the problem of LDFA minimization is NP-complete. This is a quite surprising result, as this is the first parameter in which lattice automata are more complex than weighted automata over the tropical semiring. In particular, lattice automata can always be determinized [13,15], which is not the case for weighted automata over the tropical semiring [18]. Also, language containment between nondeterministic lattice automata can be solve in PSPACE [13,14], whereas the containment problem for weighted automata on the tropical semiring is undecidable [16]. In addition, lattices have some appealing properties that general semirings do not, which make them seem simpler. Specifically, the idempotent laws (i.e., $a \vee a = a$ and $a \wedge a = a$) as well as the absorption laws (i.e., $a \vee (a \wedge b) = a$ and $a \wedge (a \vee b) = a$), do not hold in a general semiring, and do hold for lattices. Nevertheless, as mentioned, we are able to prove that their minimization is NP-complete.

Our NP-hardness proof is by a reduction from the vertex cover problem [12]. The lattice used in the reduction is $\mathcal{L} \subset 2^E$, for the set E of edges of the graph, with the usual set-inclusion order. The reduction strongly utilizes on the fact that the elements of 2^E are not fully ordered. The most challenging part of the reduction is to come up with a lattice that, on the one hand, strongly uses the fact $\mathcal{L}$ is not fully ordered, yet on the other hand is of size polynomial in E (and still satisfies the conditions of closure under meet and join).

As pointed above, the NP-hardness proof involved a partially ordered lattice, embodied in the "subset lattice", and strongly utilizes on the order being partial. This suggests that for fully ordered lattices, we may still be able to find a polynomial minimization algorithm. On the other hand, as we shall show, the property of no canonical minimal LDFA is valid already in the case of fully ordered lattice, which suggests that no polynomial algorithm exists. As good news, we show that minimization of LDFAs over fully ordered lattices can nevertheless be done in polynomial time. The idea of the algorithm is to base the minimization on linearly many minimal DFAs that correspond to the different lattice values. The fact the values are fully ordered enables us to combine these minimal automata into one minimal LDFA.

Due to lack of space, some proofs are omitted in this version and can be found in the full version, on the authors' home pages.

2 Preliminaries

This section introduces the definitions and notations related to lattices and lattice automata, as well as some background about the minimization problem.

2.1 Lattices and Lattice Automata

Let $\langle A, \leq \rangle$ be a partially ordered set, and let P be a subset of A. An element $a \in A$ is an *upper bound* on P if $a \geq b$ for all $b \in P$. Dually, a is a *lower bound* on P if $a \leq b$ for all $b \in P$. An element $a \in A$ is the *least element of* P if $a \in P$ and a is a lower bound on P. Dually, $a \in A$ is the *greatest element of* P if $a \in P$ and a is an upper bound on P. A partially ordered set $\langle A, \leq \rangle$ is a *lattice* if for every two elements $a, b \in A$ both the least upper bound and the greatest lower bound of $\{a, b\}$ exist, in which case they are denoted $a \vee b$ (a *join* b) and $a \wedge b$ (a *meet* b), respectively. A lattice is *complete* if for every subset $P \subseteq A$ both the least upper bound and the greatest lower bound of P exist, in which case they are denoted $\bigvee P$ and $\bigwedge P$, respectively. In particular, $\bigvee A$ and $\bigwedge A$ are denoted $\top$ (*top*) and $\bot$ (*bottom*), respectively. A lattice $\langle A, \leq \rangle$ is finite if A is finite. Note that every finite lattice is complete. A lattice $\langle A, \leq \rangle$ is *distributive* if for every $a, b, c \in A$, we have $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ and $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$.

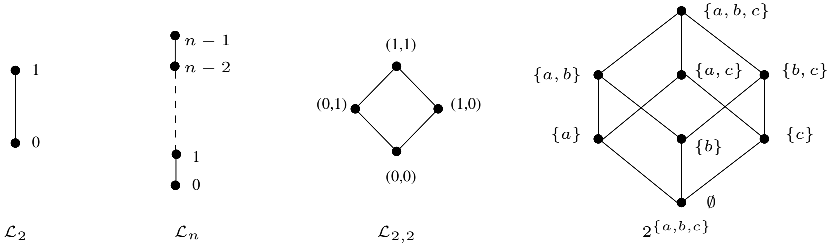


Fig. 2. Some lattices

In Figure 2 we describe some finite lattices. The elements of the lattice $\mathcal{L}_2$ are the usual truth values 1 (**true**) and 0 (**false**) with the order $0 \leq 1$. The lattice $\mathcal{L}_n$ contains the values $0, 1 ... n-1$, with the order $0 \leq 1 \leq ... \leq n-1$. The lattice $\mathcal{L}_{2,2}$ is the Cartesian product of two $\mathcal{L}_2$ lattices, thus $(a, b) \leq (a', b')$ if both $a \leq a'$ and $b \leq b'$. Finally, the lattice $2^{\{a,b,c\}}$ is the power set of $\{a, b, c\}$ with the set-inclusion order. In this lattice, for example, $\{a\} \vee \{b\} = \{a, b\}$, $\{a\} \wedge \{b\} = \bot$, $\{a, c\} \vee \{b\} = \top$, and $\{a, c\} \wedge \{b\} = \bot$.

Consider a lattice $\mathcal{L}$ (we abuse notation and refer to $\mathcal{L}$ also as a set of elements, rather than a pair of a set with an order on it). For a set X of elements, an *$\mathcal{L}$-set over X* is a function $S : X \to \mathcal{L}$ assigning to each element of X a value in $\mathcal{L}$. It is convenient to think about $S(x)$ as the truth value of the statement "x is in S". We say that an $\mathcal{L}$-set S is *Boolean* if $S(x) \in \{\top, \bot\}$ for all $x \in X$.

Consider a lattice $\mathcal{L}$ and an alphabet Σ. An *$\mathcal{L}$-language* is an $\mathcal{L}$-set over Σ^*. Thus, an $\mathcal{L}$-language $L : \Sigma^* \to \mathcal{L}$ assigns a value in $\mathcal{L}$ to each word over Σ.

A *deterministic lattice automaton* on finite words (LDFA, for short) is a tuple $\mathcal{A} = \langle \mathcal{L}, \Sigma, Q, Q_0, \delta, F \rangle$, where $\mathcal{L}$ is a finite lattice, Σ is an alphabet, Q is a finite set of states, $Q_0 \in \mathcal{L}^Q$ is an $\mathcal{L}$-set of initial states, $\delta \in \mathcal{L}^{Q \times \Sigma \times Q}$ is an $\mathcal{L}$-transition-relation, and $F \in \mathcal{L}^Q$ is an $\mathcal{L}$-set of accepting states. The fact $\mathcal{A}$ is deterministic is reflected in two conditions on Q_0 and δ. First, there is at most one state $q \in Q$, called the *initial state* of $\mathcal{A}$, such that $Q_0(q) \neq \bot$. In addition, for every state $q \in Q$ and letter $\sigma \in \Sigma$, there is at most one state $q' \in Q$, called the *σ-destination* of q, such that $\delta(q, \sigma, q') \neq \bot$. The *run* of an LDFA on a word $w = \sigma_1 \cdot \sigma_2 \cdots \sigma_n$ is a sequence $r = q_0, \ldots, q_n$ of $n+1$ states, where q_0 is the initial state of $\mathcal{A}$, and for all $1 \leq i \leq n$ it holds that q_i is the *σ_i-destination* of q_{i-1}. Note that $\mathcal{A}$ may not have a run on w. The *value* of w is $val(w) = Q_0(q_0) \wedge \bigwedge_{i=1}^{n} \delta(q_{i-1}, \sigma_i, q_i) \wedge F(q_n)$. Intuitively, $Q_0(q_0)$ is the value of q_0 being initial, $\delta((q_{i-1}, \sigma_i, q_i))$ is the value of q_i being a successor of q_{i-1} when σ_i is the input letter, $F(q_n)$ is the value of q_n being accepting, and the value of r is the meet of all these values. The *traversal value* of w is $tr\_val(w) = Q_0(q_0) \wedge \bigwedge_{i=1}^{n} \delta(q_{i-1}, \sigma_i, q_i)$, and its *acceptance value* is $F(q_n)$. The $\mathcal{L}$-language of $\mathcal{A}$, denoted $L(\mathcal{A})$, maps each word w to the value of its run in $\mathcal{A}$. Note that since $\mathcal{A}$ is deterministic, it has at most one run on w whose value is not $\bot$. This is why we talk about the traversal and acceptance values of words rather than of runs.

Note that traditional deterministic automata over finite words (DFA, for short) correspond to LDFA over the lattice $\mathcal{L}_2$. Indeed, over $\mathcal{L}_2$, a word is mapped to the value $\top$ if the run on it uses only transitions with value $\top$ and its final state has value $\top$.

An LDFA is *simple* if Q_0 and δ are Boolean. Note that the traversal value of a run r of a simple LDFA is either $\bot$ or $\top$, thus the value of r is induced by F. Simple LDFAs have been studied in the context of *fuzzy* logic and automata [17,22].

Analyzing the size of $\mathcal{A}$, one can refer to $|\mathcal{L}|$, $|Q|$, and $|\delta|$. Since the emphasize in this paper is on the size of the state space, we use $|\mathcal{A}|$ to refer to the size of its state space. Our complexity results, however, refer to the size of the input, and thus take into an account the other components of $\mathcal{A}$ as well, and in particular the size of $\mathcal{L}$.

2.2 Minimizing LDFAs

We now turn to discuss the problem of minimizing LDFA. We describe the difficulties of the problem, and present some examples that are important for the full comprehension of the issue.

A first attempt to minimize lattice automata would be to follow the classical paradigm for minimizing DFA using the Myhill Nerode theorem [20,21]. In fact, for the case of simple LDFA, the plan proceeds smoothly (see [17,22], where the problem is discussed by means of fuzzy automata)[3]: Given an $\mathcal{L}$-language L, we extend the definition of $\sim_L$ to fit the nature of $\mathcal{L}$-languages. For all $x, x' \in \Sigma^*$, we say that $x \sim_L x'$ iff for all $z \in \Sigma^*$ it holds that $L(x \cdot z)$=$L(x' \cdot z)$. Clearly, $\sim_L$ is an equivalence relation, since it is based on the equality relation. As in the case of DFA, we can build a minimal simple LDFA $\mathcal{A}_{min}$ for L such that $|\mathcal{A}_{min}| = | \sim_L |$. We construct it in the same manner, only that here the acceptance values are defined such that $F([x]) = L(x)$. Also, we can show that every simple LDFA for L must have at least $| \sim_L |$ states. Indeed, if this is not the case then we have two words $x, x' \in \Sigma^*$ reaching the same state q, while $x \not\sim_L x'$. The contradiction is reached when we read the distinguishing tail z from q as in the case of DFA, due to the fact that in simple LDFA the value of the words is solely determined by the final state.

[3] Several variants of fuzzy automata are studied in the literature. The difficulties we cope with in the minimization of our lattice automata are not applied to them, and indeed they can be minimized by variants of the minimization construction described here [19].

So, simple lattice automata can be minimized in polynomial time, and the key for proving it was a generalization of the right congruence to require agreement on the values of the words. Encouraged by this, we now turn to examine the case of general LDFAs. Unfortunately, the generalization does not seem to work here. To see the difficulties in the latticed setting, consider an $\mathcal{L}$-language L over $\Sigma = \mathcal{L}$ such that $L(l_1 \cdot l_2 \cdots l_n) = \bigwedge_{i=1}^{n} l_i$. The language L can be recognized by an LDFA $\mathcal{A}$ with a single state q. The initial and acceptance values of q are $\top$, and for every $l \in \Sigma$, there is an l-transition with value l from q to itself. Thus, the single run of $\mathcal{A}$ on an input word maps it to the meet of all letters. Clearly, there exist $x, x' \in \Sigma^*$ such that $L(x) \neq L(x')$, and still $\mathcal{A}$ has only one state. Thus, despite being mapped to different values, x and x' reach the same state in $\mathcal{A}$. This observation shows a crucial difference between the setting of DFAs or simple LDFAs and the one of general LDFA. It is only in the latter, that a "configuration" of a word is not only the state it reaches but also the traversal value accumulated when reading it. In our example, the words x and x' have a different traversal value, which is implicit in the LDFA, and an attempt to distinguish between words according to the values they have accumulated results in LDFAs with needlessly more states. Accordingly, a right congruence that may be helpful for minimization should take into an account the value accumulated by the words, and in particular, in the case of L above, should have only one equivalence class.

Following the above discussion, we now try to define an equivalence relation for LDFA that does take into an account the accumulated traversal values. Let us first consider a simpler model of LDFAs in which all acceptance values are $\top$. Note that in this model, for all $x, z \in \Sigma^*$ we have that $L(x \cdot z) \leq L(x)$. Let L be an $\mathcal{L}$-language in the model above. We define a relation $\sim_L \subseteq \Sigma^* \times \Sigma^*$ such that $x \sim_L x'$ iff for all $z \in \Sigma^*$ there exists $l \in \mathcal{L}$ such that $L(x \cdot z) = L(x) \wedge l$ and $L(x' \cdot z) = L(x') \wedge l$. Note that the relation $\sim_L$ indeed takes into a consideration the values accumulated when x and x' are read. Indeed, x and x' are equivalent iff for all tails $z \in \Sigma^*$ there exists some $l \in \mathcal{L}$ such that z can be read with the value l after reading either x or x'. Unfortunately, the relation is not even transitive. For example, for the language L of the LDFA $\mathcal{A}$ in Figure 1, we have $0 \sim_L 1$ and $1 \sim_L 2$, but $0 \not\sim_L 2$.

We have seen some evidences that minimization of LDFAs cannot follow the minimization paradigm for DFAs and even not the one for deterministic weighted automata over the tropical semiring. In the rest of the paper we formalize these evidences by showing that the problem is NP-complete. We also challenge them by describing a polynomial algorithm for minimization of LDFAs over fully ordered lattices – a special case for which the evidences apply.

3 Minimizing General LDFA

In this section we study the problem of minimizing LDFAs and show that unlike the case of DFAs, and even the case of weighted DFAs over the tropical semiring, which can be minimized in polynomial time, here the problem is NP-complete. We consider the corresponding decision problem MINLDFA=$\{\langle \mathcal{A}, k \rangle : \mathcal{A}$ is an LDFA and there exists an LDFA $\mathcal{A}'$ with at most k states such that $L(\mathcal{A}') = L(\mathcal{A})\}$.

Theorem 1. *MINLDFA is NP-complete.*

Proof. We start with membership in NP. Given $\mathcal{A}$ and k, a witness to their membership in MINLDFA is an LDFA $\mathcal{A}'$ as required. Since $|\mathcal{A}'| = k$, its size is linear in the input. Deciding language equivalence between LDFAs is NLOGSPACE-complete [13], thus we can verify that $L(\mathcal{A}') = L(\mathcal{A})$ in polynomial time.

For the lower bound, we show a polynomial time reduction from the Vertex Cover problem (VC, for short), proved to be NP-complete in [12]. Recall that VC=$\{\langle G, k\rangle : G$ is an undirected graph with a vertex cover of size $k\}$, where a *vertex cover* of a graph $G = \langle V, E\rangle$ is a set $C \subseteq V$ such that for all edges $(u, v) \in E$ we have $u \in C$ or $v \in C$.

Before we describe the reduction, we need some definitions. Consider an undirected graph $G = \langle V, E\rangle$. Let $n = |V|$ and $m = |E|$. For simplicity, we assume that V and E are ordered, thus we can refer to the minimum element in a set of vertices or edges. For $v \in V$, let $touch(v) = \{e : e = (v, u)$ for some $u\}$. For $e = (v_1, v_2) \in E$, let $far(e) = \min\{e' : e' \notin touch(v_1) \cup touch(v_2)\}$. That is, $far(e)$ is the minimal edge that is not adjacent to e. Note that if $\{e' : e' \notin touch(v_1) \cup touch(v_2)\} = \emptyset$, then $\{v_1, v_2\}$ is a VC of size two, so we can assume that $far(e)$ is well defined.

Example 1. In the graph G below, we have, for example, $touch(1) = \{a, b\}$, $touch(2) = \{d, e\}$, $far(a) = d$, $far(b) = e$, and $far(c) = e$.

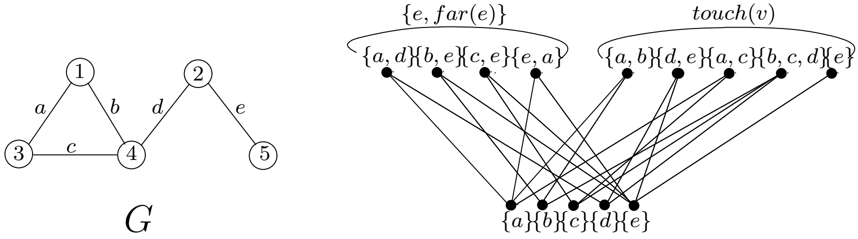


Fig. 3. A graph and its corresponding lattice

We now turn to describe the reduction. Given an input $G = \langle V, E\rangle$, we construct an LDFA $\mathcal{A} = \langle \mathcal{L}, Q, \Sigma, \delta, Q_0, F\rangle$ as follows:

- $\mathcal{L} \subseteq 2^E$ contains the following elements: $\{\emptyset, E\} \cup \{\{e\} : e \in E\} \cup \{\{e, far(e)\} : e \in E\} \cup \{touch(v) : v \in V\}$, with the usual set-inclusion relation. In particular, $\bot = \emptyset$ and $\top = E$. Note that $\mathcal{L}$ contains at most $2 + n + 2m$ elements ("at most" since $\{e, far(e)\}$ may be identical to $\{far(e), far(far(e))\}$). For example, the graph in Example 1 induces the lattice shown on its right (for clarity, we omit the elements $\top$ and $\bot$ in the figure). Note that in this example we have $\{a, far(a)\} = \{far(a), far(far(a))\}$, so we omit the unnecessary element.

  We claim that though $\mathcal{L}$ does not contain all the elements in 2^E, the operators $join$ and $meet$ are well defined for all $l_1, l_2 \in \mathcal{L}$.[4] In the case l_1 and l_2 are ordered,

[4] We note that this point has been the most challenging part of the reduction, as on the one hand, we have to strongly use the fact $\mathcal{L}$ is not fully ordered (as we show in Section 4, polynomial minimization is possible for LDFAs over fully ordered lattice), yet on the other hand the reduction has to be polynomial and thus use only polynomially many values of the subset lattice.

the closure for both $join$ and $meet$ is obvious. Otherwise, we handle the operators separately as follows. We start with the case of $meet$. Closing to $meet$ is easy since $l_1 \wedge l_2$ never contains more than one edge. Indeed, if l_1, l_2 are of the form $touch(v_1)$, $touch(v_2)$ then their $meet$ is the single edge (v_1, v_2). In all other possibilities for l_1 and l_2 that are not ordered, one of them contains at most two edges, so the fact they are not ordered implies that they have at most one edge in their $meet$. As for $join$, given l_1 and l_2 let $S = \{l : l \geq l_1 \text{ and } l \geq l_2\}$. We claim that all the elements in S are ordered, thus we can define $l_1 \vee l_2$ to be the minimal element in S. Assume by way of contradiction that S contains two elements l and l' that are not ordered. On the one hand, $l \wedge l' \geq l_1 \vee l_2$. Since l_1 and l_2 are not ordered, this implies that $l \wedge l'$ is of size at least two. On the other hand, as we argued above, the $meet$ of two elements that are not ordered contains at most one edge, and we have reached a contradiction.

- $Q = \{q_{init}, q_0, ..., q_{m-1}\}$.
- $\Sigma = \{e_0, ..., e_{m-1}\} \cup \{\#\}$.
- For $0 \leq i < m$, we define $\delta(q_{init}, e_i, q_i) = \{e_i, far(e_i)\}$ and $\delta(q_i, \#, q_{(i+1) mod m}) = \{e_i\}$. For all other $q \in Q$ and $\sigma \in \Sigma$, we define $\delta(q, \sigma, q) = \bot$.
- $Q_0(q_{init}) = \top$, and $Q_0(q) = \bot$ for all other $q \in Q$.
- $F(q) = \top$ for all $q \in Q$.

For example, the graph G in Example 1 induces the LDFA $\mathcal{A}_G$ below.

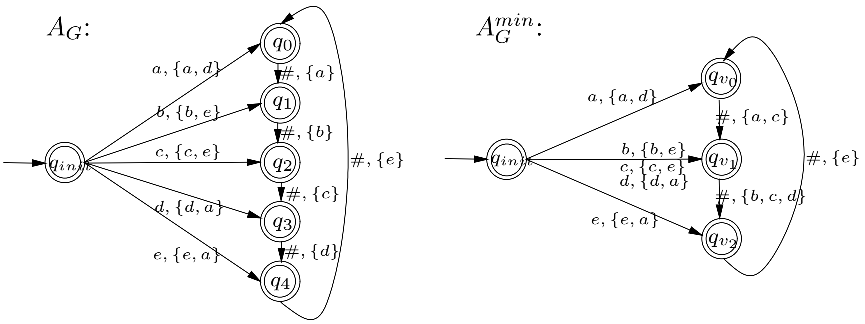


Fig. 4. The LDFA induced by G, and the minimal LDFA that corresponds to the 3-cover $\{3, 4, 5\}$

It is not hard to see that the $\mathcal{L}$-language induced by $\mathcal{A}$, denoted L, is such that for all $e \in \Sigma$, we have that $L(e) = \{e, far(e)\}$ and $L(e \cdot \#) = \{e\}$. In addition, $L(\epsilon) = \top$, and for all other $w \in \Sigma^*$, we have that $L(w) = \bot$. Also, $\mathcal{A}$ is indeed deterministic, and has $m + 1$ states. Finally, since the components of $\mathcal{A}$ are all of size polynomial in the input graph, the reduction is polynomial.

In the full version we proved that G has a k-VC iff there is an LDFA with $k + 1$ states for L.

4 Minimizing an LDFA over a Fully Ordered Lattice

In Section 3, we saw that the problem of minimizing LDFAs is NP-complete. The hardness proof involved a partially ordered lattice, embodied in the "subset lattice", and strongly utilized on the order being partial. This suggests that for fully ordered lattices, we may still be able to find a polynomial minimization algorithm. On the other hand, as we show below, the property of no canonical minimal LDFA is valid already in the case of fully ordered lattice, and there is a tight connection between this and the fact we could not come up with a polynomial algorithm in the general case.

Example 2. Let $\mathcal{L} = \langle \{0, 1, 2, 3\}, \leq \rangle$, and let L be the $\mathcal{L}$-language over $\Sigma = \{0, 1, 2\}$, where $L(\epsilon) = 3, L(0) = 3, L(0{\cdot}0) = 1, L(1) = 1, L(1{\cdot}0) = 1, L(2) = 3, L(2{\cdot}0) = 2$, and $L(x) = 0$ for all other $x \in \Sigma^*$.

Note that L is monotonic, in the sense that for all $x, z \in \Sigma^*$, we have that $L(x \cdot z) \leq L(x)$. For monotonic $\mathcal{L}$-languages, it is tempting to consider the relation $\sim_L \subseteq \Sigma^* \times \Sigma^*$ such that $x \sim_L x'$ iff for all $z \in \Sigma^*$ there exists $l \in \mathcal{L}$ such that $L(x \cdot z) = L(x) \wedge l$ and $L(x' \cdot z) = L(x') \wedge l$. It is not hard to see, however, that $0 \sim_L 1$ and $1 \sim_L 2$, but $0 \not\sim_L 2$. Thus, even for monotonic languages over a fully ordered lattice, a relation that takes the accumulated values into account is not transitive, and there are two minimal LDFAs with different topologies for L. In the first, the letters 0 and 1 lead to a state from which the letter 0 is read with the value 1, and in the second, the letters 1 and 2 lead to a state from which 0 is read with the value 2.

In this section we show that in spite of the non-canonicality, we are able to minimize them in polynomial time. We describe a polynomial time algorithm that is given an LDFA $\mathcal{A} = \langle \mathcal{L}, Q, \Sigma, \delta, Q_0, F \rangle$ over a fully ordered lattice, and returns an LDFA $\mathcal{A}_{min}$ with a minimal number of states, such that $L(\mathcal{A}_{min}) = L(\mathcal{A})$.

Let $\mathcal{L} = \{0, 1, ..., n - 1\}$ be the fully ordered lattice, let $L : \Sigma^* \to \mathcal{L}$ be the language of $\mathcal{A}$, and let $m = max_{w \in \Sigma^*} L(w)$; that is, m is the maximal value of a word in $L(\mathcal{A})$. Finally, let $q_0 \in Q$ be the single state with initial value that is not $\perp$. For each $1 \leq i \leq m$ we define a DFA $\mathcal{A}_i$ that accepts exactly all words w such that $L(w) \geq i$. Note that it is indeed enough to consider only the automata $\mathcal{A}_1, ..., \mathcal{A}_m$, as $\mathcal{A}_{m+1}, ..., \mathcal{A}_{n-1}$ are always empty and hence not needed, and $\mathcal{A}_0$ is not needed as well, as $L(\mathcal{A}_0) = \Sigma^*$.

For $1 \leq i \leq m$, we define $\mathcal{A}_i = \langle Q_i, \Sigma, \delta_i, q_0, F_i \rangle$ as follows:

- $Q_i \subseteq Q$ contains exactly all states that are both reachable from q_0 using transitions with value at least i, and also have some state with acceptance value at least i that is reachable from them using zero or more transitions with value at least i. Note that $q_0 \in Q_i$ for all i.
- δ_i contains all transitions that their value in $\mathcal{A}$ is at least i and that both their source and destination are in Q_i.
- $F_i \subseteq Q_i$ contains all states their acceptance value in $\mathcal{A}$ is at least i.

For readers that wonder why we do not define δ_i first, as these transitions with value at least i, and then define Q_i and F_i according to reachability along δ_i, note that such a definition would result in different automata that are not *trim*, as it may involve transitions that never lead to an accepting state in $\mathcal{A}_i$, and states that are equivalent to a rejecting sink. As we will see later, the fact that all the components in our $\mathcal{A}_i$ are essential is going to be important.

Note that for all $1 < i \leq m$, we have that $Q_i \subseteq Q_{i-1}$, $\delta_i \subseteq \delta_{i-1}$, and $F_i \subseteq F_{i-1}$. Also, it is not hard to see that $\mathcal{A}_i$ indeed accepts exactly all words w such that $L(w) \geq i$.

We now turn back to the given LDFA $\mathcal{A}$ and describe how it can be minimized using $\mathcal{A}_1, ..., \mathcal{A}_m$. First, we apply a pre-processing on $\mathcal{A}$ that reduces the values appearing in $\mathcal{A}$ to be the minimal possible values (without changing the language). Formally, we define $\mathcal{A}' = \langle \mathcal{L}, Q, \Sigma, \delta', Q_0, F' \rangle$, where

- For all $q, q' \in Q$ and $\sigma \in \Sigma$, we have that $\delta'(q, \sigma, q') = max\{i : (q, \sigma, q') \in \delta_i\}$.
- For all $q \in Q$, we have that $F'(q) = max\{i : q \in F_i\}$.

Note that since for all $1 < i \leq m$, we have that $\delta_i \subseteq \delta_{i-1}$ and $F_i \subseteq F_{i-1}$, then for all $1 \leq i \leq m$, we also have that $\delta'(q, \sigma, q') \geq i$ iff $(q, \sigma, q') \in \delta_i$ and $F'(q) \geq i$ iff $q \in F_i$.

Lemma 1. $L(\mathcal{A}) = L(\mathcal{A}')$.

By Lemma 1, it is enough to minimize $\mathcal{A}'$. We start with applying the algorithm for minimizing DFA on $\mathcal{A}_1, ..., \mathcal{A}_m$. Each such application generates a partition of the states of $\mathcal{A}_i$ into equivalence classes.[5] Let us denote the equivalence classes produced for $\mathcal{A}_i$ by $\mathcal{H}_i = \{S^i_1, S^i_2, ..., S^i_{n_i}\}$.

Now, we construct from $\mathcal{A}'$ a minimal automaton $\mathcal{A}_{min} = \langle \mathcal{L}, Q_{min}, \Sigma, \delta_{min}, Q_{0_{min}}, F_{min} \rangle$ as follows.

- We obtain the set Q_{min} by partitioning the states of $\mathcal{A}'$ into sets, each inducing a state in Q_{min}. The partitioning process is iterative: we maintain a disjoint partition $\mathcal{P}_i$ of the states Q, starting with one set containing all states, and refining it along the iterations. The refinement at the i-th iteration is based on $\mathcal{H}_i$, and guarantees that the new partition $\mathcal{P}_i$ agrees with $\mathcal{H}_i$, meaning that states that are separated in $\mathcal{H}_i$ are separated in $\mathcal{P}_i$ as well. At the end of this process, the sets of the final partition constitute Q_{min}.

  More specifically, the algorithm has $m + 1$ iterations, starting with $i = 0$, ending with $i = m$. Let us denote the partition obtained at the i-th iteration by $\mathcal{P}_i = \{T^i_1, ..., T^i_{d_i}\}$. At the first iteration, for $i = 0$, we have that $d_0 = 1$, and $T^0_1 = Q$. At the i-th iteration, for $i > 0$, we are given the partition $\mathcal{P}_{i-1} = \{T^{i-1}_1, ..., T^{i-1}_{d_{i-1}}\}$, and generate $\mathcal{P}_i = \{T^i_1, ..., T^i_{d_i}\}$ as follows. For each $1 \leq j \leq d_{i-1}$, we examine T^{i-1}_j and partition it further. We do it in two stages. First, we examine $S^i_1, S^i_2, ..., S^i_{n_i}$ and for each $1 \leq k \leq n_i$ we compute the set $U^i_{j,k} = T^{i-1}_j \cap S^i_k$, and if $U^i_{j,k} \neq \emptyset$, then we add $U^i_{j,k}$ to $\mathcal{P}_i$. Thus, we indeed separate the states that are separated in $\mathcal{H}_i$. At the second stage, we consider the states in T^{i-1}_j that do not belong to $U^i_{j,k}$ for all k. Note that these states do not belong to Q_i, so $\mathcal{A}_i$ is indifferent about them. This is the stage where we have a choice in the algorithm. We choose an arbitrary k for which $U^i_{j,k} \neq \emptyset$, and add these states to $U^i_{j,k}$. If no such k exists, we know that no state in T^{i-1}_j appears in Q_i, so we have no reason to refine T^{i-1}_j, and we can add T^{i-1}_j to $\mathcal{P}_i$. Finally, we define Q_{min} to be the sets of $\mathcal{P}_m$.

[5] Note that, by definition, all the states in Q_i have some accepting state reachable from them, so the fact we do not have a rejecting state is not problematic, as such a state would have constitute a singleton state in all the partitions we are going to consider.

- The transition relation δ_{min} is defined as follows. Consider a state $T \in Q_{min}$. We choose one state $q^T_{rep} \in T$ to be a *representative* of T, as follows. Let $i^T_{max} = max\{i$: there is $q \in T$ s.t. $q \in Q_i\}$, and let q^T_{rep} be a state in $T \cap Q_{i^T_{max}}$. Note that $T \cap Q_{i^T_{max}}$ may contain more than one state, in which case we can assume Q is ordered and take the minimal. We now define the transitions leaving T according to the original transitions of q^T_{rep} in $\mathcal{A}'$. For $\sigma \in \Sigma$, let $q_{dest} \in Q$ be the σ-destination of q^T_{rep} in $\mathcal{A}'$. For all $T' \in Q_{min}$, if $q_{dest} \in T'$ we define $\delta_{min}(T, \sigma, T') = \delta'(q^T_{rep}, \sigma, q_{dest})$; otherwise, $\delta_{min}(T, \sigma, T') = 0$.
- For all $T \in Q_{min}$, if $q_0 \in T$, where q_0 is the initial state of $\mathcal{A}'$, we define $Q_{0_{min}}(T) = Q_0(q_0)$; otherwise, $Q_{0_{min}}(T) = 0$.
- For all $T \in Q$, we define $F_{min}(T) = F'(q^T_{rep})$.

An example illustrating an execution of the algorithm can be found in the full version.

Let $L_{min} = L(\mathcal{A}_{min})$ and $L' = L(\mathcal{A}')$. We prove that the construction is correct. Thus, $L_{min} = L'$, $|\mathcal{A}_{min}|$ is minimal, and the time complexity of the construction of $\mathcal{A}_{min}$ is polynomial.

We first prove that $L_{min} = L'$. For $q, q' \in Q$, we say that $q \sim_i q'$ iff there exists some class $S \in \mathcal{H}_i$ such that $q, q' \in S$. Also, we say that $q \equiv_i q'$ iff for all $j \leq i$ it holds that $q \sim_j q'$. Note that although $\sim_i$ and $\equiv_i$ are equivalence relations over $Q_i \times Q_i$, they are not equivalence relations over $Q \times Q$, as they are not reflexive. Indeed, for $q \in Q \setminus Q_i$, there is no class $S \in \mathcal{H}_i$ such that $q \in S$, so $q \not\sim_i q$ and of course $q \not\equiv_i q$. However, it is easy to see that $\sim_i$ and $\equiv_i$ are both symmetric and transitive over $Q \times Q$.

Lemma 2 below explains the essence of the relation $\equiv_i$. As we shall prove, if $q \equiv_i q'$ then q and q' agree on the transition and acceptance values in $\mathcal{A}'$, if these values are less than i.

Lemma 2. *For $q, q' \in Q$, if $q \equiv_i q'$ then for all $j < i$, the following hold.*

- *For all $\sigma \in \Sigma$, we have $\delta'(q, \sigma, s) = j$ iff $\delta'(q', \sigma, s') = j$, where $s, s' \in Q$ are the σ-destinations of q, q' in $\mathcal{A}'$, respectively. .*
- *$F'(q) = j$ iff $F'(q') = j$.*

In the case of DFA, we know that each state of the minimal automaton for L corresponds to an equivalence class of $\sim_L$, and the minimization algorithm merges all the states of the DFA that correspond to each class into a single state. Consequently, the transition function of the minimal automaton can be defined according to one of the merged states, and the definition is independent of the state being chosen. In the case of our $\mathcal{A}_{min}$, things are more complicated, as states that are merged in $\mathcal{A}_{min}$ do not correspond to equivalence classes. We still were able, in the definition of the transitions and acceptance values, to chose a state q^T_{rep}, for each state T. Lemma 3 below explains why working with the chosen state is sound.

The lemma considers a word $w \in \Sigma^*$, and examines the connection between a state q_i in the run of $\mathcal{A}'$ on w and the corresponding state T_i in the run of $\mathcal{A}_{min}$ on w. It shows that if $L'(w) \geq l$ for some $l \in \mathcal{L}$, then $q_i \equiv_l q^{T_i}_{rep}$ for all i. Thus, the states along the run of $\mathcal{A}_{min}$ behave the same as the corresponding states in $\mathcal{A}'$ on values that are less than l, and may be different on values that are at least l, as long as they are both at least l. Intuitively, this is valid since after reaching a value l, we can replace all subsequent values $l' \geq l$ along the original run with any other value $l'' \geq l$.

Lemma 3. *Let $w = \sigma_1\sigma_2...\sigma_k$ be a word in Σ^*, and let $q_0, q_1, ..., q_k$ and $T_0, T_1, ..., T_k$ be the runs of $\mathcal{A}'$ and $\mathcal{A}_{min}$ on w respectively. For $l \in \mathcal{L}$, if $L'(w) \geq l$ then for all $0 \leq i \leq k$ it holds that $q_i \equiv_l q_{rep}^{T_i}$.*

Based on the above, we now turn to prove that $L_{min} = L'$. Let $w \in \Sigma^*$, and let $l = L'(w)$. We show that $L_{min}(w) = l$. Let $r' = q_0, q_1, ..., q_k$ and $r_{min} = T_0, T_1, ..., T_k$ be the runs of $\mathcal{A}'$ and $\mathcal{A}_{min}$ on w respectively.

We first show that $L_{min}(w) \geq l$. Consider the values read along r', which are $Q_0(q_0)$, $\delta'(q_0, \sigma_1, q_1)$, $\ldots$, $\delta'(q_{k-1}, \sigma_k, q_k)$, and $F'(q_k)$. Since $L'(w) = l$ we know that all these values are at least l. By Lemma 3 we get that for all $0 \leq i \leq k$ it holds that $q_i \equiv_l q_{rep}^{T_i}$. Then by applying the first part of Lemma 2 on $q_0, ..., q_{k-1}$ and the second part on q_k, we get that the values $\delta'(q_{rep}^{T_0}, \sigma_1, s_0)$, $\ldots$, $\delta_{min}(q_{rep}^{T_{k-1}}, \sigma_k, s_{k-1})$ and $F_{min}(q_{rep}^{T_k})$ are all at least l, where s_i is the σ_i-destination of $q_{rep}^{T_i}$ for all $0 \leq i < k$. Thus, we get that $\delta_{min}(T_0, \sigma_1, T_1), \ldots, \delta_{min}(T_{k-1}, \sigma_k, T_k)$ and $F_{min}(T_k)$ are all at least l as well, since these values are defined according to $q_{rep}^{T_0}, ..., q_{rep}^{T_k}$. Together with the fact that the initial value remains the same in $\mathcal{A}_{min}$, we get that $L_{min}(w) \geq l$.

In order to prove that $L_{min}(w) \leq l$, we show that at least one of the values read along r_{min} is l. Since $L'(w) = l$, at least one of the values read along r' is l. If this value is $Q_0(q_0)$ then we are done, since by definition $Q_0(T_0) = Q_0(q_0)$. Otherwise, it must be one of the values $\delta'(q_0, \sigma_1, q_1), \ldots, \delta'(q_{k-1}, \sigma_k, q_k)$ or $F'(q_k)$. Let q_d be the state from which the value l is read for the first time along r', either as a transition value ($d < k$) or as an acceptance value ($d = k$). We claim that $q_d \in Q_{l+1}$ (note that if $l = m$, then clearly $L_{min}(w) \leq l$, thus, we assume that $l < m$, so Q_{l+1} is well defined). If $d = 0$, then we are done, since $q_0 \in Q_{l+1}$ by definition. Otherwise, we look at the transition (q_{d-1}, σ_d, q_d). By the definition of q_d, we know that $\delta'(q_{d-1}, \sigma_d, q_d) \geq l+1$, and by the definition of δ' it then follows that $(q_{d-1}, \sigma_d, q_d) \in \delta_{l+1}$. Thus, we get that $q_d \in Q_{l+1}$. Now, by the definition of Q_{l+1}, there exists some state $q_{acc}^{l+1} \in Q$ with acceptance value at least $l + 1$ that is reachable from q_d in $\mathcal{A}$ using zero or more transitions with value at least $l + 1$. Let w' be the word read along these transitions from q_d to q_{acc}^{l+1}, and let $w'' = \sigma_1, ..., \sigma_d \cdot w'$. It is easy to see that $L'(w'') \geq l+1$. Thus, we can apply Lemma 3, and get that $q_d \equiv_{l+1} q_{rep}^{T_d}$. Then, by applying Lemma 2 on q_d and $q_{rep}^{T_d}$ we conclude that the value l is read from T_d along r_{min}, and we are done.

We now turn to prove that $|\mathcal{A}_{min}|$ is minimal. Let $N = |\mathcal{A}_{min}|$. We describe below N different words $w_1, ..., w_N \in \Sigma^*$, and prove that for all $i \neq j$ the words w_i and w_j cannot reach the same state in an LDFA for L. Clearly, this implies that an LDFA for L must contain at least N states, so $|\mathcal{A}_{min}|$ is indeed minimal.

We define the words $w_1, ..., w_N$ as follows. Let $T_1, ..., T_N$ be the states of $\mathcal{A}_{min}$, and let $q_{rep}^{T_1}, ..., q_{rep}^{T_N}$ be their representatives respectively. We go back to the original automaton $\mathcal{A}$, and for each such representative q, we define the following:

- $reach(q) = \{w \in \Sigma^* : \delta(q_0, w, q) > 0\}$, where q_0 is the initial state of $\mathcal{A}$.
- $maxval(q) = max\{\delta(q_0, w, q) : w \in reach(q)\}$. Note that $maxval(q)$ considers only the traversal values of the words reaching q.
- $maxw(q)$ is $w \in \Sigma^*$ for which $\delta(q_0, w, q) = maxval(q)$. Note that there may be several such words, so we can take the lowest one by lexicographic order, to make it well defined.

For all $1 \leq i \leq N$, we now define $w_i = maxw(q^{T_i}_{rep})$. Note that these words are indeed different, as they reach different states in the deterministic automaton $\mathcal{A}$. Consider two different indices i and j. We prove that the words $maxw(q^{T_i}_{rep})$ and $maxw(q^{T_j}_{rep})$ cannot reach the same state in an LDFA for L. Consider the states $q^{T_i}_{rep}$ and $q^{T_j}_{rep}$. These states belong to different sets in Q_{min}. Let $1 \leq l \leq m$ be the index of the iteration in which they were first separated.

We use the following lemmas:

Lemma 4. *The states $q^{T_i}_{rep}$ and $q^{T_j}_{rep}$ belong to different classes in $\mathcal{H}_l$.*

Lemma 5. *$tr\_val(maxw(q^{T_i}_{rep})) \geq l$ and $tr\_val(maxw(q^{T_j}_{rep})) \geq l$ in all LDFA for L.*

Based on the above, we prove that the words $maxw(q^{T_i}_{rep})$ and $maxw(q^{T_j}_{rep})$ cannot reach the same state in an LDFA for L. By Lemma 4, there is a distinguishing tail $z \in \Sigma^*$. That is, without loss of generality, z is read in $\mathcal{A}$ from $q^{T_i}_{rep}$ with value at least l, and is read from $q^{T_j}_{rep}$ with value less than l. Let us examine the words $maxw(q^{T_i}_{rep}) \cdot z$ and $maxw(q^{T_j}_{rep}) \cdot z$. By applying Lemma 5 on $\mathcal{A}$, we get that $L(\mathcal{A})(maxw(q^{T_i}_{rep}) \cdot z) \geq l$ and that $L(\mathcal{A})(maxw(q^{T_j}_{rep}) \cdot z) < l$. Now, let $\mathcal{U}$ be an LDFA for L, and assume by way of contradiction that $maxw(q^{T_i}_{rep})$ and $maxw(q^{T_j}_{rep})$ are reaching the same state in $\mathcal{U}$. Let q be that state. Applying Lemma 5 on $\mathcal{U}$, we get that both $maxw(q^{T_i}_{rep})$ and $maxw(q^{T_j}_{rep})$ are reaching q with traversal value at least l. Now, let us examine the value v_z with which z is read from q. If $v_z \geq l$, then $L(\mathcal{U})(maxw(q^{T_j}_{rep}) \cdot z) \geq l$, which contradicts the fact that $L(\mathcal{A})(maxw(q^{T_j}_{rep}) \cdot z) < l$. On the other hand, if $v_z < l$, then $L(\mathcal{U})(maxw(q^{T_i}_{rep}) \cdot z) < l$, which contradicts the fact that $L(\mathcal{A})(maxw(q^{T_i}_{rep}) \cdot z) \geq l$.

Thus, we conclude that $|\mathcal{A}_{min}|$ is minimal.

It is not hard to see that each of the three stages of the algorithm (constructing the automata $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m$ and minimizing them, generating $\mathcal{A}'$ from $\mathcal{A}$, and constructing $\mathcal{A}_{min}$ from $\mathcal{A}'$) can be implemented in polynomial time. In the full version we analyse the complexity in detail, and show that the overall complexity is $O(|\mathcal{L}|(|Q| \log |Q| + |\delta|))$.

We can now conclude with the following.

Theorem 2. *An LDFA over a fully ordered lattice can be minimized in polynomial time.*

References

1. Alur, R., Kanade, A., Weiss, G.: Ranking automata and games for prioritized requirements. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 240–253. Springer, Heidelberg (2008)
2. ESF Network programme. Automata: from mathematics to applications (AutoMathA) (2010), http://www.esf.org/index.php?id=1789
3. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999)

4. Bruns, G., Godefroid, P.: Temporal logic query checking. In: Proc. 16th LICS, pp. 409–420. IEEE Computer Society, Los Alamitos (2001)
5. Chan, W.: Temporal-logic queries. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 450–463. Springer, Heidelberg (2000)
6. Chechik, M., Devereux, B., Gurfinkel, A.: Model-checking infinite state-space systems with fine-grained abstractions using SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 16–36. Springer, Heidelberg (2001)
7. Droste, M., Kuich, W., Vogler, H. (eds.): Handbook of Weighted Automata. Springer, Heidelberg (2009)
8. Easterbrook, S., Chechik, M.: A framework for multi-valued reasoning over inconsistent viewpoints. In: Proc. 23rd Int. Conf. on Software Engineering, pp. 411–420. IEEE Computer Society Press, Los Alamitos (2001)
9. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
10. Henzinger, T.A.: From boolean to quantitative notions of correctness. In: Proc. 37th POPL, pp. 157–158 (2010)
11. Hussain, A., Huth, M.: On model checking multiple hybrid views. Technical Report TR-2004-6, University of Cyprus (2004)
12. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations, pp. 85–103 (1972)
13. Kupferman, O., Lustig, Y.: Lattice automata. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 199–213. Springer, Heidelberg (2007)
14. Kupferman, O., Lustig, Y.: Latticed simulation relations and games. International Journal on the Foundations of Computer Science 21(2), 167–189 (2010)
15. Kirsten, D., Mäurer, I.: On the determinization of weighted automata. Journal of Automata, Languages and Combinatorics 10(2/3), 287–312 (2005)
16. Krob, D.: The equality problem for rational series with multiplicities in the tropical emiring is undecidable. Journal of Algebra and Computation 4, 405–425 (1994)
17. Li, Y., Pedrycz, W.: Minimization of lattice finite automata and its application to the decomposition of lattice languages. Fuzzy Sets and Systems 158(13), 1423–1436 (2007)
18. Mohri, M.: Finite-state transducers in language and speech processing. Computational Linguistics 23(2), 269–311 (1997)
19. Malik, D.S., Mordeson, J.N., Sen, M.K.: Minimization of fuzzy finite automata. Information Sciences 113, 323–330 (1999)
20. Myhill, J.: Finite automata and the representation of events. Technical Report WADD TR-57-624, pp. 112–137, Wright Patterson AFB, Ohio (1957)
21. Nerode, A.: Linear automaton transformations. Proceedings of the American Mathematical Society 9(4), 541–544 (1958)
22. Zekai, L., Lan, S.: Minimization of lattice automata. In: Proc. 2nd ICFIE, pp. 194–205 (2007)

Regularity and Context-Freeness over Word Rewriting Systems

Didier Caucal and Trong Hieu Dinh

LIGM, UMR CNRS 8049, Université Paris-Est, Marne-la-Valle, France
{caucal,dinh}@univ-mlv.fr

Abstract. We describe a general decomposition mechanism to express the derivation relation of a word rewriting system R as the composition of a (regular) substitution followed by the derivation relation of a system $R' \cup D$, where R' is a strict sub-system of R and D is the Dyck rewriting system. From this decomposition, we deduce that the system R (resp. R^{-1}) preserves regular (resp. context-free) languages whenever $R' \cup D$ (resp. its inverse) does. From this we can deduce regularity and context-freeness preservation properties for a generalization of tagged bifix systems.

1 Introduction

A central problem in the reachability analysis of word rewriting systems is, given a language L and a system R, to determine the set $\longrightarrow_R^*(L)$ of all words which can be derived by R from some word in L. Though this set is not recursive in general for L finite, a lot of attention has been devoted to the characterization of rewriting systems whose derivation relation $\longrightarrow_R^*$ preserves classes of languages with good decidability or closure properties. In particular, a system R is said to preserve regularity (resp. context-freeness) if, for any regular (resp. context-free) language L, $\longrightarrow_R^*(L)$ is also regular (resp. context-free).

Many classes of rewriting systems preserving regularity or context-freeness can be found in the literature. For instance, it is well known that the prefix derivation of any finite rewriting system preserves regularity [8,9], and that the so-called context-free systems (systems whose left-hand sides are of length at most 1) preserve context-free languages and their inverse derivations preserve regularity (see for instance [6]). In [13], Hofbauer and Waldmann proved that the derivation of any finite deleting system preserves regularity and that its inverse derivation preserves context-freeness, thus completing a result by Hibbard [12]. They provided a clever decomposition of the derivation relation into a finite substitution followed by the derivation of an inverse context-free system and a restriction to the original alphabet. From this, they were able to deduce many previously known preservation results. In [10], Endrullis, Hofbauer and Waldmann gave a general decomposition of the derivation of any system into a context-free system followed by an inverse context-free system with empty right hand sides. The main contribution of our paper is to use this derivation decomposition idea to extend the decomposition of [13] to infinite rewriting systems with prefix and suffix rules.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 214–228, 2011.

Our construction is based on the following observation. Given a word $u = a_1 \dots a_n$, let us write $\overleftarrow{u} = \overleftarrow{a_n} \dots \overleftarrow{a_1}$ and $\overrightarrow{u} = \overrightarrow{a_n} \dots \overrightarrow{a_1}$ where $\overleftarrow{a}$ and $\overrightarrow{a}$ are fresh letters for all a. Let R be a rewriting system and $u \to v \in R$ be one of its rules, and consider factors of the form $\overleftarrow{u_1} v \overrightarrow{u_2}$ with $u_1 u_2 = u$. The intended meaning is that as an effect of this rewrite rule, right-hand side v can be inserted at a certain position i in a word provided that u_1 can be erased to the left of position i and u_2 to the right. In other words, applying rule $u \to v$ to a word can be simulated by first inserting some such factor $\overleftarrow{u_1} v \overrightarrow{u_2}$ at an appropriate position, and then erasing factors of the form $u \overleftarrow{u}$ or $\overrightarrow{u} u$. This double-phased procedure described in [10] can be performed as a substitution followed by a normalization using inverse context-free rules of the form $\overrightarrow{a} a \to \varepsilon$ and $a \overleftarrow{a} \to \varepsilon$ (constituting what we call the Dyck rewriting system, see also [15]).

Under certain syntactical criteria, this simulation step can be used to eliminate rewrite rules from the original rewriting system altogether. More precisely, we are able to decompose the derivation of a system R into a (regular) substitution h, whose role is to insert factors (as described above) corresponding to some subset R' of R, followed by the derivation according to a system $S \cup D$, where S is simply $R - R'$ and D denotes the Dyck system; we say that R can be decomposed into S. As a consequence, the derivation of R (resp. R^{-1}) preserves regularity (resp. context-freeness) if the derivation of $S \cup D$ (resp. its inverse) does. This remains true even for infinite systems, as long as the relation R' is recognizable.

This result can be used to characterize several families of systems whose derivations preserve regularity or context-freeness. First, we observe that in the case of deleting systems the decomposition yields an empty S (i.e. all rules can be simulated and eliminated from R). Since the Dyck system is inverse context-free, this indeed extends the result of [13] to infinite (recognizable) systems. Moreover, contrary to [13] our decomposition only uses a single inverse context-free system, namely D. Note however that many other systems can be directly decomposed into the empty system, for instance the well-known prefix rewriting systems (which encode pushdown system transition relations), their bifix variant, and left-to-right systems; in the finite case, most of these systems can also be simulated by deleting systems [13]. As an example, since multi-pushdown systems as defined in [7] can be seen as left-to-right systems, we can recover from our results that their transition relations preserve context-freeness and their inverse preserves regularity.

Our main application concerns tagged systems, which generalize the notions of prefix and suffix rewriting. Given a set of special symbols called tags, which we separate into prefix and suffix tags, we consider rules of the form $\#u \longrightarrow \#'v$, where $\#, \#'$ are prefix tags and u does not contain any tags. We also allow suffix, bifix, and untagged rules which are defined similarly. Since v may contain tags, this strictly extends the earlier notions of tagged systems defined in [1]. If the set of tagged rules is recognizable and the set of untagged rules is context-free, we show that our decomposition result applies, which entails that the derivation of such a system (resp. its inverse) preserves context-freeness (resp. regularity). This result still holds when we do not partition the set of tags, at the cost of imposing that tags in the left-hand side of a rule remain invariant in the corresponding right-hand side. Both results extend previously known preservation properties of simpler tagged systems [1].

The remainder of the paper is organized as follows. After some elementary notations and definitions, Section 2 presents our derivation decomposition theorem, and relates it to the class of deleting systems. Section 3 details several classes of rewriting systems whose known preservation results can be recovered using our technique (prefix, suffix and bifix systems in Section 3.1) or for which new preservation results can be shown (left-to-right systems in Section 3.2, tagged systems in Section 3.3).

2 Derivation Decomposition

This section focuses on regularity and context-freeness preservation properties for rewriting systems. After reviewing some known preservation results (Section 2.2), we generalize the derivation decomposition of [13] to arbitrary rewriting systems (Section 2.3), which allows us to deduce new preservation properties. We start by recalling some basic definitions and notations.

2.1 Notations

For ease of notation, a singleton $\{x\}$ will often be identified with x. The image by a binary relation $R \subseteq E{\times}F$ of a subset $P \subseteq E$ by a binary relation R is $R(P) = \{y \mid \exists\, x \in P,\ x\,R\,y\}$. Let N be a finite set of symbols (called an *alphabet*), we write $\mathrm{Alph}(u) = \{u(i) \mid 1 \leq i \leq |u|\}$ the set of letters occurring in a word $u \in N^*$ (whose $u(i)$ the letter of u at position i). This is extended by union to any language P over N: $\mathrm{Alph}(P) = \{a \mid \exists\, u \in P,\ a \in \mathrm{Alph}(u)\ \}$. The *concatenation* of binary relations R, S on N^* is $R.S = \{(ux, vy) \mid u\,R\,v {\wedge} x\,S\,y\}$, and the left and right concatenation of R by a language $P \subseteq N^*$ is $R.P = R.\mathrm{Id}_P = \{(uw, vw) \mid u\,R\,v \wedge w \in P\}$ and $P.R = \mathrm{Id}_P.R$, where $\mathrm{Id}_P = \{(w, w) \mid w \in P\}$ denotes the *identity relation* on P.

A *regular language* over N is the language recognized by a finite automaton labelled in N (or N^*). A *substitution* h over N is a binary relation on N^* whose image $h(a)$ is defined for every letter $a \in N$ and extended by morphism to words: $h(a_1 \ldots a_n) = h(a_1) \ldots h(a_n)$ for all $n \geq 0$ and $a_1, \ldots, a_n \in N$. It is said to be finite (resp. regular) if $h(a)$ is a finite (resp. regular) language for all $a \in N$. A *recognizable relation* R on N^* is a finite union of binary products of regular languages: $R = U_1{\times}V_1 \cup \ldots \cup U_p{\times}V_p$ for some $p \geq 0$ and regular languages $U_1, V_1, \ldots, U_p, V_p$. A *transducer* A over N is an automaton labelled in $N^*{\times}N^*$ whose language is interpreted as a binary relation, called a *rational relation* [4].

A *word rewriting system* (or just *system*) R over an alphabet N is a binary relation on N^* seen as a set of rules (u, v); we do not assume R to be finite. Let $\mathrm{Alph}(R)$ be the set of letters of R. The *rewriting relation* (or single step reduction) of R is the binary relation $\longrightarrow_R\, = N^*.R.N^*$, *i.e.* $xuy \longrightarrow_R xvy$ for all $u\,R\,v$ and $x, y \in N^*$; we also sometimes write $xuy \longrightarrow_{R,|x|} xvy$ to denote the position $|x|$ where the rule is applied. Note that $(\longrightarrow_R)^{-1} = \longrightarrow_{R^{-1}}$. The *derivation relation* (or reduction relation) $\longrightarrow_R^*$ of R is the reflexive and transitive closure (under composition) of $\longrightarrow_R$, *i.e.* $u \longrightarrow_R^* v$ if there exist $n \geq 0$ and $u_0, \ldots, u_n \in N^*$ such that $u = u_0 \longrightarrow_R u_1 \ldots \longrightarrow_R u_n = v$. Note that $(\longrightarrow_R^*)^{-1} = \longrightarrow_{R^{-1}}^*$.

A *context-free grammar* over N is a finite relation $R \subseteq M \times (M \cup N)^*$ for some alphabet M disjoint of N; it generates from $u \in (M \cup N)^*$ the *context-free language* $L(R, u) = \{v \in N^* \mid u \longrightarrow_R^* v\}$.

2.2 Preservation Properties

A first and very well-known preservation result is that any rational relation R (and its inverse) preserves regularity: the image $R(L)$ of any regular language L remains regular. It is also well-known in the field of language theory that the family of context-free languages is also closed under rational relations [4].

Lemma 2.1. *Any rational relation preserves regularity and context-freeness.*

Since both recognizable relations and regular substitutions are special cases of rational relations, it follows that regular and context-free languages are also closed under direct and inverse recognizable relations and regular substitutions.

In this paper, we are concerned with the characterization of classes of rewriting systems whose derivation relations preserve regularity or context-freeness, using a general decomposition mechanism detailed in the next subsection. A simple way to simulate the application of a rewriting rule (uv, w) to a word x is to insert, at the appropriate position in x, a factor $\overleftarrow{u}\,w\,\overrightarrow{v}$ whose intended meaning is that at this position, right-hand side w can appear after applying the rule if a factor u can be deleted on the left and v on the right. After this word is inserted, appropriate deletions are performed using a single rewriting system called the Dyck system. Therefore, the language preservation properties of that rewriting system play a central role in our study.

Formally, let $R \subseteq N^* \times N^*$ be a rewriting system, and consider a new alphabet $\overleftrightarrow{N} = \overrightarrow{N} \cup \overleftarrow{N}$ consisting of two disjoint copies of N, with $\overrightarrow{N} = \{\overrightarrow{a} \mid a \in N\}$ and $\overleftarrow{N} = \{\overleftarrow{a} \mid a \in N\}$. This notation is extended to words over N as follows: $\overrightarrow{a_1 \ldots a_n} = \overrightarrow{a_n} \ldots \overrightarrow{a_1}$ and $\overleftarrow{a_1 \ldots a_n} = \overleftarrow{a_n} \ldots \overleftarrow{a_1}$ for any $n \geq 0$ and $a_1, \ldots, a_n \in N$. The *Dyck system* $D = N\!\downarrow \cup \downarrow\!N$ defined over $\overline{N} = N \cup \overleftrightarrow{N}$ is the union of the *right* and *left Dyck systems* $N\!\downarrow = \{(\overrightarrow{a}\,a, \varepsilon) \mid a \in N\}$ and $\downarrow\!N = \{\ (a\,\overleftarrow{a}\,, \varepsilon) \mid a \in N\}$.

For any rule (uv, w) in R, the word xwy obtained by rewriting $xuvy$ can also be derived from the word $xu\overleftarrow{u}\,w\,\overrightarrow{v}\,vy$ using D. Note that when $u = \varepsilon$ (resp. $v = \varepsilon$), it suffices to use $N\!\downarrow$ (resp. $\downarrow\!N$). It is a classical and widely-used result that the derivation relation of $N\!\downarrow$ preserves regularity [3] but not context-freeness. An example [14] is to take the context-free languages L and M solutions of the equations $L = \overrightarrow{a}\,La \cup M$ and $M = b \cup aMM\overrightarrow{a}$. So $\longrightarrow_{N\downarrow}^*(L)$ is not context-free:

$$\xrightarrow[N\downarrow]{*}(L) \cap b^* \;=\; \{b^{2^n} \mid n \geq 0\}$$

Furthermore the derivation of $N{\downarrow}^{-1}$ preserves context-freeness but not regularity:

$$\xrightarrow[N{\downarrow}^{-1}]{*}(\varepsilon) \cap \overrightarrow{a}^*a^* \;=\; \{\overrightarrow{a}^n a^n \mid n \geq 0\}$$

which is not regular (but is context-free). We thus call the system $N{\downarrow}$ reg/cf-preserving, as defined below.

Definition 2.2. *A system R is reg/cf-preserving if its derivation relation preserves regularity and its inverse derivation preserves context-freeness.*
A system R is cf/reg-preserving if R^{-1} is reg/cf-preserving.

One can extend the reg/cf-preservation of the (right) Dyck system to wider classes of rewriting systems. We say that a binary relation R on N^* is a *context-free system* if $R \subseteq (N \cup \{\varepsilon\}){\times}N^*$ with $R(a)$ a context-free language for all $a \in N \cup \{\varepsilon\}$. The system D^{-1} is a context-free system.

Proposition 2.3 ([5]). *Context-free systems are cf/reg-preserving.*

Another class of cf/reg-preserving systems is defined in [12]. A system R is called *context-limited* if there exists a partial ordering $<$ on N such that for any rule $(u, v) \in R$, any letter of u is less than some letter of v: $\forall\ a \in \mathrm{Alph}(u)\ \exists\ b \in \mathrm{Alph}(v)\ a < b$. It is shown that the derivation relations of finite context-limited systems preserve context-free languages [12]. Additionally, the inverse R^{-1} of a context-limited system R is called a *deleting system*, and the derivation relation of any finite deleting system preserves regularity [13].

Proposition 2.4 ([12,13]). *Finite context-limited systems are cf/reg-preserving.*

This proposition follows from the decomposition [13] of the derivation relation of any finite deleting system R into a finite substitution h over an extended alphabet composed with the derivation of the inverse of a finite context-free system S, and followed by a restriction to the original alphabet:

$$\xrightarrow[R]{*} \;=\; \big(h \circ \xrightarrow[S^{-1}]{*}\big) \cap N^*{\times}N^*.$$

In the following section, we extend this reasoning to arbitrary rewriting system. We will see in particular that in the case of deleting systems, S^{-1} can always be chosen to be the Dyck system.

2.3 Decomposition

In this subsection we build up on the technical ideas behind Proposition 2.4 and propose a more general notion of derivation decomposition for arbitrary rewriting systems. As already sketched in the previous section, the application of a single rewriting rule (uv, w) to a word x can be simulated by inserting the factor $\overleftarrow{u}\,w\,\overrightarrow{v}$ inside x, and then deleting the extra letters using the Dyck system. We make use of this idea by identifying sets of rules whose role in the derivation can be accurately simulated by this process.

More precisely, for a given rewriting system R over some alphabet N, we identify a subset of rules $R' \subseteq R$ such that

$$\xrightarrow[R]{*} \;=\; \big(h \circ \xrightarrow[(R-R')\,\cup\, D]{*}\big) \cap N^*{\times}N^*$$

where h is a substitution inserting factors of the form $\overleftarrow{u}\,w\,\overrightarrow{v}$. This decomposition is performed by eliminating left or right recursion from the system. Formally, for any $R \subseteq N^*{\times}N^*$ and $M \subseteq N$, we define the sub-system

$$R_M \;=\; \{(u,v) \in R \mid \mathrm{Alph}(uv) \cap M \neq \emptyset\}$$

consisting of all the rules of R with a letter in M; hence $R - R_M$ is the maximal sub-system of R over $N - M$. We want to decompose the derivation of R into the composition of some substitution h together with the derivation of the system $(R - R_M) \cup D$ for suitable subsets M of N.

Definition 2.5. *A set $M \subseteq \mathrm{Alph}(R)$ is called a prefix sub-alphabet of R if*

$$R \;\subseteq\; MN^* \times M(N-M)^* \;\cup\; N^* \times (N-M)^*.$$

This definition means that for each rule $(u, av) \in R$ with $a \in N$, v has no letter in M, and if $a \in M$ then u must begin by a letter in M (see Example 2.8). Note that the set of prefix sub-alphabets of R is closed under union and we can compute its maximal element (with respect to inclusion). For any prefix sub-alphabet M of R, we define over $\overline{N}$ the language

$$P \;=\; \{\overleftarrow{u}\,w\,\overrightarrow{v} \mid uv\,R\,w \wedge u \in (N-M)^* \wedge v \in MN^*\}$$

and the substitution $h_M : \overline{N} \longrightarrow 2^{\overline{N}^*}$ with $h_M(x) = P^*x$ if $x \in M$ and $h_M(x) = x$ otherwise. Both the language P and the substitution h_M are regular whenever R_M is recognizable. When M is a prefix sub-alphabet of R, we can decompose $\longrightarrow^*_R$ by removing R_M from R.

Lemma 2.6. *For any prefix sub-alphabet M of R and for any $u \in \overline{N}^*$,*

$$\xrightarrow[R\,\cup\,D]{*}(u) \cap N^* \;=\; \xrightarrow[(R-R_M)\,\cup\,D]{*}\big(h_M(u)\big) \cap N^*.$$

Proof. Let us write $S \;=\; (R - R_M) \cup D$ and $h = h_M$.
$\supseteq$**:** We first establish two preliminary observations.

First, whenever a factor $\overrightarrow{v}$ (resp $\overleftarrow{v}$) can be removed during a derivation by $R \cup D$, one can always rearrange the derivation steps so that at some point factor v appears immediately to the right (resp. left) of $\overrightarrow{v}$ (resp. $\overleftarrow{v}$), and the resulting factor $\overrightarrow{v}\,v$ (resp. $v\overleftarrow{v}$) is deleted using D. Formally, for any $u, w \in \overline{N}^*$ and any $v, z \in N^*$,

$$w\,\overrightarrow{v}\,u \;\xrightarrow[R\,\cup\,D]{*}\; z \;\Longrightarrow\; \exists\,\overline{w},\; u \;\xrightarrow[R\,\cup\,D]{*}\; v\overline{w} \;\wedge\; w\overline{w} \;\xrightarrow[R\,\cup\,D]{*}\; z,$$
$$u\,\overleftarrow{v}\,w \;\xrightarrow[R\,\cup\,D]{*}\; z \;\Longrightarrow\; \exists\,\overline{w},\; u \;\xrightarrow[R\,\cup\,D]{*}\; \overline{w}v \;\wedge\; \overline{w}w \;\xrightarrow[R\,\cup\,D]{*}\; z.$$

This can be proven by induction on derivation length. For any $R \subseteq N^* \times N^*$, let

$$\overleftrightarrow{R} \;=\; \{\overleftarrow{u}\, w\, \overrightarrow{v} \mid (uv, w) \in R\}$$

We have

$$\xrightarrow[R\,\cup\,D]{*} \big(u[\overleftrightarrow{R}\,]\big) \cap N^* \;\subseteq\; \xrightarrow[R\,\cup\,D]{*}(u) \;\text{ for any } u \in \overline{N}^*,$$

meaning that even randomly inserting factors from $\overleftrightarrow{R}$ in u does not increase the set of words in N^* obtained by derivation using $R{\cup}D$. In other words, the specific positions at which h inserts factors is only relevant for the converse inclusion (which is proven below). The proof is done using the previous observation and for some word $x \in u[\overleftrightarrow{R}\,]$, by induction on the minimal number of insertions of words of $\overleftrightarrow{R}$ which must be performed in order to obtain x from u.

Now let $u \in \overline{N}^*$, since $h(u) \subseteq u[\overleftrightarrow{R}\,]$ and $S \subseteq R{\cup}D$ and by the above inclusion we obtain

$$\xrightarrow[S]{*}(h(u)) \cap N^* \;\subseteq\; \xrightarrow[R\,\cup\,D]{*} \big(u[\overleftrightarrow{R}\,]\big) \cap N^* \;\subseteq\; \xrightarrow[R\,\cup\,D]{*}(u).$$

$\subseteq$**:** Let $u \longrightarrow^*_{R\,\cup\,D} v$ with $u \in \overline{N}^*$. Let us show that $h(v) \subseteq \longrightarrow^*_S \big(h(u)\big)$. To prove this inclusion, we need to make sure that the insertion process performed by h does not prevent any of the words originally derivable from u using $R \cup D$ to be also derivable from $h(u)$ using S. Intuitively, this is guaranteed by the definition of the set P and the substitution h_M which only inserts factors at specific positions.

By induction on the length of the derivation of v from u, it remains to check the inclusion for $u \longrightarrow_{R\,\cup\,D} v$. Let $u = xu_0y$ and $v = xv_0y$ for some $(u_0, v_0) \in R{\cup}D$. We distinguish the three complementary cases below.

Case 1: $(u_0, v_0) \notin R_M \cup D$. By definition $u_0, v_0 \in (N - M)^*$. This means that neither u_0 nor v_0 is affected by h: we have $h(u) = h(x)u_0h(y)$ and $h(v) = h(x)v_0h(y)$. Hence

$$h(v) \;=\; h(x)v_0h(y) \;\subseteq\; \xrightarrow[\{(u_0,v_0)\}]{} \big(h(x)u_0h(y)\big) \;\subseteq\; \xrightarrow[S]{*}\big(h(u)\big).$$

Case 2: $(u_0, v_0) \in D$. By definition, $v_0 = \varepsilon$. Thus

$$h(v) \;=\; h(x)h(y) \;\subseteq\; \xrightarrow[D]{} \big(h(x)u_0h(y)\big) \;\subseteq\; \xrightarrow[S]{*} \big(h(xu_0y)\big) \;=\; \xrightarrow[S]{*} \big(h(u)\big).$$

Case 3: $(u_0, v_0) \in R_M$. This rule can be of two types, corresponding to the two subcases below.

Case 3.1: $u_0 \in MN^*$ and $v_0 \in M(N - M)^*$. We have $h(x)P^*u_0h(y) \subseteq h(u)$ and $h(v) = h(x)P^*v_0h(y)$. As $v_0\overrightarrow{u_0} \in P$, $h(x)P^*v_0\overrightarrow{u_0}u_0h(y) \subseteq h(u)$. Hence

$$h(v) \;=\; h(x)P^*v_0h(y) \;\subseteq\; \xrightarrow[D]{*}(h(x)P^*v_0\overrightarrow{u_0}u_0h(y)) \;\subseteq\; \xrightarrow[S]{*}(h(u)).$$

Case 3.2: $u_0 \in N^*MN^*$ and $v_0 \in (N-M)^*$. We have $u_0 = u_0'\#u_0''$ with $u_0' \in (N-M)^*$, $\# \in M$, $u_0'' \in N^*$, and $\overleftarrow{u_0'}v_0\overrightarrow{\#u_0''} \in P$. Hence $h(x)u_0'P^*\#u_0''h(y) \subseteq h(u)$, which implies in particular that

$$h(x)u_0'\overleftarrow{u_0'}v_0\overrightarrow{u_0''}\overrightarrow{\#}\#u_0''h(y) \ \subseteq \ h(u).$$

Finally we obtain that

$$h(v) = h(x)v_0h(y) \ \subseteq \ \xrightarrow[D]{*}\big(h(x)u_0'\overleftarrow{u_0'}v_0\overrightarrow{u_0''}\overrightarrow{\#}\#u_0''h(y)\big) \ \subseteq \ \xrightarrow[S]{*}(h(u)).$$

This concludes the proof that $h(v) \subseteq \longrightarrow_S^*(h(u))$ for $u \longrightarrow_{R\,\cup\,D}^* v$. As $v \in h(v)$, we finally get $\longrightarrow_{R\,\cup\,D}^*(u) \subseteq \longrightarrow_S^*(h(u))$. □

A similar decomposition can also be achieved using suffix sub-alphabets instead of prefix ones. A subset M of N is a *suffix sub-alphabet* of R if

$$R \ \subseteq \ N^*M{\times}(N-M)^*M \ \cup \ N^*{\times}(N-M)^*.$$

It can also be seen as a prefix sub-alphabet of $\widetilde{R} = \{(\widetilde{u},\widetilde{v}) \mid u\,R\,v\ \}$, where $\widetilde{u} = u(|u|)...u(1)$ is the *mirror* of word u. Lemma 2.6 remains true for any suffix sub-alphabet M, with the difference that $h_M(x) = xQ^*$ for all $x \in M$ with $Q = \{\overleftarrow{u}\,w\,\overrightarrow{v} \mid uv\,R\,w \wedge u \in N^*M \wedge v \in (N-M)^*\}$.

Using prefix and suffix sub-alphabets, we can now iterate this decomposition process as long as at least one such sub-alphabet remains. We say that $R \subseteq N^*{\times}N^*$ is *u-decomposable* for $u \in (2^N)^*$ if $u = \varepsilon$, or $u = Mv$ with M a prefix or suffix sub-alphabet of R and $R - R_M$ is v-decomposable. For any $u \in (2^N)^*$, we define the sub-system R_u of R as $R_u = R_{u(1)} \cup \ldots \cup R_{u(|u|)} = R_{u(1)\cup\ldots\cup u(|u|)}$ consisting of the subset of rules of R with at least one letter in $u(1) \cup \ldots \cup u(|u|)$.

When the letters of u are prefix and suffix sub-alphabets, we define the substitution $h_u : \overline{N} \longrightarrow 2^{\overline{N}^*}$ by $h_u = h_{u(1)} \circ \ \ldots \ \circ h_{u(|u|)}$ where for every $1 \leq i \leq |u|$, $h_{u(i)}$ is the substitution associated to the prefix, or suffix but not prefix, sub-alphabet $u(i)$ of R. Note that if R_u is recognizable, h_u is a regular substitution. Let us now iterate the decomposition of Lemma 2.6.

Proposition 2.7. *If R is u-decomposable then*

$$\xrightarrow[R]{*} \ = \ \Big(h_u \ \circ \ \xrightarrow[(R-R_u)\,\cup\,D]{*}\Big) \cap N^*{\times}N^*.$$

Proof. We have

$$\begin{aligned}
\xrightarrow[R]{*} \ &= \ \xrightarrow[R\,\cup\,D]{*} \cap N^*{\times}N^* \\
&= \ \Big(h_{u(1)} \ \circ \ \xrightarrow[(R-R_{u(1)})\,\cup\,D]{*}\Big) \ \cap \ N^*{\times}N^* \text{ by Lemma 2.6} \\
&= \ \Big(h_{u(1)} \ \circ \ \ldots \ \circ \ h_{u(|u|)} \ \circ \ \xrightarrow[(R-R_{u(1)}\ldots-R_{u(|u|)})\,\cup\,D]{*}\Big) \ \cap \ N^*{\times}N^* \\
&= \ \Big(h_u \ \circ \ \xrightarrow[(R-R_u)\,\cup\,D]{*}\Big) \ \cap \ N^*{\times}N^*.
\end{aligned}$$

□

We say that R is *decomposable* into S if R is u-decomposable for some u and $R - R_u = S$. This decomposition relation is reflexive and transitive. Let us illustrate this mechanism on an example.

Example 2.8. Consider the rewriting system $R = \{(abb, ab), (a, \varepsilon), (cb, cc)\}$ and the derivation $caabb \longrightarrow_R caab \longrightarrow_R cab \longrightarrow_R cb \longrightarrow_R cc$.

As $\{a\}$ is a prefix sub-alphabet of R, and by Lemma 2.6, we have

$$\xrightarrow[R]{*} \;=\; \Big(h_a \circ \xrightarrow[R' \cup D]{*}\Big) \cap N^* \times N^*$$

with $R' = \{(cb, cc)\}$ and $h_a(a) = \{\overrightarrow{a}, ab\overrightarrow{b}\,\overrightarrow{b}\,\overrightarrow{a}\}^*a$.
As $\{b\}$ is a prefix sub-alphabet of R' (but not of R), we have

$$\xrightarrow[R' \cup D]{*} \cap\, \overline{N}^* \times N^* \;=\; \Big(h_b \circ \xrightarrow[D]{*}\Big) \cap\, \overline{N}^* \times N^*$$

with $h_b(b) = \{\overleftarrow{c}\,cc\,\overrightarrow{b}\}^*b$. Thus R is $\{a\}\{b\}$-decomposable into $\emptyset$ and

$$\xrightarrow[R]{*} \;=\; \Big(h_{ab} \circ \xrightarrow[D]{*}\Big) \cap N^* \times N^*$$

with $h_{ab}(a) = h_b(h_a(a)) = \{\overrightarrow{a}, a\{\overleftarrow{c}\,cc\,\overrightarrow{b}\}^*b\,\overrightarrow{b}\,\overrightarrow{b}\,\overrightarrow{a}\}^*a$ and $h_{ab}(b) = h_b(h_a(b)) = h_b(b) = \{\overleftarrow{c}\,cc\,\overrightarrow{b}\}^*b$. For instance

$$u \;=\; c.\overrightarrow{a}\,a.\overrightarrow{a}\,a\,\overleftarrow{c}\,cc\,\overrightarrow{b}\,b\,\overrightarrow{b}\,\overrightarrow{b}\,\overrightarrow{a}\,a.b.b \in h_{ab}(caabb) \quad \text{and} \quad u \xrightarrow[D]{*} cc.$$

Finally R is terminating (no infinite derivation) although R is not match-bounded [11]. □

For any letter $a \in N - \mathrm{Im}(R)$ which does not appear in the right hand sides of rules of R, $\{a\}$ is a prefix (or suffix) sub-alphabet of R and we call a a *reducible letter* of R. We say that R is *reducible* into S if R is *u-decomposable* into S for some word u composed only of reducible letters (of the successive remaining sub-relations). Note that this is *not* the case of the rewriting system in the above example, even though it is decomposable into the empty system. The systems which can be reduced into $\emptyset$ or $\{(\varepsilon, \varepsilon)\}$ are exactly the deleting systems.

Proposition 2.9. *R is deleting if and only if R is reducible into $\emptyset$ or $\{(\varepsilon, \varepsilon)\}$.*

When R is decomposable into S and $R - S$ is recognizable, we say there is a *recognizable decomposition* of R into S. Let us apply Proposition 2.7 to that setting.

Theorem 2.10. *For R recognizable decomposable into S,*

$$\begin{array}{llcll} \xrightarrow[S \cup D]{*} & \textit{preserves regularity} & \Longrightarrow & \xrightarrow[R]{*} & \textit{preserves regularity,} \\ \xrightarrow[S^{-1} \cup D^{-1}]{*} & \textit{preserves context-freeness} & \Longrightarrow & \xrightarrow[R^{-1}]{*} & \textit{preserves context-freeness.} \end{array}$$

Proof. By Proposition 2.7, there is a regular substitution h such that for all $L \subseteq N^*$, $\longrightarrow^*_R(L) = \longrightarrow^*_{S\,\cup\,D}(h(L)) \cap N^*$ and $\longrightarrow^*_{R^{-1}}(L) = h^{-1}\big(\longrightarrow^*_{S^{-1}\,\cup\,D^{-1}}(L)\big) \cap N^*$. Since h is a regular substitution, this proves the theorem. □

Note that we cannot suppress D or D^{-1} in Theorem 2.10, and that the reverse implications are false. Indeed the system $S = \{(\#a, bb\#), (b\&, \&a)\}$ has a rational derivation, hence its derivation preserves regularity and context-freeness, but $\longrightarrow^*_{S\cup D}$ does not preserve regularity:

$$\xrightarrow[S\,\cup\,D]{*}\big((\#\overrightarrow{\&})^*\#a(\overleftarrow{\#}\&)^*\big) \cap \#a^* \;=\; \{\#a^{2^n} \mid n \geq 0\}$$

and $\longrightarrow^*_{S\cup D^{-1}}$ does not preserve context-freeness:

$$\xrightarrow[S\,\cup\,D^{-1}]{*}(a) \cap (\overrightarrow{\#}\&)^*a^*(\overleftarrow{\&}\#)^* \;=\; \{(\overrightarrow{\#}\&)^n a^{2^n}(\overleftarrow{\&}\#)^n \mid n \geq 0\}.$$

To conclude this section, let us apply Theorem 2.10 together with Proposition 2.3 to transfer regularity and context-freeness preservation properties from context-free systems to a larger class of rewriting systems.

Proposition 2.11. *If R^{-1} is recognizable decomposable into S^{-1} where S is a context-free system, then R is cf/reg-preserving.*

By Lemma 2.9, this proposition strictly generalizes Proposition 2.4 by allowing a recognizable set of rules: indeed any recognizable deleting system is recognizable-decomposable into the empty rewriting system (or $\{(\varepsilon, \varepsilon)\}$), whose inverse is trivially a context-free system. This entails that recognizable context-limited systems are cf/reg-preserving. In the following section we give several other applications of Theorem 2.10 and Proposition 2.11.

3 Applications

In this section, we provide several consequences and applications of the decomposition technique presented in the previous section. In particular, we show how to derive from Theorem 2.10 preservation properties for the classes of prefix, suffix and bifix systems as well as their tag-adding variants.

3.1 Prefix, Suffix and Bifix Systems

Proposition 2.4 was already applied in [13] to the prefix derivations of finite systems. Using Theorem 2.10, this can be extended to any recognizable system.

The *prefix rewriting* of a system R is the binary relation $\longmapsto_R \;=\; R.N^* \;=\; \longrightarrow_{R,0}$, *i.e.* $uy \longmapsto_R vy$ for any uRv and $y \in N^*$. As expected, the *prefix derivation* $\longmapsto^*_R$ of R is the reflexive and transitive closure of the prefix rewriting relation. For any finite system, the regularity of the set of words reached by prefix derivation from a given word [8] is a particular case of the rationality of the prefix derivation; this remains true for any recognizable system.

Proposition 3.1 **([9]).** *The prefix derivation of any recognizable system is a rational relation.*

Proof. Let $R = \bigcup_{i=1}^{n} U_i \times V_i$ be a recognizable rewriting system, and $\# \notin N$ be a new symbol, and consider the system $\#R = \{(\#u, \#v) \mid u\,R\,v\}$. By definition, $\{\#\}$ is a prefix sub-alphabet of $\#R$, which is thus recognizable and $\#$-decomposable into $\emptyset$. For any $L \subseteq N^*$, $\longmapsto_R^*(L) = \#^{-1}\big(\longrightarrow_{\#R}^*(\#L)\big)$ which by Theorem 2.10, is regular whenever L is regular. By Proposition 2.7, the last equality is equivalent to

$$\overset{*}{\underset{R}{\longmapsto}}(L) \;=\; \overset{*}{\underset{N\downarrow}{\longrightarrow}}\big(\{v\overrightarrow{u} \mid u\,R\,v\}^*L\big) \cap N^*.$$

Since $\longmapsto_{R^{-1}}^*(U)$ and $\longmapsto_R^*(V)$ remain regular for any regular languages U, V, the relation $\overline{R} = \bigcup_{i=1}^{n} \longmapsto_{R^{-1}}^*(U_i) \times \longmapsto_R^*(V_i)$ is recognizable, hence

$$\overset{*}{\underset{R}{\longmapsto}} \;=\; \mathrm{Id}_{N^*} \cup \underset{\overline{R}}{\longmapsto} \;=\; \mathrm{Id}_{N^*} \cup \overline{R}.N^*$$

is recognized by a finite transducer. □

The rules of a system R can also be applied only to suffixes. The *suffix rewriting* of R is the binary relation $\longrightarrow\!\!\!|_R \;=\; N^*.R$, *i.e.* $wu \longrightarrow\!\!\!|_R wv$ for any $u\,R\,v$ and $w \in N^*$. Note that the rewriting relation of a suffix system is isomorphic to that of a prefix one: $u \longrightarrow\!\!\!|_R v$ if and only if $\widetilde{u} \longmapsto_{\widetilde{R}} \widetilde{v}$ where $\widetilde{R} \;=\; \{(\widetilde{u}, \widetilde{v}) \mid u\,R\,v\}$. Hence Proposition 3.1 holds for suffix systems as well.

Finally we allow the application of rules both to prefixes and suffixes. The *bifix rewriting relation* of R is $\longmapsto\!\!\!|_R \;=\; \longmapsto_R \cup \longrightarrow\!\!\!|_R$. There exists a generalization of Proposition 3.1 to this type of rewriting.

Proposition 3.2 **([15]).** *The bifix derivation of any recognizable system is a rational relation.*

Proof. We take two new symbols $\#, \& \notin N$ and we define the recognizable system $S = \#R \cup R\& = \{(\#u, \#v) \mid u\,R\,v\} \cup \{(u\&, v\&) \mid u\,R\,v\}$. The sets $\{\#\}$ and $\{\&\}$ are respectively prefix and suffix sub-aphabets of S. Thus S is $\#\&$-decomposable in $\emptyset$. For any $L \subseteq N^*$, $\longmapsto\!\!\!|_R^*(L) \;=\; \#^{-1}\big(\longrightarrow_S^*(\#L\&)\big)\&^{-1}$ which by Theorem 2.10, is regular whenever L is regular. By Proposition 2.7, the last equality is equivalent to

$$\overset{*}{\underset{R}{\longmapsto\!\!\!|}}(L) \;=\; \overset{*}{\underset{D}{\longrightarrow}}\big(\{v\overrightarrow{u} \mid u\,R\,v\}^*L\{\overleftarrow{u}v \mid u\,R\,v\}^*\big) \cap N^*.$$

This is a possible first step of the construction, given in [15], of a finite transducer recognizing $\longmapsto\!\!\!|_R^*$. □

3.2 Left-to-Right Derivation

Let us apply Theorem 2.10 to another restriction of the derivation. The *left-to-right derivation* $\hookrightarrow_R^*$ of a system R is defined by

$$u \overset{*}{\underset{R}{\hookrightarrow}} v \iff u_0 \underset{R,p_1}{\longrightarrow} u_1 \ldots \underset{R,p_n}{\longrightarrow} u_n \;\text{ with }\; p_1 \leq \ldots \leq p_n, u_0 = u \text{ and } u_n = v.$$

The left-to-right derivation and leftmost derivation are incomparable. In particular, applying a rewrite rule at some position i could in a leftmost derivation enable another rule at some position strictly smaller than i. However, in a left-to-right derivation, successive rewriting positions must be just increasing.

Proposition 3.3. *The left-to-right derivation of any recognizable system preserves context-freeness, and its inverse preserves regularity.*

Proof. We consider a new symbol $\# \notin N$ and the system $S = \{(u, \#v) \mid u\,R\,v\}$. By choosing $\{\#\}$ as a prefix sub-alphabet, we can recognizably decompose S^{-1} into $\emptyset$. Note that S^{-1} is deleting for R finite. By Proposition 2.11, $\longrightarrow^*_S$ thus preserves context-freeness and $\longrightarrow^*_{S^{-1}}$ preserves regularity. Furthermore $\longrightarrow^*_S$ can be performed from left to right: $\longrightarrow^*_S \;=\; \hookrightarrow^*_S$. Let π be the morphism defined by $\pi(\#) = \varepsilon$ and $\pi(a) = a$ for any $a \in N$. We have

$$\overset{*}{\underset{R}{\hookrightarrow}} \;=\; \{(u, \pi(v)) \mid u \in N^* \wedge u \overset{*}{\underset{S}{\hookrightarrow}} v\}.$$

Thus for every $L \subseteq N^*$,

$$\overset{*}{\underset{R}{\hookrightarrow}}(L) \;=\; \pi\big(\overset{*}{\underset{S}{\hookrightarrow}}(L)\big) \;=\; \pi\big(\overset{*}{\underset{S}{\longrightarrow}}(L)\big)$$

hence $\hookrightarrow^*_R(L)$ is context-free whenever L is. Finally for any $L \subseteq N^*$,

$$\big(\overset{*}{\underset{R}{\hookrightarrow}}\big)^{-1}(L) \;=\; \overset{*}{\underset{S^{-1}}{\longrightarrow}}\big(\pi^{-1}(L)\big) \cap N^*$$

which is regular whenever L is regular. □

Note that Proposition 2.3, when restricted to recognizable rewriting systems, is a corollary of Proposition 3.3. Indeed for any $R \subseteq (N \cup \{\varepsilon\}) \times N^*$, the derivation $\longrightarrow^*_R$ is equal to $\hookrightarrow^*_R$. Also note that inverse preservation properties do not hold in general. For instance when $R = \{(a, bab)\}$, we have $\hookrightarrow^*_R(a) = \{b^n a b^n \mid n \geq 0\}$ hence $\hookrightarrow^*_R$ does not preserve regularity. Conversely $\longrightarrow^*_{N{\downarrow}^{-1}} \;=\; \hookrightarrow^*_{N{\downarrow}^{-1}}$ hence $\big(\hookrightarrow^*_{N{\downarrow}^{-1}}\big)^{-1} = \longrightarrow^*_{N{\downarrow}}$ which does not preserve context-freeness.

We conclude this section on left-to-right derivation by showing that the left-to-right derivation of any rewriting system can be described using prefix derivation. To any $R \subseteq N^* \times N^*$, we associate the labelled transition system (*i.e.* labelled graph) $\widehat{R}$ over $N \cup \{\varepsilon\}$ defined as

$$\widehat{R} \;=\; \{u \xrightarrow{\varepsilon} v \mid u\,R\,v\} \cup \{a \xrightarrow{a} \varepsilon \mid a \in N\}$$

and its *prefix transition graph*

$$\widehat{R}.N^* \;=\; \{uw \xrightarrow{\varepsilon} vw \mid u\,R\,v \wedge w \in N^*\} \cup \{aw \xrightarrow{a} w \mid a \in N \wedge w \in N^*\}.$$

The words obtained by left-to-right derivation by R from a word u are precisely the words v labelling paths $u \overset{v}{\underset{\widehat{R}.N^*}{\Longrightarrow}} \varepsilon$ (in other words *recognized by* $\widehat{R}.N^*$) from vertex u to vertex ε.

Lemma 3.4. *For any system R, $u \hookrightarrow_R^* v \iff u \Longrightarrow^{v}_{\widehat{R}.N^*} \varepsilon$.*

By Proposition 3.3 and Lemma 3.4, we get $\hookrightarrow_R^*(L) = L(\widehat{R}.N^*, L, \varepsilon)$, the language of words labelling paths in the graph $\widehat{R}.N^*$ between vertices in L and the vertex ε. This is a context-free language whenever L is context-free and R is recognizable [9]. When R is finite and taking a new symbol p representing a *control state*, the system $p\widehat{R} = \{pu \xrightarrow{\varepsilon} pv \mid u\,R\,v\} \cup \{pa \xrightarrow{a} p \mid a \in N\}$ can be seen as a pushdown automaton with stack alphabet N (as customary, pushdown rules can be straightforwardly obtained by adding new states and rules). We have just described the effective construction of a pushdown automaton recognizing the language $\hookrightarrow_R^*(L)$ by empty stack and with possible initial stack content each word in L.

3.3 Tagged and Tag-Adding Systems

We will now apply Theorem 2.10 to the derivation of systems generalizing bifix derivation. We consider a finite set M of special symbols called tags. Bifix systems can be easily simulated and extended by adding tags at the first and last position of the sides of the rules, enforcing that the first and/or last tags of each side of a rule must be the same [1].

Definition 3.5. *Given disjoint sets M and N, a tagged bifix system over $M \cup N$ is a system*

$$R \;\subseteq\; (M \cup \{\varepsilon\})N^*(M \cup \{\varepsilon\}) \times (M \cup N)^*$$

such that for any rule $(u, v) \in R$, $(u(1) \in M \vee u(|u|) \in M)$ and

$$u(1) \in M \implies u(1) = v(1) \quad \textit{and} \quad u(|u|) \in M \implies u(|u|) = v(|v|).$$

In such a system all tags are preserved by the rewriting process. Without this condition, we could transform any finite system R over N into the tagged system

$$\begin{aligned} R_\bullet \;=\; & \{(\bullet u, \bullet v) \mid u\,R\,v\} \cup \{(\bullet a, \#_a) \mid a \in N\} \cup \{(\&_a, \bullet a) \mid a \in N\} \\ & \cup \{(\#_a, a\bullet) \mid a \in N\} \cup \{(a\bullet, \&_a) \mid a \in N\} \end{aligned}$$

with $M = \{\#_a \mid a \in N\} \cup \{\&_a \mid a \in N\} \cup \{\bullet\}$. We have $u \longrightarrow_R^* v \iff \bullet u \longrightarrow_{R_\bullet}^* \bullet v$ for any $u, v \in N^*$. Since $\longrightarrow_R^*$ is not recursive in general, neither is $\longrightarrow_{R_\bullet}^*$. The rationality of bifix derivation can be extended to the derivation of tagged bifix systems.

Proposition 3.6 ([1]). *The derivation relation of any recognizable tagged bifix system is rational.*

Before further extending this class of systems by allowing tag-adding and infix rules, we consider systems whose tag set M is partitioned into a subset M_p of *prefix tags* and a subset M_s of *suffix tags*: $M_p \cup M_s = M$ and $M_p \cap M_s = \emptyset$.

Definition 3.7. *A tag-adding prefix/suffix system is a system*

$$R \ \subseteq \ (M_p \cup \{\varepsilon\})N^*(M_s \cup \{\varepsilon\}) \times (M \cup N)^*$$

such that for any rule $(u,v) \in R$,

$$\big(u(1) \in M_p \implies v(1) \in M_p\big) \quad \textit{and} \quad \big(u(|u|) \in M_s \implies v(|v|) \in M_s\big).$$

Considering tags $\#, \& \in M$ and a letter $a \in N$, the two-rule system $R = \{(\#a, \&), (\&, a\#)\}$ cannot be a tag-adding prefix/suffix system since the tag $\#$ would be simultaneaously prefix and suffix. For this system, neither the direct nor the inverse derivation preserve regularity:

$$\xrightarrow[R]{*}\big((\#a)^*\big) \cap a^*\#^* = \{a^n\#^n \mid n \geq 0\} \text{ and } \xrightarrow[R^{-1}]{*}\big((\#a)^*\big) \cap \#^*a^* = \{\#^n a^n \mid n \geq 0\}.$$

We say that a tag-adding prefix/suffix system R is *context-free* if $R \cap N^* \times N^*$ is a context-free system and $R - N^* \times N^*$ is recognizable. Let us apply Theorem 2.10.

Proposition 3.8. *Any context-free tag-adding prefix/suffix system is cf/reg-preserving.*

Proof. Let R be a context-free tag-adding prefix/suffix system. The sets M_p and M_s are respectively prefix and suffix sub-alphabets of R^{-1}. Thus R^{-1} is M_pM_s-decomposable into $(R \cap N^* \times N^*)^{-1}$ and by Proposition 2.11, R is cf/reg-preserving. □

A particular case of a tag-adding prefix system ($M_s = \emptyset$) is given by a recognizable system $R \subseteq MN^* \times (MN^*)^+$ which is cf/reg-preserving by Proposition 3.8, and also by Proposition 3.3: the derivation is equal to its left-to-right derivation. These particular systems generalize the first model of dynamic networks of pushdown systems [7].

We will now use Proposition 3.8 to obtain the same closure properties for the following extension of tagged bifix systems.

Definition 3.9. *A tag-adding bifix system is a system*

$$R \ \subseteq \ (M \cup \{\varepsilon\})N^*(M \cup \{\varepsilon\}) \times (M \cup N)^*$$

such that for any rule $(u,v) \in R$,

$$\big(u(1) \in M \implies u(1) = v(1)\big) \quad \textit{and} \quad \big(u(|u|) \in M \implies u(|u|) = v(|v|)\big).$$

Note that the previous system $\{(\#a, \&), (\&, a\#)\}$ is not tag-adding bifix. Proposition 3.8 remains valid for such systems when they are *context-free*, *i.e.* when $R \cap N^* \times N^*$ is a context-free system and $R - N^* \times N^*$ is recognizable.

Theorem 3.10. *Any context-free tag-adding bifix system is cf/reg-preserving.*

Note that Proposition 3.8 and Theorem 3.10 can both be generalized to the systems R such that $R - N^* \times N^*$ is recognizable and $(R \cap N^* \times N^*) \cup D^{-1}$ is cf/reg-preserving.

Theorem 3.10 generalizes Theorem 7 of [1]: for any 'tagged infix system with tag removing rules' R, its inverse R^{-1} is a particular tag-adding bifix system hence $\longrightarrow_R^*$ preserves regularity. As a corollary, Theorem 3.10 also positively answers a conjecture stated in [2] (page 99).

4 Conclusion

In this paper we presented a decomposition mechanism for word rewriting systems, allowing us to transfer the simultaneous regularity and inverse context-freeness preservation of the Dyck system to several classes of rewriting systems.

We are currently investigating more general criteria to widen the scope of this result and to extend this decomposition to terms.

Many thanks to Antoine Meyer for helping us make this paper readable, and to anonymous referees for helpful comments.

References

1. Altenbernd, J.: On bifix systems and generalizations. In: Martín-Vide, C., Otto, F., Fernau, H. (eds.) LATA 2008. LNCS, vol. 5196, pp. 40–51. Springer, Heidelberg (2008)
2. Altenbernd, J.: Reachability over word rewriting systems. Ph.D. Thesis, RWTH Aachen, Germany (2009)
3. Benois, M.: Parties rationnelles du groupe libre. C.R. Académie des Sciences, Série A 269, 1188–1190 (1969)
4. Berstel, J.: Transductions and context-free languages. Teubner, Stuttgart (1979)
5. Book, R., Jantzen, M., Wrathall, C.: Monadic thue systems. Theoretical Computer Science 19, 231–251 (1982)
6. Book, R., Otto, F.: String-rewriting systems. Texts and Monographs in Computer Science. Springer, Heidelberg (1993)
7. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 473–487. Springer, Heidelberg (2005)
8. Büchi, R.: Regular canonical systems. Archiv für Mathematische Logik und Grundlagenforschung 6, 91–111 (1964)
9. Caucal, D.: On the regular structure of prefix rewriting. Theoretical Computer Science 106, 61–86 (1992); originally published In: Arnold, A. (ed.) CAAP 1990. LNCS, vol. 431, pp. 61–86. Springer, Heidelberg (1990)
10. Endrullis, J., Hofbauer, D., Waldmann, J.: Decomposing terminating rewrite relations. In: Geser, A., Sondergaard, H. (eds.) Proc. 8^{th} WST, pp. 39–43 (2006), http://www.acm.org/corr/, Computing Research Repository
11. Geser, A., Hofbauer, D., Waldmann, J.: Match-bounded string rewriting systems. Applicable Algebra in Engineering, Communication and Computing 15, 149–171 (2004)
12. Hibbard, T.: Context-limited grammars. JACM 21(3), 446–453 (1974)
13. Hofbauer, D., Waldmann, J.: Deleting string rewriting systems preserve regularity. Theoretical Computer Science 327, 301–317 (2004); originally published In: Ésik, Z., Fülöp, Z. (eds.) DLT 2003. LNCS, vol. 2710, pp. 301–317. Springer, Heidelberg (2003)
14. Jantzen, M., Kudlek, M., Lange, K.J., Petersen, H.: Dyck_1-reductions of context-free languages. In: Budach, L., Bakharajev, R., Lipanov, O. (eds.) FCT 1987. LNCS, vol. 278, pp. 218–227. Springer, Heidelberg (1987)
15. Karhumäki, J., Kunc, M., Okhotin, A.: Computing by commuting. Theoretical Computer Science 356, 200–211 (2006)

Quantitative Robustness Analysis of Flat Timed Automata[*]

Rémi Jaubert[**] and Pierre-Alain Reynier

LIF, Université Aix-Marseille & CNRS, France
{remi.jaubert,pierre-alain.reynier}@lif.univ-mrs.fr

Abstract. Whereas formal verification of timed systems has become a very active field of research, the idealized mathematical semantics of timed automata cannot be faithfully implemented. Recently, several works have studied a parametric semantics of timed automata related to implementability: if the specification is met for some positive value of the parameter, then there exists a correct implementation. In addition, the value of the parameter gives lower bounds on sufficient resources for the implementation. In this work, we present a symbolic algorithm for the computation of the parametric reachability set under this semantics for flat timed automata. As a consequence, we can compute the largest value of the parameter for a timed automaton to be safe.

1 Introduction

Verification of real-time systems. In the last thirty years, formal verification of reactive, critical, or embedded systems has become a very active field of research in computer science. It aims at checking that (the model of) a system satisfies (a formula expressing) its specifications. The importance of taking real-time constraints into account in verification has quickly been understood, and the model of timed automata [AD94] has become one of the most established models for real-time systems, with a well studied underlying theory, the development of mature model-checking tools (UPPAAL [BDL04], KRONOS [BDM$^+$98], ...), and numerous success stories.

Implementation of real-time systems. Implementing mathematical models on physical machines is an important step for applying theoretical results on practical examples. This step is well-understood for many untimed models that have been studied (*e.g.*, finite automata, pushdown automata). In the timed setting, while timed automata are widely-accepted as a framework for modelling real-time aspects of systems, it is known that they cannot be faithfully implemented on finite-speed CPUs [CHR02]. Studying the "implementability" of timed automata is thus a challenging issue of obvious theoretical and practical interest.

A semantical approach. Timed automata are governed by a mathematical, idealized semantics, which does not fit with the digital, imprecise nature of the hardware on

[*] Partly supported by the French projects ECSPER (ANR-09-JCJC-0069) and DOTS (ANR-06-SETI-003), and by the European project QUASIMODO (FP7-ICT-STREP-214755).

[**] Funded by a doctoral grant of "Conseil Régional Provence-Alpes-Côte d'Azur".

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 229–244, 2011.

which they will possibly be implemented. An *implementation semantics* has been defined in [DDR05] in order to take the hardware into account: that semantics models a digital CPU which, every δ_P time units (at most), reads the value of the digital clock (updated every δ_L time units), computes the values of the guards, and fires one of the available transitions. A timed automaton is then said to be *implementable* if there exist positive values for those parameters (δ_P and δ_L) for which, under this new semantics, the behaviours of the automaton satisfy its specification. In order to study it efficiently, this semantics is over-approximated by the *AASAP semantics*, which consists in "enlarging" the constraints on the clocks by some parameter δ. For instance, "$x \in [a, b]$" is transformed into "$x \in [a - \delta, b + \delta]$". Moreover, a formal link is drawn in [DDR05] between these two semantics: as soon as $\delta > 4\delta_P + 3\delta_L$, the AASAP semantics simulates the semantics of the implementation. As a consequence, implementability can be ensured by establishing the existence of some positive δ for which the AASAP semantics meets the specification.

Robustness problems. We call the above problem (existence of some positive δ) the *qualitative problem of robustness*. This problem was proven decidable for different kind of properties: the problem is PSPACE-complete for safety properties [Pur00, DDMR08] and LTL formula [BMR06]. It is EXPTIME-complete for a fragment of the timed logic MTL [BMR08]. In addition, for safety properties, it is proven in [Pur00, DDMR08] that if there exists a safe positive value of δ, then the system is also safe for a specific value of the form $1/2^{|\mathcal{A}|}$. While this allows to deduce a correct value for the parameter δ, computing the largest value of δ for which the AASAP semantics meets the specification was still an open problem. We are interested here in this last problem, which we call the *quantitative problem of robustness* for safety properties.

Our contributions. In this paper, we prove that the quantitative robustness problem for safety properties is decidable for flat timed automata (*i.e.* where each location belongs to at most one cycle). In addition, we show that the maximal safe value of δ is a rational number. To this end, we solve a more general problem: we prove that it is possible to compute, for a flat timed automaton, its reachability set parameterized by δ. We call it the parametric reachability set. For this computation, we present a forward algorithm based on parametric zones (recall that a zone is a constraint on clocks). As a parametric forward analysis does not terminate for (flat) timed automata, we need some acceleration techniques. To solve the qualitative robustness problem, different algorithms have been proposed in [Pur00, DDMR08, DK06] which compute an enlarged reachability set corresponding to states reachable for any positive perturbation, and include an acceleration of cycles. The algorithm we propose can be understood as a parametric version of the symbolic algorithm proposed in [DK06] for flat timed automata. We then tackle two issues: the termination of our procedure and its correctness. For the first aspect, as we are in a parametric setting, we need completely new arguments of termination (the number of parametric zones we compute cannot be bounded as it is the case for zones). Considering a graph representation of zones introduced in [CJ99a], we obtain proofs of termination depending only on the number of clocks, and not on the constants appearing in the automaton. To our knowledge, this constitutes an original approach in the context of timed automata. Regarding correctness, we identify under which conditions the enlarged reachability set coincides with the standard reachability set. This allows us to propose an algorithm computing the parametric reachability set of a flat timed automaton.

Related work. Since its definition in [Pur00, DDR05], the approach based on the AASAP semantics has received much attention, and other kind of perturbations, like the drift of clocks, have been studied [DDMR08, ALM05, Dim07]. In the case of safety properties and under some natural assumptions, this perturbation is equivalent to constraint enlargement and relies on similar techniques, as proven in [DDMR08]. Also, several works have considered variants of the robustness problem. In [SF07, SFK08], the case of systems with bounded life-time or regular resynchronization of clocks is considered, while in [Dim07], a symbolic algorithm is proposed to handle strict constraints.

Many other notions of "robustness" have been proposed in the literature in order to relax the mathematical idealization of the semantics of timed automata, see for instance [GHJ97, OW03, BBB$^+$07]. Those approaches are different from ours, since they roughly consist in dropping "isolated" or "unlikely" executions, and are thus more related to language theoretical issues than to implementability issues.

Finally, our work is somewhat related to parametric timed automata. It is proven in [WT99] that emptiness is already undecidable for timed automata with three clocks and one parameter. In our setting, decidability results follow from strong restrictions on the use of the parameter. They correspond to the notion of upper parameter introduced in [HRSV02], but the problems we consider are different. In addition, to obtain termination, we introduce acceleration techniques based on [CJ99a]. Two recent works [BIL06, BIK10] also rely on [CJ99a] to propose acceleration techniques, but these concern flat counter automata with an integer-valued parameter.

Organisation of the paper. In Section 2, we introduce standard definitions. We present in Section 3 the definition of the enlarged reachability set, and a modification of the algorithm of [DK06] for its computation. In Section 4, we first recall the graph representation of constraints, then present how we use it to obtain a new acceleration technique, and finally we present our parametric algorithm and its proof of termination and of correction.

2 Definitions

2.1 Timed Automata, Zones

Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ be a finite set of *clock* variables. We extend it with a fictive clock x_0, whose value will always be 0, and denote by $\overline{\mathcal{X}}$ the set $\mathcal{X} \cup \{x_0\}$. An *atomic (clock) constraint* on $\mathcal{X}$ is of the form $x - y \leq k$, where $x \neq y \in \overline{\mathcal{X}}$ and $k \in \mathbb{Q}$. Note that we only consider non-strict inequalities. This makes sense as we will later enlarge these constraints. We say that the constraint is *non-diagonal* if the comparison involves the clock x_0. We denote by $\mathcal{G}(\mathcal{X})$ (resp. $\mathcal{G}_{nd}(\mathcal{X})$) the set of *(clock) constraints* (resp. *non-diagonal constraints*) defined as conjunctions of atomic constraints (resp. non-diagonal atomic constraints).

A *(clock) valuation* v for $\mathcal{X}$ is an element of $\mathbb{R}_{\geq 0}^{\mathcal{X}}$. A valuation $v \in \mathbb{R}_{\geq 0}^{\mathcal{X}}$ is extended to $\mathbb{R}_{\geq 0}^{\overline{\mathcal{X}}}$ by $v(x_0) = 0$. If $v \in \mathbb{R}_{\geq 0}^{\mathcal{X}}$ and $t \in \mathbb{R}_{\geq 0}$, we write $v + t$ for the valuation assigning $v(x) + t$ to every clock $x \in \mathcal{X}$. If $r \subseteq \mathcal{X}$, $v[r \leftarrow 0]$ denotes the valuation assigning 0 to every clock in r and $v(x)$ to every clock in $\mathcal{X} \setminus r$. Whether a valuation $v \in \mathbb{R}_{\geq 0}^{\mathcal{X}}$ satisfies a constraint $g \in \mathcal{G}(\mathcal{X})$, written $v \models g$, is defined inductively as follows: the conjunction is handled naturally, and $v \models x - y \leq k$ iff $v(x) - v(y) \leq k$ (recall that $v(x_0) = 0$). The set of valuations satisfying a constraint g is denoted $[\![g]\!]$.

A *zone* Z over $\mathcal{X}$ is a convex subset of $\mathbb{R}_{\geq 0}^{\mathcal{X}}$ which can be defined as the set of valuations satisfying a clock constraint, *i.e.* there exists $g \in \mathcal{G}(\mathcal{X})$ such that $Z = [\![g]\!]$. We note $\mathsf{Zones}(\mathcal{X})$ the set of zones on $\mathcal{X}$. The zone $\mathbb{R}_{\geq 0}^{\mathcal{X}}$ is denoted $\top$.

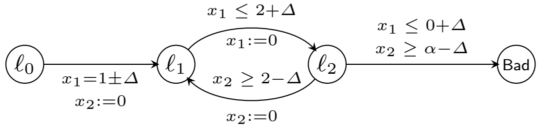
Definition 1 (Timed Automaton). *A* TA *is a tuple* $\mathcal{A} = (L, \ell_0, \mathcal{X}, \Sigma, T)$ *where* L *is a finite set of locations,* $\ell_0 \in L$ *is an initial location,* $\mathcal{X}$ *is a finite set of clocks,* Σ *is a finite set of actions, and* $T \subseteq L \times \mathcal{G}_{nd}(\mathcal{X}) \times \Sigma \times 2^{\mathcal{X}} \times L$ *is a finite set of transitions.*

We define the semantics of $\mathcal{A}$ as a timed transition system $[\![\mathcal{A}]\!] = \langle S, S_0, \Sigma, \rightarrow \rangle$. The set S of states of $[\![\mathcal{A}]\!]$ is $L \times \mathbb{R}_{\geq 0}^{\mathcal{X}}$ and $S_0 = \{(\ell_0, v_0) \mid v_0(x) = v_0(y), \forall x, y \in \mathcal{X}\}$. A transition in $[\![\mathcal{A}]\!]$ is composed either of a delay move $(\ell, v) \xrightarrow{d} (\ell, v + d)$, with $d \in \mathbb{R}_{>0}$, or of a discrete move $(\ell, v) \xrightarrow{\sigma} (\ell', v')$ when there exists a transition $(\ell, g, \sigma, r, \ell') \in T$ with $v \models g$, and $v' = v[r \leftarrow 0]$. The graph $[\![\mathcal{A}]\!]$ is thus an infinite transition system. A *run* of $[\![\mathcal{A}]\!]$ is a finite or infinite sequence $(\ell_0, v_0) \xrightarrow{\sigma_1} (\ell_1, v_1) \xrightarrow{d_1} (\ell_1, v_1 + d_1) \xrightarrow{\sigma_2} (\ell_2, v_2) \ldots$ where for each $i \geq 1$, $d_i \in \mathbb{R}_{\geq 0}$, and $(\ell_0, v_0) \in S_0$. A state (ℓ, v) is *reachable* in $[\![\mathcal{A}]\!]$ iff there exists a run from an initial state $(\ell_0, v_0) \in S_0$ to (ℓ, v); the set of reachable states is denoted $\mathsf{Reach}(\mathcal{A})$.

Note that standard definitions of timed automata also allow invariants on locations which restrict time elapsing. For the sake of simplicity, we do not consider this technical addition here, however all our results hold in presence of invariants.

A *cycle* of $\mathcal{A}$ is a finite sequence of transitions corresponding to a cycle of the underlying finite state automaton. We say that a timed automaton is *flat* if each location belongs to at most one cycle. A *progress cycle* is a cycle where each clock is reset at least once. We say $\mathcal{A}$ is *progressive* if it only contains progress cycles.

Assumptions. As our results rely on previous works on robustness in TA [Pur00, DDMR08], we assume that our TA are progressive, and that all the clocks are always bounded by some constant M. In addition, as the algorithm we propose is based on [DK06], we also require our timed automata to be flat.



with $\alpha{=}2$, $\mathcal{A}(\delta)$ avoids Bad iff $\delta{\leq}0$.

with $\alpha{=}3$, $\mathcal{A}(\delta)$ avoids Bad iff $\delta{<}1/3$.

Fig. 1. A timed automaton $\mathcal{A}$, with its parametric semantics

2.2 Parametric Objects

We define the parametric semantics introduced in [Pur00] that enlarges the set of runs of timed automata. This semantics can be defined in terms of timed automata extended with one parameter, denoted Δ, with syntactic constraints on the use of this parameter.

We denote by $\mathcal{PG}(\mathcal{X})$ the set of *parametric (clock) constraints* generated by the grammar [1] $g ::= g \wedge g \mid x - y \leq k + b\Delta$, where $x \neq y \in \overline{\mathcal{X}}$, $k \in \mathbb{Q}$ and $b \in \mathbb{N}$ (Δ represents a delay and b represents the accumulation of this delay, see Example 1). Given a parametric constraint g and $\delta \in \mathbb{Q}_{\geq 0}$, we denote by $g(\delta)$ the constraint obtained by evaluating the parameter Δ in δ. As the parameter helps in "relaxing" the clock constraint, we have that $\delta \leq \delta'$ implies $[\![g(\delta)]\!] \subseteq [\![g(\delta')]\!]$.

Definition 2 (Parametric Zone). *A* parametric zone $\mathcal{Z}$ *over* $\mathcal{X}$ *is a partial mapping from* $\mathbb{Q}_{\geq 0}$ *to zones over* $\mathcal{X}$*, which satisfies the following properties:* (i) *its domain* $\mathsf{dom}(\mathcal{Z})$ *is an interval with rational bounds, and* (ii) *it can be defined as the parametric satisfiability set of a parametric clock constraint,* i.e. *there exists* $g \in \mathcal{PG}(\mathcal{X})$ *such that for all* $\delta \in \mathsf{dom}(\mathcal{Z}), \mathcal{Z}(\delta) = [\![g(\delta)]\!]$*. We denote by* $\mathsf{PZones}(\mathcal{X})$ *the set of parametric zones on* $\mathcal{X}$[2].

By default the considered domain for a parametric zone is $\mathbb{Q}_{\geq 0}$. Given a rational interval I, we denote $\mathcal{Z}_{|I}$ the parametric zone whose domain is restricted to I *i.e.*, $\mathsf{dom}(\mathcal{Z}_{|I}) = \mathsf{dom}(\mathcal{Z}) \cap I$, and which coincides with $\mathcal{Z}$ on $\mathsf{dom}(\mathcal{Z}_{|I})$. Given $\mathcal{Z}, \mathcal{Z}' \in \mathsf{PZones}(\mathcal{X})$, we define $\mathcal{Z} \subseteq \mathcal{Z}'$ if, and only if, we have $\mathsf{dom}(\mathcal{Z}) \subseteq \mathsf{dom}(\mathcal{Z}')$, and for any $\delta \in \mathsf{dom}(\mathcal{Z})$, $\mathcal{Z}(\delta) \subseteq \mathcal{Z}'(\delta)$. We say that a parametric zone $\mathcal{Z}$ is non-empty if there exists $\delta \in \mathsf{dom}(\mathcal{Z})$ such that $\mathcal{Z}(\delta) \neq \varnothing$. Let $\mathcal{Z}$ be a non-empty parametric zone. As the mapping represented by $\mathcal{Z}$ is monotone, we define $\delta_{\neg\varnothing}(\mathcal{Z}) = \inf\{\delta \geq 0 \mid \mathcal{Z}(\delta) \neq \varnothing\}$ the minimal value of the parameter for the zone it denotes to be nonempty. As $\mathcal{Z}$ only involves non-strict linear inequalities, $\delta_{\neg\varnothing}(\mathcal{Z})$ is a rational number and we have $\mathcal{Z}(\delta_{\neg\varnothing}(\mathcal{Z})) \neq \varnothing$ (provided that $\delta_{\neg\varnothing}(\mathcal{Z}) \in \mathsf{dom}(\mathcal{Z})$).

Definition 3 (Parametric Semantics [Pur00, DDMR08]). *Let* $\mathcal{A} = (L, \ell_0, \mathcal{X}, \Sigma, T)$ *be a TA. The* parametric semantics *of* $\mathcal{A}$ *consists in replacing each constraint* $g \in \mathcal{G}_{nd}(\mathcal{X})$ *appearing in some transition of* $\mathcal{A}$ *by the parametric constraint obtained by enlarging it with the parameter* Δ*. Formally, each atomic constraint of the form* $x - y \leq k$ *is replaced by the parametric constraint* $x - y \leq k + \Delta$*.*

Given $\delta \in \mathbb{Q}_{\geq 0}$, the instantiation of all constraints of $\mathcal{A}$ in δ leads to a timed automaton that we denote by $\mathcal{A}(\delta)$. The semantics used implies the following monotonicity property: $\delta \leq \delta' \Rightarrow \mathsf{Reach}(\mathcal{A}(\delta)) \subseteq \mathsf{Reach}(\mathcal{A}(\delta'))$. An example of timed automaton is shown in Figure 1.

2.3 Symbolic Computations Using (Parametric) Zones

A *symbolic state* is a pair $(\ell, Z) \in L \times \mathsf{Zones}(\mathcal{X})$. Consider a transition $t = (\ell, g, \sigma, r, \ell') \in T$ of a TA $\mathcal{A}$. We define the operator Post^t computing the symbolic successors over t starting from the zone Z, with $Z \in \mathsf{Zones}(\mathcal{X})$, by $\mathsf{Post}^t(Z) = \{v' \in \mathbb{R}_{\geq 0}^{\mathcal{X}} \mid \exists v \in Z, \exists d \in \mathbb{R}_{>0} : v \models g \wedge v' = v[r \leftarrow 0] + d\}$. It is well known that $\mathsf{Post}^t(Z)$ is still a zone. We define similarly the operator Pre^t for the set of predecessors by t. Given a sequence of transitions ϱ, we define the operators Post^ϱ and Pre^ϱ as the compositions of these operators for each transition of ϱ. We define the set of successors from a symbolic state by $\mathsf{Succ}(\ell, Z) = \{(\ell', Z') \in L \times \mathsf{Zones}(\mathcal{X}) \mid \exists t = (\ell, g, \sigma, r, \ell') \in T \text{ s.t. } Z' = \mathsf{Post}^t(Z)\}$.

[1] Compared with L/U TA introduced in [HRSV02], our parameter is "upper".

[2] In the sequel, Z and Y denote a zone, while $\mathcal{Z}$ and $\mathcal{Y}$ denote a parametric zone.

In order to perform parametric computations, we will use parametric zones. Our parametric constraints are less expressive[3] than those considered in [AAB00]. In particular, we can perform the operations of intersection, time elapsing, clock reset, inclusion checking... and extend operators Post^{ϱ} and Pre^{ϱ} to a parametric setting. We denote these extensions by PPost^{ϱ} and PPre^{ϱ}. We also define the operator $\mathsf{Succ}(\ell, \mathcal{Z})$, where $\mathcal{Z} \in \mathsf{PZones}(\mathcal{X})$, using the PPost operator.

3 The Enlarged Reachability Set $\mathsf{Reach}^*(\mathcal{A})$

Definition of $\mathsf{Reach}^*(\mathcal{A})$. We are interested here in the *quantitative problem of robustness* for safety properties: given a set of states Bad to be avoided, compute the maximal value of δ for the system to be safe, *i.e.* the value $\delta_{\max} = \sup\{\delta \geq 0 \mid \mathsf{Reach}(\mathcal{A}(\delta)) \cap \mathsf{Bad} = \varnothing\}$ (recall the monotonicity of $\mathsf{Reach}(\mathcal{A}(\delta))$ *w.r.t.* δ). Note that the value $\delta_{\max}$ may be safe or not (see Examples in [JR10]).

In this paper, we propose an algorithm that computes a representation of the parametric reachability set of a flat timed automaton. It is then easy to derive the optimal value $\delta_{\max}$. A forward parametric computation of reachable states does not terminate in general for timed automata. Indeed, the coefficient on parameter Δ (coefficient b in Definition 2) cannot always be bounded (see Example 1). Such a phenomenon is due to cycles: it can be the case that a state (ℓ, v) is reachable for any $\delta > 0$, but the length of paths allowing to reach (ℓ, v) in $\mathcal{A}(\delta)$ diverges when δ converges to 0.

Example 1. Consider the timed automaton represented on Figure 1. State (ℓ_2, v) with $v(x_1) = 0$ and $v(x_2) = 2$ is reachable in $[\![\mathcal{A}(\delta)]\!]$ for any $\delta > 0$. Let us denote by t_1 (resp. t_2) the transition from ℓ_1 to ℓ_2 (resp. from ℓ_2 to ℓ_1), and let $\varrho = t_1 t_2$. In $[\![\mathcal{A}(\delta)]\!]$, this state is reachable only after $\lceil \frac{1}{2\delta} \rceil$ iterations of the cycle ϱ (see Figure 2).

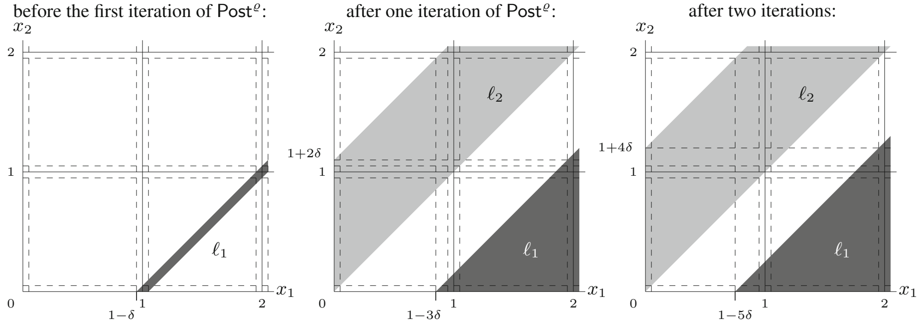


Fig. 2. Reachable states during the parametric forward analysis of $\mathcal{A}(\delta)$

[3] Note that in our setting, one can define a data structure more specific than parametric DBMs considered in [AAB00]. Indeed, we do not need to split DBMs as the constraints only involve conjunctions. Moreover, we can perform basic operations (future, reset, intersection with an atomic constraint) in quadratic time, as for DBMs, see [Jau09].

This difficulty has first been identified by Puri in [Pur00] when studying the qualitative robustness problem, and solved by computing the enlarged reachability set defined as $\mathsf{Reach}^*(\mathcal{A}) \overset{def}{=} \bigcap_{\delta \in \mathbb{Q}_{>0}} \mathsf{Reach}(\mathcal{A}(\delta))$. It is the set of states of the automaton reachable by an arbitrarily small value of the parameter. While [Pur00] proposed an algorithm based on the region graph, we use an algorithm proposed in [DK06] which relies on zones, as it is better suited for a parametric setting. The drawback of [DK06] is that it requires the timed automaton to be flat as it enumerates cycles of the automaton.

Algorithm 1. Computation of $\mathsf{Reach}^*(\mathcal{A})$

```
Require: a progressive flat timed automaton $\mathcal{A}$ with bounded clocks.
Ensure: the set $\mathsf{Reach}^*(\mathcal{A})$.
 1: Compute $\nu Y.\mathsf{Pre}^{\varrho}(Y)$, $\nu Y.\mathsf{Post}^{\varrho}(Y)$, for each cycle $\varrho$ in $\mathcal{A}$.
 2: Wait $= \{(\ell_0, Z_0)\}$ ;                                   // Initial states
 3: Passed $= \varnothing$ ;
 4: while Wait $\neq \varnothing$ do
 5:   pop $(\ell, Z)$ from Wait ;
 6:   if $\forall (\ell, Z') \in$ Passed, $Z \not\subseteq Z'$ then
 7:     if there exists a cycle $\varrho$ around location $\ell$ then
 8:       if $Z \cap \nu Y.\mathsf{Pre}^{\varrho}(Y) \neq \varnothing$ then
 9:         Wait $=$ Wait $\cup\ \mathsf{Succ}(\ell, \nu Y.\mathsf{Post}^{\varrho}(Y))$;
10:         Passed $=$ Passed $\cup\ \{(\ell, \nu Y.\mathsf{Post}^{\varrho}(Y))\}$;
11:     Wait $=$ Wait $\cup\ \mathsf{Succ}(\ell, Z)$ ;
12:     Passed $=$ Passed $\cup\ \{(\ell, Z)\}$ ;
13: return Passed ;
```

A new procedure for the computation of Reach^*. We present Algorithm 1 which is a modification of the algorithm proposed in [DK06] to compute Reach^*. This modification allows us in Section 4 to prove the termination of a parametric version of this algorithm.

The original algorithm proposed in [DK06] relies on the notion of *stable zone* of a cycle ϱ. This zone represents states having infinitely many predecessors and successors by ϱ, and is defined as the intersection of two greatest fixpoints: $W_\varrho = \nu Y.\mathsf{Post}^{\varrho}(Y) \cap \nu Y.\mathsf{Pre}^{\varrho}(Y)$. Then, the algorithm is obtained by the following modifications of the standard forward analysis of the timed automaton: for each new symbolic state (ℓ, Z) considered, if there exists a cycle ϱ around location ℓ, and if Z intersects the stable zone W_ϱ, then the stable zone is marked as reachable. The correction of this algorithm relies on the following property of the stable zone: given two valuations $v, v' \in W_\varrho$, for any $\delta > 0$, there exists a path in $[\![\mathcal{A}(\delta)]\!]$ from state (ℓ, v) to state (ℓ, v') (while such a path may not exist in $[\![\mathcal{A}]\!]$). The addition of the stable zone can be viewed as the acceleration of cycle ϱ.

Our new algorithm is obtained as follows: (i) at line 8, we test the intersection of Z with $\nu Y.\mathsf{Pre}^{\varrho}(Y)$ instead of W_ϱ, and (ii) at line 9 and 10, instead of declaring W_ϱ as reachable, we declare $\nu Y.\mathsf{Post}^{\varrho}(Y)$ reachable. We state below that this modification is correct.

Theorem 1. *Algorithm 1 is sound and complete.*

Proof. We show that Algorithm 1 is equivalent to that of [DK06]. As W_ϱ is included in both greatest fixpoints, the completeness of the algorithm is trivial. To prove the soundness, let us consider the region graph construction (see for instance [AD94]). We do not recall this standard construction as it will only be used in this proof. As there are finitely many regions, it is easy to verify that if a region is included in $\nu Y.\mathsf{Pre}^\varrho(Y)$, it has infinitely many successors by ϱ and then one of them is included in W_ϱ. In other terms, the test of line 8 of intersection with $\nu Y.\mathsf{Pre}^\varrho(Y)$ instead of W_ϱ simply anticipates the acceleration of the cycle ϱ. Similarly, any region included in $\nu Y.\mathsf{Post}^\varrho(Y)$ is the successor of a region included in W_ϱ. Thus, our modification can be understood as a speed-up of the original algorithm of [DK06]. □

We also state the following Lemma whose proof follows from a similar reasoning:

Lemma 1. *Let ϱ be a cycle of a TA $\mathcal{A}$. Then we have:*
$\nu Y.\mathsf{Pre}^\varrho(Y) \neq \varnothing \Leftrightarrow \nu Y.\mathsf{Pre}^\varrho(Y) \cap \nu Y.\mathsf{Post}^\varrho(Y) \neq \varnothing \Leftrightarrow \nu Y.\mathsf{Post}^\varrho(Y) \neq \varnothing$

4 Parametric Computation of Reach$(\mathcal{A}(\delta))$

4.1 Representing Constraints as a Graph

In the sequel, we will use a representation of clock constraints as a weighted directed graph introduced in [CJ99a, CJ99b]. Due to lack of space, we recall here only succinctly its definition. Intuitively, the value of a clock can be recovered from its date of reset and the current time. The vertices of the graph represent these values, with one duplicate for each fired transition. Constraints on clock values are expressed as weights on arcs.

More formally, recall that $n = |\mathcal{X}|$, $\mathcal{X} = \{x_1, \ldots, x_n\}$, x_0 is a fictive clock whose value is always zero, and $\overline{\mathcal{X}} = \mathcal{X} \cup \{x_0\}$. We introduce a new clock τ which is never reset and thus represents the total elapsed time. In addition, for each clock $x_i \in \overline{\mathcal{X}}$ we let variable X_i denote $X_i = \tau - x_i$. Note that for $x_i \in \mathcal{X}$, X_i thus represents last date of reset of clock x_i. For the special case of x_0, we have $X_0 = \tau$ (as x_0 always has value 0). We denote $\overrightarrow{V}$ the vector defined as $(\tau, X_1, \ldots, X_n)$ which is a vector of (symbolic) variables. For a transition $t = (\ell, g, \sigma, r, \ell')$, we define the formula $T^t(\overrightarrow{V}, \overrightarrow{V'})$ which expresses the relationship between values of the variables before (represented by $\overrightarrow{V}$) and after the firing of the transition (represented by $\overrightarrow{V'} = (\tau', X'_1, \ldots, X'_n)$):

$$T^t(\overrightarrow{V}, \overrightarrow{V'}) := \bigwedge_{i=1}^{n} \left(X_i \leq \tau \wedge X'_i \leq \tau' \wedge \tau \leq \tau' \wedge \bigwedge_{x_i \in r} \tau = X'_i \wedge \bigwedge_{x_i \notin r} X_i = X'_i \wedge \overline{g} \right)$$

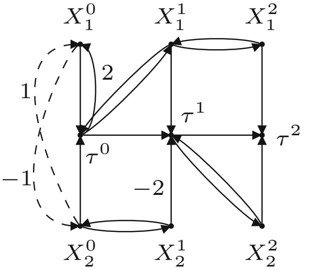
where $\overline{g}$ is the constraint g where for any i, clock x_i is replaced by $\tau - X_i$.

Let $\varrho = t_1 \ldots t_m$ be a sequence of transitions. For $j \in \{0, \ldots, m\}$, we denote by $\overrightarrow{V^j}$ the vector $(\tau^j, X_1^j, \ldots, X_n^j)$. Then we define formula $T^\varrho(\overrightarrow{V^0}, \overrightarrow{V^m})$ expressing the constraints between variables before and after the firing of the sequence ϱ as follows:

$$T^\varrho(\overrightarrow{V^0}, \overrightarrow{V^m}) := \exists \overrightarrow{V^1}, \ldots, \overrightarrow{V^{m-1}}. \bigwedge_{j=0}^{m-1} T^{t_{j+1}}(\overrightarrow{V^j}, \overrightarrow{V^{j+1}})$$

We associate with formula $T^{\varrho}(\overrightarrow{V^0}, \overrightarrow{V^m})$ a weighted directed graph whose vertices are variables used to define the formula, and arcs represent constraints of the formula:

Definition 4 (Graph $G_{\varrho}^{\top}$). *Let $\varrho = t_1 \dots t_m$ be a sequence of transitions. The weighted directed graph $G_{\varrho}^{\top}$ has a set of vertices $\mathcal{S} = \bigcup_{j=0}^{m} V^j$, where $V^j = \{\tau^j, X_1^j, \dots, X_n^j\}$. Given two vertices $X, X' \in \mathcal{S}$ and a weight $c \in \mathbb{Q}$, there is an arc from X to X' labelled by c if and only if the formula $T^{\varrho}(\overrightarrow{V^0}, \overrightarrow{V^m})$ contains the constraint $X - X' \leq c$.*



Example 2. Consider the sequence of transitions $\varrho = t_1 t_2$ in the TA of Figure 1 defined in Example 1. The graph depicted on the left-side figure with plain arcs represents $G_{\varrho}^{\top}$ (arcs without label have weight 0). For instance, the arc from vertex X_2^1 to vertex τ^1, labelled by -2, represents the lower bound for the clock x_2 in t_2 which means: $x_2 \geq 2$.

For any path p, we write $w(p)$ the total weight of the path. Suppose now that there is no cycle of negative weight in graph $G_{\varrho}^{\top}$. Let P_{beg}^{ϱ} (resp. P_{end}^{ϱ}) denote the set of minimal weighted paths from a vertex in V^0 (resp. in $V^{|\varrho|}$) to another vertex in V^0 (resp. in $V^{|\varrho|}$). We define the following mapping which interprets these shortest paths as clock constraints:

$$C(p) = x_l - x_i \leq w(p) \text{ if } \begin{cases} p \in P_{beg}^{\varrho} \text{ starts in } X_i^0 \text{ and ends in } X_l^0 \\ p \in P_{end}^{\varrho} \text{ starts in } X_i^{|\varrho|} \text{ and ends in } X_l^{|\varrho|} \end{cases}$$

From Propositions 12 and 13 of [CJ99b], we have the following properties:

Proposition 1. *Let ϱ be a sequence of transitions. Then we have:*

- *there exists a cycle γ with $w(\gamma) < 0$ in $G_{\varrho}^{\top}$* $\Leftrightarrow \mathsf{Post}^{\varrho}(\top) = \varnothing \Leftrightarrow \mathsf{Pre}^{\varrho}(\top) = \varnothing$
- *if there is no cycle of negative weight, then:* $[\![\bigwedge_{p \in P_{end}^{\varrho}} C(p)]\!] = \mathsf{Post}^{\varrho}(\top)$ *and* $[\![\bigwedge_{p \in P_{beg}^{\varrho}} C(p)]\!] = \mathsf{Pre}^{\varrho}(\top)$

More generally, given a zone Z, we define the graph denoted G_{ϱ}^{Z} by adding the constraints of Z on the vertices in V^0. Mapping C applied on paths in P_{end}^{ϱ} then defines the zone $\mathsf{Post}^{\varrho}(Z)$. Similarly, the zone $\mathsf{Pre}^{\varrho}(Z)$ can be represented by adding constraints of Z on vertices in $V^{|\varrho|}$.

Example 3 (Example 2 continued.). Consider now the zone $Z = [\![x_1 - x_2 = 1]\!]$ (it corresponds to the set of reachable valuations after firing transition $\ell_0 \rightarrow \ell_1$ in TA of Figure 1), then additional dotted arcs allow to represent G_{ϱ}^{Z}.

It is easy to verify that this construction extends to a parametric setting: considering parametric constraints on arcs, we obtain a graph representation of the parametric computation of symbolic successors or predecessors. Note that a path p in this context will have a weight of the form $k + b\Delta$, where $b \in \mathbb{N}$ represents the number of atomic constraints of the TA used in p. In particular, while the value of a path depends on the value of Δ, its existence does not.

Given a zone defined as the result of the firing of a sequence of transitions, this representation allows to recover how the constraints are obtained. Thus, the graph stores the complete history of the constraints.

In the sequel, we use this construction in the particular case of the iteration of a cycle ϱ, given as a sequence of transitions of a TA. Let Z_{init} be a zone. We consider two sequences of zones $(Z_k^{\top})_{k\geq 0}$ (resp. $(Z_k^{init})_{k\geq 0}$) defined by $Z_0^{\top} = \top$ (resp. $Z_0^{init} = Z_{init}$) and $Z_{k+1}^{*} = \mathsf{Post}^{\varrho}(Z_k^{*})$ (where * denotes either $^{\top}$ or ^{init}). Note that by monotonicity of Post^{ϱ}, the sequence $(Z_k^{\top})_{k\geq 0}$ is decreasing and converges towards $Z_{\infty}^{\top} = \nu Y.\mathsf{Post}^{\varrho}(Y)$. According to Proposition 1, note that constraints defining zone $Z_k^{\top}$ (resp. Z_k^{init}) can be obtained from shortest paths in graph $G_{\varrho^k}^{\top}$ (resp. $G_{\varrho^k}^{init}$). As the cycle ϱ will be clear from the context, we will omit to mention it in the subscript, and use notations $G_k^{\top}$ and G_k^{init} respectively.

Moreover, we will only be interested in vertices at the frontier between the different copies of the graph of ϱ. Then, given a clock $x_i \in \overline{\mathcal{X}}$ and an index $j \leq k$, *vertex* X_i^j *now denotes the date of reset of clock* x_i *after the* j*-th execution of* ϱ (this notation is a shorthand for the notation $X_i^{j\times|\varrho|}$, as this last notation will never be used anymore).

Definition 5. *Let* $N = |\overline{\mathcal{X}}|^2$. *A* return path *is a pair* $r = (p_1, p_2)$ *of paths in the graph* $G_N^{\top}$ *such that there exist two clocks* $x_u, x_v \in \overline{\mathcal{X}}$ *and two indices* $0{\leq}i{<}j{\leq}N$ *verifying:*

- p_1 *and* p_2 *are included in the subgraph associated with* i*-th to* j*-th copies of* ϱ
- p_1 *is a shortest path from vertex* X_u^j *to vertex* X_u^i
- p_2 *is a shortest path from vertex* X_v^i *to vertex* X_v^j

The weight of r *is defined as* $w(r) = w(p_1) + w(p_2)$. *The set of return paths is finite and is denoted* $\mathcal{R}$.

4.2 Accelerating Computations of Greatest Fixpoints

Let ϱ be a cycle. In this subsection, we only consider the operator Post^{ϱ}, but all our results also apply to the operator Pre^{ϱ}. We consider the decreasing sequence $(Z_k^{\top})_{k\geq 0}$ converging towards $Z_{\infty}^{\top} = \nu Y.\mathsf{Post}^{\varrho}(Y) = \bigcap_{k\geq 0} Z_k^{\top}$. We prove the following lemma which provides a bound for termination only dependant on the number of clocks. Note that this result does not require the cycle ϱ to be progressive neither the clocks to be bounded.

Lemma 2. *Let* $N = |\overline{\mathcal{X}}|^2$, *and* $k \geq N$. *If* $Z_{k+1}^{\top} \subsetneq Z_k^{\top}$, *then we have* $Z_{\infty}^{\top} = \varnothing$.

Proof. First, we prove that $Z_{k+1}^{\top} \subsetneq Z_k^{\top}$ implies that there exists $r \in \mathcal{R}$ used in some shortest path of $Z_{k+1}^{\top}$ witness of the disequality. Indeed, as $Z_{k+1}^{\top} \subsetneq Z_k^{\top}$, there exists a bound $b =$ ”$x_p - x_q \leq \cdot$” with $0 \leq p \neq q \leq n$, whose constraint is strictly smaller in $Z_{k+1}^{\top}$ than in $Z_k^{\top}$. In $Z_{k+1}^{\top}$, the constraint on b is obtained as a shortest path between vertices X_p^{k+1} and X_q^{k+1} in the graph $G_{k+1}^{\top}$. Let c be such a path. By definition of $G_k^{\top}$ and $G_{k+1}^{\top}$, the path c must use arcs in $G_1^{\top}$ (otherwise c would also exist in $G_k^{\top}$). The graph $G_{k+1}^{\top}$ is the concatenation of $k+1$ copies of the graph of ϱ. For each occurrence of ϱ, c goes through a pair of vertices when it enters/leaves it. Finally, as $k+1 > N = |\overline{\mathcal{X}}|^2$, there exists a pair that occurs twice, we denote these

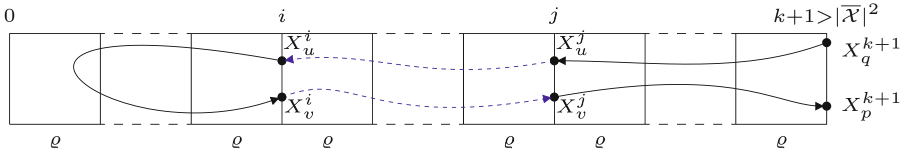


Fig. 3. Pumping lemma : a path from X_q^{k+1} to X_p^{k+1} using arcs in $G_1^\top$ exhibits a return path between pairs of vertices (X_u^i, X_v^i) and (X_u^j, X_v^j)

two clocks x_u and x_v. Thus c contains a return path $r \in \mathcal{R}$ (see Figure 3 representing the graph $G_{k+1}^\top$ and the return path r in the shortest path c).

Second, as $Z_{k+1}^\top \subsetneq Z_k^\top$, we have $w(r) < 0$. By contradiction, if $w(r) > 0$ then c would not be a shortest path and if $w(r) = 0$ then c would also exist in $G_k^\top$.

Finally, the existence of a return path $r \in \mathcal{R}$ such that $w(r) < 0$ implies that $Z_\infty^\top = \varnothing$ $(= \nu Y.\mathsf{Post}^\varrho(Y))$. When k grows, one can build new paths by repeating this return path. As its weight is negative, the weights of the paths we build diverge towards $-\infty$. In particular, the constraint of the zone $Z_\infty^\top$ on the clock difference $x_p - x_q$ cannot be finite (as it is the limit of a sequence diverging towards $-\infty$), and thus we obtain $Z_\infty^\top = \varnothing$. □

We can now compute, in the parametric setting, the greatest fixpoint of PPost^ϱ for every cycle ϱ of the automaton. We first evaluate the parametric zones $\mathcal{Z} = \mathsf{PPost}^{\varrho^N}(\top)$ and $\mathcal{Z}' = \mathsf{PPost}^\varrho(\mathcal{Z})$. Then, we determine the minimal value $\delta_0 = \min\{\delta \geq 0 \mid \mathcal{Z}(\delta) = \mathcal{Z}'(\delta)\}$. This definition is correct as $\mathcal{Z}' \subseteq \mathcal{Z}$ and, for large enough values of δ, all parametric constraints are equivalent to $\top$. Thus, we have $\nu\mathcal{Y}.\mathsf{PPost}^\varrho(\mathcal{Y})(\delta) \neq \varnothing$ which implies by Lemma 2 that $\mathcal{Z}(\delta) = \mathcal{Z}'(\delta)$. Finally the greatest fixpoint can be represented by $\mathcal{Z}_{|[\delta_0;+\infty[}$ as Lemma 2 ensures that the fixpoint is empty for all $\delta < \delta_0$.

4.3 Parametric Forward Analysis with Acceleration

We present Algorithm 2 for the parametric computation of $\mathsf{Reach}(\mathcal{A}(\delta))$. It can be understood as an adaptation in a parametric setting of Algorithm 1. First, at line 1 we perform parametric computation of greatest fixpoints using the procedure proposed in Section 4.2. Second, the test of intersection between the current zone and the greatest fixpoint of Pre^ϱ is realized in a parametric setting by the computation at line 8 of $\delta_{\min} = \delta_{\neg\varnothing}(\mathcal{Z} \cap \nu\mathcal{Y}.\mathsf{PPre}^\varrho(\mathcal{Y}))$. Finally, we split the domain of the current parametric zone into intervals I_1 and I_2. In interval I_1, no acceleration is done for cycles and thus the set $\mathsf{Reach}(\mathcal{A}(\delta))$ is computed. Acceleration techniques are used only for interval I_2, and for these values the algorithm computes the set $\mathsf{Reach}^*(\mathcal{A}(\delta))$. We prove below that in this case, the equality $\mathsf{Reach}(\mathcal{A}(\delta)) = \mathsf{Reach}^*(\mathcal{A}(\delta))$ holds. Note that the test at line 9 allows to handle differently the particular case of value $\delta_{\min}$ which does not always require to apply acceleration.

Theorem 2. *Algorithm 2 terminates and is correct.*

In the sequel, we denote $N = |\overline{\mathcal{X}}|^2$ and $\delta_{\neg\varnothing}^\varrho = \delta_{\neg\varnothing}(\nu\mathcal{Y}.\mathsf{PPre}^\varrho(\mathcal{Y})) = \delta_{\neg\varnothing}(\nu\mathcal{Y}.\mathsf{PPost}^\varrho(\mathcal{Y}))$ (by Lemma 1). Before turning to the proof, we state the following

Algorithm 2. Parametric Computation of the Reachability Set

```
Require: a progressive flat timed automaton $\mathcal{A}$ with bounded clocks.
Ensure: the set $\mathsf{Reach}(\mathcal{A}(\delta))$ for all $\delta \in \mathbb{R}_{\geq 0}$.
Compute $\nu\mathcal{Y}.\mathsf{PPre}^{\varrho}(\mathcal{Y})$ and $\nu\mathcal{Y}.\mathsf{PPost}^{\varrho}(\mathcal{Y})$ for each cycle $\varrho$ of $\mathcal{A}$.
Wait $= \{(\ell_0, \mathcal{Z}_0)\}$ ;                                   // Initial States
Passed $= \varnothing$ ;
while Wait $\neq \varnothing$ do
    pop $(\ell, \mathcal{Z})$ from Wait ;
    if $\forall(\ell, \mathcal{Z}') \in$ Passed, $\mathcal{Z} \not\subseteq \mathcal{Z}'$ then
        if there exists a cycle $\varrho$ around location $\ell$ then
            $\delta_{\min} = \delta_{\neg\varnothing}(\mathcal{Z} \cap \nu\mathcal{Y}.\mathsf{PPre}^{\varrho}(\mathcal{Y}))$ ;
            if $\delta_{\min} = \delta_{\neg\varnothing}(\nu\mathcal{Y}.\mathsf{PPre}^{\varrho}(\mathcal{Y}))$ then
                $I_1 = [0; \delta_{\min}]$ ; $I_2 = ]\delta_{\min}; +\infty[$ ;
            else
                $I_1 = [0; \delta_{\min}[$ ; $I_2 = [\delta_{\min}; +\infty[$ ;
            Wait $=$ Wait $\cup\, \mathsf{Succ}(\ell, \mathcal{Z}_{|I_1}) \cup \mathsf{Succ}(\ell, \mathcal{Z}_{|I_2}) \cup \mathsf{Succ}(\ell, \nu\mathcal{Y}.\mathsf{PPost}^{\varrho}(\mathcal{Y})_{|I_2})$ ;
            Passed $=$ Passed $\cup\, (\ell, \mathcal{Z}_{|I_1}) \cup (\ell, \mathcal{Z}_{|I_2}) \cup (\ell, \nu\mathcal{Y}.\mathsf{PPost}^{\varrho}(\mathcal{Y})_{|I_2})$ ;
        else
            Wait $=$ Wait $\cup\, \mathsf{Succ}(\ell, \mathcal{Z})$ ;
            Passed $=$ Passed $\cup\, (\ell, \mathcal{Z})$ ;
return Passed ;
```

Lemma whose proof is given in [JR10]. Intuitively, it establishes that when all return paths have a positive weight, then either (i) the starting zone has finitely many successors and then it converges to the empty set after at most N steps, or (ii) it has infinitely many successors and then it converges towards $\nu Y.\mathsf{Post}^{\varrho}(Y)$. In this last case, the enlarged reachability set corresponds to the standard reachability set. Its proof relies on pumping techniques presented in Section 4.2. To illustrate property (ii), let consider the timed automaton of Figure 1, for which the enlarged reachability set strictly contains the standard reachability set. One can verify that there exists a return path associated with $\varrho = t_1 t_2$ which has weight 0.

Lemma 3. *Let ϱ be such that for any return path $r \in \mathcal{R}$, we have $w(r) > 0$. Then we have:*

(i) *If $Z_{init} \cap \nu Y.\mathsf{Pre}^{\varrho}(Y) = \varnothing$, then $Z_N^{init} = \varnothing$.*
(ii) *If $Z_{init} \cap \nu Y.\mathsf{Pre}^{\varrho}(Y) \neq \varnothing$, then $Z_{\infty}^{init} = Z_{\infty}^{\top} (= \nu Y.\mathsf{Post}^{\varrho}(Y))$.*

Unlike Lemma 2, we use the progress cycle assumption to prove this lemma.

Recall that the TA we consider are flat. As a consequence, in the following proofs of termination and correctness, we will only consider a simple cycle ϱ.

Termination. Consider a parametric symbolic state $(\ell, \mathcal{Z})$ and a cycle ϱ starting in ℓ. We have to prove that all the elements added to the Wait list have a finite number of successors. This is trivial for the successors of $(\ell, \nu\mathcal{Y}.\mathsf{PPost}^{\varrho}(\mathcal{Y})_{|I_2})$ as $\nu\mathcal{Y}.\mathsf{PPost}^{\varrho}(\mathcal{Y})_{|I_2}$ is by definition a fixpoint of PPost^{ϱ}. We now focus on the successors of $(\ell, \mathcal{Z}_{|I_1})$ and $(\ell, \mathcal{Z}_{|I_2})$. Note that we have $\delta_{\min} \geq \delta_{\neg\varnothing}^{\varrho}$.

- **Case of** $(\ell, \mathcal{Z}_{|I_2})$**:** We prove property $(*)$ $\mathsf{PPost}^{\varrho^N}(\mathcal{Z}_{|I_2}) \subseteq \nu\mathcal{Y}.\mathsf{PPost}^{\varrho}(\mathcal{Y})_{|I_2}$. Then the computation is stopped by the test of line 6 as the greatest fixpoint has been added to the Passed list. To prove $(*)$, we prove it holds for any $\delta \in I_2$. Fix some $\delta \in I_2$ and define $Z_{init} = \mathcal{Z}_{|I_2}(\delta)$. We consider the two sequences $(Z_i^*)_{i\geq 0}$ *w.r.t.* cycle ϱ enlarged by δ. Note that as $\delta \geq \delta_{\min} \geq \delta^{\varrho}_{\neg\varnothing}$, we have $\nu\mathcal{Y}.\mathsf{PPost}^{\varrho}(\mathcal{Y})(\delta) \neq \varnothing$. By Lemma 2, this entails $Z_N^{\top} = \nu\mathcal{Y}.\mathsf{PPost}^{\varrho}(\mathcal{Y})(\delta)$. By monotonicity of Post^{ϱ}, $Z_N^{init} \subseteq Z_N^{\top}$ holds. This yields the result.
- **Case of** $(\ell, \mathcal{Z}_{|I_1})$**:** We distinguish two cases whether $\delta_{\min} > \delta^{\varrho}_{\neg\varnothing}$ or not.

  If $\delta_{\min} > \delta^{\varrho}_{\neg\varnothing}$**:** for any $\delta \in [\delta^{\varrho}_{\neg\varnothing}, \delta_{\min}[$, Lemma 3.$(i)$ can be applied on cycle ϱ enlarged by δ. This implies that for any $\delta \in [\delta^{\varrho}_{\neg\varnothing}, \delta_{\min}[$, we have $\mathsf{PPost}^{\varrho^N}(\mathcal{Z}_{|I_1})(\delta) = \varnothing$. Then this property also holds for any $\delta \in I_1$, by monotonicity of $\mathcal{Z}$ and PPost^{ϱ}.

  If $\delta_{\min} = \delta^{\varrho}_{\neg\varnothing}$**:** the complete proof of this last case is more technical and is completely described in [JR10]. We only present here a sketch of proof. First note that for any fixed value of $\delta < \delta_{\min}$, as the zone does not intersect the greatest fixpoint of Pre^{ϱ}, the zone has finitely many successors. However, this argument cannot be lifted to a parametric setting as this number diverges when δ converges towards $\delta_{\min}$. By definition of $\delta^{\varrho}_{\neg\varnothing}$, some return paths, which we call *optimal*, have a weight equal to 0 in $\delta^{\varrho}_{\neg\varnothing}$ (and are thus strictly negative on $[0, \delta^{\varrho}_{\neg\varnothing}[$). Our proof consists in first showing that there exists some integer k for which after k steps, all shortest paths go through optimal return paths. Then, considering q as the least common multiple of lengths of optimal return paths, we can prove the following inclusion $\mathsf{PPost}^{\varrho^{k+q}}(\mathcal{Z}_{|I_1}) \subseteq \mathsf{PPost}^{\varrho^k}(\mathcal{Z}_{|I_1})$. The algorithm stops by test of line 6.

Correctness. As explained before, the algorithm is a standard forward analysis which may add some additional behaviours, according to test of line 8. We distinguish three cases:

1. **For** $\delta \in [0, \delta_{\min}[$ **:** For these values, the algorithm simply performs a forward analysis. As a consequence, the correctness is trivial.
2. **For** $\delta \in]\delta_{\min}, +\infty[$**:** For all these values, the addition occurs, and then the algorithm is equivalent to Algorithm 1. By correction of Algorithm 1, this implies that it computes the set $\mathsf{Reach}^*(\mathcal{A}(\delta))$. We will prove that for all these values, we have the equality $\mathsf{Reach}(\mathcal{A}(\delta)) = \mathsf{Reach}^*(\mathcal{A}(\delta))$. Therefore we need to prove that what has been added to obtain $\mathsf{Reach}^*(\mathcal{A}(\delta))$ was already in $\mathsf{Reach}(\mathcal{A}(\delta))$. Note that the only addition is the greatest fixpoint of Post^{ϱ}. The property is then a direct consequence of Lemma 3.(ii) as it states that the greatest fixpoint is reachable from the initial states. It is easy to verify that Lemma 3.(ii) can indeed be applied.
3. **For** $\delta = \delta_{\min}$**:** There are two cases, whether $\delta_{\min} = \delta^{\varrho}_{\neg\varnothing}$ or not. If the equality holds, then $\delta_{\min} \in I_1$ and the reasoning developed at point 1. also applies. If $\delta_{\min} > \delta^{\varrho}_{\neg\varnothing}$ holds, then $\delta_{\min} \in I_2$ and we can apply reasoning of point 2. as Lemma 3.(ii) also applies because we have $\delta_{\min} > \delta^{\varrho}_{\neg\varnothing}$.

4.4 Quantitative Safety

Once the reachable state space of the automaton is computed by Algorithm 2, it is easy to compute the maximal value of the parameter such that the system avoids some set of bad states. Simply compute the value $\delta_{\neg\varnothing}$ on each parametric zone associated with a bad location and keep the lower one: $\delta_{\max} = \min\{\delta_{\neg\varnothing}(\mathcal{Z}) \mid \exists \ell \in Bad$ such that $(\ell, \mathcal{Z}) \in$ Passed$\}$. We thus obtain:

Theorem 3. *The quantitative robustness problem for safety properties is decidable for flat progressive timed automata with bounded clocks. In addition, the value $\delta_{\max}$ is a rational number.*

5 Conclusion

In this paper, we considered the quantitative robustness problem for safety properties, which aims at computing the largest value of the parameter Δ under which the TA is safe. We proposed a symbolic forward algorithm for the computation of the parametric reachability set for flat timed automata. We proved its termination by means of original arguments using a representation of zones by graphs. As a consequence, it allows us to compute the largest safe value of the parameter, and prove it is a rational number.

Among perspectives, we are first implementing the algorithm using a data structure specific to the parametric zones used in our setting. Second, we want to study the complexity of our algorithm. The difficulty is due to the argument of termination in the last case which leads to a large value and may be improved.

We also aim at enlarging the class of TA for which we can solve the quantitative robustness problem. For instance, if the parameter is not always introduced on guards with coefficient 1, but with other coefficients in $\mathbb{N}_{>0}$, we believe that our algorithm can also be applied. A challenging topic concerns the hypothesis of flatness: first, [CJ99a] proves that timed automata can be flattened, and we want to study whether their result can be combined with ours. Second, we plan to investigate a parametric extension of the algorithm introduced in [Dim07] which can be seen as an extension of that of [DK06] to non-flat TA.

Finally, we believe that it should be possible to solve the quantitative robustness problem for flat TA for other specifications like for instance LTL properties.

References

[AAB00] Annichini, A., Asarin, E., Bouajjani, A.: Symbolic techniques for parametric reasoning about counter and clock systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 419–434. Springer, Heidelberg (2000)

[AD94] Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)

[ALM05] Alur, R., La Torre, S., Madhusudan, P.: Perturbed timed automata. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 70–85. Springer, Heidelberg (2005)

[BBB$^+$07] Baier, C., Bertrand, N., Bouyer, P., Brihaye, T., Größer, M.: Probabilistic and topological semantics for timed automata. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 179–191. Springer, Heidelberg (2007)

[BDL04] Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)

[BDM$^+$98] Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: a model-checking tool for real-time systems. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 546–550. Springer, Heidelberg (1998)

[BIK10] Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010)

[BIL06] Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 577–588. Springer, Heidelberg (2006)

[BMR06] Bouyer, P., Markey, N., Reynier, P.-A.: Robust model-checking of linear-time properties in timed automata. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 238–249. Springer, Heidelberg (2006)

[BMR08] Bouyer, P., Markey, N., Reynier, P.-A.: Robust analysis of timed automata *via* channel machines. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 157–171. Springer, Heidelberg (2008)

[CHR02] Cassez, F., Henzinger, T.A., Raskin, J.-F.: A comparison of control problems for timed and hybrid systems. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 134–148. Springer, Heidelberg (2002)

[CJ99a] Comon, H., Jurski, Y.: Timed automata and the theory of real numbers. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 242–257. Springer, Heidelberg (1999)

[CJ99b] Comon, H., Jurski, Y.: Timed automata and the theory of real numbers. Research Report LSV-99-6, Laboratoire Spécification et Vérification, ENS Cachan, France, 44 pages (July 1999)

[DDMR08] De Wulf, M., Doyen, L., Markey, N., Raskin, J.-F.: Robust safety of timed automata. Formal Methods in System Design 33(1-3), 45–84 (2008)

[DDR05] De Wulf, M., Doyen, L., Raskin, J.-F.: Almost ASAP semantics: from timed models to timed implementations. Formal Aspects of Computing 17(3), 319–341 (2005)

[Dim07] Dima, C.: Dynamical properties of timed automata revisited. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 130–146. Springer, Heidelberg (2007)

[DK06] Daws, C., Kordy, P.: Symbolic robustness analysis of timed automata. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 143–155. Springer, Heidelberg (2006)

[GHJ97] Gupta, V., Henzinger, T.A., Jagadeesan, R.: Robust timed automata. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, pp. 331–345. Springer, Heidelberg (1997)

[HRSV02] Hune, T., Romijn, J., Stoelinga, M., Vaandrager, F.: Linear parametric model checking of timed automata. Journal of Logic and Algebraic Programming (2002)

[Jau09] Jaubert, R.: Aspects quantitatifs dans la réalisation de contrôleurs temps-réels robustes. Mémoire de Master Recherche, Master Informatique Fondamentale, Marseille (2009)

[JR10] Jaubert, R., Reynier, P.-A.: Quantitative robustness analysis of flat timed automata. Research Report 00534896, HAL (2010)

[OW03] Ouaknine, J., Worrell, J.B.: Revisiting digitization, robustness and decidability for timed automata. In: Proc. LICS 2003. IEEE Computer Society Press, Los Alamitos (2003)

[Pur00] Puri, A.: Dynamical properties of timed automata. Discrete Event Dynamic Systems 10(1-2), 87–113 (2000)

[SF07] Swaminathan, M., Fränzle, M.: A symbolic decision procedure for robust safety of timed systems. In: Proc. TIME 2007, p. 192. IEEE Computer Society Press, Los Alamitos (2007)

[SFK08] Swaminathan, M., Fränzle, M., Katoen, J.-P.: The surprising robustness of (closed) timed automata against clock-drift. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) Proc. TCS 2008. IFIP, vol. 273, pp. 537–553. Springer, Heidelberg (2008)

[WT99] Wong-Toi, H.: Analysis of slope-parametric rectangular automata. In: Antsaklis, P.J., Kohn, W., Lemmon, M.D., Nerode, A., Sastry, S.S. (eds.) HS 1997. LNCS, vol. 1567, pp. 390–413. Springer, Heidelberg (1999)

A Game Approach to Determinize Timed Automata

Nathalie Bertrand[1], Amélie Stainer[1], Thierry Jéron[1], and Moez Krichen[2]

[1] INRIA Rennes - Bretagne Atlantique, Rennes, France
[2] Institute of Computer Science and Multimedia, Sfax, Tunisia

Abstract. Timed automata are frequently used to model real-time systems. Their determinization is a key issue for several validation problems. However, not all timed automata can be determinized, and determinizability itself is undecidable. In this paper, we propose a game-based algorithm which, given a timed automaton with ε-transitions and invariants, tries to produce a language-equivalent deterministic timed automaton, otherwise a deterministic over-approximation. Our method subsumes two recent contributions: it is at once more general than the determinization procedure of [4] and more precise than the approximation algorithm of [11].

1 Introduction

Timed automata (TA), introduced in [1], form a usual model for the specification of real-time embedded systems. Essentially TAs are an extension of automata with guards and resets of continuous clocks. They are extensively used in the context of many validation problems such as verification, control synthesis or model-based testing. One of the reasons for this popularity is that, despite the fact that they represent infinite state systems, their reachability is decidable, thanks to the construction of the region graph abstraction.

Determinization is a key issue for several problems such as implementability, diagnosis or test generation, where the underlying analyses depend on the observable behavior. In the context of timed automata, determinization is problematic for two reasons. First, determinizable timed automata form a strict subclass of timed automata [1]. Second, the problem of the determinizability of a timed automaton, (i.e. does there exist a deterministic TA with the same language as a given non-deterministic one?) is undecidable [9,14]. Therefore, in order to determinize timed automata, two alternatives have been investigated: either restricting to determinizable classes or choosing to ensure termination for all TAs by allowing over-approximations, i.e. deterministic TAs accepting more timed words. For the first approach, several classes of determinizable TAs have been identified, such as strongly non-Zeno TAs [3], event-clock TAs [2], or TAs with integer resets [13]. In a recent paper, Baier, Bertrand, Bouyer and Brihaye [4] propose a procedure which does not terminate in general, but allows one to determinize TAs in a class covering all the aforementioned determinizable classes.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 245–259, 2011.

It is based on an unfolding of the TA into a tree, which introduces a new clock at each step, representing original clocks by a mapping; a symbolic determinization using the region abstraction; a folding up by the removal of redundant clocks. To our knowledge, the second approach has only been investigated by Krichen and Tripakis [11]. They propose an algorithm that produces a deterministic over-approximation based on a simulation of the TA by a deterministic TA with fixed resources (number of clocks and maximal constant). Its locations code (over-approximate) estimates of possible states of the original TA, and it uses a fixed policy governed by a finite automaton for resetting clocks.

Our method combines techniques from [4] and [11] and improves those two approaches, despite their notable differences. Moreover, it deals with both invariants and ε-transitions, but for clarity we present these treatments as extensions. Our method is also inspired by a game approach to decide the diagnosability of TAs with fixed resources presented by Bouyer, Chevalier and D'Souza in [8]. Similarly to [11], the resulting deterministic TA is given fixed resources (number of clocks and maximal constant) in order to simulate the original TA by a coding of relations between new clocks and original ones. The core principle is the construction of a finite turn-based safety game between two players, Spoiler and Determinizator, where Spoiler chooses an action and the region of its occurrence, while Determinizator chooses which clocks to reset. Our main result states that if Determinizator has a winning strategy, then it yields a deterministic timed automaton accepting exactly the same timed language as the initial automaton, otherwise it produces a deterministic over-approximation. Our approach is more general than the procedure of [4], thus allowing one to enlarge the set of timed automata that can be automatically determinized, thanks to an increased expressive power in the coding of relations between new and original clocks, and robustness to some language inclusions. Contrary to [4] our techniques apply to a larger class of timed automata: TAs with ε-transitions and invariants. It is also more precise than the algorithm of [11] in several respects: an adaptative and timed resetting policy, governed by a strategy, compared to a fixed untimed one and a more precise update of the relations between clocks, even for a fixed policy, allow our method to be exact on a larger class of TAs. The model used in [11] includes silent transitions, and edges are labeled with urgency status (eager, delayable, or lazy), but urgency is not preserved by their over-approximation algorithm. These observations illustrate the benefits of our game-based approach compared to existing work.

The structure of the paper is as follows. In Section 2 we recall definitions and properties relative to timed automata, and present the two recent pieces of work to determinize timed automata or provide a deterministic over-approximation. Section 3 is devoted to the presentation of our game approach and its properties. Extensions of the method to timed automata with invariants and ε-transitions are then presented in Section 4. A comparison with existing methods is detailed in Section 5.

Due to space limitation, most proofs are omitted in this paper. All details can be found in the research report [6].

2 Preliminaries

In this section, we start by introducing the model of timed automata, and then review two approaches for their determinization.

2.1 Timed Automata

We start by introducing notations and useful definitions concerning timed automata [1].

Given a finite set of clocks X, a clock *valuation* is a mapping $v : X \to \mathbb{R}_{\geq 0}$. We note $\overline{0}$ the valuation that assigns 0 to all clocks. If v is a valuation over X and $t \in \mathbb{R}_{\geq 0}$, then $v+t$ denotes the valuation which assigns to every clock $x \in X$ the value $v(x)+t$, and $\overleftrightarrow{v} = \{v+t \mid t \in \mathbb{R}\}$ denotes past and future timed extensions of v. For $X' \subseteq X$ we write $v_{[X' \leftarrow 0]}$ for the valuation equal to v on $X \setminus X'$ and to $\overline{0}$ on X', and $v_{|X'}$ for the valuation v restricted to X'.

Given a non-negative integer M, an *M-bounded guard*, or simply guard when M is clear from context, over X is a finite conjunction of constraints of the form $x \sim c$ where $x \in X$, $c \in [0, M] \cap \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$. We denote by $G_M(X)$ the set of M-bounded guards over X. Given a guard g and a valuation v, we write $v \models g$ if v satisfies g. Invariants are restricted cases of guards: given $M \in \mathbb{N}$, an *M-bounded invariant* over X is a finite conjunction of constraints of the form $x \lhd c$ where $x \in X$, $c \in [0, M] \cap \mathbb{N}$ and $\lhd \in \{<, \leq\}$. We denote by $I_M(X)$ the set of invariants. Given two finite sets of clocks X and Y, a *relation* between clocks of X and those of Y is a finite conjunction C of atomic constraints of the form $x - y \sim c$ where $x \in X$, $y \in Y$, $\sim \in \{<, =, >\}$ and $c \in \mathbb{N}$. When, moreover, the constant c is constrained to belong to $[-M', M]$, for some constants $M, M' \in \mathbb{N}$, we denote by $\mathsf{Rel}_{M,M'}(X, Y)$ the set of relations between X and Y.

Definition 1. *A timed automaton (TA) is a tuple $\mathcal{A} = (L, \ell_0, F, \Sigma, X, M, E, \mathsf{Inv})$ such that: L is a finite set of locations, $\ell_0 \in L$ is the initial location, $F \subseteq L$ is the set of final locations, Σ is a finite alphabet, X is a finite set of clocks, $M \in \mathbb{N}$, $E \subseteq L \times G_M(X) \times (\Sigma \cup \{\varepsilon\}) \times 2^X \times L$ is a finite set of edges, and $\mathsf{Inv} : L \to I_M(X)$ is the invariant function.*

The constant M is called the maximal constant of $\mathcal{A}$, and we will refer to $(|X|, M)$ as the *resources* of $\mathcal{A}$. The semantics of a timed automaton $\mathcal{A}$ is given as a timed transition system $\mathcal{T}_{\mathcal{A}} = (S, s_0, S_F, (\mathbb{R}_{\geq 0} \times (\Sigma \cup \{\varepsilon\})), \to)$ where $S = L \times \mathbb{R}^X_{\geq 0}$ is the set of states, $s_0 = (\ell_0, \overline{0})$ the initial state, $S_F = F \times \mathbb{R}^X_{\geq 0}$ the final states, and $\to \subseteq S \times (\mathbb{R}_{\geq 0} \times (\Sigma \cup \{\varepsilon\})) \times S$ the transition relation composed of moves of the form $(\ell, v) \xrightarrow{\tau, a} (\ell', v')$ whenever there exists an edge $(\ell, g, a, X', \ell') \in E$ such that $v + \tau \models g \wedge \mathsf{Inv}(\ell)$, $v' = (v+\tau)_{[X' \leftarrow 0]}$ and $v' \models \mathsf{Inv}(\ell')$.

A *run* ρ of $\mathcal{A}$ is a finite sequence of moves starting in s_0, *i.e.*, $\rho = s_0 \xrightarrow{\tau_1, a_1} s_1 \cdots \xrightarrow{\tau_k, a_k} s_k$. Run ρ is said accepting if it ends in $s_k \in S_F$. A *timed word* over Σ is an element $(t_i, a_i)_{i \leq n}$ of $(\mathbb{R}_{\geq 0} \times \Sigma)^*$ such that $(t_i)_{i \leq n}$ is nondecreasing. The timed word associated with ρ is $w = (t_{i_1}, a_{i_1}) \ldots (t_{i_m}, a_{i_m})$ where $(a_i \in$

Σ iff $\exists n,\ a_i = a_{i_n}$) and $t_i = \sum_{j=1}^{i} \tau_j$. We write $\mathcal{L}(\mathcal{A})$ for the language of $\mathcal{A}$, that is the set of timed words w such that there exists an accepting run which reads w. We say that two timed automata $\mathcal{A}$ and $\mathcal{B}$ are *equivalent* whenever $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$.

A *deterministic* timed automaton (abbreviated DTA) $\mathcal{A}$ is a TA such that for every timed word w, there is at most one run in $\mathcal{A}$ reading w. $\mathcal{A}$ is *determinizable* if there exists a deterministic timed automaton $\mathcal{B}$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$. It is well-known that some timed automata are not determinizable [1]; moreover, the determinizability of timed automata is an undecidable problem, even with fixed resources [14,9].

An example of a timed automaton is depicted in Figure 1. This nondeterministic timed automaton has ℓ_0 as initial location (denoted by a pending incoming arrow), ℓ_3 as final location (denoted by a pending outgoing arrow) and accepts the following language: $\mathcal{L}(\mathcal{A}) = \{(t_1, a) \cdots (t_n, a)(t_{n+1}, b) \mid t_{n+1} < 1\}$.

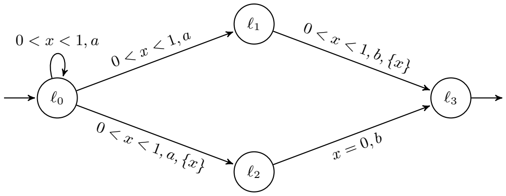


Fig. 1. A timed automaton $\mathcal{A}$

The region abstraction forms a partition of valuations over a given set of clocks. It allows one to make abstractions in order to decide properties like the reachability of a location. We let X be a finite set of clocks, and $M \in \mathbb{N}$. We write $\lfloor t \rfloor$ and $\{t\}$ for the integer part and the fractional part of a real t, respectively. The equivalence relation $\equiv_{X,M}$ over valuations over X is defined as follows: $v \equiv_{X,M} v'$ if (i) for every clock $x \in X$, $v(x) \leq M$ iff $v'(x) \leq M$; (ii) for every clock $x \in X$, if $v(x) \leq M$, then $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ and $\{v(x)\} = 0$ iff $\{v'(x)\} = 0$ and (iii) for every pair of clocks $(x, y) \in X^2$ such that $v(x) \leq M$ and $v(y) \leq M$, $\{v(x)\} \leq \{v(y)\}$ iff $\{v'(x)\} \leq \{v'(y)\}$. The equivalence relation is called the *region equivalence* for the set of clocks X w.r.t. M, and an equivalence class is called a *region*. The set of regions, given X and M, is denoted Reg_M^X. A region r' is a *time-successor* of a region r if there is $v \in r$ and $t \in \mathbb{R}_{\geq 0}$ such that $v + t \in r'$. The set of all time-successors of r is denoted $\overrightarrow{r}$.

In the following, we often abuse notations for guards, invariants, relations and regions, and write g, I, C and r, respectively, for both the constraints over clock variables and the sets of valuations they represent.

2.2 Existing Approaches to the Determinization of TAs

To overcome the non-feasibility of determinization of timed automata in general, two alternatives have been explored: either exhibiting subclasses of timed automata which are determinizable and provide determinization algorithms, or constructing deterministic over-approximations. We relate here, for each of these directions, a recent contribution.

Determinization procedure. An abstract determinization procedure which effectively constructs a deterministic timed automaton for several classes of determinizable timed automata is presented in [4]. Given a timed automaton $\mathcal{A}$, this procedure first produces a language-equivalent infinite timed tree, by unfolding $\mathcal{A}$, introducing a fresh clock at each step. This allows one to preserve all timing constraints, using a mapping from clocks of $\mathcal{A}$ to the new clocks. Then, the infinite tree is split into regions, and symbolically determinized. Under a clock-boundedness assumption, the infinite tree with infinitely many clocks can be folded up into a timed automaton (with finitely many locations and clocks). The clock-boundedness assumption is satisfied for several classes of timed automata, such as event-clock TAs [2], TAs with integer resets [13] and strongly non-Zeno TAs [3], which can thus be determinized by this procedure. The resulting deterministic timed automaton is doubly exponential in the size of $\mathcal{A}$.

Deterministic over-approximation. By contrast, Krichen and Tripakis propose an algorithm applicable to any timed automaton $\mathcal{A}$, which produces a deterministic over-approximation, that is a deterministic TA $\mathcal{B}$ accepting at least all timed words in $\mathcal{L}(\mathcal{A})$ [11]. This TA $\mathcal{B}$ is built by simulation of $\mathcal{A}$ using only information carried by clocks of $\mathcal{B}$. A location of $\mathcal{B}$ is then a state estimate of $\mathcal{A}$ consisting of a (generally infinite) set of pairs (ℓ, v) where ℓ is a location of $\mathcal{A}$ and v a valuation over the union of clocks of $\mathcal{A}$ and $\mathcal{B}$. This method is based on the use of a fixed finite automaton (the *skeleton*) which governs the resetting policy for the clocks of $\mathcal{B}$. The size of obtained deterministic timed automaton $\mathcal{B}$ is also doubly exponential in the size of $\mathcal{A}$.

3 A Game Approach

In [8], given a plant —modeled by a timed automaton— and fixed resources, the authors build a game where some player has a winning strategy if and only if the plant can be diagnosed by a timed automaton using the given resources. Inspired by this construction, given a timed automaton $\mathcal{A}$ and fixed resources, we derive a game between two players Spoiler and Determinizator, such that if Determinizator has a winning strategy, then a deterministic timed automaton $\mathcal{B}$ with $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A})$ can be effectively generated. Moreover, any strategy for Determinizator (winning or not) yields a deterministic over-approximation for $\mathcal{A}$. For simplicity, we present here the method for timed automa without ε-transitions and for which all invariants are true. The general case is presented, for clarity, as extension in Section 4.

3.1 Definition of the Game

Let $\mathcal{A} = (L, \ell_0, F, \Sigma, X, M, E)$ be a timed automaton. We aim at building a deterministic timed automaton $\mathcal{B}$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ if possible, or $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$.

In order to do so, we fix resources (k, M') for $\mathcal{B}$ and build a finite 2-player turn-based safety game $\mathcal{G}_{\mathcal{A},(k,M')}$. Players Spoiler and Determinizator alternate moves, and the objective of player Determinizator is to avoid a set of bad states (to be defined later). Intuitively, in the safe states, for sure, no over-approximation has been performed.

For simplicity, we first detail the approach in the case where $\mathcal{A}$ has no ε-transitions and all invariants are true.

Let Y be a set of clocks of cardinality k. The initial state of the game is a state of Spoiler consisting of location ℓ_0 (initial location of $\mathcal{A}$) together with the simplest relation between X and Y: $\forall x \in X$, $\forall y \in Y, x - y = 0$, and a marking $\top$ (no over-approximation was done so far), together with the null region over Y. In each of its states, Spoiler challenges Determinizator by proposing an M'-bounded region r over Y, and an action $a \in \Sigma$. Determinizator answers by deciding the set of clocks $Y' \subseteq Y$ he wishes to reset. The next state of Spoiler contains a region over Y ($r' = r_{[Y' \leftarrow 0]}$), and a finite set of configurations: triples formed of a location of $\mathcal{A}$, a relation between clocks in X and clocks in Y, and a boolean marking ($\top$ or $\bot$). A state of Spoiler thus constitutes a states estimate of $\mathcal{A}$, and the role of the markings is to indicate whether over-approximations possibly happened. A state of Determinizator is a copy of the preceding states estimate together with the move of Spoiler. Bad states player Determinizator wants to avoid are on the one hand states of the game where all configurations are marked $\bot$ and, on the other hand, states where all final configurations (if any) are marked $\bot$.

Formally, given $\mathcal{A}$ and (k, M') we define $\mathcal{G}_{\mathcal{A},(k,M')} = (\mathsf{V}, \mathsf{v}_0, \mathsf{Act}, \delta, \mathsf{Bad})$ where:

- The set of vertices V is partitioned into V_S and V_D, respectively vertices of Spoiler and Determinizator. Vertices of V_S and V_D are labeled respectively in $2^{L \times \mathsf{Rel}_{M,M'}(X,Y) \times \{\top,\bot\}} \times \mathsf{Reg}^Y_{M'}$ and $2^{L \times \mathsf{Rel}_{M,M'}(X,Y) \times \{\top,\bot\}} \times (\mathsf{Reg}^Y_{M'} \times \Sigma)$;
- $\mathsf{v}_0 = (\{\overline{0}\}, \{(\ell_0, X - Y = 0, \top)\})$ is the initial vertex and belongs to player Spoiler[1];
- $\mathsf{Act} = (\mathsf{Reg}^Y_{M'} \times \Sigma) \cup 2^Y$ is the set of possible actions;
- $\delta \subseteq \mathsf{V}_S \times (\mathsf{Reg}^Y_{M'} \times \Sigma) \times \mathsf{V}_D \cup \mathsf{V}_D \times 2^Y \times \mathsf{V}_S$ is the set of edges;
- $\mathsf{Bad} = \{(\{(\ell_j, C_j, \bot)\}_j, r)\} \cup \{(\{(\ell_j, C_j, b_j)\}_j, r) \mid \{\ell_j\}_j \cap F \neq \emptyset \wedge \forall j,\ \ell_j \in F \Rightarrow b_j = \bot\}$ is the set of bad states.

We now detail the edge relation which defines the possible moves of the players. Given $\mathsf{v}_S = (\{(\ell_j, C_j, b_j)\}_j, r) \in \mathsf{V}_S$ a state of Spoiler and (r', a) one of its moves, the successor state is defined, provided r' is a time-successor of r, as the state $\mathsf{v}_D = (\{(\ell_j, C_j, b_j)\}_j, (r', a)) \in \mathsf{V}_D$ if $\exists (\ell, C, b) \in \{(\ell_j, C_j, b_j)\}_j$ and $\exists \ell \xrightarrow{g,a,X'} \ell' \in E$ s.t. $[r' \cap C]_{|X} \cap g \neq \emptyset$.

Given $\mathsf{v}_D = (\{(\ell_j, C_j, b_j)\}_j, (r', a)) \in \mathsf{V}_D$ a state of Determinizator and $Y' \subseteq Y$ one of its moves, the successor state of v_D is the state $(\mathcal{E}, r'_{[Y' \leftarrow 0]}) \in \mathsf{V}_S$ where $\mathcal{E}$ is obtained as the set of all elementary successors of configurations in

[1] $X - Y = 0$ is a shortcut to denote the relation $\forall x \in X$, $\forall y \in Y$, $x - y = 0$.

$\{(\ell_j, C_j, b_j)\}_j$ by (r', a) and by resetting Y'. Precisely, if (ℓ, C, b) is a configuration, its elementary successors set by (r', a) and Y' is:

$$\mathsf{Succ}_e[r', a, Y'](\ell, C, b) = \left\{ (\ell', C', b') \;\middle|\; \begin{array}{l} \exists \ell \xrightarrow{g,a,X'} \ell' \in E \text{ s.t. } [r' \cap C]_{|X} \cap g \neq \emptyset \\ C' = \mathsf{up}(r', C, g, X', Y') \\ b' = b \wedge ([r' \cap C]_{|X} \cap \neg g = \emptyset) \end{array} \right\}$$

where $\mathsf{up}(r', C, g, X', Y')$ is the update of the relation between clocks in X and Y after the moves of the two players, that is after taking action a in r', resetting $X' \subseteq X$ and $Y' \subseteq Y$, and forcing the satisfaction of g. Formally, $\mathsf{up}(r', C, g, X', Y') = \overleftrightarrow{(r' \cap C \cap g)}_{[X' \leftarrow 0][Y' \leftarrow 0]}$. Boolean b' is set to $\perp$ if either $b = \perp$ or the induced guard $[r' \cap C]_{|X}$ over-approximates g. In the update, the intersection with g aims at stopping runs that for sure will correspond to timed words out of $\mathcal{L}(\mathcal{A})$; the boolean b anyway takes care of keeping track of the possible over-approximation. Region r', relation C and guard g can all be seen as zones (*i.e.* unions of regions) over clocks $X \cup Y$. It is standard that elementary operations on zones, such as intersections, resets, future and past, can be performed effectively. As a consequence, the update of a relation can also be computed effectively.

Given the labeling of states in the game $\mathcal{G}_{\mathcal{A},(k,M')}$, the size of the game is doubly exponential in the size of $\mathcal{A}$. We will see in Subsection 3.3 that the number of edges in $\mathcal{G}_{\mathcal{A},(k,M')}$ can be impressively decreased, since restricting to atomic resets (resets of at most one clock at a time) does not diminish the power of Determinizator.

As an example, the construction of the game is illustrated on the nondeterministic timed automaton $\mathcal{A}$ depicted in Figure 1, for which we construct the associated game $\mathcal{G}_{\mathcal{A},(1,1)}$ represented in Figure 2. Rectangular states belong to Spoiler and circles correspond to states of Determinizator. Note that, for the sake of simplicity, the labels of states of Determinizator are omitted in the picture. Gray states form the set Bad. Let us detail the computation of the successors of the top left state by the move $((0,1), b)$ of Spoiler and moves ($\emptyset$ or $\{y\}$) of Determinizator. To begin with, note that b cannot be fired from ℓ_0 in $\mathcal{A}$, therefore the first configuration has no elementary successor. We then consider the configuration which contains the location ℓ_1. The guard induced by $x - y = 0$ and $y \in (0,1)$ is simply $0 < x < 1$ and the guard of the corresponding transition between ℓ_1 and ℓ_3 in $\mathcal{A}$ is exactly $0 < x < 1$, moreover this transition resets x. As a consequence, the successors states contain a configuration marked $\top$ with location ℓ_3 and, respectively, relations $-1 < x - y < 0$ and $x - y = 0$ for the two different moves of Determinizator. Last, when considering the configuration with location ℓ_2, we obtain elementary successors marked $\perp$. Indeed, the guard induced by this move of Spoiler and the relation $-1 < x - y < 0$ is $-1 < x < 1$ whereas the corresponding guard in $\mathcal{A}$ is $x = 0$. To preserve all timed words accepted by $\mathcal{A}$, we represent these configurations, but they imply over-approximations. Thus the successor states contain a configuration marked

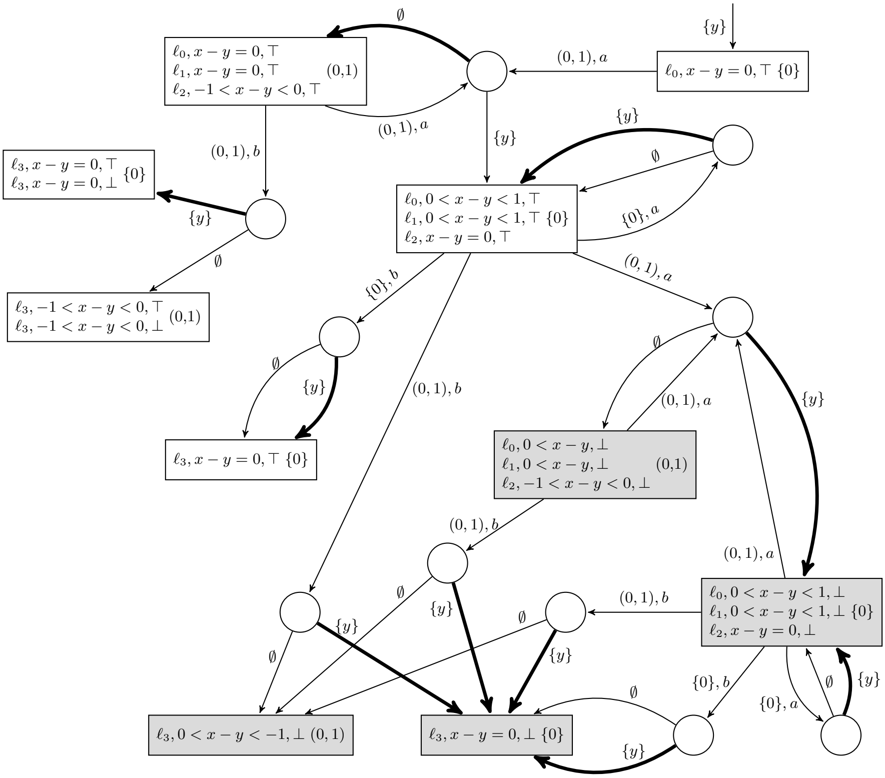


Fig. 2. The game $\mathcal{G}_{\mathcal{A},(1,1)}$ and an example of winning strategy σ for Determinizator

$\bot$ with location ℓ_3 and the same respective relations as before, thanks to the intersection with the initial guard $x = 0$ in $\mathcal{A}$.

3.2 Properties of the Strategies

Given $\mathcal{A}$ a timed automaton and resources (k, M'), the game $\mathcal{G}_{\mathcal{A},(k,M')}$ is a finite-state safety game. It is well known that, for this kind of games, winning strategies can be chosen positional and they can be computed in linear time in the size of the arena [10]. In the following, we simply write strategies for positional strategies. We will see in Subsection 3.3 that positional strategies (winning or not) are indeed sufficient in our framework. A strategy for player Determinizator thus assigns to each state $\mathsf{v}_D \in \mathsf{V}_D$ a set $Y' \subseteq Y$ of clocks to be reset; the successor state is then $\mathsf{v}_S \in \mathsf{V}_S$ such that $(\mathsf{v}_D, Y', \mathsf{v}_S) \in \delta$.

With every strategy for Determinizator σ we associate the timed automaton $\mathsf{Aut}(\sigma)$ obtained by merging a transition of Spoiler with the transition chosen by Determinizator just after, and setting final locations as states of Spoiler containing at least one final location of $\mathcal{A}$. If a strategy σ_S for Spoiler is fixed too, we denote by $\mathsf{Aut}(\sigma, \sigma_S)$ the resulting sub-automaton[2]. The main result of the paper is stated in the following theorem and links strategies of Determinizator with deterministic over-approximations of the initial timed language.

Theorem 1. *Let $\mathcal{A}$ a timed automaton, and $k, M' \in \mathbb{N}$. For every strategy σ of Determinizator in $\mathcal{G}_{\mathcal{A},(k,M')}$, $\mathsf{Aut}(\sigma)$ is a deterministic timed automaton over resources (k, M') and satisfies $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathsf{Aut}(\sigma))$. Moreover, if σ is winning, then $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathsf{Aut}(\sigma))$.*

The full proof is given in the general case (with ε-transitions and invariants in $\mathcal{A}$) in the research report [6]. We however give below its main ideas.

Proof (Sketch of proof). Given a strategy σ, we show that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathsf{Aut}(\sigma))$ by induction on the length of the runs. The induction step is based on the fact that induced guards for $\mathcal{A}$ and relations always are over-approximated in the game. Moreover, if σ is winning, we show that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathsf{Aut}(\sigma))$. Indeed, any safe state ($\mathsf{v} \notin \mathsf{Bad}$) contains at least one configuration marked $\top$. The latter is necessarily obtained from a configuration marked $\top$ without any over-approximation. Hence any run ending in a safe state has an equivalent run in $\mathcal{A}$.

Back to our running example, on Figure 2, a winning strategy for Determinizator is represented by the bold arrows. This strategy yields the deterministic equivalent for $\mathcal{A}$ depicted in Figure 3.

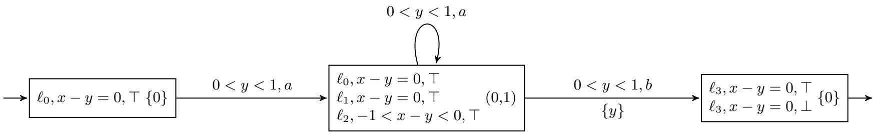


Fig. 3. The deterministic TA $\mathsf{Aut}(\sigma)$ obtained by our construction

Remark 1. Because of the undecidability of the determinizability with fixed resources [14,9], contrary to the diagnosability problem, there is no hope to have a reciprocal statement to the one of Theorem 1 in the following sense: if $\mathcal{A}$ can be determinized with resources (k, M') then Determinizator has a winning strategy in $\mathcal{G}_{\mathcal{A},(k,M')}$. Figure 4 illustrates this phenomenon by presenting a timed automaton $\mathcal{A}$ which is determinizable with resources $(1, 1)$, but for which all strategies for Determinizator in $\mathcal{G}_{\mathcal{A},(1,1)}$ are losing. Intuitively the self loop on ℓ_0 forces Determinizator to reset the clock in his first move; afterwards on each branch of the automaton (passing through ℓ_1, ℓ_2 or ℓ_3) the behavior of $\mathcal{A}$ is strictly over-approximated in the game. However, since these over-approximations cover each others, this losing strategy yields a deterministic equivalent to $\mathcal{A}$.

[2] In the case where σ and/or σ_S have arbitrary memory, we abuse notation and write $\mathsf{Aut}(\sigma)$ and $\mathsf{Aut}(\sigma, \sigma_S)$ for the resulting potentially infinite objects.

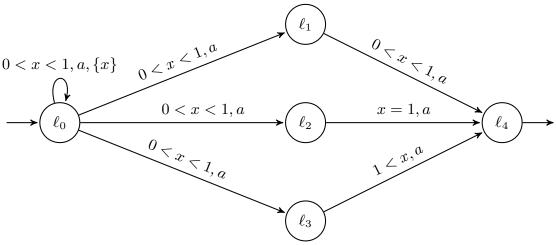


Fig. 4. A determinizable TA for which there is no winning strategy for Determinizator

3.3 Choosing a Good Losing Strategy

Standard techniques allow one to check whether there is a winning strategy for Determinizator, and in the positive case, extract such a strategy [10]. However, if Determinizator has no winning strategy to avoid the set of bad states, it is of interest to be able to choose a good losing strategy. To this aim, we introduce a natural partial order over the set of strategies of Determinizator based on the distance to the set Bad: $d_{\mathsf{Bad}}(\mathcal{A})$ denotes the minimal number of steps in some automaton $\mathcal{A}$ to reach Bad from the initial state.

Definition 2. *Let σ_1 and σ_2 be strategies of Determinizator in $\mathcal{G}_{\mathcal{A},(k,M')}$. Strategy σ_1 is said* finer *than σ_2, denoted $\sigma_1 \ll \sigma_2$, if for every strategy σ_S of Spoiler, $d_{\mathsf{Bad}}(\mathsf{Aut}(\sigma_1, \sigma_S)) \geq d_{\mathsf{Bad}}(\mathsf{Aut}(\sigma_2, \sigma_S))$.*

Given this definition, an optimal strategy for Determinizator is a minimal element for the partial order $\ll$. Note that, if they exist, winning strategies are the optimal ones since against all strategies of Spoiler, the corresponding distance to Bad is infinite. The set of optimal strategies can be computed effectively by a fix-point computation using a rank function on the vertices of the game.

With respect to this partial order on strategies, positional strategies are sufficient for Determinizator.

Proposition 1. *For every strategy σ of Determinizator with arbitrary memory, there exists a positional strategy σ' such that $\sigma' \ll \sigma$.*

Strategy σ' is obtained from σ by letting for each state the first choice made in σ; this cannot decrease the distance to Bad. Strategies of interest for Determinizator can be even more restricted. Indeed, any timed automaton can be turned into an equivalent one with atomic resets only, using a construction similar to the one that removes clock transfers (updates of the form $x := x'$) [7]. Thus, for every strategy for Determinizator there is finer one which resets at most one clock on each transition, which can be turned into a finer positional strategy thanks to Proposition 1. As a consequence, with respect to $\ll$, positional strategies that only allow for atomic resets are sufficient for Determinizator.

4 Extension to ε-Transitions and Invariants

In Section 3 the construction of the game and its properties were presented for a restricted class of timed automata. Let us now briefly explain how to extend the previous construction to deal with ε-transitions and invariants. The extension is presented in details in [6].

ε-transitions. We aim at building an over-approximation without ε-transitions. An ε-closure is performed for each state during the construction of the game. To this attempt, states of the game have to be extended since ε-transitions might be enabled only from some time-successors of the region associated with the state. Therefore, each configuration is associated with a proper region which is a time-successor of the initial region of the state. The ε-closure is effectively computed the same way as successors in the original construction when Determinizator does not reset any clock; computations thus terminate for the same reasons.

Invariants. Ignoring all invariants surely yields an over-approximation. In order to be more precise (while preserving the over-approximation) with each state of the game is associated the most restrictive invariant which contains invariants of all the configurations in the state. In the computation of the successors, invariants are treated similarly to guards and their validity is verified at the transition's target. A state whose invariant is strictly over-approximated is not safe.

5 Comparison with Existing Methods

The method we presented is both more precise than the algorithm of [11] and more general than the procedure of [4]. Let us detail these two points. Note that a deeper comparison with existing work can be found in [6].

5.1 Comparison with [11]

First of all, our method covers the application area of [11] since each time the latter algorithm produces a deterministic equivalent with resources (k, M') for a timed automaton $\mathcal{A}$, there is a winning strategy for Determinizator in $\mathcal{G}_{\mathcal{A},(k,M')}$.

Moreover, contrary to the method presented in [11], our game-approach is exact on deterministic timed automata: given a DTA $\mathcal{A}$ over resources (k, M), Determinizator has a winning strategy in $\mathcal{G}_{\mathcal{A},(k,M)}$. This is a consequence of the more general fact that, in our approach, a winning strategy can be seen as a timed generalization of the notion of skeleton [11], and solving our game amounts to finding a relevant timed skeleton.

As an example, the algorithm of [11] run on the timed automaton of Figure 1 produces a strict over-approximation, represented on Figure 5.

Our approach also improves the updates of the relations between clocks by taking the original guard into account. Precisely, when computing up_S, an intersection with the guard in the original TA is performed. This improvement allows one, even under the same resetting policy, to refine the over-approximation given by [11].

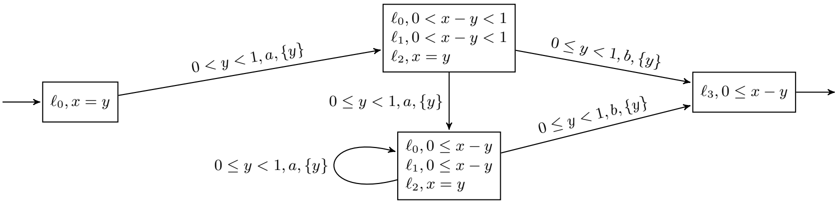


Fig. 5. The result of algorithm [11] on the running example

5.2 Comparison with [4]

Our approach generalizes the one in [4] since, for any timed automaton $\mathcal{A}$ such that the procedure in [4] yields an equivalent deterministic timed automaton with k clocks and maximal constant M', there is a winning strategy for Determinizator in $\mathcal{G}_{\mathcal{A},(k,M')}$. This can be explained by the fact that relations between clocks of $\mathcal{A}$ and clocks in the game allow one to record more information than the mapping used in [4]. Moreover, our approach strictly broadens the class of automata determinized by the procedure of [4] in two respects. First of all, our method allows one to cope with some language inclusions, contrary to [4]. For example, the TA depicted on the left-hand side of Figure 6 cannot be treated by the procedure of [4] but is easily determinized using our approach. In this example, the language of timed words accepted in location ℓ_3 is not determinizable. This will cause the failure of [4]. However, all timed words accepted in ℓ_3 also are accepted in ℓ_4 and the language of timed words accepted in ℓ_4 is clearly determinizable. Our approach allows one to deal with such language inclusions, and will thus provide an equivalent deterministic timed automaton. Second, the relations between clocks of the TA and clocks of the game are more precise than the mapping used in [4], since the mapping can be seen as restricted relations: a conjunction of constraints of the form $x - y = 0$. The precision we add by considering relations rather than mappings is sometimes crucial for the determinization. For example, the TA represented on the right-hand side of Figure 6 can be determinized by our game-approach, but not by [4].

Apart from strictly broadening the class of timed automata that can be automatically determinized, our approach performs better on some timed automata by providing a deterministic timed automaton with less resources. This is the case on the running example of Figure 1. The deterministic automaton obtained

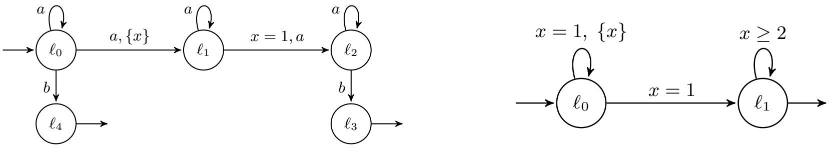


Fig. 6. Examples of determinizable TAs not treatable by [4]

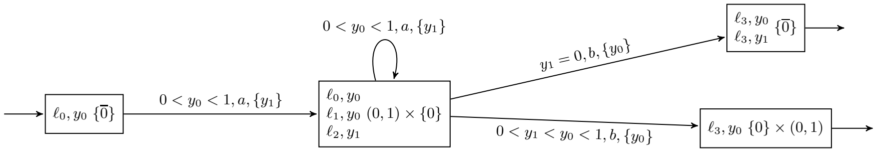

$0 < y_0 < 1, a, \{y_1\}$
ℓ_3, y_0
ℓ_3, y_1 $\{\overline{0}\}$
$y_1 = 0, b, \{y_0\}$
ℓ_0, y_0 $\{\overline{0}\}$
$0 < y_0 < 1, a, \{y_1\}$
ℓ_0, y_0
ℓ_1, y_0 $(0,1) \times \{0\}$
ℓ_2, y_1
$0 < y_1 < y_0 < 1, b, \{y_0\}$
ℓ_3, y_0 $\{0\} \times (0,1)$

Fig. 7. The result of procedure [4] on the running example

by [4] is depicted in Figure 7: it needs 2 clocks when our method produces a single-clock TA.

The same phenomenon happens with timed automata with integer resets. Timed automata with integer resets, introduced in [13], form a determinizable subclass of timed automata, where every edge (ℓ, g, a, X', ℓ') satisfies $X' \neq \emptyset$ if and only if g contains an atomic constraint of the form $x = c$ for some clock x.

Proposition 2. *For every timed automaton $\mathcal{A}$ with integer resets and maximal constant M, Determinizator has a winning strategy in $\mathcal{G}_{\mathcal{A},(1,M)}$.*

Intuitively, a single clock is needed to represent clocks of $\mathcal{A}$ since they all share a common fractional part. A complete proof is given in [6].

As a consequence of Proposition 2, any timed automaton with integer resets can be determinized into a doubly exponential single-clock timed automaton with the same maximal constant. This improves the result given in [4] where any timed automaton with integer resets and maximal constant M can be turned into a doubly exponential deterministic timed automaton, using $M + 1$ clocks. Moreover, our procedure is optimal on this class thanks to the lower-bound provided in [12].

Last, our method even when restricted to equality relations (conjunctions of constraints of the form $x - y = c$) extends the procedure of [4], whose effective construction when the number of new clocks is fixed, is detailed in the technical report [5]. Note that the latter construction is similar to our approach restricted to mappings instead of relations. We detail in [6] the benefits of (even equality) relations and explain how the sufficient conditions for termination provided in [4] can be weakened in our context.

5.3 Comparison of the Extension with ε-Transition and Invariants

Let us now compare our extended approach with the approach of [11] since the determinization procedure of [4] does not deal with invariants and ε-transitions.

The model in [11] consists of timed automata with silent transitions and actions are classified depending on their urgency: eager, lazy or delayable. First of all, the authors propose an ε-closure computation which does not terminate in general, and bring up the fact that termination can be ensured by some abstraction. Second, the urgency in the model is not preserved by their over-approximation construction which only produces lazy transitions. Note that we classically decided to use invariants to model urgency, but our approach could be adapted to the same model as the one they use, while preserving urgency much more often, the same way as we do for invariants.

6 Conclusion

In this paper, we proposed a game-based approach for the determinization of timed automata. Given a timed automaton $\mathcal{A}$ (with ε-transitions and invariants) and resources (k, M), we build a finite turn-based safety game between two players Spoiler and Determinizator, such that any strategy for Determinizator yields a deterministic over-approximation of the language of $\mathcal{A}$ and any winning strategy provides a deterministic equivalent for $\mathcal{A}$. Our construction strictly covers and improves two existing approaches [11,4].

In the research report [6] we detail how to adapt the framework to generate deterministic under-approximations, or even deterministic approximations combining under- and over-approximations. The motivation for this generalization is to tackle the problem of off-line model-based test generation for non-deterministic timed automata specifications.

In future work, we plan to investigate how our game-based approach proposed here can be applied to other models and/or other problems.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Alur, R., Fix, L., Henzinger, T.A.: A determinizable class of timed automata. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 1–13. Springer, Heidelberg (1994)
3. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: Proceedings of the 5th IFAC Symposium on System Structure and Control (SSSC 1998), pp. 469–474. Elsevier Science, Amsterdam (1998)
4. Baier, C., Bertrand, N., Bouyer, P., Brihaye, T.: When are timed automata determinizable? In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 43–54. Springer, Heidelberg (2009)
5. Baier, C., Bertrand, N., Bouyer, P., Brihaye, T.: When are timed automata determinizable? Research Report LSV-09-08, Laboratoire Spécification et Vérification, ENS Cachan, France, April 2009, 32 pages (2009)
6. Bertrand, N., Stainer, A., Jéron, T., Krichen, M.: A game approach to determinize timed automata. Research Report 7381, INRIA (September 2010), http://hal.inria.fr/inria-00524830
7. Bouyer, P.: From Qualitative to Quantitative Analysis of Timed Systems. Mémoire d'habilitation, Université Paris 7, Paris, France (January 2009)
8. Bouyer, P., Chevalier, F., D'Souza, D.: Fault diagnosis using timed automata. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 219–233. Springer, Heidelberg (2005)
9. Finkel, O.: Undecidable problems about timed automata. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 187–199. Springer, Heidelberg (2006)

10. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games: A Guide to Current Research. LNCS, vol. 2500. Springer, Heidelberg (2002)
11. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Formal Methods in System Design 34(3), 238–304 (2009)
12. Manasa, L., Krishna, S.N.: Integer reset timed automata: Clock reduction and determinizability. CoRR arXiv:1001.1215v1 (2010)
13. Suman, P.V., Pandya, P.K., Krishna, S.N., Manasa, L.: Timed automata with integer resets: Language inclusion and expressiveness. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 78–92. Springer, Heidelberg (2008)
14. Tripakis, S.: Folk theorems on the determinization and minimization of timed automata. Information Processing Letters 99(6), 222–226 (2006)

A Practical Linear Time Algorithm for Trivial Automata Model Checking of Higher-Order Recursion Schemes

Naoki Kobayashi

Tohoku University

Abstract. The model checking of higher-order recursion schemes has been actively studied and is now becoming a basis of higher-order program verification. We propose a new algorithm for trivial automata model checking of higher-order recursion schemes. To our knowledge, this is the first practical model checking algorithm for recursion schemes that runs in time linear in the size of the higher-order recursion scheme, under the assumption that the size of trivial automata and the largest order and arity of functions are fixed. The previous linear time algorithm was impractical due to a huge constant factor, and the only practical previous algorithm suffers from the hyper-exponential worst-case time complexity, under the same assumption. The new algorithm is remarkably simple, consisting of just two fixed-point computations. We have implemented the algorithm and confirmed that it outperforms Kobayashi's previous algorithm in a certain case.

1 Introduction

The model checking of higher-order recursion schemes [9,20] (higher-order model checking, for short) has been studied extensively, and recently applied to higher-order program verification [12,10,18,17,21]. A higher-order recursion scheme [9,20] is a grammar for describing a possibly infinite tree, and the higher-order model checking is concerned about whether the tree described by a given higher-order recursion scheme satisfies a given property (typically expressed by the modal μ-calculus or tree automata). Higher-order model checking can be considered a generalization of finite-state and pushdown model checking. Just as software model checkers for procedural languages like BLAST [4] and SLAM [3] have been constructed based on finite-state and pushdown model checking, one may hope to construct software model checkers for higher-order functional languages based on higher-order model checking. Some evidence for such a possibility has been provided recently [12,10,18,17,21].

The main obstacle in applying higher-order model checking to program verification is its extremely high worst-case complexity: n-EXPTIME completeness [20] (where n is the order of a given higher-order recursion scheme). Kobayashi and Ong [12,16] showed that, under the assumption that the size of properties and the largest order and arity of functions are fixed, the model

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 260–274, 2011.

checking is actually linear time in the size of the higher-order recursion scheme for the class of properties described by trivial automata [2], and polynomial time for the full modal μ-calculus. Their algorithms were however of only theoretical interest; because of a huge constant factor (which is n-fold exponential in the other parameters), they are runnable only for recursion schemes of order 2 at highest.

The only practical algorithm known to date is Kobayashi's hybrid algorithm [10], used in the first higher-order model checker TRecS [11]. According to experiments, the algorithm runs remarkably fast in practice, considering the worst-case complexity of the problem. The worst-case complexity of the hybrid algorithm is, however, actually worse than Kobayashi's naïve algorithm [12]: Under the same assumption that the other parameters are fixed, the worst-case time complexity of the hybrid algorithm [10] is still hyper-exponential in the size of the recursion scheme. In fact, one can easily construct a higher-order recursion scheme for which the hybrid algorithm suffers from an n-EXPTIME bottleneck in the size of the recursion scheme. Thus, it remained as a question whether there is a practical algorithm that runs in time polynomial in the size of the higher-order recursion scheme. The question is highly relevant for applications to program verification [12,18], as the size of a higher-order recursion scheme corresponds to the size of a program.

The present paper proposes the first (arguably) *practical*, *linear time*[1] algorithm for trivial automata model checking of recursion schemes (i.e. the problem of deciding whether the tree generated by a given recursion scheme $\mathcal{G}$ is accepted by a given trivial automaton $\mathcal{B}$). Like Kobayashi and Ong's previous algorithms [12,10,16], the new algorithm is based on a reduction of model checking to intersection type inference, but the algorithm has also been inspired by game semantics [20,1] (though the game semantics is not explicitly used). The resulting algorithm is remarkably simple, consisting of just two fixedpoint computations. We have implemented the new algorithm, and confirmed that it outperforms Kobayashi's hybrid algorithm [10] in a certain case. Another advantage of the new algorithm is that it works for *non-deterministic* trivial automata, unlike the hybrid algorithm (which works only for deterministic trivial automata).

Unfortunately, the current implementation of the new algorithm is significantly slower than the hybrid algorithm [10] (which already incorporates a number of optimizations) in most cases. However, the new algorithm provides a hope that, with further optimizations, one may eventually obtain a model checking algorithm that scales to large programs (because of the fixed-parameter linear time complexity).

The rest of this paper is structured as follows. Section 2 reviews higher-order recursion schemes and previous type-based model checking algorithms for recursion schemes. Section 3 presents a new model checking algorithm, and Section 4 proves the correctness of the algorithm. Section 5 reports preliminary experiments. Section 6 discusses related work and Section 7 concludes.

[1] Under the same assumption as [12,16] that the other parameters are fixed.

2 Preliminaries

We write $\mathit{dom}(f)$ for the domain of a map f. We write $\widetilde{x}$ for a sequence $x_1, \ldots, x_k$, and write $[t_1/x_1, \ldots, t_k/x_k]u$ for the term obtained from u by replacing $x_1, \ldots, x_k$ in u with $t_1, \ldots, t_k$. A Σ-labeled tree (where Σ is a set of symbols), written T, is a map from $[m]^*$ (where m is a positive integer and $[m] = \{1, \ldots, m\}$) to Σ such that (i) $\epsilon \in \mathit{dom}(T)$, (ii) $xi \in \mathit{dom}(T)$ implies $\{x, x1, \ldots, x(i-1)\} \subseteq \mathit{dom}(T)$ for any $x \in [m]^*$ and $i \in [m]$.

Higher-Order Recursion Schemes. A higher-order recursion scheme [20] is a tree grammar for generating an infinite tree, where non-terminal symbols can take parameters. To preclude illegal parameters, each non-terminal symbol has a sort. The set of *sorts* is given by: $\kappa ::= \mathtt{o} \mid \kappa_1 \to \kappa_2$. Intuitively, the sort $\mathtt{o}$ describes trees, and the sort $\kappa_1 \to \kappa_2$ describes functions that take an element of sort κ_1 as input, and return an element of sort κ_2. The arity and order are defined by:

$$\begin{array}{ll} \mathit{arity}(\mathtt{o}) = 0 & \mathit{arity}(\kappa_1 \to \kappa_2) = 1 + \mathit{arity}(\kappa_2) \\ \mathit{order}(\mathtt{o}) = 0 & \mathit{order}(\kappa_1 \to \kappa_2) = \mathit{max}(1 + \mathit{order}(\kappa_1), \kappa_2) \end{array}$$

Formally, a *higher-order recursion scheme* (recursion scheme, for short) is a quadruple $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$, where Σ is a set of symbols called *terminals* and $\mathcal{N}$ is a set of symbols called *non-terminals*. Each terminal or non-terminal α has an associated sort, denoted by $\mathit{sort}(\alpha)$. The order of the sort of a terminal must be 0 or 1. We write $\mathit{arity}(\alpha)$ for $\mathit{arity}(\mathit{sort}(\alpha))$ and call it the arity of α. (Thus, Σ is a ranked alphabet.) $\mathcal{R}$, called *rewriting rules*, is a finite map from $\mathcal{N}$ to λ-terms of the form $\lambda\widetilde{x}.t$ where t is an applicative term constructed from the variables $\widetilde{x}$, terminals, and non-terminals. $\mathcal{R}(F)$ must have sort $\mathit{sort}(F)$ (under the standard simple type system). S is a special non-terminal called the *start symbol*. The *order* of a recursion scheme $\mathcal{G}$ is the largest order of the sorts of its non-terminals. We use lower letters for terminals, and upper letters for non-terminals.

Given a recursion scheme $\mathcal{G}$, the rewriting relation $\longrightarrow_{\mathcal{G}}$ is the least relation that satisfies: (i) $F\, u_1 \ldots, u_k \longrightarrow_{\mathcal{G}} [u_1/x_1, \ldots, u_k/x_k]t$ if $\mathcal{R}(F) = \lambda x_1. \cdots \lambda x_k.t$, and (ii) $a\, t_1 \cdots t_n \longrightarrow_{\mathcal{G}} a\, t_1 \cdots t_{i-1}\, t_i'\, t_{i+1} \cdots t_n$ if $t_i \longrightarrow_{\mathcal{G}} t_i'$.[2] The *value tree* of $\mathcal{G}$, written $[\![\mathcal{G}]\!]$, is the (possibly infinite) $(\Sigma \cup \{\bot\})$-labelled tree generated by a fair, maximal reduction sequence from S. More precisely, define $t^{\bot}$ by $(a\, t_1 \cdots t_k)^{\bot} = a\, {t_1}^{\bot} \cdots {t_k}^{\bot}$, and $(F\, t_1 \cdots t_k)^{\bot} = \bot$. $[\![\mathcal{G}]\!]$ is $\bigsqcup\{t^{\bot} \mid S \longrightarrow_{\mathcal{G}}^{*} t\}$, where $\bigsqcup$ is the least upper bound with respect to the least compatible (i.e. closed under contexts) relation $\sqsubseteq$ on trees that satisfies $\bot \sqsubseteq T$ for every tree T.

Example 1. Consider the recursion scheme $\mathcal{G}_0 = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{S, F\}, \mathcal{R}, S)$, where $\mathcal{R}$ and the sorts of symbols are given by:

$$\begin{array}{l} \mathcal{R} = \{S \mapsto F\,\mathtt{b}\,\mathtt{c}, \quad F \mapsto \lambda f.\lambda x.(\mathtt{a}\,(f\,x)\,(F\,f\,(f\,x)))\} \\ \mathtt{a} : \mathtt{o} \to \mathtt{o} \to \mathtt{o},\ \mathtt{b} : \mathtt{o} \to \mathtt{o},\ \mathtt{c} : \mathtt{o},\ S : \mathtt{o},\ F : (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o} \end{array}$$

[2] Note that we allow only reductions of outermost redexes.

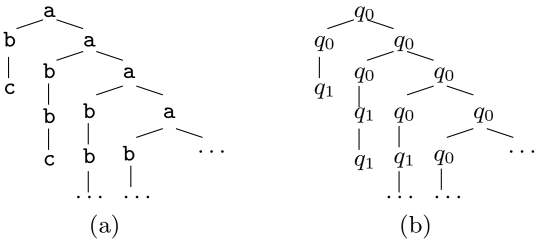


Fig. 1. The tree generated by the recursion scheme $\mathcal{G}_0$ and a run tree of it

S is rewritten as follows, and the tree in Figure 1(a) is generated.

$$S \longrightarrow F\,\mathtt{b}\,\mathtt{c} \to \mathtt{a}\,(\mathtt{b}\,\mathtt{c})\,(F\,\mathtt{b}\,(\mathtt{b}\,\mathtt{c})) \to \mathtt{a}\,(\mathtt{b}\,\mathtt{c})\,(\mathtt{a}\,(\mathtt{b}(\mathtt{b}\,\mathtt{c}))\,(F\ \mathtt{b}\ (\mathtt{b}\,(\mathtt{b}\,\mathtt{c})))) \to \cdots$$

Trivial Automata Model Checking. The aim of model-checking a higher-order recursion scheme is to check whether the tree generated by the recursion scheme satisfies a certain regular property. In the present paper, we consider the properties described by *trivial automata* [2], which are sufficient for program verification problems considered in [12,18].

A *trivial automaton* $\mathcal{B}$ is a quadruple (Σ, Q, Δ, q_0), where Σ is a set of input symbols, Q is a finite set of states, $\Delta \subseteq Q \times \Sigma \times Q^*$ is a transition function, and q_0 is the initial state. A Σ-labeled tree T is *accepted* by $\mathcal{B}$ if there is a Q-labeled tree R (called a *run tree*) such that: (i) $dom(T) = dom(R)$; (ii) $R(\epsilon) = q_0$; and (iii) for every $x \in dom(R)$, $(R(x), T(x), R(x1) \cdots R(xm)) \in \Delta$ where $m = arity(T(x))$. For a trivial automaton $\mathcal{B} = (\Sigma, Q, \Delta, q_0)$ (with $\perp \notin \Sigma$), we write $\mathcal{B}^{\perp}$ for the trivial automaton $(\Sigma \cup \{\perp\}, Q, \Delta \cup \{(q, \perp, \epsilon) \mid q \in Q\}, q_0)$.

The *trivial automata model checking* is the problem of deciding whether $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^{\perp}$, given a recursion scheme $\mathcal{G}$ and a trivial automaton $\mathcal{B}$.[3]

Example 2. Consider the trivial automaton $\mathcal{B}_0 = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{q_0, q_1\}, \Delta, q_0)$, where $\Delta = \{(q_0, \mathtt{a}, q_0 q_0), (q_0, \mathtt{b}, q_1), (q_1, \mathtt{b}, q_1), (q_0, \mathtt{c}, \epsilon), (q_1, \mathtt{c}, \epsilon)\}$. $\mathcal{B}_0$ accepts a $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$-labeled ranked tree just if $\mathtt{a}$ does not occur below $\mathtt{b}$. The tree generated by $\mathcal{G}_0$ of Example 1 is accepted by $\mathcal{B}_0$. The run tree is shown in Figure 1(b).

Type Systems Equivalent to Trivial Automata Model Checking. One can construct an intersection type system (parameterized by a trivial automaton $\mathcal{B} = (\Sigma, Q, \Delta, q_0)$) that is equivalent to trivial automata model checking, in the sense that a recursion scheme $\mathcal{G}$ is well typed if, and only if, $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^{\perp}$ [12]. Define the set of types by:

$$\theta \text{ (atomic types) } ::= q \mid \tau \to \theta \qquad \tau \text{ (intersections) } ::= \bigwedge\{\theta_1, \ldots, \theta_m\}$$

Here, q ranges over the states of $\mathcal{B}$. We often write $\theta_1 \wedge \cdots \wedge \theta_m$ or $\bigwedge_{i \in \{1,\ldots,m\}} \theta_i$ for $\bigwedge\{\theta_1, \ldots, \theta_m\}$. We also write $\top$ for $\bigwedge \emptyset$, and θ for $\bigwedge\{\theta\}$. $\bigwedge$ binds tighter

[3] In the literature [20,16], it is often assumed that the value tree of $\mathcal{G}$ does not contain $\perp$. Under that assumption, the acceptance by $\mathcal{B}^{\perp}$ and $\mathcal{B}$ are equivalent.

than →. Intuitively, q is a refinement of sort $\mathtt{o}$, describing trees accepted by $\mathcal{B}$ with the initial state replaced by q. $\theta_1 \wedge \cdots \wedge \theta_m \rightarrow \theta$ describes functions that take an element that has types $\theta_1, \ldots, \theta_m$ as input, and return an element of type θ. For example, $q_0 \wedge q_1 \rightarrow q_0$ describes a function that takes a tree that can be accepted from both q_0 and q_1, and returns a tree accepted from q_0. We define the refinement relation $\theta :: \kappa$ inductively by: (i) $q :: \mathtt{o}$ for every $q \in Q$ and (ii) $(\bigwedge_{i \in S} \theta_i \rightarrow \theta) :: (\kappa_1 \rightarrow \kappa_2)$ if $\forall i \in S. \theta_i :: \kappa_1$ and $\theta :: \kappa_2$.

The typing rules for terms and rewriting rules are given as follows.

$$\frac{(q, a, q_1 \cdots q_k) \in \Delta}{\Gamma \vdash^{\mathcal{B}} a : q_1 \rightarrow \cdots q_k \rightarrow q} \qquad \frac{x : \theta \in \Gamma}{\Gamma \vdash^{\mathcal{B}} x : \theta} \qquad \frac{\forall i \in S.(\Gamma \vdash^{\mathcal{B}} t : \theta_i)}{\Gamma \vdash^{\mathcal{B}} t : \bigwedge_{i \in S} \theta_i}$$

$$\frac{\Gamma \vdash^{\mathcal{B}} t_1 : \tau \rightarrow \theta \quad \Gamma \vdash^{\mathcal{B}} t_2 : \tau}{\Gamma \vdash^{\mathcal{B}} t_1 t_2 : \theta} \qquad \frac{\Gamma, x : \theta_1, \ldots, x : \theta_m \vdash^{\mathcal{B}} t : \theta \quad x \notin dom(\Gamma)}{\Gamma \vdash^{\mathcal{B}} \lambda x.t : \theta_1 \wedge \cdots \wedge \theta_m \rightarrow \theta} \qquad \frac{dom(\Gamma) \subseteq dom(\mathcal{R}) \quad \forall (F : \theta) \in \Gamma.(\Gamma \vdash^{\mathcal{B}} \mathcal{R}(F) : \theta)}{\vdash^{\mathcal{B}} \mathcal{R} : \Gamma}$$

Here, Γ is a set of bindings of the form $x : \theta$ where non-terminals are also treated as variables, and Γ may contain more than one binding for each variable. We write $dom(\Gamma)$ for $\{x \mid x : \theta \in S\}$. A recursion scheme $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ is well typed under Γ, written $\vdash^{\mathcal{B}} \mathcal{G} : \Gamma$, if $\vdash^{\mathcal{B}} \mathcal{R} : \Gamma$, $\forall (F : \theta) \in \Gamma.(\theta :: sort(F))$, and $S : q_0 \in \Gamma$. We write $\vdash^{\mathcal{B}} \mathcal{G}$ if $\vdash^{\mathcal{B}} \mathcal{G} : \Gamma$ for some Γ.

The following theorem guarantees the correspondence between model checking and type checking.

Theorem 1 (Kobayashi [12]). *$[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^{\perp}$ if and only if $\vdash^{\mathcal{B}} \mathcal{G}$.*

Example 3. Recall the recursion scheme $\mathcal{G}_0$ in Example 1 and the trivial automaton $\mathcal{B}_0$ in Example 2. $\vdash^{\mathcal{B}_0} \mathcal{G}_0 : \Gamma$ holds for $\Gamma = \{S : q_0, F : (q_1 \rightarrow q_1) \wedge (q_1 \rightarrow q_0) \rightarrow q_1 \rightarrow q_0\}$

Theorem 1 above yields a straightforward, fixedpoint-based model checking algorithm. Let $\mathbf{Shrink}_{\mathcal{G},\mathcal{B}}$ be the function on type environments defined by: $\mathbf{Shrink}_{\mathcal{G},\mathcal{B}}(\Gamma) = \{F : \theta \in \Gamma \mid \Gamma \vdash^{\mathcal{B}} \mathcal{R}(F) : \theta\}$, and let $\Gamma_{\mathbf{max}}$ be $\{F : \theta \mid F \in \mathcal{N}, \theta :: sort(F)\}$. Then, by the definition, $\vdash^{\mathcal{B}} \mathcal{G} : \Gamma$ if and only if there exists $\Gamma \subseteq \Gamma_{\mathbf{max}}$ such that $\Gamma \subseteq \mathbf{Shrink}_{\mathcal{G},\mathcal{B}}(\Gamma)$ and $S : q_0 \in \Gamma$. (Note that $\Gamma \subseteq \mathbf{Shrink}_{\mathcal{G},\mathcal{B}}(\Gamma)$ if and only if $\vdash^{\mathcal{B}} \mathcal{R} : \Gamma$.) Thus, to check whether $\vdash^{\mathcal{B}} \mathcal{G}$ holds, it is sufficient to compute the greatest fixedpoint $\Gamma_{\mathbf{gfp}}$ of $\mathbf{Shrink}_{\mathcal{G},\mathcal{B}}$ and checks whether $S : q_0 \in \Gamma_{\mathbf{gfp}}$. This is Kobayashi's naïve algorithm.

```
NAIVE ALGORITHM [12]:
1. $\Gamma$ := $\Gamma_{\mathbf{max}}$;
2. Repeat $\Gamma$ := $\mathbf{Shrink}_{\mathcal{G},\mathcal{B}}(\Gamma)$ until $\Gamma = \mathbf{Shrink}_{\mathcal{G},\mathcal{B}}(\Gamma)$;
3. Output whether $S : q_0 \in \Gamma$.
```

Suppose that the size of $\mathcal{B}$ and the largest size of sorts of symbols are fixed. Then, the size of $\Gamma_{\mathbf{max}}$ is linear in the size of $\mathcal{G}$, since for a given κ, the number of types that satisfy $\theta :: \kappa$ is bounded above by a constant. Thus, using Rehof and Mogensen's optimization [22], the above algorithm is made linear in the size of $\mathcal{G}$. The algorithm does not work in practice, however, as the constant factor is too large. Even at the first iteration of the fixedpoint computation, we need to pick each binding $F:\theta$ from $\Gamma_{\mathbf{max}}$ and check whether $\Gamma_{\mathbf{max}} \vdash^{\mathcal{B}} \mathcal{R}(F):\theta$ holds. This is impractical, as $\Gamma_{\mathbf{max}}$ is too large; In fact, even when $|Q|=2$, for a symbol of sort $((\mathtt{o} \rightarrow \mathtt{o}) \rightarrow \mathtt{o}) \rightarrow \mathtt{o}$, the number of corresponding types is $2^{513} \approx 10^{154}$.

Kobayashi's hybrid algorithm [10] starts the greatest fixedpoint computation from a type environment much smaller than $\Gamma_{\mathbf{max}}$. To find an appropriate startpoint of the fixedpoint computation, his algorithm reduces the recursion scheme a finite number of times, and infers candidates of the types of each non-terminal, by observing how each non-terminal is used in the reduction sequence. The following is an outline of the hybrid algorithm (see [10] for more details):

```
HYBRID ALGORITHM [10]:
1. Reduce $S$ a finite number of steps;
2. If a property violation is found, output 'no' and halt;
3. $\Gamma$ := type bindings extracted from the reduction sequence;
4. Repeat $\Gamma$ := $\mathbf{Shrink}_{\mathcal{G},\mathcal{B}}(\Gamma)$ until $\Gamma = \mathbf{Shrink}_{\mathcal{G},\mathcal{B}}(\Gamma)$;
5. If $S:q_0 \in \Gamma$ then output 'yes' and halt;
6. Go back to 1 and reduce $S$ further.
```

The algorithm works well in practice [10,18], but has some limitations: (i) No theoretical guarantee that the algorithm is efficient. In fact, the worst-case running time is hyper-exponential [13]: see Section 5. (ii) The efficiency of the algorithm crucially depends on the selection of terms to be reduced in Step 1. The implementation relies on heuristics for choosing reduced terms, and there is no theoretical justification for it. (iii) It works only for *deterministic* trivial automata (i.e. trivial automata such that $|\Delta \cap \{q\} \times \{a\} \times Q^*| \leq 1$ for every $q \in Q, a \in \Sigma$.) Though it is possible to extend the algorithm to remove the restriction, it is unclear whether the resulting algorithm is efficient in practice.

The limitations above motivated us to look for yet another algorithm, which is efficient *both in practice and in theory* (where an important criterion for the latter is that the time complexity should be linear in the size of the recursion scheme, under the assumption that the other parameters are fixed). That is the subject of this paper, discussed in the following sections.

3 The New Model Checking Algorithm

3.1 Main Idea

In the previous section, a reader may have wondered why we do not compute the *least* fixedpoint, instead of the *greatest* one. The naïve least fixedpoint computation however does not work. Let us define $\mathcal{F}_{\mathcal{B}}$ by: $\mathcal{F}_{\mathcal{B}}(\Gamma) = \{F:\theta \mid \Gamma \vdash^{\mathcal{B}} \mathcal{R}(F):\theta\}$. How about computing the least fixedpoint $\Gamma_{\mathbf{lfp}}$ of $\mathcal{F}_{\mathcal{B}}$, and checking whether $S:q_0 \in \Gamma_{\mathbf{lfp}}$? This does not work for two reasons. First of all, $S:q_0 \in \Gamma_{\mathbf{lfp}}$ is not a necessary condition for the well-typedness of $\mathcal{G}$. For example, for $\mathcal{G}_0$ of Example 1, $\Gamma_{\mathbf{lfp}} = \emptyset$. Secondly, for each iteration to compute $\mathcal{F}_{\mathcal{B}}(\emptyset), \mathcal{F}_{\mathcal{B}}^2(\emptyset), \mathcal{F}_{\mathcal{B}}^3(\emptyset), \ldots$, we have to *guess* a type θ of F and check whether $\Gamma \vdash^{\mathcal{B}} \mathcal{R}(F) : \theta$. The possible types of F are however too many, hence the same problem as the greatest fixedpoint computation.

The discussion above however suggests that a least fixedpoint computation may work if, in $\Gamma \vdash^{\mathcal{B}} \mathcal{R}(F) : \theta$, (i) we relax the condition on Γ (that Γ must have been obtained from the previous iteration steps), and (ii) we impose a restriction on θ, to disallow θ to be synthesized "out of thin air". This observation motivates us to modify $\mathcal{F}_{\mathcal{B}}(\Gamma)$ as follows:

$$\mathcal{F}'_{\mathcal{B}}(\Gamma) = \bigcup \{\{F : \theta'\} \cup \Gamma' \mid \Gamma' \vdash^{\mathcal{B}} \mathcal{R}(F) : \theta', \Gamma \preceq_O \Gamma', \theta \preceq_P \theta', (F : \theta) \in \Gamma\}.$$

Here, $\Gamma \preceq_O \Gamma'$ (which will be defined later) indicates that Γ' is not identical, but somehow similar to Γ. The condition "$\theta \preceq_P \theta'$ for some $F : \theta \in \Gamma$" also indicates that $F : \theta'$ is somehow similar to an existing type binding $F : \theta$ of Γ.

To see how $\preceq_O$ and $\preceq_P$ may be defined, let us consider $\mathcal{G}_0$ and $\mathcal{B}_0$ of Examples 1 and 2. In order for $[\![\mathcal{G}_0]\!]$ to be accepted by $\mathcal{B}_0{}^{\perp}$, S should have type q_0. So, let us first put $S : q_0$ into the initial type environment: $\Gamma_0 := \{S : q_0\}$.

Now, in order for the body $F\,\mathtt{b}\,\mathtt{c}$ of S to have type q_0, F must have a type of the form $\cdots \rightarrow \cdots \rightarrow q_0$. So, let us put $F : \top \rightarrow \top \rightarrow q_0$: $\Gamma_1 := \{S : q_0, F : \top \rightarrow \top \rightarrow q_0\}$.

Let us now look at the definition of F, to check whether the body of F has type $\top \rightarrow \top \rightarrow q_0$. The body doesn't, but it has a slightly modified type: $(\top \rightarrow q_0) \rightarrow \top \rightarrow q_0$, so we update the type of F: $\Gamma_2 := \{S{:}q_0, F{:}(\top \rightarrow q_0) \rightarrow \top \rightarrow q_0\}$.

Going back to the definition of S, we know that F is required to have a type like $(q_1 \rightarrow q_0) \rightarrow \top \rightarrow q_0$ (because $\mathtt{b}$ has type $q_1 \rightarrow q_0$, not $\top \rightarrow q_0$). Thus, we further update the type of F: $\Gamma_3 := \{S : q_0, F : (q_1 \rightarrow q_0) \rightarrow \top \rightarrow q_0\}$.

By looking at the definition of F again, we know that x should have type q_1 and that f should have q_1 as a return type, from the first and second arguments of $\mathtt{a}$ respectively. Thus, we get an updated type environment: $\Gamma_4 := \{S{:}q_0, F{:}(q_1 \rightarrow q_0) \wedge (\top \rightarrow q_1) \rightarrow q_1 \rightarrow q_0\}$. By checking the definition of S again, we get:

$$\Gamma_5 := \{S : q_0, F : (q_1 \rightarrow q_0) \wedge (q_1 \rightarrow q_1) \rightarrow q_1 \rightarrow q_0\}.$$

Thus, we have obtained enough type information for $\mathcal{G}_0$ (recall Example 3).

In the above example, the type of F has been expanded as follows.

$$\begin{aligned}&\top \preceq_O \top \rightarrow \top \rightarrow q_0 \preceq_P (\top \rightarrow q_0) \rightarrow \top \rightarrow q_0 \preceq_O (q_1 \rightarrow q_0) \rightarrow \top \rightarrow q_0\\ &\preceq_P (q_1 \rightarrow q_0) \wedge (\top \rightarrow q_1) \rightarrow q_1 \rightarrow q_0 \preceq_O (q_1 \rightarrow q_0) \wedge (q_1 \rightarrow q_1) \rightarrow q_1 \rightarrow q_0.\end{aligned}$$

Here, the expansions represented by $\preceq_O$ come from constraints on call sites of F, and those represented by $\preceq_P$ come from constraints on the definition of F. We shall formally define these expansion relations and obtain a fixedpoint-based model checking algorithm in the following sections.

Remark 1. A reader familiar with game semantics [1,20] may find a connection between the type expansion sequence above and a play of a function. For example, the type $(\top \rightarrow q_0) \rightarrow \top \rightarrow q_0$ may be considered an abstraction of a state of a play where the opponent of F has requested a tree of type q_0, and the proponent has requested a tree of type q_0 in response. The type $(q_1 \to q_0) \to \top \to q_0$ represents the next state, where the opponent has requested a tree of type q_1 in response, and the type $(q_1 \to q_0) \wedge (\top \to q_1) \to q_1 \to q_0$ represents the state where, in response to it, the proponent has requested trees of type q_1 to the first and second arguments. Thus, the expansion relations $\preceq_O$ and $\preceq_P$ represent opponent's and proponent's moves respectively. Although we do not make this connection formal, this game semantic intuition may help understand how our algorithm described below works. In particular, the intuition helps us understand why necessary type information can be obtained by gradually expanding types as in the example above; for a valid type of a function,[4] there should be a corresponding play of the function, and by following the play, one can obtain a type expansion sequence that leads to the valid type.

3.2 Expansion Relations

We now formally define the expansion relations $\preceq_O$ and $\preceq_P$ mentioned above. They are inductively defined by the following rules.

$$\frac{}{q \preceq_O q} \qquad \frac{}{q \preceq_P q} \qquad \frac{\tau \preceq_P \tau' \qquad \theta \preceq_O \theta'}{\tau \to \theta \preceq_O \tau' \to \theta'} \qquad \frac{\tau \preceq_O \tau' \qquad \theta \preceq_P \theta'}{\tau \to \theta \preceq_P \tau' \to \theta'}$$

$$\frac{\begin{array}{c}\forall j \in S' \setminus S. \exists q. \theta'_j = \top \to \cdots \to \top \to q \\ \forall j \in S. \theta_j \preceq_O \theta'_j \qquad S \subseteq S'\end{array}}{\bigwedge_{j \in S} \theta_j \preceq_O \bigwedge_{j \in S'} \theta'_j} \qquad \frac{\forall j \in S. \theta_j \preceq_P \theta'_j}{\bigwedge_{j \in S} \theta_j \preceq_P \bigwedge_{j \in S} \theta'_j}$$

Note that the four relations: $\theta \preceq_O \theta'$, $\tau \preceq_O \tau'$, $\theta \preceq_P \theta'$, and $\tau \preceq_P \tau'$ are defined simultaneously. Notice also that $\preceq_P$ and $\preceq_O$ are swapped in the argument position of arrow types; this is analogous to the contravariance of the standard subtyping relation in the argument position of function types. Another way to understand the relations is: $\theta \preceq_O \theta'$ ($\theta \preceq_P \theta'$, resp.) holds if θ' is obtained from θ by adding atomic types (of the form q) to positive (negative, resp.) positions.

In the last two rules, S can be an empty set. So, we can derive $\top \preceq_O \top \to q_0$, from which $\top \to \top \to q_0 \preceq_P (\top \to q_0) \to \top \to q_0$ follows. The expansion relation $\preceq_O$ is extended to type environments by: $\Gamma \preceq_O \Gamma'$ if and only if $\forall x \in dom(\Gamma) \cup dom(\Gamma').\Gamma(x) \preceq_O \Gamma'(x))$. Here, $\Gamma(x) = \bigwedge\{\theta \mid x : \theta \in \Gamma\}$. $\Gamma \preceq_P \Gamma'$ is defined in a similar manner.

Let $\leq$ be the standard subtyping relation on intersection types. Then, $\theta_2 \preceq_O \theta_1$ or $\theta_1 \preceq_P \theta_2$ imply $\theta_1 \leq \theta_2$, but not vice versa. For example, $\top \to q_0 \leq (q_1 \to q_0) \to q_0$ but $\top \to q_0 \not\preceq_P (q_1 \to q_0) \to q_0$.

3.3 Type Generation Rules

We now define the relation $\Gamma_1 \rhd \Gamma_2 \vdash^{\mathcal{B}}_P t : \theta_1 \rhd \theta_2$, which means that, given a candidate of type judgment $\Gamma_1 \vdash t : \theta_1$ (which may not be valid), a valid

[4] Strictly speaking, we should interpret a type as a kind of linear type; for example, the type $q_1 \to q_0$ should be interpreted as a function that takes a tree of type q_1 and uses it *at least once* to return a tree of type q_0.

judgment $\mathit{\Gamma}_2 \vdash^{\mathcal{B}} t : \theta_2$ is obtained by "adjusting" $\mathit{\Gamma}_1$ and θ_1 in a certain manner. $\mathit{\Gamma}_1 \triangleright \mathit{\Gamma}_2 \vdash^{\mathcal{B}}_P t : \theta_1 \triangleright \theta_2$ corresponds to the condition $(\mathit{\Gamma}_2 \vdash^{\mathcal{B}} t : \theta_2) \wedge (\mathit{\Gamma}_1 \preceq_O \mathit{\Gamma}_2) \wedge (\theta_1 \preceq_P \theta_2)$ used in the informal definition of $\mathcal{F}'_{\mathcal{B}}$ in Section 3.1. In fact, the rules below ensure that $\mathit{\Gamma}_1 \triangleright \mathit{\Gamma}_2 \vdash^{\mathcal{B}}_P t : \theta_1 \triangleright \theta_2$ implies that $(\mathit{\Gamma}_2 \vdash^{\mathcal{B}} t : \theta_2) \wedge (\mathit{\Gamma}_1 \preceq_O \mathit{\Gamma}_2) \wedge (\theta_1 \preceq_P \theta_2)$ holds. Thus, the "adjustment" of $\mathit{\Gamma}_1$ and θ_1 is allowed only in a restricted manner. For example, $(f\!:\!q_1 \to q_0) \triangleright (f\!:\!q_1 \to q_0, x\!:\!q_1) \vdash^{\mathcal{B}}_P f\,x : q_0 \triangleright q_0$ is allowed, but $(f : \top \to q_0) \triangleright (f : q_1 \to q_0, x : q_1) \vdash^{\mathcal{B}}_P f\,x : q_0 \triangleright q_0$ is not. In the latter, the change of the type of function f makes the assumption on the behavior of f that upon receiving a request for tree of type q_0, f requests a tree of type q_1 as an argument. That assumption is speculative and should be avoided, as its validity can be determined only by looking at the environment, not the term $f\,x$.

Definition 1. *$\mathit{\Gamma}_1 \triangleright \mathit{\Gamma}_2 \vdash^{\mathcal{B}}_P t : \theta_1 \triangleright \theta_2$ and $\mathit{\Gamma}_1 \triangleright \mathit{\Gamma}_2 \vdash^{\mathcal{B}}_P t : \tau_1 \triangleright \tau_2$ are the least relations that satisfy the following rules.*

$$\frac{\theta_1 \preceq_P \theta_2}{x : \theta_2 \triangleright x : \theta_2 \vdash^{\mathcal{B}}_P x : \theta_1 \triangleright \theta_2} \ (\textsc{VarP}) \qquad \frac{\tau_1 \preceq_O \theta_2}{x : \tau_1 \triangleright x : \theta_2 \vdash^{\mathcal{B}}_P x : \theta_2 \triangleright \theta_2} \ (\textsc{VarO})$$

$$\frac{(q, a, q_1 \cdots q_n) \in \mathit{\Delta} \qquad \forall i \in \{1, \dots, n\}.\tau_i \in \{\top, q_i\}}{\emptyset \triangleright \emptyset \vdash^{\mathcal{B}}_P a : (\tau_1 \to \cdots \to \tau_n \to q) \triangleright (q_1 \to \cdots \to q_n \to q)} \qquad (\textsc{Const})$$

$$\frac{\mathit{\Gamma}_1 \triangleright \mathit{\Gamma}'_1 \vdash^{\mathcal{B}}_P t_1 : (\tau \to \theta) \triangleright (\tau' \to \theta') \qquad \mathit{\Gamma}_2 \triangleright \mathit{\Gamma}'_2 \vdash^{\mathcal{B}}_P t_2 : \tau'' \triangleright \tau'}{\mathit{\Gamma}_1 \cup \mathit{\Gamma}_2 \triangleright \mathit{\Gamma}'_1 \cup \mathit{\Gamma}'_2 \vdash^{\mathcal{B}}_P t_1 t_2 : \theta \triangleright \theta'} \qquad (\textsc{App})$$

$$\frac{(\mathit{\Gamma}_1, x : \tau_1) \triangleright (\mathit{\Gamma}_2, x : \tau_2) \vdash^{\mathcal{B}}_P t : \theta_1 \triangleright \theta_2}{\mathit{\Gamma}_1 \triangleright \mathit{\Gamma}_2 \vdash^{\mathcal{B}}_P \lambda x.t : (\tau_1 \to \theta_1) \triangleright (\tau_2 \to \theta_2)} \qquad (\textsc{Abs})$$

$$\frac{\forall i \in S.(\mathit{\Gamma}_i \triangleright \mathit{\Gamma}'_i \vdash^{\mathcal{B}}_P t : \theta_i \triangleright \theta'_i)}{(\bigcup_{i \in S} \mathit{\Gamma}_i) \triangleright (\bigcup_{i \in S} \mathit{\Gamma}'_i) \vdash^{\mathcal{B}}_P t : (\bigwedge_{i \in S} .\theta_i) \triangleright (\bigwedge_{i \in S} .\theta'_i)} \qquad (\textsc{Int})$$

In the rules above, we write $x\!:\!\bigwedge_{i \in S} \theta_i$ for $\{x\!:\!\theta_i \mid i \in S\}$. It is implicitly assumed that the sort of each variable is respected, i.e., if $x : \theta \in \mathit{\Gamma}$, then $\theta :: \mathit{sort}(x)$ must hold. The rule VarP is used for adjusting the type of x to the type provided by the environment, while the rule VarO is used for adjusting the environment to the type of x. For example, $(x\!:\!q_1 \to q_2) \triangleright (x : q_1 \to q_2) \vdash^{\mathcal{B}}_P x : (\top \to q_2) \triangleright (q_1 \to q_2)$ is obtained from the former, and $x : \top \triangleright (x : \top \to q_2) \vdash^{\mathcal{B}}_P x : (\top \to q_2) \triangleright (\top \to q_2)$ is obtained from the latter. In the rule App, the left and right premises adjust the types of the function and the argument respectively. (In the game semantic view, the former accounts for a move of the function, and the latter accounts for a move of the argument.)

3.4 Model Checking Algorithm

We are now ready to describe the new algorithm. Let $\mathcal{G} = (\mathit{\Sigma}, \mathcal{N}, \mathcal{R}, S)$ be a recursion scheme and $\mathcal{B}$ be a trivial automaton. Define $\mathbf{Expand}_{\mathcal{G},\mathcal{B}}$ by:

$$\mathbf{Expand}_{\mathcal{G},\mathcal{B}}(\Gamma) = \Gamma \cup \Big(\bigcup \{\Gamma' \cup \{F{:}\theta'\} \mid \Gamma_1 \triangleright \Gamma' \vdash^{\mathcal{B}}_{P} \mathcal{R}(F) : \theta \triangleright \theta', F{:}\theta \in \Gamma, \Gamma_1 \subseteq \Gamma\}\Big)$$

Our new algorithm just consists of two fixedpoint computations:

```
NEW ALGORITHM:
1. $\Gamma$ := $\{S : q_0\}$;
2. Repeat $\Gamma$ := $\mathbf{Expand}_{\mathcal{G},\mathcal{B}}(\Gamma)$ until $\Gamma = \mathbf{Expand}_{\mathcal{G},\mathcal{B}}(\Gamma)$;
3. Repeat $\Gamma$ := $\mathbf{Shrink}_{\mathcal{G},\mathcal{B}}(\Gamma)$ until $\Gamma = \mathbf{Shrink}_{\mathcal{G},\mathcal{B}}(\Gamma)$;
4. Output whether $S : q_0 \in \Gamma$.
```

We first expand the set of type candidates by using $\mathbf{Expand}_{\mathcal{G},\mathcal{B}}$, and then shrink it by filtering out invalid types by $\mathbf{Shrink}_{\mathcal{G},\mathcal{B}}$. The only change from the naïve algorithm is that $\Gamma_{\mathbf{max}}$ has been replaced by the least fixedpoint of $\mathbf{Expand}_{\mathcal{G},\mathcal{B}}$.

Example 4. Recall $\mathcal{G}_0$ and $\mathcal{B}_0$ of Examples 1 and 2. Let $\Gamma_0 = \{S : q_0\}$. We have:

$$\begin{aligned}
\Gamma_1 &= \mathbf{Expand}_{\mathcal{G}_0,\mathcal{B}_0}(\Gamma_0) = \{S : q_0, F : \top \to \top \to q_0\}\\
\Gamma_2 &= \mathbf{Expand}_{\mathcal{G}_0,\mathcal{B}_0}(\Gamma_1) = \Gamma_1 \cup \{F : (\top \to q_0) \to \top \to q_0\}\\
\Gamma_3 &= \mathbf{Expand}_{\mathcal{G}_0,\mathcal{B}_0}(\Gamma_2) = \Gamma_2 \cup \{F : (q_1 \to q_0) \to \top \to q_0, \ldots\}\\
\Gamma_4 &= \mathbf{Expand}_{\mathcal{G}_0,\mathcal{B}_0}(\Gamma_3) = \Gamma_3 \cup \{F : (q_1 \to q_0) \to q_1 \to q_0, \ldots\}\\
\Gamma_5 &= \mathbf{Expand}_{\mathcal{G}_0,\mathcal{B}_0}(\Gamma_4) = \Gamma_3 \cup \{F : (q_1 \to q_0) \land (\top \to q_1) \to q_1 \to q_0, \ldots\}\\
\Gamma_6 &= \mathbf{Expand}_{\mathcal{G}_0,\mathcal{B}_0}(\Gamma_5) = \Gamma_4 \cup \{F : (q_1 \to q_0) \land (q_1 \to q_1) \to q_1 \to q_0, \ldots\}\\
\Gamma_7 &= \mathbf{Expand}_{\mathcal{G}_0,\mathcal{B}_0}(\Gamma_6) = \Gamma_6
\end{aligned}$$

In the second step (for computing Γ_2), the type $(\top \to q_0) \to \top \to q_0$ of F is obtained from the following derivation.

$$\dfrac{\dfrac{\emptyset \triangleright f : \top \to q_0 \vdash^{\mathcal{B}}_{P} \mathtt{a}\ (f\ x) : (\top \to q_0) \triangleright (q_0 \to q_0) \qquad \Delta_1 \triangleright \Delta_1 \vdash^{\mathcal{B}}_{P} (F\ f\ (f\ x)) : q_0 \triangleright q_0}{\Delta_1 \triangleright \Delta_2 \vdash^{\mathcal{B}}_{P} \mathtt{a}\ (f\ x)\ (F\ f\ (f\ x)) : q_0 \triangleright q_0}}{\Delta_1 \triangleright \Delta_1 \vdash^{\mathcal{B}}_{P} \lambda f.\lambda x.\mathtt{a}\ (f\ x)\ (F\ f\ (f\ x)) : \top \to \top \to q_0 \triangleright (\top \to q_0) \to \top \to q_0}$$

Here, $\Delta_1 = F : \top \to \top \to q_0$, $\Delta_2 = \Delta_1 \cup \{f : \top \to q_0\}$, and $\emptyset \triangleright f : \top \to q_0 \vdash^{\mathcal{B}}_{P} \mathtt{a}\ (f\ x) : (\top \to q_0) \triangleright (q_0 \to q_0)$ is derivable from $\emptyset \triangleright f : (\top \to q_0) \vdash^{\mathcal{B}}_{P} f : \top \to q_0 \triangleright \top \to q_0$ and $\emptyset \triangleright \emptyset \vdash^{\mathcal{B}}_{P} \mathtt{a} : (\top \to \top \to q_0) \triangleright (q_0 \to q_0 \to q_0)$.

By repeatedly applying $\mathbf{Shrink}_{\mathcal{G},\mathcal{B}}$ to Γ_6, we obtain: $\Gamma = \{S : q_0, F : (q_1 \to q_0) \land (q_1 \to q_1) \to q_1 \to q_0, F : (\top \to q_0) \to \top \to q_0, \ldots\}$ as a fixedpoint. Since $S : q_0 \in \Gamma$, we know that $\mathcal{G}_0$ is well-typed, i.e. $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^{\perp}$.

The least fixedpoint Γ_6 of $\mathbf{Expand}_{\mathcal{G}_0,\mathcal{B}_0}$ contains type bindings like $F : (\top \to q_0) \to \top \to q_0$, which are not required for typing $\mathcal{G}$ (recall Example 3). However, $\mathbf{Expand}_{\mathcal{G}_0,\mathcal{B}_0}$ does not add completely irrelevant type bindings like $F{:}\top \to \top \to q_1$. Thus, we can expect that the least fixedpoint of $\mathbf{Expand}_{\mathcal{G}_0,\mathcal{B}_0}$ is often much smaller than $\Gamma_{\mathbf{max}}$, which will be confirmed by the experiments in Section 5.

4 Correctness of the Algorithm

This section discusses the correctness and complexity of the algorithm.

Termination and Complexity. The termination follows immediately from the following facts: (i) Γ increases monotonically in the first loop, (ii) Γ decreases monotonically in the second loop, and (iii) Γ ranges over a finite set.

Theorem 2. *The algorithm always terminates and outputs "yes" or "no".*

From the termination argument, the complexity result also follows.

Theorem 3. *Suppose that both (i) the largest size of the sorts of non-terminals and terminals of $\mathcal{G}$ and (ii) the size of automaton $\mathcal{B}$ are fixed. Then, the algorithm terminates in time quadratic in $|\mathcal{G}|$.*

Proof. By the assumption, the size of $\Gamma_{\mathbf{max}}$ is $O(|\mathcal{G}|)$. Thus, the two loops terminate in $O(|\mathcal{G}|)$ iterations. At each iteration, $\mathbf{Expand}_{\mathcal{G},\mathcal{B}}(\mathcal{G})$ and $\mathbf{Shrink}_{\mathcal{G},\mathcal{B}}(\mathcal{G})$ can be computed in time $O(|\mathcal{G}|)$,[5] hence the result. □

Actually, we can use Rehof and Mogensen's algorithm [22] to accelerate the above algorithm, and obtain a linear time algorithm. (The idea is just to recompute $\mathbf{Expand}_{\mathcal{G},\mathcal{B}}$ and $\mathbf{Shrink}_{\mathcal{G},\mathcal{B}}$ only for variables whose relevant bindings were updated. As $\Gamma(x)$ is updated only a constant number of times for each variable x, and the number of typing bindings that are affected by the update is a constant, the resulting algorithm runs in time linear in $|\mathcal{G}|$.) More precisely, if the number of states of $\mathcal{B}$ is $|Q|$ and the largest arity of functions is A, the algorithm runs in time $O(|\mathcal{G}|\mathbf{exp}_n(p(A|Q|))$, where $p(x)$ is a polynomial of x, and $\mathbf{exp}_n(x)$ is defined by: $\mathbf{exp}_0(x) = x$, $\mathbf{exp}_{k+1}(x) = 2^{\mathbf{exp}_k(x)}$.

Soundness. The soundness of the algorithm follows immediately from that of Kobayashi's type system [12] (or Theorem 1): note that Γ at the last line of the algorithm satisfies $\Gamma \subseteq \mathbf{Shrink}_{\mathcal{G},\mathcal{B}}(\Gamma)$.

Theorem 4. *If the algorithm outputs "yes", then $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^{\perp}$.*

Completeness. A recursion scheme is in *normal form* if each rewrite rule is either of the form (i) $F \mapsto \lambda\widetilde{x}.t$ where t does not contain terminals, or (ii) $F \mapsto \lambda\widetilde{x}.a\,\widetilde{x}$. We show that the algorithm is complete when the given recursion scheme is in normal form.[6] We now prove the completeness, i.e., if $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^{\perp}$, then the algorithm outputs "yes". From the game semantic intuition described in Remark 1, the reason for the completeness is intuitively clear: If a function behaves as described by a type θ in a reduction sequence of the given recursion scheme, then there should be a sequence of interactions between the function and the environment that conforms to θ. As the sequence of interactions evolves, the

[5] To guarantee this, we need to normalize the rewrite rules of $\mathcal{G}$ in advance [16], so that the size of body $\mathcal{R}(F)$ of each non-terminal F is bounded by a constant.

[6] Note that this does not lose generality, as we can always transform a recursion scheme into an equivalent recursion scheme in normal form before applying the algorithm, by introducing the rule $A \mapsto \lambda\widetilde{x}.a\,\widetilde{x}$ for each terminal a, and replace all the other occurrences of a with A. We conjecture that the algorithm is complete without the normal form assumption.

function's behavior should gradually evolve from $\top \to \cdots \to \top \to q$ (which represents a state where the environment has just called the function to ask for a tree of type q) to $\top \to \cdots \to (\top \to \cdots \to \top \to q') \to \cdots \to \top \to q$ (which represents a state where the function has responded to ask the environment to provide a tree of type q'), and eventually to θ. Such evolution of the function's type can be computed by $\mathbf{Expand}_{\mathcal{G},\mathcal{B}}$, and θ should be eventually generated.

The actual proof is however rather involved. We defer the proof to the extended version [14] and just state the theorem here.

Theorem 5. *Suppose $\mathcal{G}$ is in normal form. If $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^{\perp}$, then the algorithm outputs "yes".*

5 Experiments

We have implemented the new model checking algorithm, and tested it for several recursion schemes. The result of the preliminary experiments is shown in Table 1. The experiments were conducted on a machine with Intel(R) Xeon(R) CPU with 3Ghz and 8GB memory. More information about the benchmark is available at `http://www.kb.ecei.tohoku.ac.jp/~koba/gtrecs/`.

The column "order" shows the order of the recursion scheme. The columns "hybrid" and "new" show the running times of the hybrid and new algorithms respectively, measured in seconds. The cell marked by "–" in the column "hybrid" shows that the hybrid algorithm has timed out (where the time limit is 10 min.) or run out of stack. The columns "Γ_1" and "Γ_2" show the numbers of atomic types in the type environment Γ after the first and second loops of the new algorithm.

The table on the lefthand side shows the result for the following recursion scheme $\mathcal{G}_{n,m}$ [13]:

$$\begin{array}{l}\{S \mapsto F_0\ G_{n-1}\ \cdots\ G_2\ G_1\ G_0,\\ F_0 \mapsto \lambda f.\lambda\widetilde{x}.F_1\,(F_1\ f)\ \widetilde{x}, \cdots, F_{m-1} \mapsto \lambda f.\lambda\widetilde{x}.F_m\,(F_m\ f)\ \widetilde{x}, F_m \mapsto \lambda f.\lambda\widetilde{x}.G_n\ f\ \widetilde{x},\\ G_n \mapsto \lambda f.\lambda z.\lambda.\widetilde{x}.f\,(f\,z)\,\widetilde{x}, \cdots, G_2 \mapsto \lambda f.\lambda z.f\ (f\ z), G_1 \mapsto \lambda z.\mathtt{a}\ z, G_0 \mapsto \mathtt{c}\}\end{array}$$

S is reduced to $\mathtt{a}^{\mathbf{exp}_n(m)}(G_0)$ and then to $\mathtt{a}^{\mathbf{exp}_n(m)}(\mathtt{c})$. The verified property is that the number of $\mathtt{a}$ is even. The hybrid algorithm [10] requires $O(\mathbf{exp}_n(m))$ expansions to extract the type information on G_0, so that it times out except for the case $n = 3, m = 1$. In contrast, the new algorithm works even for the case $n = 4, m = 10$. For a fixed n, the size of type environments (the columns Γ_1 and Γ_2) is almost linear in m. The running times are not linear in m due to the naïveness of the current implementation, but exponential slowdown with respect to m is not observed. As expected, the sizes of type environments (the columns Γ_1 and Γ_2) are much smaller than that of $\Gamma_{\mathbf{max}}$. For $\mathcal{G}_{3,1}$, the size of $\Gamma_{\mathbf{max}}$ is about 3×2^{2057}, so that the naïve algorithm does not work even for $\mathcal{G}_{3,1}$.

In the table on the righthand side, `Example1` is the recursion scheme given in Example 1, where the trivial automaton is given in Example 2. The recursion schemes `Twofiles` – `Lock2` have been taken from the benchmark set used in

Table 1. The result of experiments. Times are in seconds.

	order	hybrid	new	Γ_1	Γ_2
$\mathcal{G}_{3,1}$	3	0.002	0.021	61	41
$\mathcal{G}_{3,5}$	3	–	0.135	161	97
$\mathcal{G}_{3,10}$	3	–	0.382	286	167
$\mathcal{G}_{4,1}$	4	–	0.563	302	206
$\mathcal{G}_{4,5}$	4	–	14.856	1079	703
$\mathcal{G}_{4,10}$	4	–	43.815	2054	1328

	order	hybrid	new	Γ_1	Γ_2
`Example1`	2	0.002	0.002	15	13
`Twofiles`	3	0.001	0.228	468	187
`FileWrong`	3	0.001	0.116	398	142
`FileOcamlc`	3	0.003	1.162	1610	414
`Lock2`	3	0.013	98.785	2464	1191
`Nondet`	3	N.A.	0.013	77	63

[10], obtained by encoding the resource usage verification problems [12]. We have used the refined encoding given in [15] however.[7] Unfortunately, for these recursion schemes, the new algorithm is slower than the hybrid algorithm by several orders of magnitude. Further optimization is required to see whether this is a fundamental limitation of the new algorithm. Finally, `Nondet` is an order-3 recursion scheme that generates a tree representation of an infinite list $[(0,1);(1,2);(2,3);\cdots]$, where each natural number n is represented by the tree $s^n(z)$. A non-deterministic trivial automaton is used for expressing the property that each pair in the list is either a pair of an even number and an odd number, or a pair of an odd number and an even number. Our new algorithm works well, while the hybrid algorithm (which works only for deterministic trivial automata) is not directly applicable.[8]

6 Related Work

We have already discussed the main related work in Section 1. There are several algorithms for the model checking of higher-order recursion schemes (some of which are presented in the context of showing the decidability). Besides those already mentioned [20,12,10,16], Hague et al. [6] reduce the modal μ-calculus model checking to a parity game over the configuration graph of a collapsible pushdown automaton. Aehlig [2] gives a trivial automata model checking algorithm based on a finite semantics, which runs in a fixed-parameter *non-deterministic* linear time in the size of the recursion scheme. For recursion schemes with the so called *safety* restriction, Knapik et al. [9] give another decision procedure, which reduces a model checking problem for an order-n recursion scheme to that for an order-$(n-1)$ recursion scheme. As mentioned already, however, the only practical previous algorithm (which was ever implemented) is Kobayashi's hybrid algorithm [10], to our knowledge. Its worst-case complexity is hyper-exponential in the size of recursion schemes, unlike our new algorithm. Recently, Lester et al. [19] extended

[7] The encoding in [12] produces order-4 recursion schemes, while that of [15] produces order-3 recursion schemes. An additional optimization is required to handle the encoding of [12]: see [14].

[8] As mentioned in Section 6, Lester et al. [19] recently extended the hybrid algorithm to deal with alternating Büchi automata.

the hybrid algorithm to deal with alternating Büchi automata. As the basic mechanism for collecting type information remains the same, their algorithm also suffers from the same worst-case behavior as Kobayashi's hybrid algorithm.

As mentioned already, though our new algorithm is type-based, it has been inspired from game semantics [1,20]. In the previous type-based approach [12], the types of a function provide coarse-grained information about the function's behavior, in the sense that the types tell us information about *complete* runs of the function. On the other hand, the game-semantic view provides more fine-grained information, about partial runs of a function. For example, $F : \top \rightarrow q_0$ belonging to the least fixedpoint of $\mathbf{Expand}_{\mathcal{G},\mathcal{B}}$ means that F may be called in a context where a tree of type q_0 is required, not necessarily that F returns a tree of type q_0 for arbitrary arguments. This enabled us to collect type information by a least fixedpoint computation, yielding a realistic linear time algorithm.

As explained in Section 2, the model checking of higher-order recursion schemes has been reduced to the type checking problem for an intersection type system. Thus, our algorithm may have some connection to intersection type inference algorithms [7,5]. The connection is however not so clear. To our knowledge, the existing inference algorithms have a process corresponding to β-normalization [5], so that even for terms without recursion, the worst-case complexity of intersection type inference is non-elementary in the program size.

7 Conclusion

Studies of the model checking of higher-order recursion schemes have started from theoretical interests [8,9], but it is now becoming the basis of automated verification tools for higher-order functional programs [12,18,17,21]. Thus, it is very important to develop an efficient model checker for higher-order recursion schemes. The new algorithm presented in this paper is the first one that is efficient both in theory (in the sense that it is fixed-parameter linear time) and in practice (in the sense that it is runnable for recursion schemes of order 3 or higher). The practical efficiency is however far from satisfactory (recall Section 5), so that further optimization of the algorithm is necessary. As the structure of the new algorithm is simple, we expect that it is more amenable to various optimization techniques, such as BDD representation of types. A combination of the hybrid and new algorithms also seems useful. It does not seem so difficult to extend the new algorithm to obtain a practical fixed-parameter polynomial time algorithm for the full modal μ-calculus; It is left for future work.

References

1. Abramsky, S., McCusker, G.: Game semantics. In: Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School, pp. 1–56. Springer, Heidelberg (1999)
2. Aehlig, K.: A finite semantics of simply-typed lambda terms for infinite runs of automata. Logical Methods in Computer Science 3(3) (2007)

3. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: Proc. of POPL, pp. 1–3 (2002)
4. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. International Journal on Software Tools for Technology Transfer 9(5-6), 505–525 (2007)
5. Carlier, S., Polakow, J., Wells, J.B., Kfoury, A.J.: System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 294–309. Springer, Heidelberg (2004)
6. Hague, M., Murawski, A., Ong, C.-H.L., Serre, O.: Collapsible pushdown automata and recursion schemes. In: Proceedings of 23rd Annual IEEE Symposium on Logic in Computer Science, pp. 452–461. IEEE Computer Society, Los Alamitos (2008)
7. Kfoury, A.J., Wells, J.B.: Principality and type inference for intersection types using expansion variables. Theor. Comput. Sci. 311(1-3), 1–70 (2004)
8. Knapik, T., Niwinski, D., Urzyczyn, P.: Deciding monadic theories of hyperalgebraic trees. In: Abramsky, S. (ed.) TLCA 2001. LNCS, vol. 2044, pp. 253–267. Springer, Heidelberg (2001)
9. Knapik, T., Niwinski, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 205–222. Springer, Heidelberg (2002)
10. Kobayashi, N.: Model-checking higher-order functions. In: Proceedings of PPDP 2009, pp. 25–36. ACM Press, New York (2009), see also [13]
11. Kobayashi, N.: TRecS (2009), http://www.kb.ecei.tohoku.ac.jp/~koba/trecs/
12. Kobayashi, N.: Types and higher-order recursion schemes for verification of higherorder programs. In: Proc. of POPL, pp. 416–428 (2009), see also [13]
13. Kobayashi, N.: Model checking higher-order programs. A revised and extended version of [12] and [10], available from the author's web page (2010)
14. Kobayashi, N.: A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes (2010), an extended version http://www.kb.ecei.tohoku.ac.jp/~koba/gtrecs/
15. Kobayashi, N., Ong, C.-H.L.: Complexity of model checking recursion schemes for fragments of the modal mu-calculus. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 223–234. Springer, Heidelberg (2009)
16. Kobayashi, N., Ong, C.-H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: Proceedings of LICS 2009, pp. 179–188. IEEE Computer Society Press, Los Alamitos (2009)
17. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and cegar for higher-order model checking (July 2010) (unpublished manuscript)
18. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: Proc. of POPL, pp. 495–508 (2010)
19. Lester, M.M., Neatherway, R.P., Ong, C.-H.L., Ramsay, S.J.: Model checking liveness properties of higher-order functional programs (2010) (unpublished manuscript)
20. Ong, C.-H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS 2006, pp. 81–90. IEEE Computer Society Press, Los Alamitos (2006)
21. Ong, C.-H.L., Ramsay, S.: Verifying higher-order programs with pattern-matching algebraic data types. In: Proceedings of POPL 2011 (to appear, 2011)
22. Rehof, J., Mogensen, T.: Tractable constraints in finite semilattices. Science of Computer Programming 35(2), 191–221 (1999)

Church Synthesis Problem for Noisy Input

Yaron Velner and Alexander Rabinovich

The Blavatnik School of Computer Science, Tel Aviv University, Israel

Abstract. We study two variants of infinite games with imperfect information. In the first variant, in each round player-1 may decide to hide his move from player-2. This captures situations where the input signal is subject to fluctuations (noises), and every error in the input signal can be detected by the controller. In the second variant, all of player-1 moves are visible to player-2; however, after the game ends, player-1 may change some of his moves. This captures situations where the input signal is subject to fluctuations; however, the controller cannot detect errors in the input signal.

We consider several cases, according to the amount of errors allowed in the input signal: a fixed number of errors, finitely many errors and the case where the rate of errors is bounded by a threshold. For each of these cases we consider games with regular and mean-payoff winning conditions. We investigate the decidability of these games.

There is a natural reduction for some of these games to (perfect information) multidimensional mean-payoff games recently considered in [7]. However, the decidability of the winner of multidimensional mean-payoff games was stated as an open question. We prove its decidability and provide tight complexity bounds.

1 Introduction

The algorithmic theory of infinite games is a powerful and flexible framework for the design of reactive systems (see e.g., [12]). It is well known for instance, that the construction of a controller acting indefinitely within its environment amounts to the computation of a winning strategy in an infinite game. For the case of regular games, algorithmic solutions of the synthesis problem have been developed, providing methods for the automatic construction of controllers. The basis of this approach is the Büchi-Landweber Theorem, which states that in a regular infinite game, i.e., a game over a finite game graph with a winning condition given by an ω-regular language, a finite state winning strategy for the winner can be constructed [4]. Much work has been devoted to the generalizations and extensions of this fundamental result. One well known extension are mean-payoff games where the winning condition is an ω language recognized by an automaton with a *mean-payoff acceptance condition*. These games have been studied since the end of the seventies [11,17] and still attract a large interest. Another well known extension are games with imperfect information. In most of the previous work, the setting of an imperfect information game is given by a game graph with a coloring of the state space that defines equivalence

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 275–289, 2011.

classes of indistinguishable states called observations [9,15], and the strategies are observation-based (i.e., they rely on the past sequence of observations rather than on states).

In the present paper we investigate *games with errors* which are different kind of imperfect information games.

To present games with errors it is convenient to refer to the simplest format of infinite games, also called Gale-Stewart games. In such game we abstract from graphs but just let the two players choose letters from a finite alphabets Σ_1, Σ_2 in turn. A play is built up a sequence $\binom{b_0}{a_0}\binom{b_1}{a_1}\binom{b_2}{a_2}\cdots \in (\Sigma_1 \times \Sigma_2)^\omega$. A natural view is to consider the sequence $a = a_0a_1\ldots$ as the input stream and the sequence $b = b_0b_1\ldots$ as the output stream. In the Gale-Stewart game, the play is won by player-2 if the ω word $\binom{b_0}{a_0}\binom{b_1}{a_1}\binom{b_2}{a_2}\ldots$ satisfies the winning condition, i.e., if it belongs to a given *specification* language $L \subseteq (\Sigma_1 \times \Sigma_2)^\omega$.

We consider a *game with detected errors* where in each round of the game, player-1 has the possibility to hide his move from the opponent; however, player-2 can detect whether player-1 hides a move. Player-1 needs to decide on the value of the hidden moves, only at the end of the game. Hence, at the end of the game he replaces his hidden moves in the produced play ρ by letters from Σ_1 and this defines an *interpretation* $Int(\rho) \in (\Sigma_1 \times \Sigma_2)^\omega$ for the play ρ. Round $i \in \mathbb{N}$ has an *error* if player-1 decides to hide his move in round i. This game captures the cases where the input signal is noisy and the controller can detect errors in the input signal. We also consider a *game with undetected errors* where in each round player-2 is fully aware of player-1 moves; however, at the end of the game player-1 can change some of his moves, i.e., player-1 provides an *interpretation* $Int(\rho) \in (\Sigma_1 \times \Sigma_2)^\omega$ for the play ρ. An *error* is made in round $i \in \mathbb{N}$ if at the end of the game player-1 decides to change his move in round i. Note that in this game, an error is determined according to the interpretation of the play. This game captures the cases where the input signal is noisy and the controller cannot detect errors in the input signal.

We measure the amount of errors in a play according to two scales. The *error count* scale counts the number of errors in a play (the result is in $\mathbb{N} \cup \{\infty\}$). The *error rate* of a play ρ is $\limsup_{n\to\infty} \frac{1}{n} \cdot$ (number of errors in first n rounds).

In both games a limitation on the amount of errors allowed for player-1 is given by one of the following conditions types. The first type of conditions is a bound $n \in \mathbb{N}$ on the error count. The second type of conditions requires that the error count of a play is finite. The third type of conditions is a bound $\delta \in \mathbb{Q}$ on the error rate. The last type seems to be the most interesting for real life applications.

For a specification language L: Player-1 is the winner of a play with detected errors if the play satisfies the amount of errors limitation and there exists an interpretation $Int \notin L$. Player-1 is the winner of a play with undetected errors if there exists an interpretation $Int \notin L$ which satisfies the amount of errors limitation.

We consider the cases where the specification language is either an ω regular language or when the language is recognizable by a mean-payoff automaton.

We investigate the decidability of who is the winner, and the computability of the winning strategies. In addition, we investigate the *bounded number of errors* problem which asks if there exists $n \in \mathbb{N}$ such that player-1 is the winner of a game when he allowed to do at most n errors.

Table 1 summarizes our decidability results for games with detected and undetected errors with regular and mean-payoff winning condition.

Table 1. ✓ - Decidable. × - Undecidable. ? - Open.

	Bounded[1]		Finite[2]		Rate[3] $\delta = 0$		Rate[3] $\delta \in (0,1)$		Rate[3] $\delta = 1$	
	Regular	MP	Regular	MP	Regular	MP	Regular	MP	Regular	MP
Detected	✓	✓	✓	✓	✓	✓	✓	×	✓	×
Undetected	?	?	✓	?	?	?	×	×	✓	×

We reduced games with detected errors and a mean-payoff winning condition to *multidimensional mean-payoff games.*

A multidimensional mean-payoff game, introduced in [7], is played on a finite weighted graph by two players. The edges of the graphs are labeled by k-dimensional vectors $w \in \mathbb{Z}^k$. The game is played as follows. A pebble is placed on a designated initial vertex of the game graph $G = (V, E)$. The game is played in rounds in which the player owning the position where the pebble lies moves the pebble to an adjacent position of the graph using an outgoing edge. An infinite play results an infinite path $\pi = e_0 e_1 \dots \in E^\omega$ through the game graph. The energy level vector of the finite path $e_0 e_1 \dots e_{n-1}$ is $EL(e_0 e_1 \dots e_{n-1}) = \sum_{i=1}^{n-1} w(e_i)$. The infinite path π produces two mean-payoff vectors. $\underline{MP}(\pi) = \liminf_{n\to\infty} \frac{1}{n} \cdot EL(e_0 e_1 \dots e_{n-1})$ and $\overline{MP}(\pi) = \limsup_{n\to\infty} \frac{1}{n} \cdot EL(e_0 e_1 \dots e_{n-1})$. Hence, the vector associated to a play ρ which induces an infinite path π_ρ is the $2k$ dimensional vector $\overrightarrow{MP}(\rho) = (\underline{MP}(\pi_\rho), \overline{MP}(\pi_\rho))$. The winning condition for player-2 is given by a threshold $\nu = \langle \nu_1, \dots, \nu_{2k} \rangle \in \mathbb{Q}^{2k}$, which induces $2k$ boolean variables $x_i^\nu \doteq \overrightarrow{MP}(\rho)_i \geq \nu_i$, and by a boolean formula on x_i^ν (for $i = 1, \dots, 2k$).

In [7] the players were restricted to use only finite state strategies. The play π produced by finite state strategies is always quasi periodic and $\overline{MP}(\pi) = \underline{MP}(\pi)$.

It was proved in [7] that when the players are restricted to use the finite state strategies and the winning condition is a conjunction of the form $\bigwedge_{i=1}^{2k} x_i^\nu$, then it is decidable who is the winner. The decidability of who is the winner when players are allowed to use arbitrary strategies was stated as an open question. We prove that this problem is decidable and provide tight bounds on its complexity. The lower complexity bounds are easily derived from [7,3].

We investigate the case where the winning condition is given by the formula $\varphi_{\bigwedge \text{MeanPayoffInf}^{\geq}(\nu)} \doteq \bigwedge_{i=1}^{k} x_i^\nu$ or by the formula $\varphi_{\bigwedge \text{MeanPayoffSup}^{\geq}(\nu)} \doteq$

[1] Decidability of the bounded number of errors problem.

[2] Games with finitely many errors.

[3] Games with the errors rate limitation δ.

$\bigwedge_{i=k+1}^{2k} x_i^{\nu}$. We show that the problem who wins $\varphi_{\bigwedge \text{MeanPayoffInf}^{\geq}(\nu)}$ games is Strongly coNP complete, while there exists a pseudo-polynomial algorithm which determines the winner for a condition of the form $\varphi_{\bigwedge \text{MeanPayoffSup}^{\geq}(\nu)}$.

We also consider the case where the winning condition is given by a positive boolean formula which depends only on the $\underline{MP}(\pi_\rho)$ vector or on the $\overline{MP}(\pi_\rho)$ vector, and prove decidability for this case.

This paper is organized as follows. In the next section we introduce notations and well known definitions. In section 3, we provide formal definitions of games with errors. In sections 4-5, we investigate regular games with errors. In section 6, we investigate mean-payoff games with errors. In section 7, we investigate multidimensional mean-payoff games and complete the proof of the main theorem of section 6. Due to lack of space, only the outlines of the proofs are presented.

2 Definitions

Words and languages. Let Σ be a finite *alphabet*. A *finite word* over Σ is $w = \sigma_0\sigma_1 \ldots \sigma_n$, for $n \in \mathbb{N}$ and $\sigma_i \in \Sigma$. An *infinite word* over Σ is $w = \sigma_0\sigma_1 \ldots \sigma_n \ldots$ Let $w = \sigma_0\sigma_1 \ldots \sigma_n \ldots$, we denote σ_i by $w(i)$. For $j > i \geq 0$, $w[i,j] = w(i)w(i+1)\ldots w(j-1)w(j)$, $w[0, n-1]$ is a *prefix* of length n of w. $w[i,\infty]$ is a *suffix* of w starting from position i. For every $i \geq 0$, $w[i, i-1]$ is defined to be the *empty word* ϵ. We denote by Σ^* the set of all finite words, and by Σ^ω the set of all infinite words. A *language* is a subset of Σ^*, and an ω *language* is a subset of Σ^ω. In the sequel, when it is clear from the context, we shall also refer to an ω language $L \subseteq \Sigma^\omega$ as a language. A word over $\Sigma = \Sigma_1 \times \Sigma_2$ is $w = \binom{\sigma_2(0)}{\sigma_1(0)}\binom{\sigma_2(1)}{\sigma_1(1)} \ldots \binom{\sigma_2(n)}{\sigma_1(n)} \ldots$ We denote $\binom{\sigma_2(i)}{\sigma_1(i)}$ by $w(i)$, $\sigma_1(i)$ by $w(i)_1$ and $\sigma_2(i)$ by $w(i)_2$, for $j = 1, 2$ we denote $\sigma_j(0)\sigma_j(1)\ldots$ by w_j.

Gale-Stewart game. A *Gale-Stewart* game is a two-player game of perfect information. The game is defined using an alphabet $\Sigma = \Sigma_1 \times \Sigma_2$ and a language $L \subseteq (\Sigma_1 \times \Sigma_2)^\omega$, and is denoted by G_L. The two players alternate turns, and each player is aware of all moves before making the next one. A *play* of game has ω rounds. At round $i \in \mathbb{N}$: first player-1 chooses $\sigma_1 \in \Sigma_1$, then player-2 chooses $\sigma_2 \in \Sigma_2$. At the end of the play, an ω word $w = \binom{\sigma_2(0)}{\sigma_1(0)}\binom{\sigma_2(1)}{\sigma_1(1)} \ldots$ is formed. Player-2 wins the play if $w \in L$.

Strategies. A player-1 *strategy* is $\tau : (\Sigma_1 \times \Sigma_2)^* \to \Sigma_1$. A play $\rho = \binom{\sigma_2(0)}{\sigma_1(0)}\binom{\sigma_2(1)}{\sigma_1(1)} \ldots$ is consistent with player-1 strategy τ if $\tau(\binom{\sigma_2(0)}{\sigma_1(0)}\binom{\sigma_2(1)}{\sigma_1(1)} \ldots \binom{\sigma_2(i)}{\sigma_1(i)}) = \sigma_1(i+1)$ for every $i \in \mathbb{N}$. A player-2 strategy is $\tau : (\Sigma_1 \times \Sigma_2)^* \times \Sigma_1 \to \Sigma_2$. The consistency of a play with player-2 strategy is defined similarly. Player-j (for $j = 1, 2$) is the *winner* of game G_L if it has a strategy τ such that in every play consistent with τ, player-j wins.

Automaton. An *automaton* over Σ is a tuple $\mathcal{A} = (\Sigma, Q, Q_0, E)$ Where Σ is a finite alphabet, Q is a finite set of *states*, $Q_0 \subseteq Q$ is a set of *initial states* and $E \subseteq Q \times Q \times \Sigma$ is a *transition relation*. Automaton $\mathcal{A}$ is *deterministic* if $|Q_0| = 1$ and $\forall q \in Q, \forall \sigma \in \Sigma, \forall q_1, q_2 \in Q$ if $(q, q_1, \sigma) \in E$ and $(q, q_2, \sigma) \in E$ then $q_1 = q_2$. For infinite word $\rho = \sigma_1\sigma_2 \ldots$ a *run* of ρ is an infinite sequence $\pi = q_0e_0q_1e_1 \ldots$ with

$q_0 \in Q_0$ and $e_i = (q_i, q_{i+1}, \rho(i)) \in E$ for every $i \in \mathbb{N}$. The set of all states reachable by a finite word ρ is $E^*(\rho) = \{q \in Q \mid \rho$ has a (finite) run that ends in $q\}$.

Language recognized by automaton. Let $\mathcal{A} = (\Sigma, Q, Q_0, E)$. An *acceptance condition* for $\mathcal{A}$ is a set $\phi \subseteq (Q \times E)^\omega$, note that we are interested only in the cases where ϕ has a finite description. An infinite word is *accepted* by automaton $\mathcal{A}$ and condition ϕ if it has a run π such that $\pi \in \phi$. The set of all ω words accepted by automaton $\mathcal{A}$ and condition ϕ is denoted by $L_{\mathcal{A},\phi}$. In the sequel, when the acceptance condition ϕ it is clear from the context, we shall omit it and denote the language by $L_{\mathcal{A}}$.

Acceptance conditions (objectives). Let $p : Q \to \mathbb{N}$ be a *priority function* and $w : E \to \mathbb{Z}$ be a *weight function*. The *energy level* of a finite path $\gamma = q_0e_0q_1e_1...e_{n-1}q_n$ is $EL(w, \gamma) = \sum_{i=0}^{n-1} w(e_i)$, and the *mean-payoff value* of an infinite path $\pi = q_0e_0q_1e_1...$ is defined by either $\overline{MP}(w, \pi) = \limsup_{n\to\infty} \frac{1}{n} \cdot EL(w, \pi[0, n-1])$ or by $\underline{MP}(w, \pi) = \liminf_{n\to\infty} \frac{1}{n} \cdot EL(w, \pi[0, n-1])$. In the sequel, when the weight function w is clear from the context we will omit it and simply write $EL(\gamma)$ and $\overline{MP}(\pi)$ or $\underline{MP}(\pi)$. We denote by $Inf(\pi)$ the set of states that occur infinitely often in π. We consider the following conditions:

- *Parity condition.* The *parity* condition $Parity_{\mathcal{A}}(p) = \{\pi \in (Q \times E)^\omega \mid \min\{p(q) \mid q \in Inf(\pi)\}$ is even$\}$ requires that the minimum priority visited infinitely often is even.
- *Mean-payoff condition.* Given a threshold $\nu \in \mathbb{Q}$, and $\sim \in \{>, \geq\}$ the *mean-payoff* condition is either
  - $\text{MeanPayoffSup}^{\sim}(\nu) = \{\pi \in (Q \times E)^\omega \mid \overline{MP}(\pi) \sim \nu\}$
  - $\text{MeanPayoffInf}^{\sim}(\nu) = \{\pi \in (Q \times E)^\omega \mid \underline{MP}(\pi) \sim \nu\}$

  Mean-payoff conditions defined with $\sim \in \{<, \leq\}$ are obtained by duality since $\limsup_{n\to\infty} x_n = -\liminf_{n\to\infty} -x_n$.
- *Tail objective*, Informally the class of tail objectives is the class of objectives that are independent of all finite prefixes. Formally $\phi \subseteq (Q \times E)^\omega$ is a *tail objective* if for every $\pi \in (Q \times E)^\omega$, $\pi \in \phi \Leftrightarrow$ every suffix of π is in ϕ.

Language defined by a MSO formula. Let $\varphi(X_1, X_2)$ be a formula of Monadic Second-order logic over the signature $\{<\}$, where X_1 and X_2 are set (second-order) variables. We define the language $L_\varphi \subseteq (\{0,1\} \times \{0,1\})^\omega$ as usual. Let $w = \binom{\sigma_2(0)}{\sigma_1(0)}\binom{\sigma_2(1)}{\sigma_1(1)}\ldots$ be an ω word. Let P_1, P_2 be monadic predicates on $\mathbb{N}$ defined as: $P_1(t) \Leftrightarrow w(t)_1 = 1$ and $P_2(t) \Leftrightarrow w(t)_2 = 1$ for every $t \in \mathbb{N}$. Then $w \in L_\varphi$ iff $(\mathbb{N}, <) \models \varphi(P_1, P_2)$. A language $L \subseteq (\{0,1\} \times \{0,1\})^\omega$ is *MSO definable* if there exists a MSO formula $\varphi(X_1, X_2)$ such that $L = L_\varphi$. The MSO definable languages over an alphabet $\Sigma_1 \times \Sigma_2$ are defined similarly [16].

ω **regular languages.** It is well-known that the class of ω language definable by MSO formulas is the same as the class recognizable by parity automata. An ω language is *regular* if it is definable by a MSO formula.

3 Games with Errors

We define two versions of games with errors. Informally we want to distinguish the case where player-2 can detect errors made by player-1, from the case where errors cannot be detected during the play.

The first case is captured by *games with detected errors*, and the second case is captured by *games with undetected errors.*

3.1 Games with Detected Errors

Let z be a symbol not in Σ_1. We use z to represent a (detected) error.

Play with detected errors. A *play with detected errors* has ω rounds. In round $i \in \mathbb{N}$: First player-1 chooses $\sigma_1 \in \Sigma_1 \cup \{z\}$, then player-2 responds with $\sigma_2 \in \Sigma_2$.

A play with detected errors $\rho \in ((\Sigma_1 \cup \{z\}) \times \Sigma_2)^\omega$ is formed. In the sequel if it is clear from the context, we shall refer a to play with detected errors simply as a play.

Interpretation of a play with detected errors. Let ρ be a play with detected errors. $\rho' \in (\Sigma_1 \times \Sigma_2)^\omega$ is an *interpretation* of ρ if for every $i \in \mathbb{N}$, $\rho(i)_2 = \rho'(i)_2$ and if $\rho(i)_1 \neq z$ then $\rho(i)_1 = \rho'(i)_1$. The *set of interpretations* of a play ρ is denoted by $\mathit{Interp}(\rho)$. We define the interpretation of a play prefix similarly.

Error count and error rate. Let ρ be a play with detected errors.

The *error count* of a play prefix $\rho[0,n]$ is denoted by $EC(\rho[0,n])$ and it is the number of occurrences of z in $\rho[0,n]$. The *error rate* of a play ρ is denoted by $\mathit{ER}(\rho)$ and is defined as $\limsup_{n\to\infty} \frac{1}{n} \cdot EC(\rho[0,n-1])$.

Winning conditions for plays with detected errors. A *winning condition* is a tuple (L,χ), where $L \subseteq (\Sigma_1 \times \Sigma_2)^\omega$ is the target objective, and $\chi \subseteq ((\Sigma_1 \cup \{z\}) \times \Sigma_2)^\omega$ is the *quantitative threshold* objective. For a play with detected errors ρ, we consider three types of quantitative thresholds.

- *Fixed number of errors.* For $m \in \mathbb{N}$, $\rho \in \chi \Leftrightarrow EC(\rho[0,n-1]) \leq m$ for every $n \geq 0$.
- *Finite number of errors.* $\rho \in \chi$ if z does not occur infinitely often in ρ.
- *Error rate.* For $\delta \in \mathbb{Q}$, $\rho \in \chi \Leftrightarrow \mathit{ER}(\rho) \leq \delta$.

Let ρ be a play with detected errors, player-2 wins in ρ if $\mathit{Interp}(\rho) \subseteq L$ or if $\rho \notin \chi$.

We name a class of games according to the target objective and the quantitative threshold. For example, the class of games with ω regular language target objective and any quantitative threshold χ is named: Regular games with detected errors.

In the sequel, if the target objective is clear from the context, we omit the name of the target objective. In addition instead of defining χ explicitly we simply state the threshold value (i.e., n, *fin* or δ).

Definition 1. *Let $L \subseteq (\Sigma_1 \times \Sigma_2)^\omega$ be the target objective.*

1. *$\mathit{DE}_n(L)$ is a game with detected errors and a winning condition (L,n).*
2. *$\mathit{DE}_{\mathit{fin}}(L)$ is a game with detected errors and a winning condition (L,fin).*
3. *$\mathit{DE}_\delta(L)$ is a game with detected errors and a winning condition (L,δ).*

Strategies for plays with detected errors. A *strategy* determines player next move according to the history of moves (of both players) in the play. Formally, a player-1 strategy is a function $\tau_1 : ((\Sigma_1 \cup \{z\}) \times \Sigma_2)^* \to (\Sigma_1 \cup \{z\})$, a player-2 strategy is a function $\tau_2 : ((\Sigma_1 \cup \{z\}) \times \Sigma_2)^* \times (\Sigma_1 \cup \{z\}) \to \Sigma_2$.

A play with detected errors ρ is played according to player-1 strategy τ_1 if $\rho(i)_1 = \tau_1(\rho[0, i-1])$ for $i \geq 0$. A play according to player-2 strategy is defined similarly.

Finite-memory strategies. A strategy of player-1 is a *finite-memory* strategy if it can be encoded by a deterministic transducer $(M, m_0, \alpha_u, \alpha_n)$ where M is a finite set (the memory of the strategy), $m_0 \in M$ is the initial memory value, $\alpha_u : M \times (\Sigma_1 \cup \{z\}) \times \Sigma_2 \to M$ is an update function, and $\alpha_n : M \to \Sigma_1 \cup \{z\}$ is the next move function. The *size* of the strategy is the cardinality of M. In every round, let m be the current memory value, then the strategy chooses $\sigma_1 = \alpha_n(m)$ as player-1 next move and the memory is updated to $\alpha_u(m, \sigma_1, \sigma_2)$, where σ_2 is player-2 move. Finite-memory strategy for player-2 is defined similarly.

Winning strategy. Player-i strategy τ_i (for $i = 1, 2$) is a *winning strategy* if for every play played according to τ_i, player-i wins. If player-i has a winning strategy, player-i is said to be the *winner* of the game.

3.2 Games with Undetected Errors

In this game, player-1 errors are not detected by player-2, thus the error count and error rate cannot be measured by the amount of zs in the play. In this subsection we give appropriate definitions for plays, play interpretation, error count, error rate, and winning conditions.

Play with undetected errors. A play with undetected error is defined exactly as a play (without errors) in section 2. Thus every play with undetected errors forms $\rho \in (\Sigma_1 \times \Sigma_2)^\omega$.

Interpretation of play with undetected errors. Let $\rho \in (\Sigma_1 \times \Sigma_2)^\omega$ be a play with undetected errors. $\rho' \in (\Sigma_1 \times \Sigma_2)^\omega$ is an *interpretation* of ρ if $\rho(i)_2 = \rho'(i)_2$ for every $i \geq 0$. The *set of interpretations* of a play ρ is denoted by *Interp*(ρ). The interpretations of a play prefix are defined similarly.

Error count and error rate. Since errors are not detected, the existence of an error in a certain position of the play is relative to a specific interpretation of the play. i.e., for play ρ, different interpretations ρ_1, ρ_2 may define different error positions.

Let ρ be a play with undetected errors, and $\rho' \in$ *Interp*(ρ). ρ' is said to have an *error* in position i if $\rho(i)_1 \neq \rho'(i)_1$ The *Error count* of the finite play prefix ρ and $\rho' \in$ *Interp*(ρ), denoted by $EC(\rho, \rho')$, is the number of positions in ρ with errors relatively to ρ'.

The *error rate* of a play ρ and $\rho' \in$ *Interp*(ρ) is denoted by $ER(\rho, \rho') = \limsup_{n\to\infty} \frac{1}{n} \cdot EC(\rho[0, n-1], \rho'[0, n-1])$.

Winning conditions for plays with undetected errors. For $L \subseteq (\Sigma_1 \times \Sigma_2)^\omega$, the *winning condition* is a tuple (L, χ). We refer to L as the *target objective*, and to $\chi \subseteq ((\Sigma_1 \times \Sigma_2)^\omega \times (\Sigma_1 \times \Sigma_2)^\omega)$ as the *quantitative threshold* objective. For a play with undetected errors ρ, we consider three types of quantitative thresholds.

- *Fixed number of errors.* For $m \in \mathbb{N}$, $(\rho, \rho') \in \chi \Leftrightarrow EC(\rho[0, n-1], \rho'[0, n-1]) \leq m$ for every $n \geq 0$.
- *Finite number of errors.* $(\rho, \rho') \in \chi \Leftrightarrow \rho' \in Interp(\rho)$ and ρ' differs from ρ in finite number of positions.
- *Error rate.* For $\delta \in \mathbb{Q}$, $(\rho, \rho') \in \chi \Leftrightarrow ER(\rho, \rho') \leq \delta$.

Let ρ be a play with undetected errors, player-2 wins in play ρ if for every $\rho' \in Interp(\rho)$, either $\rho' \in L$ or $(\rho, \rho') \notin \chi$.

We name a class of games according to the target objective and the quantitative threshold. For example a class of games with languages recognized by a mean-payoff automatons as a target objective, and any quantitative threshold χ is named: mean-payoff games with undetected errors.

In the sequel, if the target objective is clear from the context, we omit the name of the target objective. In addition instead of defining χ explicitly we simply state the threshold value (i.e., n, *fin* or δ).

Definition 2. *Let L be the target objective.*

1. *$UDE_n(L)$ is a game with undetected errors and a winning condition (L, n).*
2. *$UDE_{fin}(L)$ is a game with undetected errors and a winning condition (L, fin).*
3. *$UDE_\delta(L)$ is a game with undetected errors and a winning condition (L, δ).*

4 Regular Games with Detected Errors

Regular games with detected errors consists of ω regular language L and an error threshold objective χ.

Decision problem. *Deciding who is the winner*:

- Input: An ω regular language $L \subseteq (\Sigma_1 \times \Sigma_2)^\omega$ (given either as a MSO formula or as a parity automaton), numbers $n \in \mathbb{N}$ and $\delta \in \mathbb{Q}$.
- Output: decide who is the winner of $\text{DE}_n(L)$, $\text{DE}_{fin}(L)$ and $\text{DE}_\delta(L)$.

Computation problems

- The *computability of winning parameters* problem is:
  - Input: ω regular language $L \subseteq (\Sigma_1 \times \Sigma_2)^\omega$
  - Output: Find the minimal $n \in \mathbb{N}$ such that player-1 is the winner of $\text{DE}_n(L)$ and the minimal $\delta \in \mathbb{Q}$ such that player-1 is the winner of $\text{DE}_\delta(L)$.
- The *computability of winning strategy* problem is:
  - Input: An ω regular language $L \subseteq \Sigma_1 \times \Sigma_2)^\omega$, numbers $n \in \mathbb{N}$ and $\delta \in \mathbb{Q}$.
  - Output: A finite description of the winning strategies for the winning player of $\text{DE}_n(L)$, $\text{DE}_{fin}(L)$ and $\text{DE}_\delta(L)$.

The main result of this section is

Theorem 1. *Let L be an ω regular language.*

1. *For every $n \in \mathbb{N}$ and $\delta \in \mathbb{Q}$ It is decidable who is the winner of $DE_n(L)$, $DE_{fin}(L)$ and $DE_\delta(L)$.*
2. *The winning parameters problem is computable.*
3. *The winning strategy problem is computable.*

A proof outline for Theorem 1. We present an immediate reduction from $\mathrm{DE}_n(L)$ and $\mathrm{DE}_{fin}(L)$ games, where L is an ω regular language to regular games without errors.

In addition we present a reduction from $\mathrm{DE}_\delta(L)$ games, where L is an ω regular language to mean-payoff parity games (studied in [5]). This reduction together with results of [5] imply the decidability of who is the winner of $\mathrm{DE}_\delta(L)$ game and the computability of the minimal δ such that player-1 is the winner of $\mathrm{DE}_\delta(L)$.

The computability of the minimal $m \in \mathbb{N}$ such that player-1 is the winner of $\mathrm{DE}_m(L)$ for an ω regular language L is obtained by the following lemma.

Lemma 1. *Let $\mathcal{A} = (\Sigma_1 \times \Sigma_2, Q, Q_0, E)$ be a an automaton, and let ϕ be a tail objective. Then player-2 is the winner of $DE_{fin}(L_{\mathcal{A},\phi}) \Leftrightarrow$ player-2 is the winner of $DE_m(L_{\mathcal{A},\phi})$ for $m = 2^{|Q|}$.*

Due to Lemma 1 and Theorem 1(1) and since every parity objective is a tail objective, one can decide the winner of $\mathrm{DE}_n(L)$ for every $n = 0, 1, \ldots, 2^{|Q|}$ and return the minimal n such that player-1 is the winner of $\mathrm{DE}_n(L)$.

Need of infinite memory strategies. The following proposition asserts the need of infinite memory strategies for player-1 for regular games with detected errors and error rate threshold.

Proposition 1. *There exists an ω regular language $L \subseteq (\Sigma_1 \times \Sigma_2)^\omega$ and a threshold $\delta \in \mathbb{Q}$ such that player-1 is the winner of $DE_\delta(L)$, however player-1 does not have a finite memory winning strategy.*

Proof (of Proposition 1). Fix $\Sigma_1 = \Sigma_2 = \{a, b\}$. Consider the following ω language $L \subseteq (\Sigma_1 \times \Sigma_2)^\omega$. A tuple $(X_1, X_2) \in (\Sigma_1 \times \Sigma_2)^\omega$ is in the language L if the following conditions are hold.

1. $(\exists^\omega t X_1(t) = b) \rightarrow (\exists t(\forall t' > t(X_2(t') = a)))$
2. $(\forall t X_1(t) = a) \rightarrow (\exists^\omega t(X_2(t) = b))$

First, let us show that for $\delta = 0$, player-1 is the winner of $\mathrm{DE}_\delta(L)$. The following strategy τ is a winning strategy for player-1: In round $i \in \mathbb{N}$, if there exists $n \in \mathbb{N}$ such that $i = 2^n$, play z, otherwise play a.

Clearly the error rate of a play played according to τ is 0. Player-1 violates the target objective in the following way. If player-2 played infinitely often b, then the play interpretation which replaces every occurrence of z with b violates the first condition. If player-2 played finitely often b, then the play interpretation which replaces every occurrence of z with a violates the second condition.

Thus, player-1 is the winner of $\mathrm{DE}_\delta(L)$.

Second, the reader can verify that for every $n \in \mathbb{N}$, every player-1 finite memory winning strategy for $\mathrm{DE}_{\frac{1}{n}}(L)$ requires memory size of at least n. Hence, player-1 does not have a finite memory winning strategy for $\mathrm{DE}_\delta(L)$ for $\delta = 0$, which concludes the proof of proposition 1. □

5 Regular Games with Undetected Errors

Analyzing games with errors is much more difficult when the errors are not detected. As in section 4 we are interesting in the decidability of who is the winner and in the computation of winning parameters and winning strategies. However, we are able to give only partial answer to these questions.

The main result of this section is:

Theorem 2. *The following problem is undecidable.*

- *Input: $\delta \in (0, 1) \cap \mathbb{Q}$ and an ω regular language L.*
- *Question: Decide who is the winner of $UDE_\delta(L)$.*

A proof outline for Theorem 2. In [10,8] it was proved that the universality problem of a non-deterministic mean-payoff automaton is undecidable. We provide a reduction from this problem to the problem of who is the winner of $\mathrm{UDE}_\delta(L)$ game.

The next remark deals with the simple cases.

Remark 1. *Let L be an ω regular language.*

1. *For $n \in \mathbb{N}$ it is decidable who is the winner of $UDE_n(L)$, and a winning strategy is computable.*
2. *It is decidable who is the winner of $UDE_{fin}(L)$, and the winning strategy is computable.*
3. *For $\delta = 1$, it is decidable who is the winner of $UDE_\delta(L)$, and the winning strategy is computable.*

Proof (of Remark 1). All items in Remark 1 are immediately proved by encoding the winning condition as a MSO formula. □

The following two decision problems remain open.

1. The *bounded errors* problem asks if for an ω-regular language L, there exists $n \in \mathbb{N}$ such that player-1 is the winner of $\mathrm{UDE}_n(L)$.
2. The *zero error rate* problem, asks for an ω-regular language L and for $\delta = 0$, who is the winner of $\mathrm{DE}_\delta(L)$.

At first glance one can hope to answer the bounded number of errors problem with the same technique used in Lemma 1, i.e., try to prove that if for an ω-regular language L player-2 is the winner of $\mathrm{UDE}_{fin}(L)$, then there exists $m_L \in \mathbb{N}$ such that player-2 is the winner of $\mathrm{UDE}_{m_L}(L)$. The next proposition show that this is not the case.

Proposition 2. *There exists an ω regular language L such that player-1 is the winner of $UDE_{fin}(L)$, however for every $n \in \mathbb{N}$ player-2 is the winner of $UDE_n(L)$.*

6 Mean-Payoff Games with Errors

Quantitative languages do not enjoy the same robustness as ω regular languages. In particular, the class of quantitative languages recognized by the deterministic mean-payoff automata is a strict subset of the class of quantitative languages recognized by the non-deterministic mean-payoff automata [8]. In addition, deterministic mean-payoff automatons are not closed under the conjunction operation [8].

In this section we consider only quantitative languages defined by a **deterministic** mean-payoff automaton. We also assume that the mean-payoff condition is an objective of the form of $\text{MeanPayoffInf}^{\geq}(0)$. However unless noted otherwise, the proofs can be easily modified for $\text{MeanPayoffInf}^{>}(0)$, $\text{MeanPayoffSup}^{\geq}(0)$ and $\text{MeanPayoffSup}^{>}(0)$ objectives.

The next theorem is the main result of this section.

Theorem 3. *Let $L \subseteq (\Sigma_1 \times \Sigma_2)^\omega$ be a quantitative language recognized by a mean-payoff automaton $\mathcal{A}$.*

1. *For every $\delta > 0$, it is undecidable who is the winner of $DE_\delta(L)$.*
2. *For every $\delta > 0$, it is undecidable who is the winner of $UDE_\delta(L)$.*
3. *For every $n \in \mathbb{N}$ it is decidable who is the winner of $DE_n(L)$ and of $DE_{fin}(L)$.*
4. *For $\delta = 0$, it is decidable who is the winner of $DE_\delta(L)$. Moreover, player-2 is the winner of $DE_\delta(L)$, for $\delta = 0 \Leftrightarrow$ player-2 is the winner $DE_{fin}(L)$*[1].

A proof outline for Theorem 3. The first two items are proved by an immediate reduction from the universality problem of non-deterministic mean-payoff automatons. This problem was proved to be undecidable in [10,8].

For the last two items we prove that for every quantitative language defined by a mean-payoff automaton $\mathcal{A}$, there exists a computable $m_\mathcal{A}$ such that: Player-1 is the winner of $\text{DE}_{m_\mathcal{A}}(L) \Leftrightarrow$ Player-1 is the winner of $\text{DE}_{\mathit{fin}}(L) \Leftrightarrow$ Player-1 is the winner of $\text{DE}_\delta(L)$ for $\delta = 0$.

Determining the winner of $\text{DE}_m(L)$ for fixed $m \in \mathbb{N}$ is done by a reduction to a game without errors with winning condition of the form $\bigcap_{i=1}^{f(m)} L_i$, where $f(m)$ is computable from m and L_i is a quantitative language computable from L, m and i. The proof is concluded with the following lemma.

Lemma 2. *Let $\mathcal{A}_1, \ldots, \mathcal{A}_k$ be mean-payoff automatons. Let $L = \bigcap_{i=1}^{k} L_{\mathcal{A}_i, MeanPayoffInf^{\geq}(0)}$. Then it is decidable who is the winner of the Gale-Stewart game G_L.*

The proof of Lemma 2 is given is section 7.

Need for infinite memory strategies. The following proposition assert that player-2 has to use infinite memory strategies for DE_n games even for $n = 1$.

Proposition 3. *There exists an ω language L recognized by a mean-payoff automaton such that player-2 is the winner of $DE_n(L)$ for every $n \in \mathbb{N}$, however player-2 does not have a finite memory winning strategy for $DE_n(L)$ even for $n = 1$.*

[1] The proof of item 4 holds only for $\text{MeanPayoffInf}^{\geq}(0)$ objective.

7 Multidimensional Mean-Payoff Games

Multidimensional mean-payoff games are an extension of mean-payoff games to graphs with multidimensional weights. A multidimensional mean-payoff objective is a boolean combination of one dimensional mean-payoff objectives. For example, let G be a graph with weight function $w : E \to \mathbb{Z}^2$, and $\nu \in \mathbb{Q}$. Let ϕ_1 be the $\text{MeanPayoffInf}^{\geq}(\nu)$ objective according to the projection of w to the first dimension, and ϕ_2 be the $\text{MeanPayoffInf}^{\geq}(\nu)$ objective according to the projection of w to the second dimension. A possible multidimensional objective can be any boolean combination of ϕ_1 and ϕ_2.

One interesting form of multidimensional objective is a conjunction of one dimensional mean-payoff objectives. This objective was introduced in [7]. However, the decidability of who is the winner was proved only for the case where player-2 is restricted to finite memory strategies. The general case (i.e., when player-2 strategy is not limited to finite memory) was stated as an open question. In this section we prove decidability and present tight complexity bounds to the question of who is the winner. Thus we answer an open question from [7], and complete the proof of Theorem 3.

Another interesting multidimensional objective is a positive boolean combination of one dimensional mean-payoff objectives. For this case we prove decidability of who is the winner.

Since in this section we consider games without errors, we find it convenient to use the standard definitions of games on graphs presented below.

Game graph. A *game graph* $G = \langle Q, E\rangle$ consists of a finite set Q of states partitioned into player-1 states Q_1 and player-2 states Q_2. The graph is bipartite, i.e., $E \subseteq (Q_1 \times Q_2) \cup (Q_2 \times Q_1)$. In addition, every state must have an outdegree of at least 1.

Plays and strategies. A game on G starting from an *initial state* $q_0 \in Q$ is played in rounds as follows. If the game is in a player-1 state, then player-1 chooses an outgoing edges; otherwise the game is in a player-2 state, and player-2 chooses an outgoing edges. The game results is a *play* from q_0, i.e., an infinite path $\rho = q_0e_0q_1e_1\dots$ such that e_i is an edge from q_i to q_{i+1} for all $i \geq 0$. A strategy for player-1 is a function $\tau : (Q \times E)^* \times Q_1 \to E$. A strategy for player-2 is a function $\tau : (Q \times E)^* \times Q_2 \to E$.

Mean-payoff vector. Let G be a game graph, $k \in \mathbb{N}$, and $w : E \to \mathbb{Z}^k$ be a multi-dimension weight function for the edges. We denote by w_i the projection of w to dimension i. Let $\pi \in (Q \times E)^\omega$ be an infinite path (i.e., play) in G. The *mean-payoff vector* $\overrightarrow{MP}(\pi) = (\underline{MP}(\pi)_1, \dots, \underline{MP}(\pi)_k, \overline{MP}(\pi)_1, \dots, \overline{MP}(\pi)_k)$ has $2k$ dimensions. In the first k dimensions, $\underline{MP}(\pi)_i = \underline{MP}(w_i, \pi)$, for $i = 1, \dots, k$. Similarly, in the last k dimensions, $\overline{MP}(\pi)_i = \overline{MP}(w_i, \pi)$, for $i = 1, \dots, k$.

Multidimensional mean-payoff objectives. For a k-dimensional game and $S \subseteq \{1, \dots, k\}$ and $\nu \in \mathbb{Q}$ we denote by $\bigwedge \text{MeanPayoffSup}^{\geq}_S(\nu)$ the following objectives:

Player-2 wins $\bigwedge$ MeanPayoffSup$^{\geq}_{S}(\nu)$ in a play π if $\overline{MP}(\pi)_i \geq \nu$, for every $i \in S$.

The objectives $\bigwedge$ MeanPayoffSup$^{>}_{S}(\nu)$, $\bigwedge$ MeanPayoffInf$^{\geq}_{S}(\nu)$ and $\bigwedge$ MeanPayoffInf$^{>}_{S}(\nu)$, are defined similarly. When $S = \{1, \ldots, k\}$ we will drop the subscript and write $\bigwedge$ MeanPayoffSup$^{\geq}(\nu)$ instead of $\bigwedge$ MeanPayoffSup$^{\geq}_{\{1,\ldots,k\}}(\nu)$.

An objective is *conjunctive* if it is a conjunction of conditions of the form $\overline{MP}(\pi)_i \sim \nu$ and $\underline{MP}(\pi)_i \sim \nu$, where $\sim \in \{>, \geq\}$.

$\bigvee\bigwedge$ MeanPayoffInf$^{\geq,>}(\nu)$ is the class of objectives of the form $\bigvee_{i\in\{1,\ldots,m\}} \bigwedge_{j\in S_i} \underline{MP}(\pi)_j \sim \nu$, where $\sim \in \{>, \geq\}$ and $S_1, \ldots, S_m \subseteq \{1, \ldots, k\}$.

$\bigwedge\bigvee$ MeanPayoffSup$^{\geq,>}(\nu)$ is the class of objectives of the form $\bigvee_{i\in\{1,\ldots,m\}} \bigwedge_{j\in S_i} \overline{MP}(\pi)_j \sim \nu$, where $\sim \in \{>, \geq\}$ and $S_1, \ldots, S_m \subseteq \{1, \ldots, k\}$.

Let w be a weight function and $b, c \in \mathbb{Q}$ such that $b > 0$, and π be a play. Then $\overline{MP}(w, \pi)_i \geq \nu$ iff $\overline{MP}(w', \pi)_i \geq b\nu + c$, where w' is a weight function equal to w in all dimensions except i and $w'_i = bw_i + c$. Similar equivalences hold for $\underline{MP}$.

For $\overrightarrow{\nu} \in \mathbb{Q}^{2k}$ an objective MultiDimensionMeanPayoff($\overrightarrow{\nu}$) is defined as $\{\pi \in (Q \times E)^{\omega} | \overrightarrow{MP}(\pi) \geq \overrightarrow{\nu}\}$. Clearly determining the winner of a game with MultiDimensionMeanPayoff($\overrightarrow{\nu}$) objective is log-space reducible to determining the winner of a game with conjunctive objective.

The next two theorems are the main result of this section.

Theorem 4. *For every finite game graph $G = \langle Q, E\rangle$ with a weight function $w : E \to \mathbb{Z}^k$, and a conjunctive objective ϕ: player-1 has a winning strategy iff he has a memoryless winning strategy.*

Theorem 5. *For input: game graph $G = \langle Q, E, w : E \to \mathbb{Z}^k\rangle$, initial state $q \in Q$ and $\nu \in \mathbb{Q}$. The problem of deciding whether player-2 is the winner for a multidimensional mean-payoff objective ϕ is*

1. *In coNP $\cap$ NP when $\phi \in \{\bigwedge$ MeanPayoffSup$^{\geq}(\nu), \bigwedge$ MeanPayoffSup$^{>}(\nu)\}$. Moreover the problem of determining the winner is in $\widetilde{P}$ (i.e., it has a pseudo-polynomial-time algorithm) and there is a polynomial time Cook reduction to the problem of determining the winner of a (one dimensional) mean-payoff game.*
2. *(Strongly) coNP complete when $\phi \in \{\bigwedge$ MeanPayoffInf$^{\geq}(\nu), \bigwedge$ MeanPayoffInf$^{>}(\nu)\}$.*
3. *(Strongly) coNP complete when ϕ is an arbitrary conjunctive objective.*
4. *(Strongly) coNP complete when $\phi \in \bigvee\bigwedge$ MeanPayoffInf$^{\geq,>}(\nu)$ and (Strongly) NP complete when $\phi \in \bigwedge\bigvee$ MeanPayoffSup$^{\geq,>}(\nu)$.*

We recall that an algorithm runs in pseudo-polynomial time, for input $G = \langle Q, E, w : E \to \mathbb{Z}^k\rangle$, if its running time is polynomial in the size of G and the description of w, when the values of the weight function w are represented in unary. A (co)NP-complete problem is called strongly (co)NP-complete if it is proven that it cannot be solved by a pseudo-polynomial time algorithm unless P=NP.

We would like to note that the lower bounds of Theorems 5(2)-(3) are easily obtained from proofs of lower bounds in [7]. Our proof of Theorems 5(2) relies on the corresponding result of [7] for the case when the players are restricted to use only finite state memory strategies.

We are now ready to prove Lemma 2.

Proof of Lemma 2. We begin with the following remark.

Remark 2. *Let $\mathcal{A}_1$, $\mathcal{A}_2$ be automatons with a $\bigwedge$ MeanPayoffInf$^{\geq}(0)$ objective. Then there exists an automaton $\mathcal{A}_3$ with a $\bigwedge$ MeanPayoffInf$^{\geq}(0)$ objective such that $L_{\mathcal{A}_3} = L_{\mathcal{A}_1} \cap L_{\mathcal{A}_2}$. Moreover $\mathcal{A}_3$ is computable from $\mathcal{A}_1$ and $\mathcal{A}_2$, and $|\mathcal{A}_3| = |\mathcal{A}_1| \times |\mathcal{A}_2|$.*

By Remark 2 we conclude the proof of Lemma 2 in the following way. Let $\mathcal{A}_1, \ldots, \mathcal{A}_k$ be a one dimension mean-payoff automatons. Let $L = \bigcap_{i=1}^{k} L_{A_i,\text{MeanPayoffInf}^{\geq}(0)}$. Then one can compute a k dimensional mean-payoff automaton $\mathcal{A}$ such that $L_{\mathcal{A},\bigwedge \text{MeanPayoffInf}^{\geq}(0)} = L$. By Theorem 5 deciding the winner in the Gale-Stewart game defined by L is decidable.

Moreover, Remark 2 implies that Theorem 3 can be extended also to quantitative languages defined by a multidimensional mean-payoff automaton with $\bigwedge$ MeanPayoffInf$^{\geq}(0)$ objective.

Note that by the same arguments we can prove Lemma 2 for $\bigwedge$ MeanPayoffInf$^{>}(0)$, $\bigwedge$ MeanPayoffSup$^{\geq}(0)$ and $\bigwedge$ MeanPayoffSup$^{>}(0)$ objectives.

8 Conclusion and Further Work

In this work we investigated games with errors and obtained decidability results described in Table 1. Our proofs immediately imply a 2-EXPTIME upper bound for the complexity of determining the winner of a game with errors (for the decidable fragments). Further work is required to give tighter complexity bounds. Further work may also consider additional classes of specification languages. While Table 1 contains five open problems, we believe that the following two open problems may have applications outside the framework of games with errors.

1. Decidability of the bounded number of errors problem for regular games with undetected errors.
2. Decidability of who is the winner of UDE$_\delta(L)$ game for ω regular language L and $\delta = 0$.

We also investigated multidimensional mean-payoff games and provided complexity bounds for interesting fragments of these games. The following two interesting problems are open.

1. Decidability of who is the winner for arbitrary multidimensional mean-payoff objective (as defined in section 7). For example, the decidability of who is the winner for the objective
$$\{\pi \in (Q \times E)^\omega | (\underline{MP}(\pi)_1 \geq 0) \wedge (\overline{MP}(\pi)_2 \geq 0) \vee (\underline{MP}(\pi)_3 \geq 0)\}$$
was not determined in this paper.

2. Complexity of solving multidimensional mean-payoff games with fixed number of dimensions and a $\bigwedge \mathrm{MeanPayoffInf}^{\geq}(0)$ objective. Specifically whether the problem of determining the winner is in NP∩coNP, and/or in $\widetilde{P}$?

References

1. Alur, R., Degorre, A., Maler, O., Weiss, G.: On omega-languages defined by mean-payoff conditions. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 333–347. Springer, Heidelberg (2009)
2. Bjorklund, H., Sandberg, S., Vorobyov, S.: Memoryless determinacy of parity and mean payoff games: a simple proof. Theoretical Computer Science 310, 365–378 (2004)
3. Brázdil, T., Jancar, P., Kucera, A.: Reachability games on extended vector addition systems with states. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 478–489. Springer, Heidelberg (2010)
4. Büchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Transactions of the AMS 138, 295–311 (1969)
5. Chatterjee, K., Henzinger, T.A., Jurdzinski, M.: Mean payoff parity games. In: Proc. of LICS, pp. 178–187. IEEE Computer Society, Los Alamitos (2005)
6. Chatterjee, K., Doyen, L.: Energy Parity Games. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 599–610. Springer, Heidelberg (2010)
7. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.F.: Generalized mean payoff and Energy Games. To appear in Proc. of FSTTCS (2010)
8. Chatterjee, K., Doyen, L., Edelsbrunner, H., Henzinger, T.A., Rannou, P.: Mean-Payoff Automaton Expressions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 269–283. Springer, Heidelberg (2010)
9. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Algorithms for omega-regular games with imperfect information'. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 287–302. Springer, Heidelberg (2006)
10. Degorre, A., Doyen, L., Gentilini, R., Raskin, J.-F., Torunczyk, S.: Energy and Mean-Payoff Games with Imperfect Information. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 260–274. Springer, Heidelberg (2010)
11. Ehrenfeucht, A., Mycielski, J.: International journal of game theory. Positional Strategies for Mean-Payoff Games 8, 109–113 (1979)
12. Grädel, E., Thomas, W., Wilke, T.: Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)
13. Holtmann, M., Kaiser, L., Thomas, W.: Degrees of Lookahead in Regular Infinite Games. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 252–266. Springer, Heidelberg (2010)
14. Kosaraju, S.R., Sullivan, G.F.: Detecting cycles in dynamic graphs in polynomial time. In: Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, pp. 398–406 (1988)
15. Reif, J.H.: The complexity of two-player games of incomplete information. Journal of Computer and System Sciences 29(2), 274–301 (1984)
16. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, vol. B, pp. 133–191. Elsevier Science Pub., Amsterdam (1990)
17. Zwick, U., Paterson, M.: The Complexity of Mean Payoff Games on Graphs. Theoretical Computer Science 158, 343–359 (1996)

Probabilistic Modal μ-Calculus with Independent Product

Matteo Mio

LFCS, School of Informatics, University of Edinburgh

Abstract. The probabilistic modal μ-calculus pLμ (often called the quantitative μ-calculus) is a generalization of the standard modal μ-calculus designed for expressing properties of probabilistic labeled transition systems. The syntax of pLμ formulas coincides with that of the standard modal μ-calculus. Two equivalent semantics have been studied for pLμ, both assigning to each process-state p a value in $[0, 1]$ representing the probability that the property expressed by the formula will hold in p: a *denotational semantics* and a *game semantics* given by means of two player stochastic games. In this paper we extend the logic pLμ with a second conjunction called *product*, whose semantics interprets the two conjuncts as probabilistically independent events. This extension allows one to encode useful operators, such as the modalities *with probability one* and *with non-zero probability*. We provide two semantics for this extended logic: one denotational and one based on a new class of games which we call *tree games*. The main result is the equivalence of the two semantics. The proof is carried out in ZFC set theory extended with Martin's Axiom at the first uncountable cardinal.

1 Introduction

The modal μ-calculus Lμ [10] is a very expressive logic, for expressing properties of reactive systems (labeled transition systems), obtained by extending classical propositional modal logic with least and greatest fixed point operators. In the last decade, a lot of research has focused on the study of reactive systems that exhibit some kind of probabilistic behavior, and logics for expressing their properties [14,5,9,6,8]. Probabilistic labeled transition systems (PLTS's) [15] are a natural generalization of standard LTS's to the probabilistic setting, as they allow both (countable) non-deterministic and probabilistic choices.

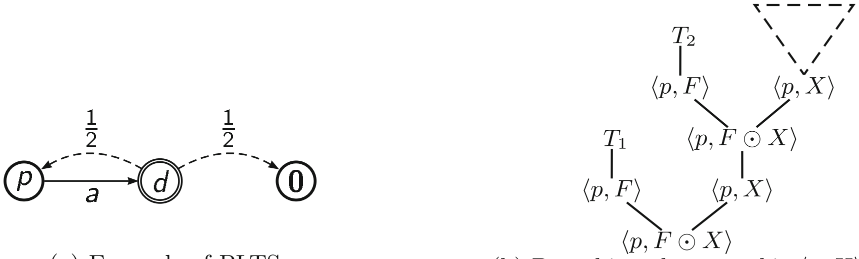
The probabilistic modal μ-calculus pLμ, introduced in [14,9,5], is a generalization of Lμ designed for expressing properties of PLTS's. This logic was originally named the *quantitative* μ-calculus, but since other μ-calculus-like logics, designed for expressing properties of non-probabilistic systems, have been given the same name (e.g. [7]), we adopt the *probabilistic* adjective. The syntax of the logic pLμ coincides with that of the standard μ-calculus. The denotational semantics of pLμ [14,5] generalizes that of Lμ, by interpreting every formula F as a map $[\![F]\!] : P \rightarrow [0, 1]$, which assigns to each process p a *degree of truth*. In [12], the authors introduce an alternative semantics for the logic pLμ. This semantics,

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 290–304, 2011.

given in term of two player stochastic (parity) games, is a natural generalization of the two player (non-stochastic) game semantics for the logic Lμ [16]. As in Lμ games, the two players play a game starting from a configuration $\langle p, F\rangle$, where the objective for Player 1 is to produce a path of configurations along which the outermost fixed point variable X unfolded infinitely often is bound by a greatest fixed point in F. On a configuration of the form $\langle p, G_1 \vee G_2\rangle$, Player 1 chooses one of the disjuncts G_i, $i \in \{1, 2\}$, by moving to the next configuration $\langle p, G_i\rangle$. On a configuration $\langle p, G_1 \wedge G_2\rangle$, Player 2 chooses a conjunct G_i and moves to $\langle p, G_i\rangle$. On a configuration $\langle p, \mu X.G\rangle$ or $\langle p, \nu X.G\rangle$ the game evolves to the configuration $\langle p, G\rangle$, after which, from any subsequent configuration $\langle q, X\rangle$ the game again evolves to $\langle q, G\rangle$. On configurations $\langle p, \langle a\rangle G\rangle$ and $\langle p, [a]\, G\rangle$, Player 1 and 2 respectively choose a transition $p \xrightarrow{a} d$ in the PLTS and move the game to $\langle d, G\rangle$. Here d is a probability distribution (this is the key difference between pLμ and Lμ games) and the configuration $\langle d, G\rangle$ belongs to Nature, the probabilistic agent of the game, who moves on to the next configuration $\langle q, G\rangle$ with probability $d(q)$. This game semantics allows one to interpret formulae as expressing, for each process p, the (limit) probability of a *property*, specified by the formula, holding at the state p. In [12], the equivalence of the denotational and game semantics for pLμ on finite models, was proven. The result was recently extended to arbitrary models by the present author [13].

In this paper we consider an extension of the logic pLμ obtained by adding to the syntax of the logic a second conjunction operator ($\cdot$) called *product* and its De Morgan dual operator called *coproduct* ($\odot$). We call this extension the *probabilistic modal μ-calculus with independent product*, or just pL$\mu^{\odot}$. The denotational semantics of the product operator is defined as $[\![F \cdot G]\!](p) = [\![F]\!](p) \cdot [\![G]\!](p)$, where the product symbol in the right hand side is multiplication on reals. Such an operator was already considered in [9] as a possible generalization of standard boolean conjunction to the lattice $[0, 1]$. Our logic pL$\mu^{\odot}$ is novel in containing both ordinary conjunctions and disjunctions ($\wedge$ and $\vee$) and independent products and coproducts ($\cdot$ and $\odot$). While giving a denotational semantics to pL$\mu^{\odot}$ is straightforward, the major task we undertake in this paper is to extend the game semantics of [12] to the new connectives. The game semantics implements the intuition that $H_1 \cdot H_2$ expresses the probability that H_1 and H_2 both hold if verified independently of each other.

To capture formally this intuition we introduce a game semantics for the logic pL$\mu^{\odot}$ in which independent execution of many instances of the game is allowed. Our games build on those for pLμ outlined above. Novelty arises in the game interpretation of the game-states $\langle p, H_1 \cdot H_2\rangle$ and $\langle p, H_1 \odot H_2\rangle$: when during the execution of the game one of these kinds of nodes is reached, the game is split into two concurrent and independent sub-games continuing their executions from the states $\langle p, H_1\rangle$ and $\langle p, H_2\rangle$ respectively. The difference between the game-interpretation of product and coproduct operators is that on a product configuration $\langle p, H_1 \cdot H_2\rangle$, Player 1 has to win in both generated sub-games, while on a coproduct configuration $\langle p, H_1 \odot H_2\rangle$ Player 1 needs to win just one of the two generated sub-games.



(a) Example of PLTS

(b) Branching play rooted in $\langle p, H\rangle$

Fig. 1. Illustrative example

To illustrate the main ideas, let us consider the PLTS of figure 1(a) and the pLμ formula $F = \langle a\rangle\langle a\rangle tt$ which asserts the possibility of performing two consecutive a-steps. The probability of F being satisfied at p is $\frac{1}{2}$, since after the first a-step, the process $\mathbf{0}$ is reached with probability $\frac{1}{2}$ and no further a-step is possible. Let us consider the pL$\mu^{\odot}$ formula $H = \mu X.F \odot X$. Figure 1(b) depicts a play in the game starting from the configuration $\langle p, H\rangle$ (fixed-point unfolding steps are omitted). The branching points represent places where coproduct is the main connective, and each T_i represents play in one of the independent subgames for $\langle p, F\rangle$ thereupon generated. We call such a tree, describing play on all independent subgames, a *branching play.* Since all branches are coproducts, and the fixpoint is a least fixpoint, the objective for Player 1 is to win at least one of the games T_i. Since the probability of winning a particular game T_i is $\frac{1}{2}$, and there are infinitely many independent such games, almost surely Player 1 will win one of them. Therefore the game semantics assigns H at p the value 1.

The above example illustrates an interesting application of the new operators, namely the possibility of encoding the *qualitative* probabilistic modalities $\mathbb{P}_{>0}F$ (F holds with *probability greater than zero*) and $\mathbb{P}_{=1}F$ (F holds *with probability one*), which are equivalent to the pL$\mu^{\odot}$ formulae $\mu X.F \odot X$ and $\nu X.F \cdot X$ respectively. These encodings, which are easily seen to be correct denotationally, provide a novel game interpretation for the qualitative probabilistic modalities, which makes essential use of the new branching features of pL$\mu^{\odot}$ games (giving a direct game interpretation to the qualitative modalities seems no easier than giving the game semantics for the whole of pL$\mu^{\odot}$.) Moreover, they show that the interpretation of pL$\mu^{\odot}$ formulae is, in general, not continuous on the free variables: $\mathbb{P}_{>0}Y$ is an example of pL$\mu^{\odot}$ formula discontinuous on Y, since $[\![\mathbb{P}_{>0}Y]\!]_{\rho}(p) = 1$ if $\rho(Y)(p) > 0$ and $[\![\mathbb{P}_{>0}Y]\!]_{\rho}(p) = 0$ otherwise, where ρ interprets every variable Y as a map from process-states to $[0, 1]$. Other useful properties can be expressed by using these probabilistic modalities in the scope of fixed point operators. Some interesting formulae include $\mu X.\big(\langle a\rangle X \vee (\mathbb{P}_{=1}H)\big)$, $\nu X.\big(\mathbb{P}_{>0}\langle a\rangle X\big)$ and $\mathbb{P}_{>0}\big(\nu X.\langle a\rangle X\big)$: the first assigns to a process p the probability of eventually reaching, by means of a sequence of a-steps, a state in which

H holds with probability one; the second, interpreted on a process p, has value 1 if there exists an infinite sequence of possible (in the sense of having probabilty greater than 0) a-steps starting from p, and 0 otherwise; the third formula, express a stronger property, namely it assigns to a process p value 1 if the probability of making (starting from p) an infinite sequence of a-steps is greater than 0, and value 0 otherwise. Moreover, every property expressible in the qualitative fragment of PCTL [8] can be formulated as a pL$\mu^{\odot}$ formula.

Formalizing the pL$\mu^{\odot}$ games outlined above is a surprisingly technical undertaking. To account for the *branching plays* that arise, we introduce a general notion of *tree game* which is of interest in its own right. Tree games generalize 2-player stochastic games, and are powerful enough to encode certain classes of games of imperfect information such as Blackwell games [2]. Tree games can also be used to formulate other notions that appear in the literature on stochastic games such as *qualitative determinacy* [3,1] and branching-time winning objectives [4]. This, as well as the encoding of qualitative PCTL mentioned above, will appear in the author's forthcoming PhD thesis.

A further level of difficulty arises in expressing when a branching play in a pL$\mu^{\odot}$ game is considered an objective for Player 1. This is delicate because branching plays can contain infinitely many interleaved occurrences of product and coproduct operations (so our simple explanation of such nodes above does not suffice). To account for this, branching plays are themselves considered as ordinary 2-player (parity) games with coproduct nodes as Player 1 nodes, and product nodes as Player 2 nodes. Player 1's goal in the *outer* pL$\mu^{\odot}$ game is to produce a branching play for which, when itself considered as a game, the *inner* game, he has a winning strategy. To formalize the class of tree games whose objective is specified by means of *inner* games, we introduce the notion of 2*-player stochastic meta-game.*

Our main technical result is the equivalence of the denotational semantics and the game semantics for the logic pL$\mu^{\odot}$. As in [13] the proof of equivalence of the two semantics is based on the *unfolding method* of [7]. However there are significant complications, notably, the transfinite inductive characterization of the set of winning branching plays in a given pL$\mu^{\odot}$-game (section 6) and the lack of denotational continuity on the free variables taken care by the game-theoretic notion of *robust* Markov branching play (section 7). Moreover, because of the complexity of the objectives described by means of *inner games*, the proof is carried out in ZFC set theory extended with MA$_{\aleph_1}$ (Martin's Axiom at $\aleph_1$) which is known to be consistent with ZFC. We leave open the question of whether our result is provable in ZFC alone; we do not know if this is possible even restricting the equivalence problem to finite models.

The rest of the paper is organized as follows: in section 2, we fix some terminology and discuss the property MA$_{\aleph_1}$. In section 3, we define the syntax and the denotational semantics of the logic pL$\mu^{\odot}$. In section 4, the class of stochastic tree games, and its sub-class given by two player stochastic meta-parity games, are introduced in detail. In section 5, the game semantics of pL$\mu^{\odot}$ is defined in terms of two player stochastic meta-parity games. In section 6, we provide a transfinite inductive characterization of the objective of the game associated with a formula $\mu X.F$. In section 7, we sketch the main ideas of the proof of the main theorem, which states the equivalence of the two semantics.

2 Background Definitions and Notation

Definition 1 (Probability distributions). A probability distribution d over a set X is a function $d : X \rightarrow [0,1]$ such that $\sum_{x \in X} d(x) = 1$. The *support* of d, denoted by $supp(d)$ is defined as the set $\{x \in X \mid d(x) > 0\}$. We denote with $\mathcal{D}(X)$ the set of probability distributions over X.

Definition 2 (PLTS [15]). Given a countable set L of labels, a *Probabilistic Labeled Transition System* is a pair $\langle P, \{\xrightarrow{a}\}_{a \in L}\rangle$, where P is a countable set of states and $\xrightarrow{a} \subseteq P \times \mathcal{D}(P)$ for every $a \in L$. In this paper we restrict our attention to those PLTS such that for every $p \in P$ and every $a \in L$, the set $\{d \mid p \xrightarrow{a} d\}$ is countable. We refer to the countable set $\bigcup_{a \in L} \bigcup_{p \in P} \{d \mid p \xrightarrow{a} d\}$ as the set of probability distributions of the PLTS.

Definition 3 (Lattice operations). Given a set X, we denote with 2^X the set of all subsets $Y \subseteq X$. Given a complete lattice $(X, <)$, we denote with $\bigsqcup : 2^X \rightarrow X$ and $\bigsqcap : 2^X \rightarrow X$ the operations of join and meet respectively.

In the following we assume standard notions of basic topology and basic measure theory. The topological spaces we consider will always be 0-dimensional Polish spaces. We specify a probability measure on such a space by assigning compatible values in $[0,1]$ to basic clopen sets. Such an assignment extends, using standard technology, to a probability measure μ on Borel sets, whence to a complete probability measure, again called μ, on all μ-measurable sets.

Martin's Axiom (MA), from set theory, states that, for every infinite cardinal $\kappa < 2^\omega$, a certain property MA_κ holds. In this paper we use the property $\text{MA}_{\aleph_1}$ as an axiom. This is implied by MA+¬CH (where CH is the *Continuum Hypothesis*), itself implies ¬CH, and is relatively consistent with ZFC set theory. Rather than explaining $\text{MA}_{\aleph_1}$ in detail, we instead list the consequences of it that we need. Let μ be a σ-finite Borel measure on a Polish space X, and let Ω be the collection of μ-measurable sets; then every $\boldsymbol{\Sigma}^1_2$ subset of X is in Ω and for every $\{X_\alpha\}_{\alpha < \omega_1}$ increasing $\subseteq$-chain of sets $X_\alpha \in \Omega$ indexed by the ordinals $\alpha < \omega_1$ (where ω_1 is the first uncountable ordinal), the statements $\bigcup_{\alpha < \omega_1} X_\alpha \in \Omega$ (ω_1-*completeness*) and $\mu(\bigcup_{\alpha<\omega_1} X_\alpha) = \bigsqcup_{\alpha<\omega_1} \mu(X_\alpha)$ (ω_1-*continuity*) hold. We refer [11] for a detailed proof of equivalent properties, in the special case of the Lebesgue measure on reals. As asserted there, the proofs generalize to the measure spaces considered in this paper.

3 The Logic pL$\mu^\odot$

Given a set Var of propositional variables ranged over by the letters X, Y and Z, and a set of labels L ranged over by the letters a, b and c, the formulae of the logic pL$\mu^\odot$ are defined by the following grammar:

$$F, G ::= X \mid [a]\,F \mid \langle a\rangle F \mid F \wedge G \mid F \vee G \mid F \cdot G \mid F \odot G \mid \nu X.F \mid \mu X.F$$

which extends the syntax of the standard μ-calculus with a new kind of conjunction $(\cdot)$ and disjunction $(\odot)$ operators called *product* and *coproduct* respectively. As usual the operators $\nu X.F$ and $\mu X.F$ bind the variable X in F. A formula is *closed* if it has no *free* variables.

Given a PLTS $\langle P, \{\xrightarrow{a}\}_{a\in L}\rangle$ we denote with $(P \to [0,1])$ and with $(\mathcal{D}(P) \to [0,1])$ the complete lattices of functions from P and from $\mathcal{D}(P)$ respectively, to the real interval $[0,1]$ with the pointwise order. Given a function $f : P \to [0,1]$, we denote with $\overline{f} : \mathcal{D}(P) \to [0,1]$ the lifted function defined as follows:

$$\overline{f} \overset{\text{def}}{=} \lambda d. \Big(\textstyle\sum_{p\in P} d(p)\cdot f(p)\Big).$$

A function $\rho : \mathrm{Var} \to (P \to [0,1])$ is called an *interpretation* of the variables. Given a function $f : P \to [0,1]$ we denote with $\rho[f/X]$ the interpretation that assigns f to the variable X, and $\rho(Y)$ to all other variables Y. The denotational semantics $[\![F]\!]_\rho : P \to [0,1]$ of the $\mathrm{pL}\mu^{\odot}$ formula F, under the interpretation ρ, is defined by structural induction on F as follows:

$$\begin{array}{ll}
[\![X]\!]_\rho & = \rho(X) \\
[\![G \vee H]\!]_\rho & = [\![G]\!]_\rho \sqcup [\![H]\!]_\rho \\
[\![G \wedge H]\!]_\rho & = [\![G]\!]_\rho \sqcap [\![H]\!]_\rho \\
[\![G \odot H]\!]_\rho & = \lambda p.\big([\![G]\!]_\rho(p) \odot [\![H]\!]_\rho(p)\big) \\
[\![G \cdot H]\!]_\rho & = \lambda p.\big([\![G]\!]_\rho(p) \cdot [\![H]\!]_\rho(p)\big) \\
[\![\langle a\rangle G]\!]_\rho & = \lambda p.\Big(\bigsqcup\{\ \overline{[\![G]\!]_\rho}(d) \mid p \xrightarrow{a} d\}\Big) \\
[\![[a]\,G]\!]_\rho & = \lambda p.\Big(\bigsqcap\{\ \overline{[\![G]\!]_\rho}(d) \mid p \xrightarrow{a} d\}\Big) \\
[\![\mu X.G]\!]_\rho & = \text{ least fixed point of } \lambda f.([\![G]\!]_{\rho[f/X]}) \\
[\![\nu X.G]\!]_\rho & = \text{ greatest fixed point of } \lambda f.([\![G]\!]_{\rho[f/X]})
\end{array}$$

where the symbols $\cdot$ and $\odot$ in the definitions of $[\![G \cdot H]\!]_\rho$ and $[\![G \odot H]\!]_\rho$ are standard multiplication on reals and the function $x \odot y = x + y - xy$, which is the De Morgan dual of multiplication with respect to the negation $\neg x = 1 - x$. Since the interpretation assigned to every $\mathrm{pL}\mu^{\odot}$ operator is monotone, the existence of the least and greatest fixed points is guaranteed by the Knaster-Tarski theorem. Moreover the least and the greatest fixed points can be computed inductively as follows: $[\![\mu X.G]\!]_\rho = \bigsqcup_\alpha [\![\mu X.G]\!]^\alpha_\rho$ and $[\![\nu X.G]\!]_\rho = \bigsqcap_\alpha [\![\nu X.G]\!]^\alpha_\rho$ where α, β are ordinals and $[\![\mu X.G]\!]^\alpha_\rho$ and $[\![\nu X.G]\!]^\alpha_\rho$ are defined as follows:

$$[\![\mu X.G]\!]^\alpha_\rho = \textstyle\bigsqcup_{\beta<\alpha} [\![G]\!]_{\rho[[\![\mu X.G]\!]^\beta_\rho/X]} \quad \text{and} \quad [\![\nu X.G]\!]^\alpha_\rho = \textstyle\bigsqcap_{\beta<\alpha} [\![G]\!]_{\rho[[\![\nu X.G]\!]^\beta_\rho/X]}$$

4 Stochastic Tree Games

In this (unavoidably long) section we introduce a new class of games which we call *stochastic two player tree games*, or just $2\frac{1}{2}$*-player tree games*. Stochastic tree games generalizes standard two player stochastic games by allowing a new class of *branching nodes* on which the execution of the game is split in independent concurrent sub-games. Formally, stochastic tree games are infinite duration games played by Player 1, Player 2 and a third probabilistic agent named *Nature*, on a Arena $\mathcal{A}=\langle (S,E),(S_1,S_2,S_N,B),\pi\rangle$, where (S,E) is a directed graph with countable set of vertices S and transition relation E, (S_1,S_2,S_N,B) is a partition of S and $\pi:S_N\rightarrow\mathcal{D}(S)$. The states in S_1, S_2, S_N and B are called *Player* 1 states, *Player* 2 states, *probabilistic* states and *branching* states respectively. We denote with $E(s)$, for $s\in S$, the set $\{s' \mid (s,s')\in E\}$. As a technical constraint, we require that $supp(\pi(s))\subseteq E(s)$, for every $s\in S_N$.

Definition 4 (Paths in $\mathcal{A}$). We denote with $\mathcal{P}^{\omega}$ and $\mathcal{P}^{<\omega}$ the sets of infinite and finite paths in $\mathcal{A}$. Given a finite path $\boldsymbol{s}\in\mathcal{P}^{<\omega}$ we denote with $last(\boldsymbol{s})$ the last state $s\in S$ of $\boldsymbol{s}$. We denote with $\mathcal{P}^{<\omega}_1$ and $\mathcal{P}^{<\omega}_2$ the sets of finite paths $\boldsymbol{s}$ such that $last(\boldsymbol{s})\in S_1$ and $last(\boldsymbol{s})\in S_2$ respectively. We write $\boldsymbol{s}\lhd\boldsymbol{s'}$, with $\boldsymbol{s},\boldsymbol{s'}\in\mathcal{P}^{<\omega}$, if $\boldsymbol{s'}=\boldsymbol{s}.s$, for some $s\in S$, where as usual the *dot* symbol denotes the concatenation operator. We denote with $\mathcal{P}^t$ the set of *terminated paths*, i.e. the set of paths $\boldsymbol{s}$ such that $E(last(\boldsymbol{s}))=\emptyset$. We denote with $\mathcal{P}$ the set $\mathcal{P}^{\omega}\cup\mathcal{P}^t$ and we refer to this set as the set of *completed* paths in A. Given a finite path $\boldsymbol{s}\in\mathcal{P}^{<\omega}$, we denote with $O_{\boldsymbol{s}}$ the set of all completed paths having $\boldsymbol{s}$ as prefix. We consider the standard topology on $\mathcal{P}$ where the countable basis for the open sets is given by the clopen sets $O_{\boldsymbol{s}}$, for $\boldsymbol{s}\in\mathcal{P}^{<\omega}$.

Definition 5 (Tree in $\mathcal{A}$). A *tree* in the arena $\mathcal{A}$ is a collection $C=\{\boldsymbol{s}_i\}_{i\in I}$ of finite paths $\boldsymbol{s}_i\in\mathcal{P}^{<\omega}$, such that

1. C is down-closed: if $\boldsymbol{s}\in C$ and $\boldsymbol{s'}$ is a prefix of $\boldsymbol{s}$, then $\boldsymbol{s'}\in C$.
2. C has a root: there exists exactly one finite path $\{s\}$ of length one in C. The path $\{s\}$, denoted by $root(C)$, is called the root of the tree C.

We consider the nodes $\boldsymbol{s}$ of C as labeled by the *last* function.

Definition 6 (Uniquely and fully branching nodes of a tree). A node $\boldsymbol{s}$ in a tree T is said to be *uniquely branching* in T if either $E(last(\boldsymbol{s}))=\emptyset$ or $\boldsymbol{s}$ has a unique successor in T. Similarly, $\boldsymbol{s}$ is *fully branching* in T if, for every $s\in E(s)$, it holds that $\boldsymbol{s}.s\in T$.

An outcome of the game in $\mathcal{A}$, which we call a *branching play*, is a possibly infinite tree T in $\mathcal{A}$ defined as follows:

Definition 7 (Branching play in $\mathcal{A}$). A *branching play* in the arena $\mathcal{A}$ is a tree T in $\mathcal{A}$ such that, for every node $\boldsymbol{s}\in T$ the following conditions holds:

1. If $last(\boldsymbol{s})\in S_1\cup S_2\cup S_N$ then $\boldsymbol{s}$ branches uniquely in T.
2. If $last(\boldsymbol{s})\in B$ then $\boldsymbol{s}$ branches fully in T.

We denote with $\mathcal{T}$ the set of branching plays T in the arena $\mathcal{A}$.

A branching play T represents a possible execution of the game from the state s labeling the root of T. The nodes of T with more than one child are all labeled with a state $s\in B$ and are the branching points of the game; their children represent the independent instances of play generated at the branching point.

Definition 8 (Topology on $\mathcal{T}$). Given a finite tree F in $\mathcal{A}$, we say that F is a *branching-play prefix*, if there exists some $T \in \mathcal{T}$, such that $F \subseteq T$. Given a branching-play prefix F, we denote with $O_F \subseteq \mathcal{T}$ the set of all branching plays T, such that $F \subseteq T$. We fix the topology on $\mathcal{T}$, where the basis for the open sets is given by the clopen sets O_F, for every branching-play prefix F. It is routine to show that this is a 0-dimensional Polish space.

As usual when working with *stochastic* games, it is useful to look at the possible outcomes of a play up-to the behavior of Nature. In the context of standard two player stochastic games this amounts at considering Markov chains. In our setting the following definition of Markov branching play is natural:

Definition 9 (Markov branching play in $\mathcal{A}$). A *Markov branching play* in $\mathcal{A}$ is a tree M in $\mathcal{A}$ such that for every node $\boldsymbol{s} \in M$, the following conditions holds:

1. If $last(\boldsymbol{s}) \in S_1 \cup S_2$ then $\boldsymbol{s}$ branches uniquely in T.
2. If $last(\boldsymbol{s}) \in S_N \cup B$ then $\boldsymbol{s}$ branches fully in T.

A Markov branching play, is similar to a branching play except that probabilistic choices of Nature have not been resolved.

Definition 10 (Probability measure $\mathcal{M}_M$). Every Markov branching play M determines a probability assignment $\mathcal{M}_M$ to every basic clopen set $O_F \subseteq \mathcal{T}$, for F a branching-play prefix, defined as follows:

$$\mathcal{M}_M(O_F) \stackrel{\text{def}}{=} \begin{cases} \prod\{\pi(s,s') \mid \boldsymbol{s}.s.s' \in F \wedge s \in S_N\} & \text{if } F \subseteq M \\ 0 & \text{otherwise} \end{cases}$$

It is the above definition that implements the *probabilistic independence* of the sub-branching plays that follows a branching node s. The assignment $\mathcal{M}_M$ extends to a unique complete probability measure $\mathcal{M}_M$ on the measurable space Ω_M of all $\mathcal{M}_M$-measurable sets. By $\mathrm{MA}_{\aleph_1}$, the collection Ω_M is ω_1-complete, and the probability measure $\mathcal{M}_M$ is ω_1-continuous.

Definition 11 (Measurable space of branching plays in $\mathcal{A}$). We define the measurable space $(\mathcal{T}, \Omega)$ of branching plays in $\mathcal{A}$ taking $\Omega = \bigcap_M \Omega_M$, where M ranges over Markov branching plays in $\mathcal{A}$. We say that a set $X \subseteq \mathcal{T}$ is *measurable* if $X \in \Omega$. The fact that the σ-algebra Ω is closed under arbitrary ω_1-unions follows from the remark above that Ω_M, for every M, is ω_1-complete. Given any Markov branching play M in $\mathcal{A}$, the (ω_1-continuous) probability measure $\mathcal{M}_M$, induced by M on the measurable space Ω_M restricts to a unique (ω_1-continuous) probability measure on the smaller space Ω, which we denote again with $\mathcal{M}_M$.

Definition 12 (Two player stochastic tree game). A *two player stochastic tree game* (or a $2\frac{1}{2}$-player tree game) is given by a pair $\langle \mathcal{A}, \Phi \rangle$, where $\mathcal{A}$ is a stochastic tree game arena as described above, and $\Phi \subseteq \mathcal{T}$, which is the *objective* for Player 1, is a measurable set of branching plays in $\mathcal{A}$.

Definition 13 (Expected value of a Markov branching play). Let $\langle \mathcal{A}, \Phi \rangle$ be a $2\frac{1}{2}$-player tree game, and M a Markov branching play in $\mathcal{A}$. We define the *expected value* of M as: $E(M) = \mathcal{M}(\Phi)$. The value $E(M)$ should be understood as the probability for Player 1 to win the play.

As usual in game theory, players' moves are determined by strategies.

Definition 14 (Deterministic strategies). An (*unbounded memory deterministic*) *strategy* σ_1 for Player 1 in $\mathcal{A}$ is defined as a function $\sigma_1 : \mathcal{P}_1^{<\omega} \rightarrow S \cup \{\bullet\}$ such that $\sigma_1(\boldsymbol{s}) \in E(last(s))$ if $E(last(\boldsymbol{s})) \neq \emptyset$ and $\sigma_1(\boldsymbol{s}) = \bullet$ otherwise. Similarly a *strategy* σ_2 for Player 2 is defined as a function $\sigma_2 : \mathcal{P}_2^{<\omega} \rightarrow S \cup \{\bullet\}$. A pair $\langle \sigma_1, \sigma_2 \rangle$ of strategies, one for each player, is called a *strategy profile* and determines the behaviors of both players.

Note that the above definition of strategy captures the intended behavior of the game; both players when acting on a given instance of the game, know all the history of the actions happened on that sub-game, but have no knowledge of the evolution of the other independent parallel sub-games.

Definition 15 ($M^{s_0}_{\sigma_1,\sigma_2}$). Given an initial state $s_0 \in S$ and a strategy profile $\langle \sigma_1, \sigma_2 \rangle$ a unique Markov branching play $M^{s_0}_{\sigma_1,\sigma_2}$ is determined:

1. the root of M is labeled with s_0,
2. For every $\boldsymbol{s} \in M^{s_0}_{\sigma_1,\sigma_2}$, if $last(\boldsymbol{s}) = s$ with $s \in S_1$ not a terminal state, then the unique child of $\boldsymbol{s}$ in $M^{s_0}_{\sigma_1,\sigma_2}$ is $\boldsymbol{s}.\big(\sigma_1(\boldsymbol{s})\big)$.
3. For every $\boldsymbol{s} \in M^{s_0}_{\sigma_1,\sigma_2}$, if $last(\boldsymbol{s}) = s$ with $s \in S_2$ not a terminal state, then the unique child of $\boldsymbol{s}$ in $M^{s_0}_{\sigma_1,\sigma_2}$ is $\boldsymbol{s}.\big(\sigma_2(\boldsymbol{s})\big)$.

Definition 16 (Upper and lower values of a $2\frac{1}{2}$-player tree game). Let $G = \langle \mathcal{A}, \Phi \rangle$ be a $2\frac{1}{2}$-player tree game. We define the lower and upper values of G on the state s, denoted by $Val^s_{\downarrow}(G)$ and $Val^s_{\uparrow}(G)$ respectively, as follows:

$$Val^s_{\downarrow}(G) = \bigsqcup_{\sigma_1} \bigsqcap_{\sigma_2} E(M^s_{\sigma_1,\sigma_2}) \qquad Val^s_{\uparrow}(G) = \bigsqcap_{\sigma_2} \bigsqcup_{\sigma_1} E(M^s_{\sigma_1,\sigma_2})$$

$Val^s_{\downarrow}(G)$, represents the limit probability of Player 1 winning, when the game begins in s, by choosing his strategy σ_1 first and then letting Player 2 pick an appropriate counter strategy σ_2. Similarly $Val^s_{\uparrow}(G)$ represents the limit probability of Player 1 winning, when the game begins in s, by first letting Player 2 choose a strategy σ_2 and then picking an appropriate counter strategy σ_1. In case $Val^s_{\downarrow}(G) = Val^s_{\uparrow}(G)$, we say that the game G at s is *determined.*

Definition 17 (ϵ-optimal strategies). Let $G = \langle \mathcal{A}, \Phi \rangle$ be a $2\frac{1}{2}$-player tree game. We say that a strategy σ_1 for Player 1 in G is ϵ-optimal, for $\epsilon > 0$, if for every state s, the following inequality holds: $\bigsqcap_{\sigma_2} E(M^s_{\sigma_1,\sigma_2}) > Val^s_{\downarrow}(G) - \epsilon$. Similarly we say that a strategy σ_2 for Player 2 in G is ϵ-optimal, for $\epsilon > 0$, if for every state s, the following inequality holds: $\bigsqcup_{\sigma_1} E(M^s_{\sigma_1,\sigma_2}) < Val^s_{\uparrow}(G) + \epsilon$. Clearly ϵ-optimal strategies for Player 1 and Player 2 always exist for every $\epsilon > 0$.

An interesting class of $2\frac{1}{2}$-player tree games is given by what we call meta-games. A *meta-game* is a $2\frac{1}{2}$-player tree game G, which we refer to as the *outer* game, in which branching plays are themselves interpreted as (ordinary) two player games and the objective Φ of G is defined as the set of branching plays T in which this *inner game* is winnable for a given player taking part in it. We illustrate this notion by formalizing the class of $2\frac{1}{2}$-player meta-parity games. A $2\frac{1}{2}$*-player meta-parity game* G is specified by a $2\frac{1}{2}$-player tree game arena $\mathcal{A}{=}\langle(S,E),\{S_1,S_2,S_N,B\},\pi\rangle$ and a parity structure $\mathbb{P}$ which is a pair $\langle Pr, Pl\rangle$, where $Pr : S \to \{0,...,n\}$, for some $n \in \mathbb{N}$, and $Pl : B \to \{1,2\}$. The function Pr assigns a *priority* to each state $s \in S$. This is needed to define the set $W \subseteq \mathcal{P}$, of completed paths $\boldsymbol{s}$ such that

- if $\boldsymbol{s}$ is finite, then $Pr(last(\boldsymbol{s}))$ is even,
- if $\boldsymbol{s}$ is infinite, then the least priority among those that appear (assigned by Pr to the states of $\boldsymbol{s}$) infinitely often in $\boldsymbol{s}$ is even.

The function Pl assigns a *player identifier* to each state $s \in B$. This allows to consider each branching play T in $\mathcal{A}$ as the game G_T played by Player 1 and Player 2 on the tree T: Player 1 moves on a node $\boldsymbol{s}$ of T, such that $Pl(last(\boldsymbol{s})) = 1$, by choosing a successor in the (possibly empty) set of children of $\boldsymbol{s}$. Similarly Player 2 moves on the node $\boldsymbol{s}$ of T such that $Pl(last(\boldsymbol{s})) = 2$. The result of a play in the game G_T is a completed path in T. We say that Player 1 wins a play if the resulting path is in W; Player 2 wins otherwise. Since W is a parity objective we have that G_T is a parity game. The meta-parity game G can therefore be defined formally as a stochastic two player tree game, as follows:

Definition 18 (Two player stochastic meta-parity game). A *two player stochastic meta-parity game* specified by the pair $\langle\mathcal{A},\mathbb{P}\rangle$, is formally defined as the $2\frac{1}{2}$-player tree game $\langle\mathcal{A},\Phi\rangle$ where Φ is defined as follows:

$$\Phi = \{T \mid T \in \mathcal{T} \text{ and Player 1 has a winning strategy in } G_T\}$$

The definition is good because the measurabilty of Φ follows from $\mathrm{MA}_{\aleph_1}$and the following Lemma.

Lemma 1. *The set Φ is a $\boldsymbol{\Delta}^1_2$ set and hence a $\boldsymbol{\Sigma}^1_2$ set in $\mathcal{T}$.*

5 Game Semantics of pL$\boldsymbol{\mu}^{\odot}$

In this section we describe the pL$\mu^{\odot}$ game G^F_ρ associated to each triple consisting of a PLTS $\langle P,\{\xrightarrow{a}\}_{a\in L}\rangle$, a (possibly open) pL$\mu^{\odot}$ formula F, and an interpretation of the variables ρ. For convenience, we assume that F is *normal* [16], i.e., every occurrence of a μ or ν binder binds a distinct variable, and no variable appears both free and bound. The game G^F_ρ is a $2\frac{1}{2}$-player meta-parity game specified by the arena $\mathcal{A}^F_\rho = \langle(S,E),(S_1,S_2,S_N,B),\pi\rangle$ and parity structure $\mathbb{P} = \langle Pr, Pl\rangle$ defined as follows. The countable set S of vertices of the directed graph (S,E) is given by the set.

$$S = (P \times Sub(F)) \cup (D \times Sub(F)) \cup \{\bot, \top\}$$

where P is the set of processes, $Sub(F)$ is the set of sub-formulae of F (defined as usual, e.g. [16]), D is the set of distributions in the PLTS (see definition 2) and $\{\bot, \top\}$ are two special states. The relation E is defined as follows: $E(\langle d, G\rangle) = \{\langle p, G\rangle \mid p \in supp(d)\}$ for every $d \in D$; $E(\langle p, G\rangle)$ is defined by case analysis on the outermost connective of G as follows:

1. if $G = X$, with X free in F, then $E(\langle p, G\rangle) = \{\bot, \top\}$.
2. if $G = X$, with X bound in F by the subformula $\star X.H$, with $\star \in \{\mu, \nu\}$, then $E(\langle p, G\rangle) = \{\langle p, H\rangle\}$.
3. if $G = \star X.H$, with $\star \in \{\mu, \nu\}$, then $E(\langle p, G\rangle) = \{\langle p, H\rangle\}$.
4. if $G = \langle a\rangle H$ then $E(\langle p, G\rangle) = \{\langle d, H\rangle \mid p \xrightarrow{a} d\}$.
5. if $G = [a]\, H$ then $E(\langle p, G\rangle) = \{\langle d, H\rangle \mid p \xrightarrow{a} d\}$.
6. if $G = H * H'$ with $* \in \{\vee, \wedge, \odot, \cdot\}$ then $E(\langle p, G\rangle) = \{\langle p, H\rangle, \langle p, H'\rangle\}$

The relation E is defined on the two special game states $\top$ and $\bot$ as $E(\top) = E(\bot) = \emptyset$. This makes $\top$ and $\bot$ *terminal* states of the game. The partition (S_1, S_2, S_N, B) of S is defined as follows: every state $\langle p, G\rangle$ with G's main connective in $\{\langle a\rangle, \vee, \mu X\}$ or with $G = X$ where X is a μ-variable, is in S_1; dually every state $\langle p, G\rangle$ with G's main connective in $\{[a]\, , \wedge, \nu X\}$ or with $G = X$ where X is a ν-variable, is in S_2. Every state of the form $\langle d, G\rangle$ or $\langle p, X\rangle$, with X free in F, is in S_N. Every state $\langle p, G\rangle$ whose G's main connective is $\cdot$ or $\odot$ is in B. Lastly we define the terminal states $\bot$ and $\top$ to be in S_1 and S_2 respectively. The function $\pi : S_N \to \mathcal{D}(S)$ assigns a probability distribution to every state under the control of Nature (thus specifying its indended probabilistic behavior) and it is defined as $\pi(\langle d, G\rangle)(\langle p, G\rangle) = d(p)$ on all states of the form $\langle d, G\rangle$; all other states in S_N are of the form $\langle p, X\rangle$, with X free in F; the function π is defined on these states as follows:

$$\pi(\langle p, X\rangle)(s) \stackrel{\text{def}}{=} \begin{cases} \rho(X)(p) & \text{if } \ s = \top \\ 1 - \rho(X)(p) & \text{if } \ s = \bot \\ 0 & \text{otherwise} \end{cases}$$

The priority assignment $Pr : S \to \{0, ..., n\}$ is defined, by picking a sufficiently large n, as usual in μ-calculus games: an odd priority is assigned to the states $\langle p, X\rangle$ with X a μ-variable and dually an even priority is assigned to the states $\langle p, X\rangle$ with X a ν-variable, in such a way that if Z subsumes Y in F then $Pr(\langle p, Z\rangle) < Pr(\langle p, Y\rangle)$. Moreover, for every terminal state $s \in S$, we define $Pr(s) = 1$ if $s \in S_1$, and $Pr(s) = 0$ if $s \in S_2$. Lastly, the fuction $Pl : B \to \{1, 2\}$ is defined as $Pl(\langle p, G_1 \odot G_2) = 1$ and $Pl(\langle p, G_1 \cdot G_2) = 2$ for every $p \in P$ and $G_1, G_2 \in Sub(F)$.

Note that if no (co)product operators occur in F, then $B = \emptyset$, and the game is equivalent to the one in [12,13] for the logic pLμ. We are now ready to state our main result.

Theorem 1 (MA$_{\aleph_1}$). *Given a PLTS $\langle P, \{\xrightarrow{a}\}_{a \in L}\rangle$, for every process $p \in P$, interpretation of the variables ρ and pL$\mu^{\odot}$ formula F, the equalities*

$$Val_{\downarrow}^{\langle p, F\rangle}(G_\rho^F) = Val_{\uparrow}^{\langle p, F\rangle}(G_\rho^F) = [\![F]\!]_\rho(p)$$

hold. In particular the game G_ρ^F is determined.

6 Inductive Characterization of the Objective of $G_\rho^{\mu X.F}$

In this section we provide a transfinite inductive characterization of the set $\Phi^{\mu X.G}$ of winning branching plays of the game $G_\rho^{\mu X.F}$ needed in the proof of Theorem 1. Let us consider the game G_ρ^F, where X appears free in F. Note that the two arenas $\mathcal{A}_\rho^F$ and $\mathcal{A}_\rho^{\mu X.F}$ are similar as they differ only in the following aspects:

1. The set of states $S^{\mu X.F}$ of $\mathcal{A}_\rho^{\mu X.F}$ is the same as the set of states S^F of $\mathcal{A}_\rho^F$, plus the set of states of the form $\langle p, \mu X.F\rangle$, which however play almost no role in the game because these nodes have only one successor ($\langle p, F\rangle$) and are not reachable by any other state.
2. More significantly, the nodes of the form $\langle p, X\rangle$, which are present in both game arenas, are Player 1 states in $G_\rho^{\mu X.F}$ (they have unique successor $\langle p, F\rangle$), and Nature states in G_ρ^F (they have two *terminal* successors $\top$ and $\bot$ reachable with probabilities $\rho(X)(p)$ and $1-\rho(X)(p)$ respectively).

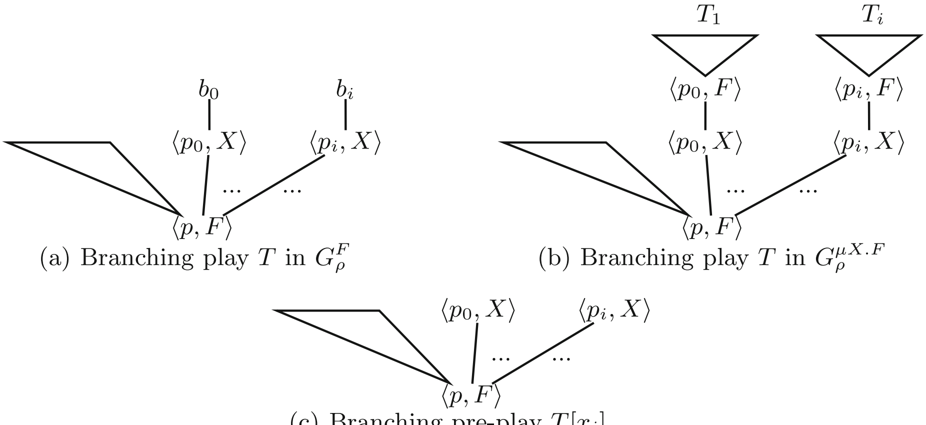
Moreover observe that the player assignments Pl^F and $Pl^{\mu X.F}$ are identical, and the priority assigments Pr^F and $Pr^{\mu X.F}$ differ only on the game-states s of the form $\langle p, X\rangle$: $Pr^F(s)=0$ and $Pr^{\mu X.F}(s)=m$ for some odd $m\in\mathbb{N}$. A G_ρ^F branching play T, rooted in $\langle p, F\rangle$ can be depicted as in figure 2(a), where the triangle represents the set of paths in T never reaching a state of the form $\langle q, X\rangle$, for $q \in P$, and the other edges represents the, possibly empty, collection of paths $\{\boldsymbol{s}_i\}_{i\in I\subseteq\mathbb{N}}$ reaching a state of the form $\langle p_i, X\rangle$ which is (necessarily) followed by a terminal state $b_i \in \{\top, \bot\}$. Similarly a branching play T in $G_\rho^{\mu X.F}$, rooted in $\langle p, F\rangle$, can be depicted as in figure 2(b). We extract the common part between the branching plays G_ρ^F and $G_\rho^{\mu X.F}$ by defining the notion of branching pre-play.

Definition 19 (Branching pre-play). Let T be a branching play in G_ρ^F and let I index the (necessarily countable) collection of nodes of the form $\langle p_i, X\rangle$ in T. The *branching pre-play* $T[x_i]_{i\in I}$, which can be depicted as in figure 2(c), is the tree obtained from the branching play T by removing all subtrees rooted in states of the form $\langle p_i, X\rangle$.

Given a I-indexed family $\{b_i\}_{i\in I}$, where $b_i\in\{\top,\bot\}$, we denote with $T[b_i]_{i\in I}$ the branching play in G_ρ^F obtained by adding, for every $i \in I$, the child b_i to the leaf $\langle p_i, X\rangle$ of $T[x_i]_{i\in I}$. Similarly given a family $\{T_i\}_{i\in I}$ of branching plays in $G^{\mu X.F}$, where each T_i is rooted at $\langle p_i, G\rangle$, we denote with $T[T_i]_{i\in I}$ the branching play in $G_\rho^{\mu X.F}$ obtained by adding the subtree T_i after the leaf $\langle p_i, X\rangle$. Clearly every branching play T rooted at $\langle p, F\rangle$ in G_ρ^F is uniquely of the form $T'[b_i]_{i\in I}$ for appropriate $T'[x_i]_{i\in I}$ and $\{b_i\}_{i\in I}$. Similarly every branching play T rooted at $\langle p, F\rangle$ in $G_\rho^{\mu X.F}$ is of the form $T'[T_i]_{i\in I}$ for appropriate $T'[x_i]_{i\in I}$ and $\{T_i\}_{i\in I}$.

Definition 20. The function $m_X : \mathcal{T}^{\mu X.F} \to \mathcal{T}^F$, from branching plays in $G_\rho^{\mu X.F}$ to branching plays in G_ρ^F, is defined for every subset $X \subseteq \mathcal{T}^{\mu X.F}$ as follows:

$$m_X(T[T_i]_{i\in I}) = T[T_i \underline{\in} X]_{i\in I} \qquad \text{where } T_i \underline{\in} X \overset{\text{def}}{=} \begin{cases} \top \text{ if } \; T_i \in X \\ \bot \text{ otherwise} \end{cases}$$



(a) Branching play T in G^F_ρ

(b) Branching play T in $G^{\mu X.F}_\rho$

(c) Branching pre-play $T[x_i]_{i\in I}$

Fig. 2. Branching plays and pre-plays

Lemma 2. *If $X \subseteq \mathcal{T}^{\mu X.F}$ is a measurable set, then m_X is a measurable map.*

We now define the operator R, of which $\Phi^{\mu X.G}$ is the least fixed point.

Definition 21. The operator R: $2^{\mathcal{T}^{\mu X.F}} \rightarrow 2^{\mathcal{T}^{\mu X.F}}$ is defined as follows:

$$R(X) \overset{\text{def}}{=} m_X^{-1}(\Phi^F) = \{T[T_i]_{i\in I} \mid T[T_i \underline{\in} X]_{i\in I} \in \Phi^F\}$$

Theorem 2. *The set $\Phi^{\mu X.F}$ is the least fixed point of the monotone operator R, which is guaranteed to exists by the Knaster-Tarski theorem. Hence the set $\Phi^{\mu X.F}_\rho$ can be defined as $\bigcup_\alpha R^\alpha$ where R^α is defined for every ordinal α as $\bigcup_{\beta<\alpha} R(R^\beta)$.*

The following lemma, which is essential for our proof of Theorem 1, states that the fixed points is reached in at most ω_1 steps.

Lemma 3. *$\bigcup_\alpha R^\alpha = \bigcup_{\alpha<\omega_1} R^\alpha$.*

7 Proof of Theorem 1

In this section we sketch the main ideas of the proof of Theorem 1. We first introduce a property that is going to be useful in proving the main result:

Definition 22 (Robust Markov branching plays). Fix a pL$\mu^\odot$ game G^F_ρ, a free variable X in F and an $\mathbb{N}$-indexed collection $\{\epsilon_n\}_{n\in\mathbb{N}}$ of reals in $(0, 1]$. Let M be a Markov branching play in G^F_ρ. Let $\{x_i\}_{i\in I\subseteq\mathbb{N}}$ be the set of vertices in M labeled with states of the form $\langle p, X\rangle$. Since X is free in F, these vertices are necessarily connected to the two leafs $\bot$ and $\top$ by two edges $e^\top_i$ and $e^\bot_i$ marked with the probabilities λ_i and $1-\lambda_i$ respectively. Let M^+ be the same Markov branching play where, for each $i \in I$, the probability attached to the edge $e^\top_i$

is replaced with $\min\{1, \lambda_i + \epsilon_i\}$, and the probability attached to the edge $e_i^{\perp}$ is replaced by $\max\{0, (1-\lambda_i) - \epsilon_i\}$. Similarly let M^- be as M where, for all $i \in I$, the probabilities attached to the edge $e_i^{\top}$ is replaced with $\max\{0, \lambda_i - \epsilon_i\}$, and the probability attached to the edge $e_i^{\perp}$ is replaced by $\min\{1, (1-\lambda_i) + \epsilon_i\}$. We say that M is *robust* if and only if both inequalities $E(M^+) \leq E(M) + \sum_{i\in I} \epsilon_i$ and $E(M^-) \geq E(M) - \sum_{i \in I} \epsilon_i$ hold, for every collection $\{\epsilon_n\}_{n\in\mathbb{N}}$.

The notion of robustness can be informally described as follows: in the Markov branching play M^+ we increase the probability associated with the branching plays having paths ending in $\top$ immediately following a configuration $\langle p, X\rangle$; by doing so we increase the value, see Definition 13, of the original M and, in a similar way we decrease the value of M by moving to M^-. A Markov branching play is robust if small changes (either in the direction of M^+ or M^-) in the probabilities (labeling the edges $e_i^{\top}$ and $e_i^{\perp}$, $i \in I$) produce bounded (by $\sum_{i\in I}\epsilon_i$) changes in the overall value of the Markov branching play. Note that altering the probabilities in M uniformly (i.e. taking $\{\epsilon_i\}_{i\in I}$ such that for every $i \neq j$, $\epsilon_i = \epsilon_j$), may produce, in general, unbounded changes in the value of M; this reflect the discontinuity of the denotational interpretation of pL$\mu^{\odot}$ formulae on the free variables.

In order to prove Theorem 1, we prove the following stronger theorem:

Theorem 3 (MA$_{\aleph_1}$). *Given a PLTS $\langle P, \{\xrightarrow{a}\}_{a\in L}\rangle$, for every pL$\mu^{\odot}$ formula F and for every interpretation ρ, the following assertions hold for every $p \in P$:*

1. $[\![F]\!]_\rho(p) \geq Val_{\uparrow}^{\langle p,F\rangle}(G_\rho^F)$
2. $[\![F]\!]_\rho(p) \leq Val_{\downarrow}^{\langle p,F\rangle}(G_\rho^F)$
3. *Every Markov branching play M rooted in $\langle p, F\rangle$ in G_ρ^F is robust.*

The proof is by induction on the structure of F, and resembles the *unfolding method* of [7,13]. The most difficult case in proving point 1 is when F is of the form $\mu X.H$ (and dually the case $\nu X.H$ is difficult for point 2). This is proven showing that, for every $\epsilon > 0$, there exists a strategy σ_2^{ϵ} for Player 2 in $G_\rho^{\mu X.H}$, such that for every counter-strategy σ_1 for Player 1, the inequality $E(M_{\sigma_1,\sigma_2^\epsilon}^{\langle p,\mu X.G\rangle}) < [\![\mu X.H]\!]_\rho(p) + \epsilon$ holds. As in [7,13], the strategy σ_2^ϵ is constructed using δ-optimal strategies for Player 2 in the game $G_{\rho^\gamma}^H$ (where $\rho^\gamma = [\![\mu X.H]\!]_\rho$) which exist by induction hypothesis. The idea behind the construction of σ_2^ϵ is the following: initially the strategy σ_2^ϵ behaves as some δ_0-optimal strategy τ_0 for Player 2 in $G_{\rho^\gamma}^H$; if at some point of the play the game reaches a configuration of the form $\langle p, X\rangle$, then Player 2 *improves* his play and, depending on the history of the previously played moves, starts behaving as some δ_1-optimal strategy τ_1 for Player 2 in $G_{\rho^\gamma}^H$ and so on; the strictly decreasing sequence $\{\delta_i\}_{i\in\mathbb{N}}$ is carefully chosen, so that the desired ϵ-bound follows from the induction hypothesis of robustness for Markov branching plays of $G_{\rho^\gamma}^H$. The desired inequality $E(M_{\sigma_1,\sigma_2^\epsilon}^{\langle p,\mu X.G\rangle}) < [\![\mu X.H]\!]_\rho(p) + \epsilon$, thanks to Theorem 2, Lemma 3 and the fact that $\mathcal{M}_{\sigma_1,\sigma_2^\epsilon}^{\langle p,\mu X.H\rangle}$ is ω_1-continuous under MA$_{\aleph_1}$, is equivalent to

$\bigsqcup_{\alpha<\omega_1} \mathcal{M}^{\langle p,\mu X.H\rangle}_{\sigma_1,\sigma_2^{\epsilon}}(R^{\alpha}) < [\![\mu X.H]\!]_{\rho}(p) + \epsilon$. The proof that this last inequality holds, is by ordinal induction.

Acknowledgments. I thank my PhD supervisor, Alex Simpson, for helpful suggestions. This research was supported by studentships from LFCS and the IGS at the School of Informatics, Edinburgh, and by EPSRC research grant EP/F042043/1.

References

1. Bertrand, N., Genest, B., Gimbert, H.: Qualitative determinacy and decidability of stochastic games with signals. In: LICS 2009, IEEE Symposium on Logic in Computer Science, Los Angeles, USA (2009)
2. Blackwell, D.: Infinite Gδ games with imperfect information. Matematyki Applicationes Mathematicae, Hugo Steinhaus Jubilee Volume (1969)
3. Brázdil, T., Brozek, V., ín Kucera, A., Obdrzálek, J.: Qualitative reachability in stochastic BPA games. In: STACS, pp. 207–218 (2009)
4. Brázdil, T., Brozek, V., Forejt, V., Kucera, A.: Stochastic games with branching-time winning objectives. In: LICS, pp. 349–358 (2006)
5. de Alfaro, L., Majumdar, R.: Quantitative solution of omega-regular games. Journal of Computer and System Sciences 68, 374–397 (2004)
6. Deng, Y., van Glabbeek, R.: Characterising probabilistic processes logically. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 278–293. Springer, Heidelberg (2010)
7. Fischer, D., Gradel, E., Kaiser, L.: Model checking games for the quantitative μ-calculus. In: Theory of Computing Systems. Springer, New York (2009)
8. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6 (1994)
9. Huth, M., Kwiatkowska, M.: Quantitative analysis and model checking. In: LICS 1997, Washington, DC, USA, p. 111. IEEE Computer Society, Los Alamitos (1997)
10. Kozen, D.: Results on the propositional mu-calculus. Theoretical Computer Science, 333–354 (1983)
11. Martin, D.A., Solovay, R.M.: Internal Cohen extensions. Ann. Math. Logic 2, 143–178 (1970)
12. McIver, A., Morgan, C.: Results on the quantitative μ-calculus qMμ. ACM Trans. Comput. Logic 8(1), 3 (2007)
13. Mio, M.: The equivalence of denotational and game semantics for the probabilistic μ-calculus. In: 7th Workshop on Fixed Points in Computer Science (2010)
14. Morgan, C., McIver, A.: A probabilistic temporal calculus based on expectations. In: Groves, L., Reeves, S. (eds.) Proc. Formal Methods. Springer, Heidelberg (1997)
15. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems. PhD thesis, Laboratory for Computer Science, M.I.T. (1995)
16. Stirling, C.: Modal and temporal logics for processes. Springer, Heidelberg (2001)

A Step-Indexed Kripke Model of Hidden State via Recursive Properties on Recursively Defined Metric Spaces

Jan Schwinghammer[1], Lars Birkedal[2], and Kristian Støvring[3]

[1] Saarland University
[2] IT University of Copenhagen
[3] DIKU, University of Copenhagen

Abstract. Frame and anti-frame rules have been proposed as proof rules for modular reasoning about programs. Frame rules allow one to hide irrelevant parts of the state during verification, whereas the anti-frame rule allows one to hide local state from the context. We give the first sound model for Charguéraud and Pottier's type and capability system including both frame and anti-frame rules. The model is a possible worlds model based on the operational semantics and step-indexed heap relations, and the worlds are constructed as a recursively defined predicate on a recursively defined metric space.

We also extend the model to account for Pottier's *generalized* frame and anti-frame rules, where invariants are generalized to *families* of invariants indexed over pre-orders. This generalization enables reasoning about some well-bracketed as well as (locally) monotonic uses of local state.

1 Introduction

Reasoning about higher-order stateful programs is notoriously difficult, and often involves the need to track aliasing information. A particular line of work that addresses this point are substructural type systems with regions, capabilities and singleton types [2,8,9]. In this context, Pottier [14] presented the anti-frame rule as a proof rule for hiding invariants on encapsulated state: the description of a piece of mutable state that is *local* to a procedure can be removed from the procedure's external interface (expressed in the type system). The benefits of hiding invariants on local state include simpler interface specifications, simpler reasoning about client code, and fewer restrictions on the procedure's use because potential aliasing is reduced. Thus, in combination with frame rules that allow the irrelevant parts of the state to be hidden during verification, the anti-frame rule provides an important ingredient for modular reasoning about programs.

Essentially, the frame and anti-frame rules exploit the fact that programs cannot access non-local state directly. However, in an ML-like language with higher-order procedures and the possibility of call-backs, the dependencies on non-local state can be complex; consequently, the soundness of frame and anti-frame rules is anything but obvious.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 305–319, 2011.

Pottier [14] sketched a soundness proof for the anti-frame rule by a progress and preservation argument, which rests on assumptions about the existence of certain recursively defined types and capabilities. (He has since formalized the details in Coq.) More recently, Birkedal et al. [6] developed a step-indexed model of Charguéraud and Pottier's type and capability system with higher-order frame rules, but without the anti-frame rule. This was a Kripke model in which capabilities are viewed as assertions (on heaps) that are indexed over recursively defined worlds: intuitively, these worlds are used to represent the invariants that have been added by the frame rules.

Proving soundness of the anti-frame rule requires a refinement of this idea, as one needs to know that additional invariants do not invalidate the invariants on local state which have been hidden by the anti-frame rule. This requirement can be formulated in terms of a monotonicity condition for the world-indexed assertions, using an order on the worlds that is induced by invariant extension, i.e., the addition of new invariants [17]. (The fact that ML-style untracked references can be encoded from strong references with the anti-frame rule [14] also indicates that a monotonicity condition is required: Kripke models of ML-style references involve monotonicity in the worlds [7,1].) More precisely, in the presence of the anti-frame rule, it turns out that the recursive domain equation for the worlds involves monotonic functions with respect to an order relation on worlds, and that this order is specified using the isomorphism of the recursive world solution itself. This circularity means that standard existence theorems, in particular the one used in [6], cannot be applied to define the worlds. Thus Schwinghammer et al. [17], who considered a separation logic variant of the anti-frame rule for a simple language (without higher-order functions, and untyped), had to give the solution to a similar recursive domain equation by a laborious inverse-limit construction.

In the present paper we develop a new model of Charguéraud and Pottier's system, which can also be used to show soundness of the anti-frame rule. Moreover, we show how to extend our model to show soundness of Pottier's *generalized* frame and anti-frame rules, which allow hiding of *families* of invariants [15]. The new model is a non-trivial extension of the earlier work because, as pointed out above, the anti-frame rule is the "source" of a circular monotonicity requirement.

Our approach can loosely be described as a metric space analogue of Pitts' approach to relational properties of domains [13] and thus consists of two steps. First, we consider a recursive metric space domain equation without any monotonicity requirement, for which we obtain a solution by appealing to a standard existence theorem. Second, we carve out a suitable subset of what might be called *hereditarily monotonic* functions. We show how to define this recursively specified subset as a fixed point of a suitable operator. The resulting subset of monotonic functions is, however, not a solution to the original recursive domain equation; hence we verify that the semantic constructions used to justify the anti-frame rule in [17] suitably restrict to the recursively defined subset of hereditarily monotonic functions. This results in a considerably simpler model construction than the earlier one in *loc. cit.* We show that our approach scales by extending the model to also allow for hiding of families of invariants, and using it to prove the soundness of Pottier's generalized frame and anti-frame rules [15].

Contributions. In summary, the contributions of this paper are (1) the development of a considerably simpler model of recursive worlds for showing the soundness of the anti-frame rule; (2) the use of this model to give the first soundness proof of the anti-frame rule in the expressive type and capability system of Charguéraud and Pottier; and (3) the extension of the model to include hiding of families of invariants, and showing the soundness of generalized frame and anti-frame rules. Moreover, at a conceptual level, we augment our earlier approach to constructing (step-indexed) recursive possible worlds based on a programming language's operational semantics via metric spaces [6] by a further tool, *viz.*, defining worlds as recursive subsets of recursive metric spaces.

Outline. In the next section we give a brief overview of Charguéraud and Pottier's type and capability system [8,14] with higher-order frame and anti-frame rules. Section 3 summarizes some background on ultrametric spaces and presents the construction of a set of hereditarily monotonic recursive worlds. The worlds thus constructed are then used (Section 4) to give a model of the type and capability system. Finally, in Section 5 we show how to extend the model to also prove soundness of the generalized frame and anti-frame rules.

2 A Calculus of Capabilities

Syntax and operational semantics. We consider a standard call-by-value, higher-order language with general references, sum and product types, and polymorphic and recursive types. For concreteness, the following grammar gives the syntax of values and expressions, keeping close to the notation of [8,14]:

$$\begin{aligned} v &::= x \mid () \mid \mathsf{inj}^i\, v \mid (v_1, v_2) \mid \mathsf{fun}\, f(x){=}t \mid l \\ t &::= v \mid (v\, t) \mid \mathsf{case}(v_1, v_2, v) \mid \mathsf{proj}^i\, v \mid \mathsf{ref}\, v \mid \mathsf{get}\, v \mid \mathsf{set}\, v \end{aligned}$$

Here, the term $\mathsf{fun}\, f(x){=}t$ stands for the recursive procedure f with body t, and locations l range over a countably infinite set Loc. The operational semantics is given by a relation $(t \mid h) \longmapsto (t' \mid h')$ between configurations that consist of a (closed) expression t and a heap h. We take a heap h to be a finite map from locations to closed values, we use the notation $h\#h'$ to indicate that two heaps h, h' have disjoint domains, and we write $h \cdot h'$ for the union of two such heaps. By Val we denote the set of closed values.

Types. Charguéraud and Pottier's type system uses *capabilities*, *value types*, and *memory types*, as summarized in Figure 1. A capability C describes a heap property, much like the assertions of a Hoare-style program logic. For instance, $\{\sigma : \mathsf{ref\ int}\}$ asserts that σ is a valid location that contains an integer value. More complex assertions can be built by separating conjunctions $C_1 * C_2$ and universal

Variables	$\xi ::= \alpha \mid \beta \mid \gamma \mid \sigma$
Capabilities	$C ::= C \otimes C \mid \emptyset \mid C * C \mid \{\sigma : \theta\} \mid \exists\sigma.C \mid \gamma \mid \mu\gamma.C \mid \forall\xi.C$
Value types	$\tau ::= \tau \otimes C \mid 0 \mid 1 \mid \mathsf{int} \mid \tau + \tau \mid \tau \times \tau \mid \chi \to \chi \mid [\sigma] \mid \alpha \mid \mu\alpha.\tau \mid \forall\xi.\tau$
Memory types	$\theta ::= \theta \otimes C \mid \tau \mid \theta + \theta \mid \theta \times \theta \mid \mathsf{ref}\,\theta \mid \theta * C \mid \exists\sigma.\theta \mid \beta \mid \mu\beta.\theta \mid \forall\xi.\theta$
Computation types	$\chi ::= \chi \otimes C \mid \tau \mid \chi * C \mid \exists\sigma.\chi$
Value contexts	$\Delta ::= \Delta \otimes C \mid \varnothing \mid \Delta, x{:}\tau$
Linear contexts	$\Gamma ::= \Gamma \otimes C \mid \varnothing \mid \Gamma, x{:}\chi \mid \Gamma * C$

Fig. 1. Capabilities and types

and existential quantification over names σ. Value types τ classify values; they include base types, singleton types $[\sigma]$, and are closed under products, sums, and universal quantification. (We do not consider existential types in this paper.) *Memory types* (and the subset of computation types χ) describe the result of computations. They extend the value types by a type of references, and also include all types of the form $\exists\vec{\sigma}.\tau * C$ which describe both the value and heap that result from the evaluation of an expression. Arrow types (which are value types) have the form $\chi_1 \to \chi_2$ and thus, like the pre- and post-conditions of a triple in Hoare logic, make explicit which part of the heap is accessed and modified by a procedure call. We allow recursive capabilities, value types, and memory types, resp., provided the recursive definition is formally contractive [11], i.e., the recursion must go through a type constructor such as × or →.

Since Charguéraud and Pottier's system tracks aliasing, so-called strong (i.e., non-type preserving) updates are permitted: a possible type for such an update operation is $\forall\sigma,\sigma'.([\sigma]\times[\sigma']) * \{\sigma : \mathsf{ref}\,\tau\} \to \mathbf{1} * \{\sigma : \mathsf{ref}\,[\sigma']\}$. Here, the argument to the procedure is a pair consisting of a location (named σ) and the value to be stored (named σ'), and the location is assumed to be allocated in the initial heap (and store a value of some type τ). The result of the procedure is unit, but as a side-effect σ' will be stored at the location σ.

Frame and anti-frame rules. Each of the syntactic categories is equipped with an *invariant extension* operation, $\cdot \otimes C$. Intuitively, this operation conjoins C to the domain and codomain of every arrow type that occurs within its left hand argument, which means that the capability C is preserved by all procedures of this type. This intuition is made precise by regarding capabilities and types modulo a structural equivalence which subsumes the "distribution axioms" for $\otimes$ that are used to express generic higher-order frame rules [5]. The two key cases of the structural equivalence are the distribution axioms for arrow types, $(\chi_1 \to \chi_2) \otimes C = (\chi_1 \otimes C * C) \to (\chi_2 \otimes C * C)$, and for successive extensions, $(\chi \otimes C_1) \otimes C_2 = \chi \otimes (C_1 \circ C_2)$ where the derived operation $C_1 \circ C_2$ abbreviates the conjunction $(C_1 \otimes C_2) * C_2$.

There are two typing judgements, $x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash v : \tau$ for values, and $x_1{:}\chi_1, \ldots, x_n{:}\chi_n \Vdash t : \chi$ for expressions. The latter is similar to a Hoare triple where (the separating conjunction of) $\chi_1, \ldots, \chi_n$ serves as a precondition and χ

as a postcondition. This view provides some intuition for the following "shallow" and "deep" frame rules, and for the (essentially dual) anti-frame rule:

$$[SF]\frac{\mathit{\Gamma} \Vdash t : \chi}{\mathit{\Gamma} * C \Vdash t : \chi * C} \qquad [DF]\frac{\mathit{\Gamma} \Vdash t : \chi}{(\mathit{\Gamma} \otimes C) * C \Vdash t : (\chi \otimes C) * C}$$
$$[AF]\frac{\mathit{\Gamma} \otimes C \Vdash t : (\chi \otimes C) * C}{\mathit{\Gamma} \Vdash t : \chi} \tag{1}$$

As in separation logic, the frame rules can be used to add a capability C (which might assert the existence of an integer reference, say) as an invariant to a specification $\mathit{\Gamma} \Vdash t : \chi$, which is useful for local reasoning. The difference between the shallow variant $[SF]$ and the deep variant $[DF]$ is that the former adds C only on the top-level, whereas the latter also extends all arrow types nested inside $\mathit{\Gamma}$ and χ, via $\cdot \otimes C$. While the frame rules can be used to reason about certain forms of information hiding [5], the anti-frame rule expresses a hiding principle more directly: the capability C can be removed from the specification if C is an invariant that is established by t, expressed by $\cdot * C$, and that is guaranteed to hold whenever control passes from t to the context and back, expressed by $\cdot \otimes C$.

Pottier [14] illustrates the anti-frame rule by a number of applications. One of these is a fixed-point combinator implemented by means of "Landin's knot", i.e., recursion through heap. Every time the combinator is called with a functional $f : (\chi_1 \rightarrow \chi_2) \rightarrow (\chi_1 \rightarrow \chi_2)$, a new reference cell σ is allocated in order to set up the recursion required for the resulting fixed point $\mathit{fix}\, f$. Subsequent calls to $\mathit{fix}\, f$ still rely on this cell, and in Charguéraud and Pottier's system this is reflected in the type $(\chi_1 \rightarrow \chi_2) \otimes I$ of $\mathit{fix}\, f$, where the capability $I = \{\sigma : \mathsf{ref}\,(\chi_1 \rightarrow \chi_2) \otimes I\}$ describes the cell σ after it has been initialized. However, the anti-frame rule allows one to hide the existence of σ, and leads to a purely functional interface of the fixed point combinator. In particular, after hiding I, $\mathit{fix}\, f$ has the much simpler type $(\chi_1 \rightarrow \chi_2)$, which means that we can reason about aliasing and type safety of programs that *use* the fixed-point combinator without considering the reference cells used internally by that combinator.

3 Hereditarily Monotonic Recursive Worlds

Intuitively, capabilities describe heaps. A key idea of the model that we present next is that capabilities (as well as types and type contexts) are parameterized by invariants – this will make it easy to interpret the invariant extension operation $\otimes$, as in [16,17]. That is, rather than interpreting a capability C directly as a set of heaps, we interpret it as a function $[\![C]\!] : W \rightarrow \mathit{Pred}(\mathit{Heap})$ that maps "invariants" from W to sets of heaps. Intuitively, invariant extension of C is then interpreted by applying $[\![C]\!]$ to the given invariant. In contrast, a simple interpretation of C as a set of heaps would not contain enough information to determine the meaning of every invariant extension of C.

The question is now what the set W of invariants should be. As the frame and anti-frame rules in (1) indicate, invariants are in fact arbitrary capabilities, so W

should be the set used to interpret capabilities. But, as we just saw, capabilities should be interpreted as functions from W to $\mathit{Pred}(\mathit{Heap})$. Thus, we are led to consider a Kripke model where the worlds are *recursively defined*: to a first approximation, we need a solution to the equation

$$W \;=\; W \to \mathit{Pred}(\mathit{Heap})\ . \tag{2}$$

In fact, we will also need to consider a preorder on W and ensure that the interpretation of capabilities and types is *monotonic*. We will find a solution to a suitable variant of (2) using ultrametric spaces.

Ultrametric spaces. We recall some basic definitions and results about ultrametric spaces; for a less condensed introduction to ultrametric spaces we refer to [18]. A *1-bounded ultrametric space* (X, d) is a metric space where the distance function $d : X \times X \to \mathbb{R}$ takes values in the closed interval $[0, 1]$ and satisfies the "strong" triangle inequality $d(x, y) \leq \max\{d(x, z), d(z, y)\}$. A metric space is *complete* if every Cauchy sequence has a limit. A function $f : X_1 \to X_2$ between metric spaces (X_1, d_1), (X_2, d_2) is *non-expansive* if $d_2(f(x), f(y)) \leq d_1(x, y)$ for all $x, y \in X_1$. It is *contractive* if there exists some $\delta < 1$ such that $d_2(f(x), f(y)) \leq \delta \cdot d_1(x, y)$ for all $x, y \in X_1$. By the Banach fixed point theorem, every contractive function $f : X \to X$ on a complete and non-empty metric space (X, d) has a (unique) fixed point. By multiplication of the distances of (X, d) with a non-negative factor $\delta < 1$, one obtains a new ultrametric space, $\delta \cdot (X, d) = (X, d')$ where $d'(x, y) = \delta \cdot d(x, y)$.

The complete, 1-bounded, non-empty, ultrametric spaces and non-expansive functions between them form a Cartesian closed category CBUlt_{ne}. Products are given by the set-theoretic product where the distance is the maximum of the componentwise distances. The exponential $(X_1, d_1) \to (X_2, d_2)$ has the set of non-expansive functions from (X_1, d_1) to (X_2, d_2) as underlying set, and the distance function is given by $d_{X_1 \to X_2}(f, g) = \sup\{d_2(f(x), g(x)) \mid x \in X_1\}$.

The notation $x \overset{n}{=} y$ means that $d(x, y) \leq 2^{-n}$. Each relation $\overset{n}{=}$ is an equivalence relation because of the ultrametric inequality; we refer to this relation as "n-equality." Since the distances are bounded by 1, $x \overset{0}{=} y$ always holds, and the n-equalities become finer as n increases. If $x \overset{n}{=} y$ holds for all n then $x = y$.

Uniform predicates, worlds and world extension. Let $(A, \sqsubseteq)$ be a partially ordered set. An *upwards closed, uniform predicate* on A is a subset $p \subseteq \mathbb{N} \times A$ that is downwards closed in the first and upwards closed in the second component: if $(k, a) \in p$, $j \leq k$ and $a \sqsubseteq b$, then $(j, b) \in p$. We write $\mathit{UPred}(A)$ for the set of all such predicates on A, and we define $p_{[k]} = \{(j, a) \mid j < k\}$. Note that $p_{[k]} \in \mathit{UPred}(A)$. We equip $\mathit{UPred}(A)$ with the distance function $d(p, q) = \inf\{2^{-n} \mid p_{[n]} = q_{[n]}\}$, which makes $(\mathit{UPred}(A), d)$ an object of CBUlt_{ne}.

In our model, we use $\mathit{UPred}(A)$ with the following concrete instances for the partial order $(A, \sqsubseteq)$: (1) *heaps* $(\mathit{Heap}, \sqsubseteq)$, where $h \sqsubseteq h'$ iff $h' = h \cdot h_0$ for some $h_0 \# h$, (2) *values* $(\mathit{Val}, \sqsubseteq)$, where $u \sqsubseteq v$ iff $u = v$, and (3) *stateful values* $(\mathit{Val} \times \mathit{Heap}, \sqsubseteq)$, where $(u, h) \sqsubseteq (v, h')$ iff $u = v$ and $h \sqsubseteq h'$. We also use variants of the latter two instances where the set *Val* is replaced by the set of value substitutions, *Env*, and by the set of closed expressions, *Exp*. On *UPred*(*Heap*), ordered by subset inclusion, we have a complete Heyting BI algebra structure [4]. Below we only need the separating conjunction and its unit I, given by

$$p_1 * p_2 \;=\; \{(k,h) \mid \exists h_1, h_2.\ h = h_1 \cdot h_2 \ \wedge\ (k,h_1) \in p_1 \wedge\ (k,h_2) \in p_2\}$$

and $I = \mathbb{N} \times \mathit{Heap}$. Still, this observation on *UPred*(*Heap*) suggests that Pottier and Charguéraud's system could be extended to a full-blown program logic.

It is well-known that one can solve recursive domain equations in CBUlt_{ne} by an adaptation of the inverse-limit method from classical domain theory [3]. In particular, with regard to the domain equation (2) above:

Theorem 1. *There exists a unique (up to isomorphism) metric space* $(X,d) \in \mathit{CBUlt}_{ne}$ *and an isomorphism* ι *from* $\frac{1}{2} \cdot X \to \mathit{UPred}(\mathit{Heap})$ *to* X.

Using the pointwise lifting of separating conjunction to $\frac{1}{2} \cdot X \to \mathit{UPred}(\mathit{Heap})$ we define a *composition operation* on X. More precisely, $\circ : X \times X \to X$ is a non-expansive operation that for all $p, q, x \in X$ satisfies

$$\iota^{-1}(p \circ q)(x) \;=\; \iota^{-1}(p)(q \circ x) \,*\, \iota^{-1}(q)(x)\ ,$$

and it can be defined by an easy application of Banach's fixed point theorem as in [16]. This operation reflects the syntactic abbreviation $C_1 \circ C_2 = C_1 \otimes C_2 * C_2$ of conjoining C_1 and C_2 and additionally applying an invariant extension to C_1; the isomorphism ι^{-1} lets us view p, q and $p \circ q$ as *UPred*(*Heap*)-valued functions on $\frac{1}{2} \cdot X$. One can show that this operation $\circ$ is associative and has a left and right unit given by $\mathit{emp} = \iota(\lambda w.I)$; thus $(X, \circ, \mathit{emp})$ is a monoid in CBUlt_{ne}.

Then, using $\circ$ we define an *extension operation* $\otimes : Y^{(1/2 \cdot X)} \times X \to Y^{(1/2 \cdot X)}$ for any $Y \in \mathit{CBUlt}_{ne}$ by $(f \otimes x)(x') = f(x \circ x')$. Not going into details here, let us remark that $\otimes$ is the semantic counterpart to the syntactic invariant extension, and thus plays a key role in the model. However, for Pottier's anti-frame rule we also need to ensure that specifications are not invalidated by invariant extension. This requirement is stated via monotonicity, as we discuss next.

Relations on ultrametric spaces and hereditarily monotonic worlds As a conseqence of the fact that $\circ$ defines a monoid structure on X there is an induced preorder on X: $x \sqsubseteq y \ \Leftrightarrow\ \exists x_0.\ y = x \circ x_0$.

For modelling the anti-frame rule, we aim for a set of worlds similar to $X \cong 1/2 \cdot X \to \mathit{UPred}(\mathit{Heap})$ but where the function space consists of the non-expansive functions that are additionally monotonic, with respect to the order induced by $\circ$ on X and with respect to set inclusion on *UPred*(*Heap*):

$$(W, \sqsubseteq) \;\cong\; \tfrac{1}{2} \cdot (W, \sqsubseteq) \to_{mon} (\mathit{UPred}(\mathit{Heap}), \subseteq)\ . \tag{3}$$

Because the definition of the order $\sqsubseteq$ (induced by $\circ$) already uses the isomorphism between left-hand and right-hand side, and because the right-hand side depends on the order for the monotonic function space, the standard existence theorems for solutions of recursive domain equations do not appear to apply to (3). Previously we have constructed a solution to this equation explicitly as inverse limit of a suitable chain of approximations [17]. We show in the following that we can alternatively carve out from X a suitable subset of what we call *hereditarily monotonic* functions. This subset needs to be defined recursively.

Let $\mathcal{R}$ be the collection of all non-empty and closed relations $R \subseteq X$. We set

$$R_{[n]} \overset{def}{=} \{y \mid \exists x \in X.\ x \overset{n}{=} y \ \wedge\ x \in R\} \ .$$

for $R \in \mathcal{R}$. Thus, $R_{[n]}$ is the set of all points within distance 2^{-n} of R. Note that $R_{[n]} \in \mathcal{R}$. In fact, $\emptyset \neq R \subseteq R_{[n]}$ holds by the reflexivity of n-equality, and if $(y_k)_{k\in\mathbb{N}}$ is a sequence in $R_{[n]}$ with limit y in X then $d(y_k, y) \leq 2^{-n}$ must hold for some k, i.e., $y_k \overset{n}{=} y$. So there exists $x \in X$ with $x \in R$ and $x \overset{n}{=} y_k$, and hence by transitivity $x \overset{n}{=} y$ which then gives $\lim_n y_n \in R_{[n]}$.

We make some further observations that follow from the properties of n-equality on X. First, $R \subseteq S$ implies $R_{[n]} \subseteq S_{[n]}$ for any $R, S \in \mathcal{R}$. Moreover, using the fact that the n-equalities become increasingly finer it follows that $(R_{[m]})_{[n]} = R_{[\min(m,n)]}$ for all $m, n \in \mathbb{N}$, so in particular each $(\cdot)_{[n]}$ is a closure operation on $\mathcal{R}$. As a consequence, we have $R \subseteq \ldots \subseteq R_{[n]} \subseteq \ldots \subseteq R_{[1]} \subseteq R_{[0]}$. By the 1-boundedness of X, $R_{[0]} = X$ for all $R \in \mathcal{R}$. Finally, $R = S$ if and only if $R_{[n]} = S_{[n]}$ for all $n \in \mathbb{N}$.

Proposition 2. *Let $d : \mathcal{R} \times \mathcal{R} \to \mathbb{R}$ be defined by $d(R, S) = \inf\{2^{-n} \mid R_{[n]} = S_{[n]}\}$. Then $(\mathcal{R}, d)$ is a complete, 1-bounded, non-empty ultrametric space. The limit of a Cauchy chain $(R_n)_{n\in\mathbb{N}}$ with $d(R_n, R_{n+1}) \leq 2^{-n}$ is given by $\bigcap_n (R_n)_{[n]}$, and in particular $R = \bigcap_n R_{[n]}$ for any $R \in \mathcal{R}$.*

We will now define the set of hereditarily monotonic functions W as a recursive predicate on the space X. Let the function $\Phi : \mathcal{P}(X) \to \mathcal{P}(X)$ on subsets of X be given by $\Phi(R) = \{\iota(p) \mid \forall x, x_0 \in R.\ p(x) \subseteq p(x \circ x_0)\}$.

Lemma 3. *Φ restricts to a contractive function on $\mathcal{R}$: if $R \in \mathcal{R}$ then $\Phi(R)$ is non-empty and closed, and $R \overset{n}{=} S$ implies $\Phi(R) \overset{n+1}{=} \Phi(S)$.*

While the proof of this lemma is not particularly difficult, we include it here to illustrate the kind of reasoning that is involved.

Proof. It is clear that $\Phi(R) \neq \emptyset$ since $\iota(p) \in \Phi(R)$ for every constant function p from $\frac{1}{2} \cdot X$ to $\mathit{UPred}(\mathit{Heap})$. Limits of Cauchy chains in $\frac{1}{2} \cdot X \to \mathit{UPred}(\mathit{Heap})$ are given pointwise, hence $(\lim_n p_n)(x) \subseteq (\lim_n p_n)(x \circ x_0)$ holds for all Cauchy chains $(p_n)_{n\in\mathbb{N}}$ in $\Phi(R)$ and all $x, x_0 \in R$. This proves $\Phi(R) \in \mathcal{R}$.

We now show that Φ is contractive. To this end, let $n \geq 0$ and assume $R \overset{n}{=} S$. Let $\iota(p) \in \Phi(R)_{[n+1]}$. We must show that $\iota(p) \in \Phi(S)_{[n+1]}$. By definition of the closure operation there exists $\iota(q) \in \Phi(R)$ such that p and q are $(n{+}1)$-equal. Set $r(w) = q(w)_{[n+1]}$. Then r and p are also $(n{+}1)$-equal, hence it suffices to show that $\iota(r) \in \Phi(S)$. To establish the latter, let $w_0, w_1 \in S$ be arbitrary. By the assumption that R and S are n-equal there exist elements $w_0', w_1' \in R$ such that $w_0' \overset{n}{=} w_0$ and $w_1' \overset{n}{=} w_1$ in holds X, or equivalently, such that w_0' and w_0 as well as w_1' and w_1 are $(n{+}1)$-equal in $\frac{1}{2} \cdot X$. By the non-expansiveness of $\circ$, this implies that also $w_0' \circ w_1'$ and $w_0 \circ w_1$ are $(n{+}1)$-equal in $\frac{1}{2} \cdot X$. Since

$$q(w_0) \ \overset{n+1}{=}\ q(w_0') \ \subseteq\ q(w_0' \circ w_1') \ \overset{n+1}{=}\ q(w_0 \circ w_1)$$

holds by the non-expansiveness of q and the assumption that $\iota(q) \in \Phi(R)$, we obtain the required inclusion $r(w_0) \subseteq r(w_0 \circ w_1)$ by definition of r. □

By Proposition 2 and the Banach theorem we can now define the hereditarily monotonic functions W as the uniquely determined fixed point of Φ, for which

$$w \in W \ \Leftrightarrow\ \exists p.\ w = \iota(p) \ \wedge\ \forall w, w_0 \in W.\ p(w) \subseteq p(w \circ w_0)\ .$$

Note that W thus constructed does not quite satisfy (3). We do not have an isomorphism between W and the non-expansive and monotonic functions from W (viewed as an ultrametric space itself), but rather between W and all functions from X that *restrict* to monotonic functions whenever applied to hereditarily monotonic arguments. Keeping this in mind, we abuse notation and write

$$\begin{aligned}&\tfrac{1}{2} \cdot W \to_{mon} \mathit{UPred}(A)\\ &\qquad \overset{\mathit{def}}{=}\ \{p : \tfrac{1}{2} \cdot X \to \mathit{UPred}(A) \mid \forall w_1, w_2 \in W.\ p(w_1) \subseteq p(w_1 \circ w_2)\}\ .\end{aligned}$$

Then, for our particular application of interest, we also have to ensure that all the operations restrict appropriately (*cf.* Section 4 below). Here, as a first step, we show that the composition operation $\circ$ restricts to W. In turn, this means that the $\otimes$ operator restricts accordingly: if $w \in W$ and p is in $\frac{1}{2} \cdot W \to_{mon} \mathit{UPred}(A)$ then so is $p \otimes w$.

Lemma 4. *For all $n \in \mathbb{N}$, if $w_1, w_2 \in W$ then $w_1 \circ w_2 \in W_{[n]}$. In particular, since $W = \bigcap_n W_{[n]}$ it follows that $w_1, w_2 \in W$ implies $w_1 \circ w_2 \in W$.*

Proof. The proof is by induction on n. The base case is immediate as $W_{[0]} = X$. Now suppose $n > 0$ and let $w_1, w_2 \in W$; we must prove that $w_1 \circ w_2 \in W_{[n]}$. Let w_1' be such that $\iota^{-1}(w_1')(w) = \iota^{-1}(w_1)(w)_{[n]}$. Observe that $w_1' \in W$, that w_1' and w_1 are n-equal, and that w_1' is such that n-equality of w, w' in $\frac{1}{2} \cdot X$ already implies $\iota^{-1}(w_1')(w) = \iota^{-1}(w_1')(w')$. Since w_1' and w_1 are n-equivalent, the non-expansiveness of the composition operation implies $w_1 \circ w_2 \overset{n}{=} w_1' \circ w_2$. Thus it suffices to show that $w_1' \circ w_2 \in W = \Phi(W)$. To see this, let $w, w_0 \in W$ be arbitrary, and note that by induction hypothesis we have $w_2 \circ w \in W_{[n-1]}$. This means that there exists $w' \in W$ such that $w' \overset{n}{=} w_2 \circ w$ holds in $\frac{1}{2} \cdot X$, hence

$$\begin{aligned}\iota^{-1}(w_1' \circ w_2)(w) &= \iota^{-1}(w_1')(w_2 \circ w) * \iota^{-1}(w_2)(w) && \text{by definition of } \circ\\ &= \iota^{-1}(w_1')(w') * \iota^{-1}(w_2)(w) && \text{by } w' \overset{n}{=} w_2 \circ w\\ &\subseteq \iota^{-1}(w_1')(w' \circ w_0) * \iota^{-1}(w_2)(w \circ w_0) && \text{by hereditariness}\\ &= \iota^{-1}(w_1')((w_2 \circ w) \circ w_0) * \iota^{-1}(w_2)(w \circ w_0) && \text{by } w' \overset{n}{=} w_2 \circ w\\ &= \iota^{-1}(w_1' \circ w_2)(w \circ w_0) && \text{by definition of } \circ.\end{aligned}$$

Since w, w_0 were chosen arbitrarily, this calculation establishes $w_1' \circ w_2 \in W$. □

4 Step-Indexed Possible World Semantics of Capabilities

We define semantic domains for the capabilities and types of the calculus described in Section 2,

$$\begin{aligned} Cap &= \tfrac{1}{2}\cdot W \rightarrow_{mon} \mathit{UPred}(\mathit{Heap}) \\ VT &= \tfrac{1}{2}\cdot W \rightarrow_{mon} \mathit{UPred}(\mathit{Val}) \\ MT &= \tfrac{1}{2}\cdot W \rightarrow_{mon} \mathit{UPred}(\mathit{Val} \times \mathit{Heap})\ , \end{aligned}$$

so that $p \in Cap$ if and only if $\iota(p) \in W$. Next, we define operations on the semantic domains that correspond to the syntactic type and capability constructors. The most interesting of these is the one for arrow types. Given $T_1, T_2 \in 1/2 \cdot X \rightarrow \mathit{UPred}(\mathit{Val} \times \mathit{Heap})$, $T_1 \rightarrow T_2$ in $\frac{1}{2} \cdot X \rightarrow \mathit{UPred}(\mathit{Val})$ is defined on $x \in X$ as

$$\begin{aligned} \{(k, \mathsf{fun}\, f(y){=}t) \mid \forall j < k.\ \forall w \in W.\ \forall r \in \mathit{UPred}(\mathit{Heap}). \\ \forall v, h.\ (j,(v,h)) \in T_1(x \circ w) * \iota^{-1}(x \circ w)(\mathit{emp}) * r \ \Rightarrow \\ (j, (t[f{:=}\mathsf{fun}\, f(y){=}t, y{:=}v], h)) \in \mathcal{E}(T_2 * r)(x \circ w)\}\ , \end{aligned} \tag{4}$$

where $\mathcal{E}(T)$ is the extension of a world-indexed, uniform predicate on $\mathit{Val} \times \mathit{Heap}$ to one on $\mathit{Exp} \times \mathit{Heap}$. It is here where the index is linked to the operational semantics: $(k,(t,h)) \in \mathcal{E}(T)(x)$ if and only if for all $j \le k, t', h'$,

$$\begin{aligned} (t \,|\, h) \longmapsto^{j} (t' \,|\, h') \ \wedge\ (t' \,|\, h') \text{ irreducible} \\ \Rightarrow\ (k{-}j, (t', h')) \in \textstyle\bigcup_{w' \in W} T(x \circ w') * \iota^{-1}(x \circ w')(\mathit{emp})\ . \end{aligned}$$

Definition (4) realizes the key ideas of our model as follows. First, the universal quantification over $w \in W$ and subsequent use of the world $x \circ w$ builds in monotonicity, and intuitively means that $T_1 \rightarrow T_2$ is parametric in (and hence preserves) invariants that have been added by the procedure's context. In particular, (4) states that procedure application preserves this invariant, when viewed as the predicate $\iota^{-1}(x \circ w)(\mathit{emp})$. By also conjoining r as an invariant we "bake in" the first-order frame property, which results in a subtyping axiom $T_1 \rightarrow T_2 \le T_1 * C \rightarrow T_2 * C$ in the type system. The existential quantification over w', in the definition of $\mathcal{E}$, allows us to "absorb" a part of the local heap description into the world. Finally, the quantification over indices $j < k$ in (4) achieves that $(T_1 \rightarrow T_2)(x)$ is uniform. There are three reasons why we require that j be *strictly* less than k. Technically, the use of $\iota^{-1}(x \circ w)$ in the definition "undoes" the scaling by 1/2, and $j < k$ is needed to ensure the non-expansiveness of $T_1 \rightarrow T_2$ as a function $1/2 \cdot X \rightarrow \mathit{UPred}(\mathit{Val})$. Moreover, it lets us prove the typing rule for *recursive* functions by induction on k. Finally, it means that $\rightarrow$ is a contractive type constructor, which justifies the formal contractiveness assumption about arrow types that we made earlier. Intuitively, the use of $j < k$ for the arguments suffices since application consumes a step.

The function type constructor, as well as all the other type and capability constructors, restrict to *Cap*, *VT* and *MT*, respectively. With their help it becomes straightforward to define the interpretation $[\![C]\!]_\eta$ and $[\![\tau]\!]_\eta$ of capabilities and types, given an environment η which maps region names $\sigma \in \mathit{RegName}$ to closed values $\eta(\sigma) \in \mathit{Val}$, capability variables γ to semantic capabilities $\eta(\gamma) \in \mathit{Cap}$, and type variables α and β to semantic types $\eta(\alpha) \in \mathit{VT}$ and $\eta(\beta) \in \mathit{MT}$. The type equivalences can then be verified with respect to this interpretation. We state this for the case of arrow types:

Lemma 5. *Let T_1, T_2 non-expansive functions from $\frac{1}{2}{\cdot}X$ to $\mathit{UPred}(\mathit{Val} \times \mathit{Heap})$.*

1. *$T_1 \to T_2$ is non-expansive, and $(T_1 \to T_2)(x)$ is uniform for all $x \in X$.*
2. *$T_1 \to T_2 \in \mathit{VT}$.*
3. *The assignment of $T_1 \to T_2$ to T_1, T_2 is contractive.*
4. *Let $c \in \mathit{Cap}$ and $w \stackrel{\mathit{def}}{=} \iota(c)$. Then $(T_1 \to T_2) \otimes w = (T_1 \otimes w * c) \to (T_2 \otimes w * c)$.*

Recall that there are two kinds of typing judgments, one for typing of values and the other for the typing of expressions. The semantics of a value judgement simply establishes truth with respect to all worlds w, environments η, and $k \in \mathbb{N}$:

$$\models (\Delta \vdash v : \tau) \stackrel{\mathit{def}}{\iff} \forall \eta.\ \forall w.\ \forall k.\ \forall \rho.\ (k, \rho) \in [\![\Delta]\!]_\eta\, w \ \Rightarrow\ (k, \rho(v)) \in [\![\tau]\!]_\eta\, w\ .$$

Here $\rho(v)$ means the application of the substitution ρ to v. The judgement for expressions mirrors the interpretation of the arrow case for value types, in that there is also a quantification over heap predicates $r \in \mathit{UPred}(\mathit{Heap})$ and an existential quantification over $w' \in W$ through the use of $\mathcal{E}$:

$$\models\ (\Gamma \Vdash t : \chi) \stackrel{\mathit{def}}{\iff} \forall \eta.\ \forall w.\ \forall k.\ \forall r \in \mathit{UPred}(\mathit{Heap}).$$
$$\forall \rho, h.\ (k, (\rho, h)) \in [\![\Gamma]\!]_\eta\, w * \iota^{-1}(w)(\mathit{emp}) * r \ \Rightarrow\ (k, (\rho(t), h)) \in \mathcal{E}([\![\chi]\!]_\eta * r)(w).$$

Theorem 6 (Soundness). *If $\Delta \vdash v : \tau$ then $\models (\Delta \vdash v : \tau)$, and if $\Gamma \Vdash t : \chi$ then $\models (\Gamma \Vdash t : \chi)$.*

To prove the theorem, we show that each typing rule preserves the truth of judgements. Detailed proofs for the shallow and deep frame rules are included in the appendix. Here, we consider the anti-frame rule. Its proof employs so-called commutative pairs [14,17], a property expressed by the following lemma.

Lemma 7. *For all worlds $w_0, w_1 \in W$, there exist $w_0', w_1' \in W$ such that*

$$w_0' = \iota(\iota^{-1}(w_0) \otimes w_1'), \qquad w_1' = \iota(\iota^{-1}(w_1) \otimes w_0'), \qquad \mathit{and} \qquad w_0 \circ w_1' = w_1 \circ w_0'\ .$$

Lemma 8 (Soundness of the anti-frame rule). *Suppose $\models (\Gamma \otimes C \Vdash t : \chi \otimes C * C)$. Then $\models (\Gamma \Vdash t : \chi)$.*

Proof. We prove $\models (\Gamma \Vdash t : \chi)$. Let $w \in W$, η an environment, $r \in \mathit{UPred}(\mathit{Heap})$ and

$$(k, (\rho, h)) \in [\![\Gamma]\!]_\eta\, (w) * \iota^{-1}(w)(\mathit{emp}) * r\ .$$

We must prove $(k, (\rho(t), h)) \in \mathcal{E}([\![\chi]\!]_\eta * r)(w)$. By Lemma 7,

$$w_1 = \iota(\iota^{-1}(w) \otimes w_2), \quad w_2 = \iota([\![C]\!]_\eta \otimes w_1) \quad \text{and} \quad \iota([\![C]\!]_\eta) \circ w_1 = w \circ w_2 \tag{5}$$

holds for some worlds w_1, w_2 in W.

First, we find a superset of the precondition $\llbracket \mathit{\Gamma} \rrbracket_\eta (w) * \iota^{-1}(w)(\mathit{emp}) * r$ in the assumption above, replacing the first two $*$-conjuncts as follows:

$$\begin{aligned}
\llbracket \mathit{\Gamma} \rrbracket_\eta (w) &\subseteq \llbracket \mathit{\Gamma} \rrbracket_\eta (w \circ w_2) && \text{by monotonicity of } \llbracket \mathit{\Gamma} \rrbracket_\eta \text{ and } w_2 \in W \\
&= \llbracket \mathit{\Gamma} \rrbracket_\eta (\iota(\llbracket C \rrbracket_\eta) \circ w_1) && \text{since } \iota(\llbracket C \rrbracket_\eta) \circ w_1 = w \circ w_2 \\
&= \llbracket \mathit{\Gamma} \otimes C \rrbracket_\eta (w_1) && \text{by definition of } \otimes.
\end{aligned}$$

$$\begin{aligned}
\iota^{-1}(w)(\mathit{emp}) &\subseteq \iota^{-1}(w)(\mathit{emp} \circ w_2) && \text{by monotonicity of } \iota^{-1}(w) \text{ and } w_2 \in W \\
&= \iota^{-1}(w)(w_2 \circ \mathit{emp}) && \text{since } \mathit{emp} \text{ is the unit} \\
&= (\iota^{-1}(w) \otimes w_2)(\mathit{emp}) && \text{by definition of } \otimes \\
&= \iota^{-1}(w_1)(\mathit{emp}) && \text{since } w_1 = \iota(\iota^{-1}(w) \otimes w_2).
\end{aligned}$$

Thus, by the monotonicity of separating conjunction, we have that

$$(k, (\rho, h)) \in \llbracket \mathit{\Gamma} \rrbracket_\eta (w) * \iota^{-1}(w)(\mathit{emp}) * r \;\subseteq\; \llbracket \mathit{\Gamma} \otimes C \rrbracket_\eta (w_1) * \iota^{-1}(w_1)(\mathit{emp}) * r \,. \quad (6)$$

By the assumed validity of the judgement $\mathit{\Gamma} \otimes C \Vdash t : \chi \otimes C * C$, (6) entails

$$(k, (\rho(t), h)) \in \mathcal{E}(\llbracket \chi \otimes C * C \rrbracket_\eta * r)(w_1) \,. \quad (7)$$

We need to show that $(k, (\rho(t), h)) \in \mathcal{E}(\llbracket \chi \rrbracket_\eta * r)(w)$, so assume $(\rho(t) \,|\, h) \longmapsto^j (t' \,|\, h')$ for some $j \leq k$ such that $(t' \,|\, h')$ is irreducible. From (7) we then obtain

$$(k{-}j, (t', h')) \in \textstyle\bigcup_{w'} \llbracket \chi \otimes C * C \rrbracket_\eta (w_1 \circ w') * \iota^{-1}(w_1 \circ w')(\mathit{emp}) * r \,. \quad (8)$$

Now observe that we have

$$\begin{aligned}
&\llbracket \chi \otimes C * C \rrbracket_\eta (w_1 \circ w') * \iota^{-1}(w_1 \circ w')(\mathit{emp}) \\
&\quad = \llbracket \chi \rrbracket_\eta (\iota(\llbracket C \rrbracket_\eta) \circ w_1 \circ w') * \llbracket C \rrbracket_\eta (w_1 \circ w') * \iota^{-1}(w_1 \circ w')(\mathit{emp}) \\
&\quad = \llbracket \chi \rrbracket_\eta (\iota(\llbracket C \rrbracket_\eta) \circ w_1 \circ w') * \iota^{-1}(\iota(\llbracket C \rrbracket_\eta) \circ w_1 \circ w')(\mathit{emp}) \\
&\quad = \llbracket \chi \rrbracket_\eta (w \circ w'') * \iota^{-1}(w \circ w'')(\mathit{emp})
\end{aligned}$$

for $w'' \overset{\mathit{def}}{=} w_2 \circ w'$, since $w \circ w_2 = \iota(\llbracket C \rrbracket_\eta) \circ w_1$. Thus, (8) entails that $(k{-}j, (t', h'))$ is in $\bigcup_{w''} \llbracket \chi \rrbracket_\eta (w \circ w'') * \iota^{-1}(w \circ w'')(\mathit{emp}) * r$, and we are done. □

5 Generalized Frame and Anti-frame Rules

The frame and anti-frame rules allow for hiding of *invariants*. However, to hide uses of local state, say for a function, it is, in general, not enough only to allow hiding of global invariants that are preserved across arbitrary sequences of calls and returns. For instance, consider the function f with local reference cell r:

$$\mathsf{let}\ r = \mathsf{ref}\ 0\ \mathsf{in}\ \mathsf{fun}\ f(g) {=} (\mathit{inc}(r); g(); \mathit{dec}(r)) \quad (9)$$

If we write $\mathsf{int}\ n$ for the singleton integer type containing n, we may wish to hide the capability $I = \{\sigma : \mathsf{ref}\,(\mathsf{int}\ 0)\}$ to capture the intuition that the cell $r : [\sigma]$

stores 0 upon termination. However, there could well be re-entrant calls to f and $\{\sigma : \mathsf{ref}\,(\mathsf{int}\ 0)\}$ is not an invariant for those calls.

Thus Pottier [15] proposed two extensions to the anti-frame rule that allows for hiding of families of invariants. The first idea is that each invariant in the family is a *local* invariant that holds for one level of the recursive call of a function. This extension allows us to hide "well-bracketed" [10] uses of local state. For instance, the $\mathbb{N}$-indexed family of invariants $I\,n = \{\sigma : \mathsf{ref}\,(\mathsf{int}\ n)\}$ can be used for (9); see the examples in [15]. The second idea is to allow each local invariant to *evolve* in some monotonic fashion; this allows us to hide even more uses of local state. The idea is related to the notion of evolving invariants for local state in recent work on reasoning about contextual equivalence [1,10]. (Space limitations preclude us from including examples; please see [15] for examples.)

In summary, we want to allow the hiding of a family of capabilities $(I\,i)_{i\in\kappa}$ indexed over a preordered set $(\kappa, \leq)$. The preorder is used to capture that the local invariants can evolve in a monotonic fashion, as expressed in the new definition of the action of $\otimes$ on function types (note that I on the right-hand side of $\otimes$ now has kind $\kappa \to \textsc{cap}$):

$$(\chi_1 \to \chi_2) \otimes I \;=\; \forall i.\; \big((\chi_1 \otimes I) * I\,i \to \exists j \geq i.\; ((\chi_2 \otimes I) * I\,j)\big)$$

Observe how this definition captures the intuitive idea: if the invariant $I\,i$ holds when the function is called then, upon return, we know that an invariant $I\,j$ (for $j \in \kappa$, $j \geq i$) holds. Different recursive calls may use different local invariants due to the quantification over i. The generalized frame and anti-frame rules are:

$$[GF]\frac{\Gamma \Vdash t : \chi}{\Gamma \otimes I * I\,i \Vdash t : \exists j \geq i.\,(\chi \otimes I) * I\,j} \qquad [GAF]\frac{\Gamma \otimes I \Vdash t : \exists i.\,(\chi \otimes I) * I\,i}{\Gamma \Vdash t : \chi}$$

We now show how to extend our model of the type and capability calculus to accomodate hiding of such more expressive families of invariants. Naturally, the first step is to refine our notion of world, since the worlds are used to describe hidden invariants.

Generalized worlds and generalized world extension. Suppose $\mathcal{K}$ is a (small) collection of preordered sets. We write $\mathcal{K}^*$ for the finite sequences over $\mathcal{K}$, ε for the empty sequence, and use juxtaposition to denote concatenation. For convenience, we will sometimes identify a sequence $\alpha = \kappa_1, \ldots, \kappa_n$ over $\mathcal{K}$ with the preorder $\kappa_1 \times \cdots \times \kappa_n$. As in Section 3, we define the worlds for the Kripke model in two steps, starting from an equation without any monotonicity requirements: CBUlt_{ne} has all non-empty coproducts, and there is a unique solution to the two equations

$$X \cong \sum_{\alpha\in\mathcal{K}^*} X_\alpha\ , \quad X_{\kappa_1,\ldots,\kappa_n} = (\kappa_1 \times \cdots \times \kappa_n) \to (\tfrac{1}{2}\cdot X \to \mathit{UPred}(\mathit{Heap}))\ , \quad (10)$$

with isomorphism $\iota : \sum_{\alpha\in\mathcal{K}^*} X_\alpha \to X$ in CBUlt_{ne}, where each $\kappa \in \mathcal{K}$ is equipped with the discrete metric. Each X_α consists of the α-indexed families of (world-dependent) predicates so that, in comparison to Section 3, X consists of all these families rather than individual predicates.

The composition operation $\circ : X \times X \to X$ is now given by $x_1 \circ x_2 = \iota(\langle \alpha_1\alpha_2, p\rangle)$ where $\langle \alpha_i, p_i\rangle = \iota^{-1}(x_i)$, and where $p \in X_{\alpha_1\alpha_2}$ is defined by

$$p(i_1 i_2)(x) \;=\; p_1(i_1)(x_2 \circ x) * p_2(i_2)(x) \;.$$

for $i_1 \in \alpha_1$, $i_2 \in \alpha_2$. That is, the combination of an α_1-indexed family p_1 and an α_2-indexed family p_2 is a family p over $\alpha_1\alpha_2$, but there is no interaction between the index components i_1 and i_2: they concern disjoint regions of the heap.

From here on we can proceed essentially as in Section 3: The composition operation can be shown associative, with a left and right unit given by $emp = \iota(\langle \varepsilon, \lambda\_, \_.I\rangle)$. For $f : \frac{1}{2} \cdot X \to Y$ the extension operation $(f \otimes x)(x') = f(x \circ x')$ is also defined as before (but with respect to the solution (10) and the new $\circ$ operation). We then carve out from X the subset of hereditarily monotonic functions W, which we again obtain as fixed point of a contractive function on the closed and non-empty subsets of X. Let us write $\sim$ for the (recursive) partial equivalence relation on X where $\iota(\langle \alpha_1\alpha_2, p\rangle) \sim \iota(\langle \alpha_2\alpha_1, q\rangle)$ holds if $p(i_1 i_2)(x_1) = q(i_2 i_1)(x_2)$ for all $i_1 \in \alpha_1$, $i_2 \in \alpha_2$ and $x_1 \sim x_2$. Then $w \in W$ iff $w \sim w$ and

$$\exists \alpha, p.\; w = \iota\langle \alpha, p\rangle \;\wedge\; \forall i \in \alpha.\, \forall w_1, w_2 \in W.\; p(i)(w_1) \subseteq p(i)(w_1 \circ w_2) \;.$$

Finally, the proof of Lemma 4 can be adapted to show that the operation $\circ$ restricts to the subset W.

Semantics of capabilities and types. The definition of function types changes as follows: given $x \in X$, $(k, \mathsf{fun}\, f(y){=}t) \in (T_1 \to T_2)(x)$ if and only if

$$\begin{aligned}&\forall j < k.\ \forall w \in W \text{ where } \iota^{-1}(x \circ w) = \langle \alpha, p\rangle.\ \forall r \in \mathit{UPred}(\mathit{Heap}).\ \forall i \in \alpha.\ \forall v, h.\\ &(j, (v, h)) \in T_1(x \circ w) * p(i)(emp) * r \;\Rightarrow\\ &\qquad (j, t[f{:=}\mathsf{fun}\, f(y){=}t, y{:=}v], h)) \in \mathcal{E}(T_2 * r, x \circ w, i) \;,\end{aligned}$$

where the extension to expressions now depends on $i \in \alpha$: $(k, t) \in \mathcal{E}(T, x, i)$ if

$$\begin{aligned}&\forall j \le k, t', h'.\ (t \,|\, h) \longmapsto^j (t' \,|\, h') \;\wedge\; (t' \,|\, h') \text{ irreducible}\\ &\qquad\qquad \Rightarrow\; (k - j, (t', h')) \in \textstyle\bigcup_{w \in W,\, i_1 \in \alpha,\, i_2 \in \beta,\, i_1 \geq i} T(x \circ w) * q(i_1 i_2)(emp)\end{aligned}$$

for $\langle \alpha\beta, q\rangle = \iota^{-1}(x \circ w)$.

Next, one proves the analogue of Lemma 5 which shows the well-definedness of $T_1 \to T_2$ and (a semantic variant of) the distribution axiom for generalized invariants: in particular, given $p \in \kappa \to \mathit{Cap}$ and setting $w \overset{\mathit{def}}{=} \iota(\langle \kappa, p\rangle)$,

$$(T_1 \to T_2) \otimes w = \forall_{i \in \kappa}\big((T_1 \otimes w) * p\, i) \to \exists_{j \geq i}((T_2 \otimes w) * p\, j)\big)$$

where $\forall$ and $\exists$ denote the pointwise intersection and union of world-indexed uniform predicates.

Once similar properties are proved for the other type and capability constructors (which do not change for the generalized invariants), we obtain:

Theorem 9 (Soundness). *The generalized frame and anti-frame rules* $[GF]$ *and* $[GAF]$ *are sound.*

In particular, this theorem shows that all the reasoning about the use of local state in the (non-trivial) examples considered by Pottier in [15] is sound.

6 Conclusion and Future Work

We have developed the first soundness proof of the anti-frame rule in the expressive type and capability system of Charguéraud and Pottier by constructing a Kripke model of the system. For our model, we have used a new approach to the construction of worlds by definining them as a recursive subset of a recursively defined metric space, thus avoiding a tedious explicit inverse-limit construction. We have shown that this approach scales, by also extending the model to show soundness of Pottier's generalized frame and anti-frame rules. Future work includes exploring some of the orthogonal extensions of the basic type and capability system: group regions [8] and fates & predictions [12].

References

1. Ahmed, A., Dreyer, D., Rossberg, A.: State-dependent representation independence. In: POPL (2009)
2. Ahmed, A., Fluet, M., Morrisett, G.: L3: A linear language with locations. Fundam. Inf. 77(4), 397–449 (2007)
3. America, P., Rutten, J.J.M.M.: Solving reflexive domain equations in a category of complete metric spaces. J. Comput. Syst. Sci. 39(3), 343–375 (1989)
4. Biering, B., Birkedal, L., Torp-Smith, N.: BI-hyperdoctrines, higher-order separation logic, and abstraction. ACM Trans. Program. Lang. Syst. 29(5) (2007)
5. Birkedal, L., Torp-Smith, N., Yang, H.: Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. LMCS 2(5:1) (2006)
6. Birkedal, L., Reus, B., Schwinghammer, J., Støvring, K., Thamsborg, J., Yang, H.: Step-indexed Kripke models over recursive worlds. In: POPL (to appear, 2011)
7. Birkedal, L., Støvring, K., Thamsborg, J.: Realizability semantics of parametric polymorphism, general references, and recursive types. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 456–470. Springer, Heidelberg (2009)
8. Charguéraud, A., Pottier, F.: Functional translation of a calculus of capabilities. In: Proceedings of ICFP, pp. 213–224 (2008)
9. Crary, K., Walker, D., Morrisett, G.: Typed memory management in a calculus of capabilities. In: Proceedings of POPL, pp. 262–275 (1999)
10. Dreyer, D., Neis, G., Birkedal, L.: The impact of higher-order state and control effects on local relational reasoning. In: Proceedings of ICFP (2010)
11. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
12. Pilkiewicz, A., Pottier, F.: The essence of monotonic state (July 2010) (unpublished)
13. Pitts, A.M.: Relational properties of domains. Inf. Comput. 127(2), 66–90 (1996)
14. Pottier, F.: Hiding local state in direct style: a higher-order anti-frame rule. In: Proceedings of LICS, pp. 331–340 (2008)
15. Pottier, F.: Generalizing the higher-order frame and anti-frame rules (July 2009) (unpublished), http://gallium.inria.fr/~fpottier
16. Schwinghammer, J., Birkedal, L., Reus, B., Yang, H.: Nested hoare triples and frame rules for higher-order store. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 440–454. Springer, Heidelberg (2009)
17. Schwinghammer, J., Yang, H., Birkedal, L., Pottier, F., Reus, B.: A semantic foundation for hidden state. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 2–17. Springer, Heidelberg (2010)
18. Smyth, M.B.: Topology. In: Handbook of Logic in Computer Science, vol. 1. Oxford Univ. Press, Oxford (1992)

A Modified GoI Interpretation for a Linear Functional Programming Language and Its Adequacy

Naohiko Hoshino

Research Institute for Mathematical Science, Kyoto university
naophiko@kurims.kyoto-u.ac.jp

Abstract. Geometry of Interaction (GoI) introduced by Girard provides a semantics for linear logic and its cut elimination. Several extensions of GoI to programming languages have been proposed, but it is not discussed to what extent they capture behaviour of programs as far as the author knows. In this paper, we study GoI interpretation of a linear functional programming language (LFP). We observe that we can not extend the standard GoI interpretation to an adequate interpretation of LFP, and we propose a new adequate GoI interpretation of LFP by modifying the standard GoI interpretation. We derive the modified interpretation from a realizability model of LFP. We also relate the interpretation of recursion to cyclic computation (the trace operator in the category of sets and partial maps) in the realizability model.

1 Introduction

Geometry of Interaction (GoI) introduced by Girard [11] models linear logic by bidirectional computation which is executed by passing data along edges of a proof net. The main purpose of the original GoI is to capture dynamics of cut-elimination of linear logic (β-reduction of the linear lambda calculus). The original GoI interprets a proof as a matrix of operator algebras, and the *execution formula* captures the cut-elimination process. Since we can describe the computation of GoI by a set of local transition rules, we can implement GoI by token machines [14,20,10]. These implementations are free from the notion of substitution, and local transition rules specifies them and we have a graphical presentation of local transition rules. As emphasised in [14], GoI interpretation provides a semantic tool to verify graph reductions.

GoI has some applications to studies of programming languages and lambda calculi such as optimization of reductions [14], a compilation of call-by-name PCF into a low level language [20], full completeness for ML-types in the polymorphic lambda calculus [2] and implicit complexity theory [4]. Among these studies, the following questions are fundamental:

- Can we extend GoI to a semantics of programming languages?
- To what extent does GoI interpretation capture behaviour of programming languages? For example, is GoI adequate for the polymorphic linear functional programming language `lily` in [8]?

There are two approaches to the first question. In [12], Girard gave an algorithm to compute fixed points of linear functions, from which we can calculate fixed points for

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 320–334, 2011.

intuitionistic functions. The algorithm is similar to the computation of the least fixed points in domain theory. On the other hand, in the GoI interpretation of the call-by-name PCF [20], Mackie interpreted the fixed point operator by means of cyclic computation (Section 2 in [20]). This approach corresponds exactly to the coding of recursion in graph reduction, which is more efficient with respect to the use of memory, see [18]. However it is not discussed whether these interpretations are adequate or not.

Our aim in this paper is to give an adequate GoI interpretation of a linear functional programming language LFP which is essentially equal to `lily`. As we will observe, the standard GoI interpretation does not extend to an adequate interpretation of LFP. The main problem is that the standard GoI interpretations of LFP programs are not strict: for some LFP programs, their standard GoI interpretations do not evaluate their arguments. Therefore, so as to give an adequate GoI interpretation of LFP, we need to modify the standard GoI interpretation by imposing strictness on the interpretations of terms. In this paper, we give a modification of the standard GoI interpretation by extracting realizers from a realizability model of LFP.

Our main contributions are:

- We give a modified GoI interpretation of LFP in Mackie style: we interpret recursion by cyclic computation.
- We prove adequacy of the modified GoI interpretation.

This paper is organized as follows. In Section 2, We describe a linear functional programming language LFP. In Section 3, We recall the standard GoI interpretation and define modified GoI interpretation. In Section 4, we illustrate how we extract the modified GoI interpretation from a categorical realizability model. We capture a fixed point operator in the categorical model by means of the trace operator of the category of sets and partial maps which is given by cyclic computation. In Section 5, We prove adequacy of our modified GoI interpretation.

2 A Linear Functional Programming Language LFP and Its Operational Semantics

We describe the syntax of a linear functional programming language LFP. This language is essentially equal to `lily` in [8]. LFP is also a fragment of $\mathrm{PILL}_{\mathrm{Y}}$ in [9].

Type $\mathsf{A} := \mathsf{X} \mid \mathsf{A} \multimap \mathsf{A} \mid \,!\mathsf{A} \mid \forall \mathsf{X}.\mathsf{A}$

Term $\mathsf{M} := \mathsf{x} \mid \mathsf{M}\ \mathsf{M} \mid \lambda \mathsf{x}^{\mathsf{A}}.\mathsf{M} \mid \mathtt{let}\ !\mathsf{x}\ \mathsf{be}\ \mathsf{M}\ \mathtt{in}\ \mathsf{M} \mid \,!\mathsf{M} \mid \mathsf{M}\ \mathsf{A} \mid \Lambda \mathsf{X}.\mathsf{M} \mid \mu \mathsf{x}^{\mathsf{A}}.\mathsf{M}$

$$\frac{\Theta \vdash \Gamma, A}{\Theta \mid \Gamma; x : A \vdash x : A} \qquad \frac{\Theta \vdash \Gamma, A}{\Theta \mid \Gamma, x : A; - \vdash x : A} \qquad \frac{\Theta \mid \Gamma; \Delta, x : A \vdash M : B}{\Theta \mid \Gamma; \Delta \vdash \lambda x^{A}.M : A \multimap B}$$

$$\frac{\Theta \mid \Gamma; \Delta \vdash M : A \multimap B \quad \Theta \mid \Gamma; \Delta' \vdash N : A}{\Theta \mid \Gamma; \Delta \# \Delta' \vdash M\ N : B} \qquad \frac{\Theta \mid \Gamma, x : A; - \vdash M : A}{\Theta \mid \Gamma; - \vdash \mu x^{A}.M : A}$$

$$\frac{\Theta \mid \Gamma; - \vdash M : A}{\Theta \mid \Gamma; - \vdash !M : !A} \qquad \frac{\Theta \mid \Gamma, x : A; \Delta \vdash M : B \quad \Theta \mid \Gamma; \Delta' \vdash N : !A}{\Theta \mid \Gamma; \Delta \# \Delta' \vdash \mathtt{let}\ !x\ \mathsf{be}\ N\ \mathtt{in}\ M : B}$$

$$\frac{\Theta, X \mid \Gamma; \Delta \vdash M : A \quad \Theta \vdash \Gamma; \Delta}{\Theta \mid \Gamma; \Delta \vdash \Lambda X.M : \forall X.A} \qquad \frac{\Theta \mid \Gamma; \Delta \vdash M : \forall X.A \quad \Theta \vdash B}{\Theta \mid \Gamma; \Delta \vdash M\ B : A[B/X]}$$

where Θ is a finite sequence of type variables, and Γ and Δ are finite sequences of pairs of type variables and types. $\Delta \# \Delta'$ is a *merge* of Δ and Δ' [6]. We inductively define a relation "$\Delta \# \Delta'$ is a merge of finite lists Δ and Δ'" by (i) Δ is a merge of the empty sequence ε and Δ, (ii) Δ is a merge of Δ and ε, (iii) if Δ'' is a merge of Δ and Δ', then $\mathsf{x} : \mathsf{A}, \Delta''$ is a merge of $\mathsf{x} : \mathsf{A}, \Delta$ and Δ', (iv) if Δ'' is a merge of Δ and Δ', then $\mathsf{x} : \mathsf{A}, \Delta''$ is a merge of Δ and $\mathsf{x} : \mathsf{A}, \Delta'$. We write $|\Gamma|$ and $|\Delta|$ for the length of these sequences. $\Theta \vdash \mathsf{A}$ means that A is a well formed type under Θ, and $\Theta \vdash \Gamma; \Delta$ means that every type A in $\Gamma; \Delta$ satisfies $\Theta \vdash \mathsf{A}$. We write Ty for the set of closed types, and we write Term(A) for the set of closed terms for $\mathsf{A} \in$ Ty. When we write Term(A), we assume that A is a closed type without mentioning it.

Operational semantics for LFP is the standard call-by-name evaluation strategy. For more about operational semantics of LFP, see [8].

$$\mathsf{V} := \lambda \mathsf{x}^{\mathsf{A}}.\mathsf{M} \mid \Lambda \mathsf{X}.\mathsf{M} \mid !\mathsf{M} \qquad \frac{}{\mathsf{V} \Downarrow \mathsf{V}} \quad \frac{\mathsf{M} \Downarrow \lambda \mathsf{x}^{\mathsf{A}}.\mathsf{L} \quad \mathsf{L}[\mathsf{N}/\mathsf{x}] \Downarrow \mathsf{V}}{\mathsf{M}\,\mathsf{N} \Downarrow \mathsf{V}} \quad \frac{\mathsf{M}[\mu \mathsf{x}^{\mathsf{A}}.\mathsf{M}/\mathsf{x}] \Downarrow \mathsf{V}}{\mu \mathsf{x}^{\mathsf{A}}.\mathsf{M} \Downarrow \mathsf{V}}$$

$$\frac{\mathsf{M} \Downarrow !\mathsf{L} \quad \mathsf{N}[\mathsf{L}/\mathsf{x}] \Downarrow \mathsf{V}}{\mathtt{let}\ !\mathsf{x}\ \mathtt{be}\ \mathsf{M}\ \mathtt{in}\ \mathsf{N} \Downarrow \mathsf{V}} \quad \frac{\mathsf{M} \Downarrow \Lambda \mathsf{X}.\mathsf{N} \quad \mathsf{N}[\mathsf{A}/\mathsf{X}] \Downarrow \mathsf{V}}{\mathsf{M}\,\mathsf{A} \Downarrow \mathsf{V}}$$

When $\mathsf{M} \Downarrow \mathsf{V}$ is derivable, we write $\mathsf{M} \Downarrow$. Otherwise, we write $\mathsf{M} \Uparrow$.

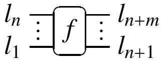
We informally define soundness and adequacy. We define adequacy by termination at !-types. As noted in [8], if we add a type $\mathbb{B}$ of Boolean values then adequacy defined for termination at !-types is equivalent to adequacy defined for termination at $\mathbb{B}$. In Section 3.3, we formally define these notions for the modified GoI interpretation.

Definition 1. *An* interpretation *of* LFP *is a map* $[\![-]\!]$ *that assigns a mathematical object to each term* M*. We suppose that there is a distinguished object* $\perp$ *among the mathematical objects. We say that the interpretation is* sound *when* $\mathsf{M} \Downarrow \mathsf{V} \implies [\![\mathsf{M}]\!] = [\![\mathsf{V}]\!]$ *for every closed term* M*. We say that the interpretation is* adequate *when the interpretation is sound and* $[\![\mathsf{M}]\!] = \perp \iff \mathsf{M} \Uparrow$ *for every closed type* A *and a term* M *in* Term(!A).

3 GoI Interpretation of LFP

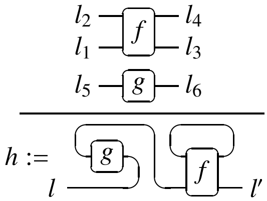
3.1 Graphical Presentation of Partial Maps

In this section, we give graphical presentation of partial maps, which correspond precisely to string diagrams in the category **Int**(**Pfn**) (see Section 4.3). Let $\mathbb{N}$ be the set of natural numbers. We present a partial map $f : \{l_1, \cdots, l_{n+m}\} \times \mathbb{N} \rightharpoonup \{l_1, \cdots, l_{n+m}\} \times \mathbb{N}$ as a diagram with labeled ports:



When the number of ports on the left hand side is n and the number of ports on the right hand side is m as above, we write $f \in \mathcal{D}(n, m)$. By the definition of $\mathcal{D}(n, m)$, we have $\mathcal{D}(n + 1, m) = \mathcal{D}(n, m + 1)$ and so on. This equation corresponds to a rearrangement of labeled ports of a diagram. For labels l_i and l_j, we write $f(l_i, n) \simeq (l_j, m)$ or $(l_i, n) \overset{f}{\mapsto} (l_j, m)$ when $f(l_i, n)$ is defined and is equal to (l_j, m). If we do not need to write labels, especially when a diagram has at most one port, we do not write labels. As a special partial map, we write $\emptyset_{n,m} \in \mathcal{D}(n, m)$ for the partial map whose value is always undefined. When we can infer n and m from context, we omit them.

By combining diagrams, we can construct new partial maps. For example, h given in the following presents a partial map from $\{l, l'\} \times \mathbb{N}$ to $\{l, l'\} \times \mathbb{N}$. The value of h for (l', n) is calculated by the algorithm on the left hand side:

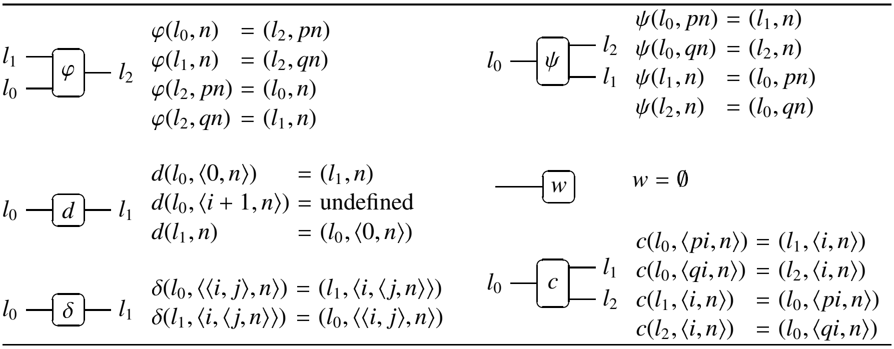


1. Input n to l_3.
2. If we get an output m from l_1 then input m to l_5.
3. If we get an output m from l_2 then input m to l_4.
4. If we get an output m from l_3 then h outputs m from l.
5. If we get an output m from l_4 then input m to l_2.
6. If we get an output m from l_5 then input m to l_1.
7. If we get an output m from l_6 then h outputs m from l.

If there is an infinite loop or no output, then $h(l', n)$ is undefined. The value of $h(l, n)$ is calculated by the same algorithm. Similar argument appears in the composition of strategies in game semantics called "parallel composition + hiding".

3.2 The Standard GoI Interpretation

For LFP without recursion, we give the standard GoI interpretation, which is a restriction of GoI interpretation of classical linear logic to intuitionistic linear logic. For the GoI interpretation of classical linear logic, see [16].



$$\begin{aligned} \varphi(l_0, n) &= (l_2, pn) \\ \varphi(l_1, n) &= (l_2, qn) \\ \varphi(l_2, pn) &= (l_0, n) \\ \varphi(l_2, qn) &= (l_1, n) \end{aligned} \qquad \begin{aligned} \psi(l_0, pn) &= (l_1, n) \\ \psi(l_0, qn) &= (l_2, n) \\ \psi(l_1, n) &= (l_0, pn) \\ \psi(l_2, n) &= (l_0, qn) \end{aligned}$$

$$\begin{aligned} d(l_0, \langle 0, n\rangle) &= (l_1, n) \\ d(l_0, \langle i+1, n\rangle) &= \text{undefined} \\ d(l_1, n) &= (l_0, \langle 0, n\rangle) \end{aligned} \qquad w = \emptyset$$

$$\begin{aligned} \delta(l_0, \langle\langle i, j\rangle, n\rangle) &= (l_1, \langle i, \langle j, n\rangle\rangle) \\ \delta(l_1, \langle i, \langle j, n\rangle\rangle) &= (l_0, \langle\langle i, j\rangle, n\rangle) \end{aligned} \qquad \begin{aligned} c(l_0, \langle pi, n\rangle) &= (l_1, \langle i, n\rangle) \\ c(l_0, \langle qi, n\rangle) &= (l_2, \langle i, n\rangle) \\ c(l_1, \langle i, n\rangle) &= (l_0, \langle pi, n\rangle) \\ c(l_2, \langle i, n\rangle) &= (l_0, \langle qi, n\rangle) \end{aligned}$$

Fig. 1. Components for the standard GoI iterpretation

In the following, we fix two bijections: $\alpha : \mathbb{N} + \mathbb{N} \to \mathbb{N}$ and $\langle -, - \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. We define maps $p, q : \mathbb{N} \to \mathbb{N}$ by $\alpha \circ \mathrm{inl}$ and $\alpha \circ \mathrm{inr}$ respectively. For the purpose of this paper, Any infinite set is sufficient, and the choice is a matter of taste. Generalisation of GoI interpretation to Figure 1 is the list of components for the standard GoI interpretation. We use maps p, q and $\langle -, - \rangle$ for tagging. They tell where a natural number came from. An output pn at l_2 of φ means that the input was a number n coming from l_0, and an

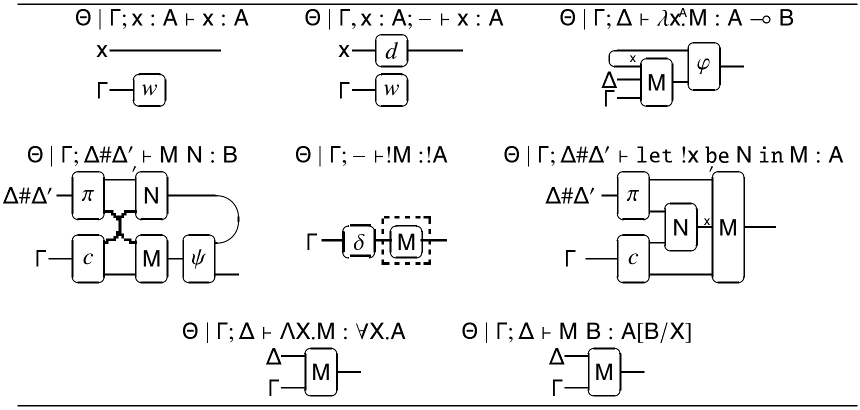
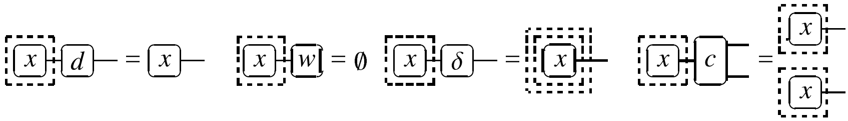
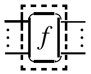


Fig. 2. The standard GoI interpretation of LFP without recursion

output qn at l_2 means that the input was a number n coming from l_1. A number $\langle i, n\rangle$ means the i-th copy of n. By the definition, ψ is a right inverse of φ, which corresponds to the tensor cell of proof nets [13]. Components d, δ, w and c correspond to dereliction $!A \multimap A$, comultiplication $!A \multimap !!A$, weakening $!A \multimap$ I and contraction $!A \multimap !A \otimes !A$ of linear logic respectively. In fact, we have



where we define dotted line box, which corresponds to ! of linear logic, as follows.

$Tf :=$  $(l, \langle i, n\rangle) \overset{Tf}{\mapsto} (l', \langle j, m\rangle) \overset{\Delta}{\iff} (l, n) \overset{f}{\mapsto} (l', m) \wedge i = j$

In Figure 2, we define the standard GoI interpretation of LFP without recursion. We interpret each term $\Theta \mid \Gamma; \Delta \vdash \mathsf{M} : \mathsf{A}$ by an element in $\mathcal{D}(|\Gamma| + |\Delta|, 1)$. On the left hand side of the interpretation of M, the i-th port counted from the bottom corresponds to the i-th variable in Γ for $1 \leq i \leq |\Gamma|$, and the $|\Gamma| + j$-th port corresponds to the j-th variable in Δ for $1 \leq j \leq |\Delta|$. In the definition, πs are the appropriate permutations, and we write diagrams as if $|\Gamma|$ and $|\Delta|$ were 1. We can infer precise definitions from Figure 2.

We can inductively show soundness of the standard GoI interpretation. However, the standard GoI interpretation identifies certain terms that ought to be distinguished: the interpretation of `let` !x be M `in` $!(\lambda \mathsf{x}^{\mathsf{A}}.\mathsf{x})$ for a term $\mathsf{M} \in$ Term(!A) is equal to the interpretation of $!(\lambda \mathsf{x}^{\mathsf{A}}.\mathsf{x})$. Because of this equality, the standard GoI interpretation does not extend to an adequate interpretation of the whole LFP. In fact, we have $!(\lambda \mathsf{x}^{\mathsf{A}}.\mathsf{x}) \Downarrow$ as well as `let` !x be $\mu \mathsf{x}^{\mathsf{A}}.\mathsf{x}$ `in` $!(\lambda \mathsf{x}^{\mathsf{A}}.\mathsf{x}) \Uparrow$. The reason of this undesirable equality is that the interpretation of a term is not strict on its arguments. This means that the interpretation of a term does not necessarily evaluate its free variables. If a term $\Theta \mid \Gamma, \mathsf{x} : \mathsf{A}; \Delta \vdash \mathsf{M} : \mathsf{B}$ does not have free x, then the GoI interpretation $[\mathsf{M}]$ of M ignores x, i.e. the port on the right hand side of $[\mathsf{M}]$ has no path to the port for x on the left hand side of $[\mathsf{M}]$.

3.3 Modified GoI Interpretation

As noted at the end of previous section, the standard GoI interpretation does not extend to an adequate interpretation of LFP. In this section, in order to obtain adequacy, we modify the standard GoI interpretation and extend it to the whole language. Idea of our modification is to impose strictness on the interpretations of terms.

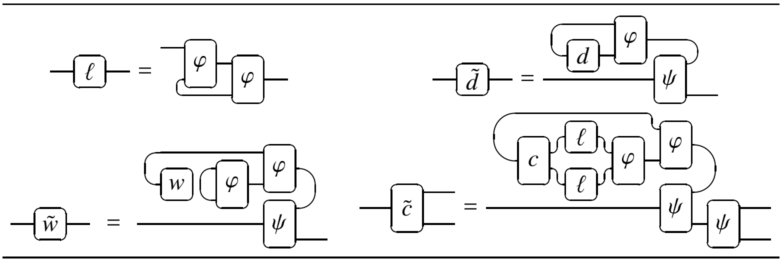
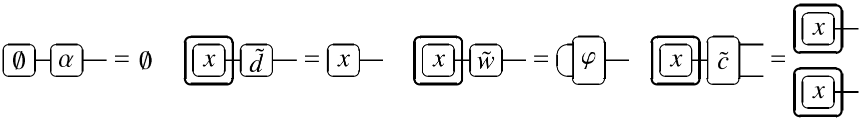


Fig. 3. Basic components for the modified GoI interpretation of LFP

Figure 3 is a list of components for the modified GoI interpretation. $\tilde{d}, \tilde{w}$ and $\tilde{c}$ correspond to d, w and c of the standard GoI interpretation. Note that $\tilde{w} \in \mathcal{D}(1, 1)$ whereas $w \in \mathcal{D}(1, 0)$. Informally, we "lift" elements $x \in \mathcal{D}(0, 1)$ by concatenating ℓ and x. Let α be one of $\tilde{d}, \tilde{w}$ and $\tilde{c}$. Then the concatenation of α and $\emptyset$ is equal to $\emptyset$, and the concatenation of α and a lifting of Tx for $x \in \mathcal{D}(0, 1)$ is equal to the following:



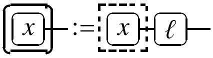
where . So as to show the equalities, readers are not required to compute the partial functions represented by the above diagrams. We can show the above 4 equalities by means of graph rewriting using the equations noted in Section 3.2.

Figure 4 is the definition of the modified GoI interpretation of LFP. As for the standard GoI interpretation, we interpret a term $\Theta \mid \Gamma; \Delta \vdash \mathsf{M} : \mathsf{A}$ by a partial map in $\mathcal{D}(|\Gamma| + |\Delta|, 1)$, and cells named π in Figure 4 are the appropriate permutations. Note that cyclic computation appears in the interpretation of $\mu\mathsf{x}^{\mathsf{A}}.\mathsf{M}$ which starts from c in front of M going back to δ. We write the modified GoI interpretation of M by $[\![\mathsf{M}]\!]$. The interpretation $[\![\mathsf{M}]\!]$ induces a map from $\mathcal{D}(0, 1)^{|\Gamma;\Delta|}$ to $\mathcal{D}(0, 1)$, which are strict on their arguments: the value of the map at $(\boldsymbol{x}, \boldsymbol{y})$ is $\emptyset$ when each component of $\boldsymbol{x}$ is $\emptyset$ or surrounded by a thick line box, and at least one of $\boldsymbol{x}$ is $\emptyset$.

We formally define soundness and adequacy for the modified GoI interpretation.

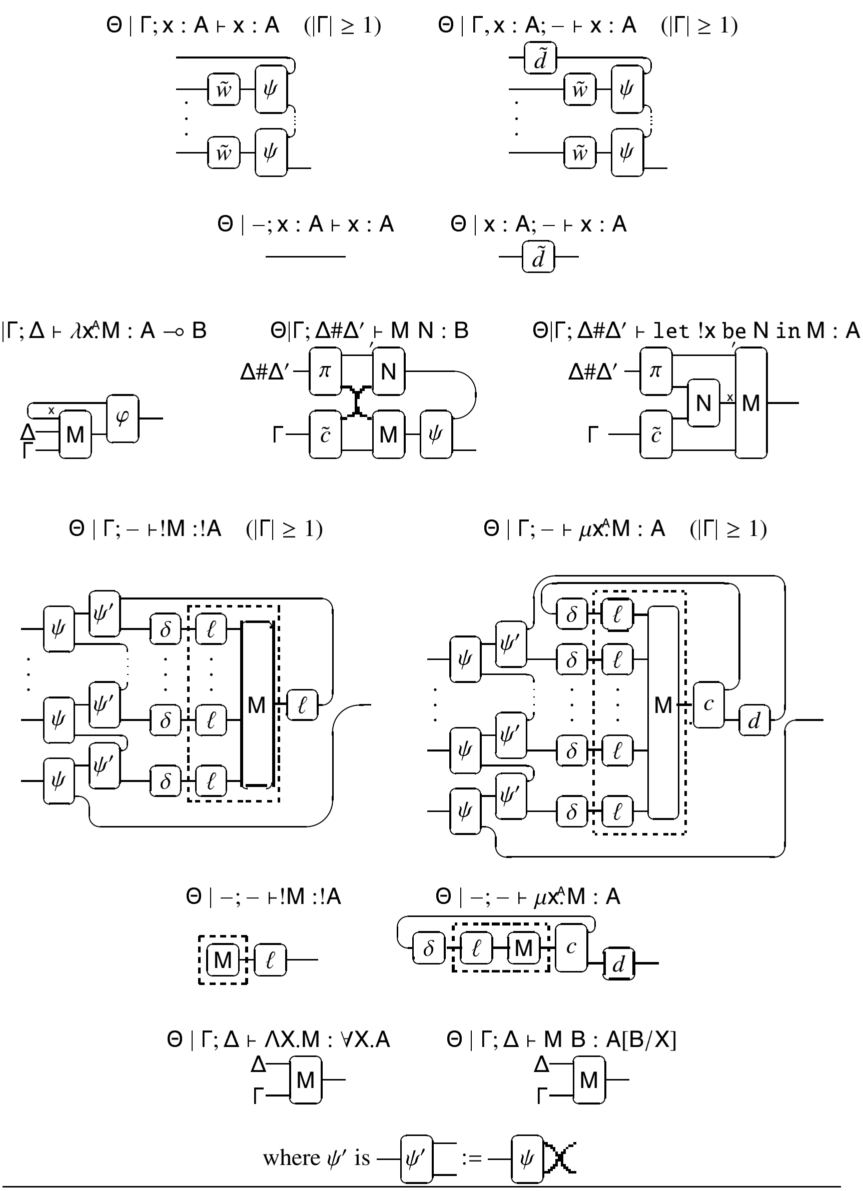


Fig. 4. Modified GoI interpretation of LFP

Definition 2. *We say that the modified GoI interpretation is* sound *when* $\mathsf{M}\Downarrow\mathsf{V}$ *implies* $[\![\mathsf{M}]\!] = [\![\mathsf{V}]\!]$ *for every closed term* M. *We say that the modified GoI interpretation is* adequate *when the interpretation is sound and* $[\![\mathsf{M}]\!] = \emptyset \iff \mathsf{M}\Uparrow$ *for every closed type* A *and a term* M *in* Term(!A).

Theorem 1. *The modified GoI interpretation* $[\![-]\!]$ *is adequate.*

Proof. We will prove this theorem in Section 5.

4 Deriving the Modification from a Realizability Interpretation

Before showing the adequacy of the modified GoI interpretation, we sketch how we derived the modification. Our technique stems from a realizability interpretation. We construct a realizability model of LFP from the ω-cpo $\mathscr{D}(0,1)$. We interpret each LFP term as an equivalence class of a partial equivalence relation (per) on $\mathscr{D}(0,1)$. We can derive our modified GoI interpretation by inductively extracting elements from the equivalence classes. For lack of space, we just outline the construction of the realizability model and illustrate several extractions of the modified GoI interpretation.

4.1 Interpretation of the Untyped Linear Lambda Calculus

As a preparation, we give a syntax for diagrams representing partial maps. We add the tensor to the untyped linear lambda calculus given in [22]. We need the tensor to describe the monoidal product, which is a fundamental structure of linear categories, of the realizability model that we are going to construct.

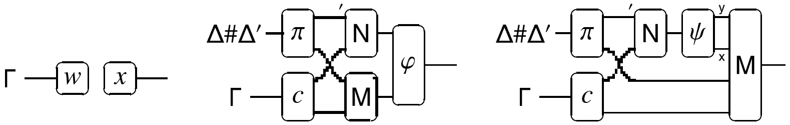
$$\mathsf{M} := \mathsf{x} \mid \mathsf{k}_x \mid \lambda\mathsf{x}.\mathsf{M} \mid !\mathsf{M} \mid \mathsf{M}\;\mathsf{M} \mid \mathtt{let}\ !\mathsf{x}\ \mathtt{be}\ \mathsf{M}\ \mathtt{in}\ \mathsf{M} \mid \mathsf{M}\otimes\mathsf{M} \mid \mathtt{let}\ \mathsf{x}\otimes\mathsf{y}\ \mathtt{be}\ \mathsf{M}\ \mathtt{in}\ \mathsf{M}$$

$$\frac{}{\Gamma;\mathsf{x}\vdash\mathsf{x}}\qquad\frac{}{\Gamma,\mathsf{x};-\vdash\mathsf{x}}\qquad\frac{}{\Gamma;-\vdash\mathsf{k}_x}\qquad\frac{\Gamma;\Delta\vdash\mathsf{M}\quad\Gamma;\Delta'\vdash\mathsf{N}}{\Gamma;\Delta\#\Delta'\vdash\mathsf{M}\;\mathsf{N}}\qquad\frac{\Gamma;\Delta,\mathsf{x}\vdash\mathsf{M}}{\Gamma;\Delta\vdash\lambda\mathsf{x}.\mathsf{M}}\qquad\frac{\Gamma;-\vdash\mathsf{M}}{\Gamma;-\vdash!\mathsf{M}}$$

$$\frac{\Gamma,\mathsf{x};\Delta\vdash\mathsf{M}\quad\Gamma;\Delta'\vdash\mathsf{N}}{\Gamma;\Delta\#\varDelta'\vdash\mathtt{let}\ !\mathsf{x}\ \mathtt{be}\ \mathsf{N}\ \mathtt{in}\ \mathsf{M}}\qquad\frac{\Gamma;\Delta\vdash\mathsf{M}\quad\Gamma;\Delta'\vdash\mathsf{N}}{\Gamma;\Delta\#\Delta'\vdash\mathsf{M}\otimes\mathsf{N}}\qquad\frac{\Gamma;\Delta,\mathsf{x},\mathsf{y}\vdash\mathsf{M}\quad\Gamma;\Delta'\vdash\mathsf{N}}{\Gamma;\Delta\#\Delta'\vdash\mathtt{let}\ \mathsf{x}\otimes\mathsf{y}\ \mathtt{be}\ \mathsf{N}\ \mathtt{in}\ \mathsf{M}}$$

where k_x is a constant symbol, and x runs through $\mathscr{D}(0,1)$. We assign each term to a partial map. For k_x, $\mathsf{M}\otimes\mathsf{N}$ and $\mathtt{let}\ \mathsf{x}\otimes\mathsf{y}\ \mathtt{be}\ \mathsf{N}\ \mathtt{in}\ \mathsf{M}$, assignments are as follows:

$\Gamma;-\vdash\mathsf{k}_x$ $\qquad\Gamma;\Delta\#\Delta'\vdash\mathsf{M}\otimes\mathsf{N}$ $\qquad\Gamma;\Delta\#\Delta'\vdash\mathtt{let}\ \mathsf{x}\otimes\mathsf{y}\ \mathtt{be}\ \mathsf{N}\ \mathtt{in}\ \mathsf{M}$



For other terms, we can associate partial maps as in Figure 2. For example, the partial map associated to $-;\mathsf{x}\vdash\lambda\mathsf{k}.\mathsf{k}\;\mathsf{x}$ is equal to ℓ in Figure 3. We identify a term and its associated partial map. There are expected equations.

Proposition 1. *As partial maps, the following equations hold.*

$$\begin{array}{lll}(\lambda\mathsf{x}.\mathsf{M})\;\mathsf{N} & = \mathsf{M}[\mathsf{N}/\mathsf{x}] & \\ \mathtt{let}\ !\mathsf{x}\ \mathtt{be}\ !\mathsf{M}\ \mathtt{in}\ \mathsf{P} & = \mathsf{P}[\mathsf{M}/\mathsf{x}] & (\mathsf{M}\ \textit{is closed}) \\ \mathtt{let}\ \mathsf{x}\otimes\mathsf{y}\ \mathtt{be}\ \mathsf{M}\otimes\mathsf{N}\ \mathtt{in}\ \mathsf{P} & = \mathsf{P}[\mathsf{M}/\mathsf{x},\mathsf{N}/\mathsf{y}] & (\mathsf{M}\ \textit{and}\ \mathsf{N}\ \textit{are closed})\end{array}$$

4.2 Admissible Pers and Strict Morphisms

We regard $\mathcal{D}(0,1)$ as an ω-cpo by the inclusion order of graph relations of partial maps. The empty map $\emptyset$ is the least element in $\mathcal{D}(0,1)$. We construct a realizability model on the ω-cpo $\mathcal{D}(0,1)$. Like in [9], we consider admissible pers on $\mathcal{D}(0,1)$ which is a subclass of partial equivalence relations.

Definition 3. *For a pointed ω-cpo D, an* admissible per *R on D is a per on D such that $(\bot,\bot) \in R$, and R is closed under least upper bounds (lub) of ω-chains, i.e. for any ascending chain $(x_1,y_1) \le (x_2,y_2) \le \cdots$ in R, the lub $(\bigvee_i x_i, \bigvee_i y_i)$ is also in R. We write $|R|$ for $\{x \mid (x,x) \in R\}$. For $x \in |R|$, we write $[x]_R$ for the equivalence class of x.*

In the following, we simply write x for a closed term k_x. For example, for $x, y \in \mathcal{D}(0,1)$, $x\ y$ means $\mathsf{k}_x\ \mathsf{k}_y$. We use italics for elements in $\mathcal{D}(0,1)$ and sans serifs for syntactic variables of the untyped linear lambda calculus.

Definition 4. *We define a category* **Adm***. Objects are admissible pers on $\mathcal{D}(0,1)$. For objects X and Y, we define $\sim_{X,Y}$ to be a partial equivalence relation on $\mathcal{D}(0,1)$ such that $r \sim_{X,Y} s$ if and only if $\forall (x,x') \in X.\ (r\ x, s\ x') \in Y$. A morphism $\boldsymbol{f} : X \to Y$ is an equivalence class of $\sim_{X,Y}$. We call a representative r of $\boldsymbol{f}$ a* realizer of $\boldsymbol{f}$*, and we say that r* realizes $\boldsymbol{f}$*. When $\boldsymbol{f}[\emptyset]_X = [\emptyset]_Y$, we say a morphism $\boldsymbol{f} : X \to Y$ is* strict*. We define* $\mathbf{Adm}_\bot$ *to be a subcategory of* **Adm** *consisting of* **Adm***-objects and strict* **Adm***-morphisms. We use bold face for morphisms in* **Adm** *and* $\mathbf{Adm}_\bot$*.*

Adm as well as $\mathbf{Adm}_\bot$ is a model of intuitionistic linear logic called *linear category*: A linear category is a symmetric monoidal closed category with a comonad called *linear exponential comonad* [5,7]. An intuition of **Adm** is the category of ωcpos and continuous maps, and an intuition of $\mathbf{Adm}_\bot$ is the category of pointed ωcpos and strict continuous maps. We list the unit object, the linear implication and the linear exponential comonad of **Adm** and $\mathbf{Adm}_\bot$ respectively.

$$
\begin{array}{lrl}
\mathbf{Adm} & \mathrm{I} & = \{(\emptyset,\emptyset)\} \\
 & X \otimes Y & = \{(x \otimes y, x' \otimes y') \mid (x,x') \in X, (y,y') \in Y\} \\
 & X \multimap Y & = \{(r,s) \mid \forall (x,x') \in X.\ (r\ x, s\ x') \in Y\} \\
 & !X & = \{(!x, !x') \mid (x,x') \in X\}
\end{array}
$$

$$
\begin{array}{lrl}
\mathbf{Adm}_\bot & \dot{\mathrm{I}} & = \{(\emptyset,\emptyset)\} \cup \{(\lambda \mathsf{x}.\mathsf{x}, \lambda \mathsf{x}.\mathsf{x})\} \\
 & X \dot{\otimes} Y & = \{(x \otimes y, x' \otimes y') \mid (x,x') \in X, (y,y') \in Y\} \cup \\
 & & \quad \left\{ (x \otimes y, x' \otimes y') \,\middle|\, \begin{array}{l} (x, x' \in |X|) \wedge (y, y' \in |Y|) \wedge \\ (x \in [\emptyset]_X \vee y \in [\emptyset]_Y) \wedge (x' \in [\emptyset]_X \vee y' \in [\emptyset]_Y) \end{array} \right\} \\
 & X \dot{\multimap} Y & = \{(r,s) \mid \forall (x,x') \in X.\ (r\ x, s\ x') \in Y \wedge r\ \emptyset \in [\emptyset]_Y\} \\
 & \dot{!}X & = \{(\emptyset,\emptyset)\} \cup \{(\lambda \mathsf{k}.\mathsf{k}\ !x, \lambda \mathsf{k}.\mathsf{k}\ !x') \mid (x,x') \in X\}
\end{array}
$$

It is not hard to show that both **Adm** and $\mathbf{Adm}_\bot$ provide models of the polymorphic linear lambda calculus whose linear exponential comonad ! is idempotent, i.e. !! is isomorphic to !. Readers can consult [2,9]. In [9], Birkedal et al. showed that $\mathbf{Adm}_\bot$ on a certain domain provides a model of the polymorphic linear lambda calculus with recursion, and they discussed relationship between **Adm** and $\mathbf{Adm}_\bot$. We can find a similar relationship for **Adm** and $\mathbf{Adm}_\bot$ on $\mathcal{D}(0,1)$ (Lemma 1).

Extraction of realizers. We illustrate extraction of realizers of the dereliction $\dot{\boldsymbol{d}} : \dot{!}X \to X$ and the weakening $\dot{\boldsymbol{w}} : \dot{!}X \to \dot{\mathrm{I}}$ of $\mathbf{Adm}_\perp$. They are given by the following:

$$\dot{\boldsymbol{d}}[\emptyset]_{\dot{!}X} = [\emptyset]_X \quad \dot{\boldsymbol{d}}[\lambda\mathsf{k}.\mathsf{k}\ !x]_{\dot{!}X} = [x]_X \quad \dot{\boldsymbol{w}}[\emptyset]_{\dot{!}X} = [\emptyset]_{\mathrm{I}} \quad \dot{\boldsymbol{w}}[\lambda\mathsf{k}.\mathsf{k}\ !x]_{\dot{!}X} = [\lambda\mathsf{x}.\mathsf{x}]_{\mathrm{I}}\,.$$

$\dot{\boldsymbol{d}}$ and $\dot{\boldsymbol{w}}$ are realized by $\dot{d}$ and $\dot{w}$ in the following:

$$\dot{d} := \lambda\mathsf{x}.\mathsf{x}\ (\lambda\mathsf{x}'.\mathtt{let}\ !\mathsf{y}\ \mathsf{be}\ \mathsf{x}'\ \mathtt{in}\ \mathsf{y}) \qquad \dot{w} := \lambda\mathsf{x}.\mathsf{x}\ (\lambda\mathsf{x}'.\mathtt{let}\ !\mathsf{y}\ \mathsf{be}\ \mathsf{x}'\ \mathtt{in}\ \lambda\mathsf{z}.\mathsf{z})$$

By uncurrying $\dot{d}$, we obtain $\mathsf{x}\ (\lambda\mathsf{x}'.\mathtt{let}\ !\mathsf{y}\ \mathsf{be}\ \mathsf{x}'\ \mathtt{in}\ \mathsf{y})$, whose corresponding diagram is exactly $\tilde{d}$ in Figure 3. Similarly, we obtain $\tilde{w}$ in Figure 3 by uncurrying $\dot{w}$. Extraction of $\tilde{\delta}$ and the modified GoI interpretation of $!\mathsf{M}$ are more complicated. Still, we can extract realizers by elementary calculation. First, we give a realizer of these terms in a form of untyped terms, then we write down the corresponding diagrams. In the end, by ad hoc optimization, we obtain simpler realizers of these terms which are the modified GoI interpretations of them.

It is not automatic to find $\dot{d}$ and $\dot{w}$. However, if you write down a proof of $\mathbf{Adm}_\perp$ being a linear category, then you must have given an algorithm constructing realizers of $\dot{\boldsymbol{d}}$ and $\dot{\boldsymbol{w}}$ (unless you did not use excluded middle in the proof). Then, you can extract realizers $\dot{d}$ and $\dot{w}$ by the algorithm.

4.3 Interpretation of Recursion in Mackie Style

We give an interpretation of recursion in $\mathbf{Adm}_\perp$ and illustrate extraction of a realizer of $\mu\mathsf{x}^A.\mathsf{M}$ in Mackie style. At the level of realizers, we interpret recursion by means of the trace operator in $\mathbf{Int}(\mathbf{Pfn})$ (See the following definitions). Before that, we briefly recall several necessary notions: traced symmetric monoidal categories, compact closed categories and $\mathbf{Int}$-construction. For further details, see [19,17].

Definition 5. *A symmetric monoidal category* $\mathbb{C}$ *is* traced *when* $\mathbb{C}$ *has an operator* $\mathrm{tr}^A_{X,Y} : \mathbb{C}(X \otimes A, Y \otimes A) \to \mathbb{C}(X, Y)$ *subject to*

- $\mathrm{tr}^A_{X',Y'}((k \otimes \mathrm{id}_A) \circ f \circ (h \otimes \mathrm{id}_A)) = k \circ \mathrm{tr}^A_{X,Y}(f) \circ h$
- $\mathrm{tr}^X_{X,X}(\sigma_{X,X}) = \mathrm{id}_X$
- $\mathrm{tr}^A_{Z\otimes X,Z\otimes Y}(\mathrm{id}_Z \otimes f) = \mathrm{id}_Z \otimes \mathrm{tr}^A_{X,Y}(f)$
- $\mathrm{tr}^A_{X,Y}(\mathrm{tr}^B_{X\otimes A,Y\otimes A}(f)) = \mathrm{tr}^B_{X,Y}(\mathrm{tr}^A_{X\otimes B,Y\otimes B}((\mathrm{id}_Y \otimes \sigma_{A,B}) \circ f \circ (\mathrm{id}_X \otimes \sigma_{B,A})))$

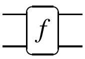
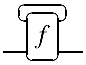
Definition 6. *A* compact closed category $\mathbb{C}$ *is a symmetric monoidal category with a function* $* : ob(\mathbb{C}) \to ob(\mathbb{C})$ *and morphisms* $\epsilon_X : X^* \otimes X \to \mathrm{I}$ *and* $\eta_X : \mathrm{I} \to X \otimes X^*$ *satisfying* $(X \otimes \epsilon_X) \circ \alpha \circ (\eta_X \otimes X) = \mathrm{id}_X$ *and* $(\epsilon_X \otimes X^*) \circ \alpha^{-1} \circ (X^* \otimes \eta_X) = \mathrm{id}_{X^*}$ *where* $\alpha_{X,Y,Z} : (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z)$ *is the coherence isomorphism.*

Definition 7. *Let* $\mathbb{C}$ *be a traced symmetric monoidal category* $\mathbb{C}$*. We define a category* $\mathbf{Int}(\mathbb{C})$*. Objects are pairs* (X_+, X_-) *of* $\mathbb{C}$*-objects, and a morphism* $f : (X_+, X_-) \to (Y_+, Y_-)$ *is a* $\mathbb{C}$*-morphism* $f : X_+ \otimes Y_- \to Y_+ \otimes X_-$*.*

As is known, every compact closed category has a canonical structure of a traced symmetric monoidal category. On the other hand, for a traced symmetric monoidal category $\mathbb{C}$, $\mathbf{Int}(\mathbb{C})$ is a compact closed category whose monoidal product $(X_+, X_-) \otimes (Y_+, Y_-)$ is $(X_+ \otimes Y_+, X_- \otimes Y_-)$.

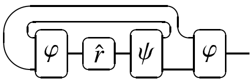
The following theorem shown in [17,21] relates a trace operator on a linear category to an interpretation of recursion.

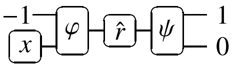
Theorem 2. *Let $\mathbb{C}$ be a linear category. If $\mathbb{C}$ is traced, and the linear exponential comonad ! is idempotent, then $\mathbb{C}$ has an operator $(-)^\dagger : \mathbb{C}(!A \otimes !X, X) \to \mathbb{C}(!A, X)$ that satisfies $f = f \circ (!A \otimes (!f^\dagger \circ \delta)) \circ c$ for every $f : !A \otimes !X \to X$ where c is the contraction $!A \to !A \otimes !A$ of $\mathbb{C}$, and δ is the comultiplication $!X \to !!X$ of $\mathbb{C}$.*

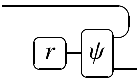
In this paper, we are interested in the traced symmetric monoidal category **Pfn** of sets and partial maps and the compact closed category **Int**(**Pfn**). The monoidal product of **Pfn** is the set theoretic coproduct (i.e. disjoint union), and the trace operator of **Pfn** is given by cyclic computation, see [1]. Diagrams in this paper are exactly string diagrams in **Int**(**Pfn**), and $\mathbf{Int}(\mathbf{Pfn})((\mathbb{N}, \mathbb{N})^{\otimes n}, (\mathbb{N}, \mathbb{N})^{\otimes m})$ is equal to $\mathcal{D}(n, m)$ where $(\mathbb{N}, \mathbb{N})^{\otimes n}$ is the n-fold monoidal product of $(\mathbb{N}, \mathbb{N})$. As noted in the above, **Int**(**Pfn**) is also a traced monoidal category. Its trace operator $\mathrm{tr}^{(\mathbb{N},\mathbb{N})}_{(\mathbb{N},\mathbb{N}),(\mathbb{N},\mathbb{N})}$ is depicted as  ↦  . Observe that $\mathrm{tr}^{(\mathbb{N},\mathbb{N})}_{(\mathbb{N},\mathbb{N}),(\mathbb{N},\mathbb{N})}(f)$ is calculated by cyclic computation.

Proposition 2. **Adm** *is a traced symmetric monoidal category.*

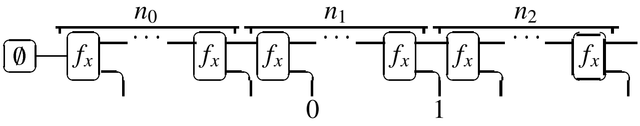
Proof. Let r be a realizer of $\boldsymbol{f} : X \otimes A \to Y \otimes A$. We define $\mathrm{tr}(r) \in \mathcal{D}(0, 1)$ by the diagram (i) below. We show that $\mathrm{tr}(r)$ realizes a morphism from X to Y.

(i) 

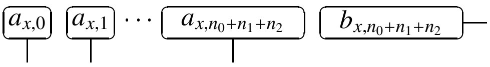
(ii) 

where $\hat{r} =$ 

Note that the trace operator $\mathrm{tr}^{(\mathbb{N},\mathbb{N})}_{(\mathbb{N},\mathbb{N}),(\mathbb{N},\mathbb{N})}$ of **Int**(**Pfn**) appears in $\mathrm{tr}(r)$. For $x \in \mathcal{D}(0, 1)$, let $f_x \in \mathcal{D}(1, 2)$ be the diagram (ii) in the above. Then, $n \overset{\mathrm{tr}(r)\ x}{\mapsto} m$ if and only if either $(0, n) \overset{f_x}{\mapsto} (0, m)$, or there exists $k \geq 0$ and $i_0, \cdots, i_k \in \{-1, 1\}$ such that $(0, n) \overset{f_x}{\mapsto} (i_0, x_0)$, $(-i_p, x_p) \overset{f_x}{\mapsto} (i_{p+1}, x_{p+1})$ for $0 \leq p \leq k - 1$ and $(-i_k, x_k) \overset{f_x}{\mapsto} (0, m)$. This is equivalent to the existence of $n_0, n_2 \geq 0$, $n_1 \geq 1$ and $i, j \in \{0, 1\}$ such that the following partial map g_{x,n_0,n_1,n_2} sends (i, n) to (j, m).

$g_{x,n_0,n_1,n_2} =$ 

Since r is a realizer of $\boldsymbol{f} : X \otimes A \to Y \otimes A$, the partial map g_{x,n_0,n_1,n_2} is equal to



for some $a_{x,n} \in |Y|$ and $b_{x,n} \in |A|$. Note that we have $a_{x,0} \le a_{x,1} \le \cdots$. Therefore,

$$\begin{aligned} n \xmapsto{\mathrm{tr}(r)\ x} m &\iff \exists n_0, n_1, n_2 \ge 0.\ \exists i, j \in \{0,1\}.\ g_{x,n_0,n_1,n_2}(i,n) \simeq (j,m) \\ &\iff \exists k \ge 0.\ a_{x,k}(n) \simeq m \\ &\iff (\textstyle\bigvee_{k\ge 0} a_{x,k})(n) \simeq m \end{aligned}$$

By induction on a natural number n, we can show that if $(x,x') \in X$ then $(a_{x,n}, a_{x',n}) \in Y$ and $(b_{x,n}, b_{x',n}) \in A$. Therefore, $(\mathrm{tr}(r)\ x, \mathrm{tr}(r)\ x') = (\bigvee_{n\ge 0} a_{x,n}, \bigvee_{n\ge 0} a_{x',n}) \in Y$, i.e. $\mathrm{tr}(r)$ realizes a morphism from X to Y. Let s be another realizer of $\boldsymbol{f}$, and we define $a'_{x,n}$ and $b'_{x,n}$ in $\mathcal{D}(0,1)$ similarly. Then, we can inductively show that $(a_{x,n}, a'_{x,n}) \in Y$ and $(b_{x,n}, b'_{x,n}) \in A$ for each $x \in |X|$ and $n \ge 0$. Therefore, $\mathrm{tr}(r)$ and $\mathrm{tr}(s)$ realize the same morphism. We define $\mathrm{tr}^A_{X,Y}(\boldsymbol{f})$ to be the morphism realized by $\mathrm{tr}(r)$. The axioms in Definition 5 follow from the fact that **Int**(**Pfn**) is a compact closed category, which is in particular a traced symmetric monoidal category.

Corollary 1. **Adm** *has an operator* $(-)^\dagger : \mathbf{Adm}(!A\otimes!X, X) \to \mathbf{Adm}(!A, X)$ *that satisfies* $\boldsymbol{f} = \boldsymbol{f} \circ (!A \otimes (!\boldsymbol{f}^\dagger \circ \boldsymbol{\delta})) \circ \boldsymbol{c}$ *for every* $\boldsymbol{f} : !A\otimes!X \to X$ *where* $\boldsymbol{c}$ *is the contraction* $!A \to !A\otimes!A$ *of* **Adm**, *and* $\boldsymbol{\delta}$ *is the comultiplication* $!X \to !!X$ *of* **Adm**.

Proof. Since **Adm** is traced and the linear exponential comonad of **Adm** is idempotent, the statement holds by Remark 5.1 in [17]. For the proof, see [17,21].

The following lemma relates **Adm** with $\mathbf{Adm}_\perp$. We omit the proof. Explicitly, LX in the following Lemma is given by $\{(\emptyset,\emptyset)\} \cup \{(\lambda \mathsf{k}.\mathsf{k}\ x, \lambda \mathsf{k}.\mathsf{k}\ y) \mid (x,y) \in X\}$, and the unit $\boldsymbol{h} : X \to ULX$ of the adjunction is given by $\boldsymbol{h}[x]_X = [\lambda \mathsf{k}.\mathsf{k}\ x]_{LX}$. In fact, the linear exponential comonad $\dot{!}$ of $\mathbf{Adm}_\perp$ is the induced linear exponential comonad $L!U$.

Lemma 1. *The forgetful functor* $U : \mathbf{Adm}_\perp \to \mathbf{Adm}$ *has a left adjoint functor* $L : \mathbf{Adm} \to \mathbf{Adm}_\perp$ *which is strong monoidal.*

Proposition 3. $\mathbf{Adm}_\perp$ *has an operator* $(-)^\ddagger : \mathbf{Adm}_\perp(\dot{!}A\dot{\otimes}\dot{!}X, X) \to \mathbf{Adm}_\perp(\dot{!}A, X)$ *that satisfies* $\boldsymbol{f} = \boldsymbol{f} \circ (\dot{!}A\dot{\otimes}(\dot{!}\boldsymbol{f}^\ddagger \circ \dot{\boldsymbol{\delta}})) \circ \dot{\boldsymbol{c}}$ *for every* $\boldsymbol{f} : \dot{!}A\dot{\otimes}\dot{!}X \to X$ *where* $\dot{\boldsymbol{c}}$ *is the contraction* $\dot{!}A \to \dot{!}A\dot{\otimes}\dot{!}A$ *of* $\mathbf{Adm}_\perp$, *and* $\dot{\boldsymbol{\delta}}$ *is the comultiplication* $\dot{!}X \to \dot{!}\dot{!}X$ *of* $\mathbf{Adm}_\perp$.

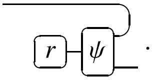
Proof. Let $\boldsymbol{g}$ be the transpose of $L(!UA\otimes!UX) \xrightarrow{\cong} L!UA\dot{\otimes}L!UX \xrightarrow{\boldsymbol{f}} X$. We define $\boldsymbol{f}^\ddagger : \dot{!}A \to X$ for $\boldsymbol{f} : \dot{!}A\dot{\otimes}\dot{!}X \to X$ to be $L!UA \xrightarrow{L(\boldsymbol{g}^\dagger)} LUX \longrightarrow X$. By diagram chasing, we can check $\boldsymbol{f} = \boldsymbol{f} \circ (\dot{!}A\dot{\otimes}(\dot{!}\boldsymbol{f}^\ddagger \circ \dot{\boldsymbol{\delta}})) \circ \dot{\boldsymbol{c}}$.

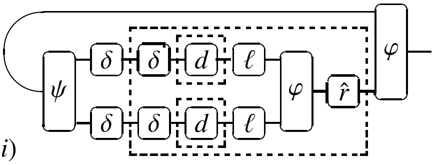
By Proposition 3, we can interpret recursion in $\mathbf{Adm}_\perp$ by $(-)^\ddagger$. As in the proof, ‡ is constructed from †, and † is constructed from the trace operator of **Adm**. Since the trace of **Adm** is calculated by taking trace of a realizer of $\boldsymbol{f}$ in **Int**(**Pfn**), the trace operator of **Int**(**Pfn**) (cyclic computation) appears in the interpretation of recursion.

Extraction of realizers. We calculate a realizer of $\boldsymbol{f}^\ddagger : \dot{!}A \to X$ for $\boldsymbol{f} : \dot{!}A\dot{\otimes}\dot{!}X \to X$. We take a realizer r of $\boldsymbol{f}$. Let $\boldsymbol{g} : !UA\otimes!UX \to UX$ be an **Adm**-morphism obtained by the transpose of $L(!UA\otimes!UX) \xrightarrow{\cong} L!UA\dot{\otimes}L!UX \xrightarrow{\boldsymbol{f}} X$. We define h and M by:

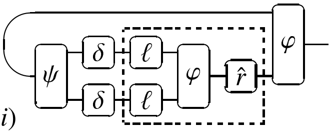
$$h := \ !UA\otimes!UX \longrightarrow !!UA\otimes!!UX \longrightarrow !(!UA\otimes!UX) \xrightarrow{!\boldsymbol{g}} !UX$$

$$\mathsf{M} := \lambda \mathsf{x}.\mathtt{let}\ \mathsf{p}\otimes\mathsf{q}\ \mathsf{be}\ \mathsf{x}\ \mathtt{in}\ \mathtt{let}\ !\mathsf{s}\ \mathsf{be}\ \mathsf{p}\ \mathtt{in}\ \mathtt{let}\ !\mathsf{t}\ \mathsf{be}\ \mathsf{q}\ \mathtt{in}\ !(r\ ((\lambda \mathsf{k}.\mathsf{k}\ !\mathsf{s})\otimes(\lambda \mathsf{k}.\mathsf{k}\ !\mathsf{t}))).$$

Then M realizes h. The diagram of M is (i), where $\hat{r}$ is  .

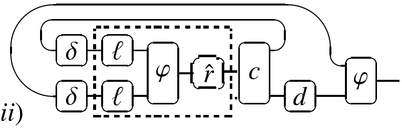


(i)

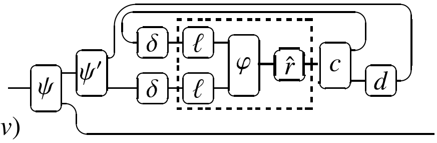


(ii)

Since the diagrams (i) and (ii) realize the same morphism, we can optimize (i) by (ii). Then $\boldsymbol{g}^{\dagger}$ is realized by the following diagram (iii).



(iii)



(iv)

We write s for the diagram (iii). Then $\lambda\mathsf{k}.\mathsf{k}\ s$ realizes $\boldsymbol{f}^{\dagger}$. By uncurrying the diagram of $\lambda\mathsf{k}.\mathsf{k}\ s$ and arranging the uncurried diagram, we obtain a diagram (iv) in the above. By replacing $\hat{r}$ and φ in the dotted line box with the modified GoI interpretation of M, we obtain the modified GoI interpretation of $\mu\mathsf{x}^{\mathsf{A}}.\mathsf{M}$ in Figure 4 for $|\Gamma| = 1$.

5 Proof of Soundness and Adequacy

We show Theorem 1 by means of logical relations. Theorem 1 follows from Proposition 4 and Proposition 5 in this section.

For a closed type A, we define $\mathcal{R}(\mathsf{A})$ to be the set of relations R between $\mathcal{D}(0, 1)$ and $\mathrm{Term}(\mathsf{A})$ such that

- $\forall \mathsf{M} \in \mathrm{Term}(\mathsf{A}).\ (\emptyset, \mathsf{M}) \in R$.
- If $(x_i, \mathsf{M}) \in R$ for an ascending chain $\{x_i\}_{i\in\mathbb{N}}$, then $(\bigvee_i x_i, \mathsf{M}) \in R$.
- If $(x, \mathsf{M}) \in R$ and $\mathsf{M} \Downarrow \mathsf{V} \Longleftrightarrow \mathsf{N} \Downarrow \mathsf{V}$, then $(x, \mathsf{N}) \in R$.

For $\tau : \Theta \to \mathrm{Ty}$ and $\{\rho(\mathsf{X}) \in \mathcal{R}(\tau(\mathsf{X}))\}_{\mathsf{X}\in\Theta}$, we define $\langle \mathsf{A} \rangle_{\tau,\rho} \in \mathcal{R}(\mathsf{A}[\tau(\Theta)/\Theta])$ for $\Theta \vdash \mathsf{A}$:

$$\begin{array}{rcl}
\langle \mathsf{X} \rangle_{\tau,\rho} &=& \rho(\mathsf{X}) \\
\langle \mathsf{A} \multimap \mathsf{B} \rangle_{\tau,\rho} &=& \{(x, \mathsf{M}) \mid \forall (y, \mathsf{N}) \in \langle \mathsf{A} \rangle_{\tau,\rho}.\ (x\ y, \mathsf{M}\ \mathsf{N}) \in \langle \mathsf{B} \rangle_{\tau,\rho}\} \\
\langle !\mathsf{A} \rangle_{\tau,\rho} &=& \begin{array}{l} \{(\emptyset, \mathsf{M}) \mid \mathsf{M} \in \mathrm{Term}(!\mathsf{A}[\tau(\Theta)/\Theta])\} \\ \cup\{(\lambda \mathsf{k}.\mathsf{k}\ !x, \mathsf{N}) \mid \mathsf{N} \Downarrow !\mathsf{M} \wedge (x, \mathsf{M}) \in \langle \mathsf{A} \rangle_{\tau,\rho}\} \end{array} \\
\langle \forall \mathsf{X}.\mathsf{A} \rangle_{\tau,\rho} &=& \bigcap_{\mathsf{B}\in\mathrm{Ty}, R\in\mathcal{R}(\mathsf{B})} \{(x, \mathsf{M}) \mid (x, \mathsf{N}\ \mathsf{B}) \in \langle \mathsf{A} \rangle_{\tau[\mathsf{B}/\mathsf{X}],\rho[R/\mathsf{X}]}\}
\end{array}$$

For $\Theta \mid \Gamma; \Delta \vdash \mathsf{M} : \mathsf{A}$, since $[\![\mathsf{M}]\!] \in \mathcal{D}(|\Gamma| + |\Delta|, 1)$, we have a map from $\mathcal{D}(0, 1)^{|\Gamma|} \times \mathcal{D}(0, 1)^{|\Delta|}$ to $\mathcal{D}(0, 1)$. We write its value at $(\boldsymbol{x}, \boldsymbol{y})$ by $[\![\mathsf{M}]\!](\boldsymbol{x}, \boldsymbol{y})$.

Lemma 2. *For* $\Theta \mid \Gamma ; \Delta \vdash \mathsf{M} : \mathsf{A}$ *and* $\mathbf{P} \in \mathrm{Term}(\Gamma)$ *and* $\mathbf{Q} \in \mathrm{Term}(\Delta)$*, we have* $[\![\mathsf{M}]\!]\,([\![\mathbf{!P}]\!], [\![\mathbf{Q}]\!]) = [\![\mathsf{M}[\mathbf{P}/\Gamma, \mathbf{Q}/\Delta]]\!]$.

Proposition 4 (Soundness). *If* $\mathsf{M} \Downarrow \mathsf{V}$ *then* $[\![\mathsf{M}]\!] = [\![\mathsf{V}]\!]$.

Proof. By induction on the derivation of $\mathsf{M} \Downarrow \mathsf{V}$. In the induction step of $\mu \mathsf{x}^{\mathsf{A}}.\mathsf{M}$, we can show that the modified GoI interpretation is equal to the least fixed point of a continuous map on $\mathcal{D}(0,1)$ by an argument similar to the proof of Proposition 2. Then, the induction step of $\mu \mathsf{x}^{\mathsf{A}}.\mathsf{M}$ follows from the induction hypothesis. Other cases follow from Lemma 2.

Proposition 5. *For a term* $\Theta \mid \Gamma ; \Delta \vdash \mathsf{M} : \mathsf{A}$*, a map* $\tau : \Theta \to \mathrm{Ty}$*, relations* $\rho(\mathsf{X}) \in \mathcal{R}(\tau(\mathsf{X}))$ *for* $\mathsf{X} \in \Theta$*, pairs* $(\xi_0(\mathsf{x}), \xi_1(\mathsf{x})) \in \langle !\mathsf{B} \rangle_{\tau,\rho}$ *for* $(\mathsf{x} : \mathsf{B}) \in \Gamma$ *and pairs* $(\delta_0(\mathsf{x}), \delta_1(\mathsf{x})) \in \langle \mathsf{C} \rangle_{\tau,\rho}$ *for* $(\mathsf{x} : \mathsf{C}) \in \Delta$*, the following holds:*

- *If* $\xi_1(\mathsf{x}) \Downarrow !\xi'_1(\mathsf{x})$ *for every* $(\mathsf{x} : \mathsf{B})$ *in* Γ *then*

$$([\![\mathsf{M}]\!]\,(\xi_0(\Gamma), \delta_0(\Delta)), \mathsf{M}[\tau(\Theta)/\Theta, \xi'_1(\Gamma)/\Gamma, \delta_1(\Delta)/\Delta]) \in \langle \mathsf{A} \rangle_{\tau,\rho}.$$

- *If there exists* $\mathsf{x} : \mathsf{B}$ *in* Γ *such that* $\xi_1(\mathsf{x}) \Uparrow$ *then* $[\![\mathsf{M}]\!]\,(\xi_0(\Gamma), \delta_0(\Delta)) = \emptyset$.

Proof. By induction on M. In the induction step of $\mu \mathsf{x}^{\mathsf{A}}.\mathsf{M}$, we argue as in the proof of Proposition 2. We can calculate the interpretation of $\mu \mathsf{x}^{\mathsf{A}}.\mathsf{M}$ by the lub of an ω-chain. Then, the induction step of $\mu \mathsf{x}^{\mathsf{A}}.\mathsf{M}$ follows from the closedness of $\langle \mathsf{A} \rangle_{\tau,\rho}$ under lubs. We can directly show the other cases by the induction hypothesis.

Related works. The GoI interpretation in this paper is based on the traced symmetric monoidal category **Pfn**. In [1], Abramsky, Haghverdi and Scott captured a categorical background of GoI interpretation by *GoI situation*. In [15], Haghverdi introduced *unique decomposition category* so as to capture the original GoI interpretation. The standard GoI interpretation in this paper is based on a GoI situation on the unique decomposition category **Pfn**.

We can describe our modified GoI interpretation by data of a GoI situation on the traced symmetric monoidal category **Pfn**, and we used the unique decomposition structure of **Pfn** to show that **Adm** is traced. However, we do not know to which class of unique decomposition categories we can generalise our argument. Since we used the ω-cpo structure on $\mathcal{D}(0,1) = \mathbf{Pfn}(\mathbb{N}, \mathbb{N})$ in the definition of **Adm** and the proof of Proposition 2, we might need to require some additional domain theoretic structures on a unique decomposition category. We also want to explore relationship between trace operators on linear categories and fixed point operators on cartesian closed categories from our concrete example.

At this point, we do not know how our modified GoI interpretation and the category **Adm** are related to the combinatory algebra called $\mathcal{B}$ in [23] and call-by-value game semantics [3].

Acknowledgement. I thank Masahito Hasegawa and Sin-ya Katsumata for discussions and advice. The author is supported by JSPS Research Fellowships for Young Scientists.

References

1. Abramsky, S., Haghverdi, E., Scott, P.J.: Geometry of interaction and linear combinatory algebras. Math. Struct. in Comput. Sci. 12(5), 625–665 (2002)
2. Abramsky, S., Lenisa, M.: A Fully Complete PER Model for ML Polymorphic Types. In: Clote, P., Schwichtenberg, H. (eds.) CSL 2000. LNCS, vol. 1862, pp. 140–155. Springer, Heidelberg (2000)
3. Abramsky, S., McCusker, G.: Call-by-value games. In: Nielsen, M. (ed.) CSL 1997. LNCS, vol. 1414, pp. 1–17. Springer, Heidelberg (1998)
4. Baillot, P., Pedicini, M.: Elementary complexity and geometry of interaction. Fundam. Inf. 45(1-2), 1–31 (2001)
5. Barber, A., Plotkin, G.: Dual intuitionistic linear logic, Technical Reprot ECS-LFCS-96-347, LFCS, Universit of Edinburgh (1997)
6. Barber, A.G.: Linear Type Theories, Semantics and Action Calculi. PhD thesis, University of Edinburgh (1997)
7. Bierman, G.M.: What is a categorical model of intuitionistic linear logic? In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 78–93. Springer, Heidelberg (1995)
8. Bierman, G.M., Pitts, A.M., Russo, C.V.: Operational properties of lily, a polymorphic linear lambda calculus with recursion. Electr. Notes Theor. Comput. Sci. 41(3) (2000)
9. Birkedal, L., Møgelberg, R.E., Petersen, R.L.: Domain-theoretical models of parametric polymorphism. Theor. Comput. Sci. 388(1-3), 152–172 (2007)
10. Danos, V., Regnier, L.: Reversible, irreversible and optimal lambda-machines. Electr. Notes Theor. Comput. Sci. 3 (1996)
11. Girard, J.-Y.: Geometry of Interaction I: Interpretation of System F. In: Ferro, R., et al. (eds.) Logic Colloquium 1988. North-Holland, Amsterdam (1989)
12. Girard, J.-Y.: Geometry of Interaction II: Deadlock-free Algorithms. In: Martin-Löf, P., Mints, G. (eds.) COLOG 1988. LNCS, vol. 417, pp. 76–93. Springer, Heidelberg (1990)
13. Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge University Press, Cambridge (1989)
14. Gonthier, G., Abadi, M., Lévy, J.-J.: The geometry of optimal lambda reduction. In: POPL, pp. 15–26 (1992)
15. Haghverdi, E.: A Categorical Approach to Linear Logic, Geometry of Proofs and Full Completeness. PhD thesis, University of Ottawa (2000)
16. Haghverdi, E., Scott, P.J.: A categorical model for the geometry of interaction. Theor. Comput. Sci. 350(2-3), 252–274 (2006)
17. Hasegawa, M.: On traced monoidal closed categories. Mathematical Structures in Computer Science 19(2), 217–244 (2009)
18. Jones, S.P.: The Implementation of Functional Programming Languages. Prentice Hall, Englewood Cliffs (1987)
19. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. Mathematical Proceedings of the Cambridge Philosophical Society 119(3), 447–468 (1996)
20. Mackie, I.: The geometry of interaction machine. In: POPL, pp. 198–208 (1995)
21. Melliès, P.-A.: Functorial Boxes in String Diagrams. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 1–30. Springer, Heidelberg (2006)
22. Simpson, A.K.: Reduction in a Linear Lambda-Calculus with Applications to Operational Semantics. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 219–234. Springer, Heidelberg (2005)
23. van Oosten, J.: A combinatory algebra for sequential functionals of finite type. In: Cooper, S.B., Truss, J.K. (eds.) Models and Computatbility. Cambridge University Press, Cambridge (1999)

Estimation of the Length of Interactions in Arena Game Semantics

Pierre Clairambault

University of Bath
p.clairambault@bath.ac.uk

Abstract. We estimate the maximal length of interactions between strategies in HO/N game semantics, in the spirit of the work by Schwichtenberg and Beckmann for the length of reduction in simply typed λ-calculus. Because of the operational content of game semantics, the bounds presented here also apply to head linear reduction on λ-terms and to the execution of programs by abstract machines (PAM/KAM), including in presence of computational effects such as non-determinism or ground type references. The proof proceeds by extracting from the games model a combinatorial rewriting rule on trees of natural numbers, which can then be analysed independently of game semantics or λ-calculus.

1 Introduction

Among the numerous notions of execution that one can consider on higher-order programming languages (in particular on the λ-calculus) *head linear reduction* [9] plays a particular role. Although it is not as widespread and specifically studied as, say, β-reduction, it is nonetheless implicit to various approaches of higher-order computation, such as geometry of interaction, game semantics, optimal reduction and ordinary operational semantics. It is also implicit to several abstract machines, including the Krivine Abstract Machine (KAM) [17] and the Pointer Abstract Machine (PAM) [9], in the sense that it is the reduction they perform [9,5] and as such is a valuable abstraction of how programs are executed in the implementation of higher order languages.

Despite being closer to the implementation of programming languages, head linear reduction never drew a lot of attention from the community. Part of the reason for that is that it is not a usual notion of reduction: defining it properly on λ-terms requires both to extend the notion of redex and to restrict to linear substitution, leading to rather subtle and tricky definitions which lack the canonicity of β-reduction[1]. Moreover, its associated observational equivalence is the same as for the usual head β-reduction, which makes it non relevant as long as one is interested in the equational theory of λ-calculus. However, head linear

[1] However, there are syntaxes on which head linear reductions appear more canonical that β-reduction, for instance proof nets [21].

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 335–349, 2011.

reduction should appear in the foreground as soon as one is interested in quantitative aspects of computation, such as complexity. On the contrary, although very precise bounds are known for the possible length of β-reduction chains in simply typed λ-calculus [24,3], to the author's knowledge, the situation for head linear reduction remains essentially unexplored. Even if it is generally expected that the bounds remain hyper-exponential[2] (and this indeed what we will prove), it does not seem to follow easily from the bounds known for β-reduction.

Rather than reasoning directly on head linear reduction, we will instead look at it through game semantics [16]. Indeed, there is a close relationship between head linear reduction and interaction in games model of programming languages [8]. More precisely, given two β-normal and η-long λ-terms S and T, there is a step-by-step correspondence between head linear reduction chains of ST and game-theoretic interactions between the strategies $[\![S]\!]$ and $[\![T]\!]$. Of course, game semantics are not central to our analysis: as is often the case, our methods and results could be adapted to a purely syntactical framework. However, games have this considerable advantage of accommodating in a single framework purely functional programming languages such as the λ-calculus or PCF and a number of computational features such as non-determinism [15], control operators [19] and references [1]. This will allow us to do our study with an increased generality: our complexity results will hold for a variety of settings, from simply typed λ-calculus to richer languages possibly featuring the computational effects mentioned above, as long as there is no fixed point operator.

Outline. In Section 2 we will recall some of the basic definitions of Hyland-Ong game semantics, define the central notion of size of a strategy, and introduce our main question as the problem of finding the maximal length of an interaction between two strategies of fixed size. Our approach will be then to progressively simplify this problem in order to reach its underlying combinatorial nature. In Section 3 we first introduce the notion of *visible pointer structures*, *i.e.* plays where the identity of moves has been forgotten. This allows a more elementary (strategy-free) equivalent statement of our problem. Then we show how each position in a visible pointer structure can be characterised by a tree of natural numbers called an *agent*. We then show that the problem can be once again reformulated as the maximal length of a reduction on these agents. In Section 4 we study the length of this reduction, giving in particular an upper bound. We also give a corresponding lower bound, and finally we use our result to estimate the maximal length of head linear reduction sequences on simply typed λ-terms.

Related works. Our results and part of our methods are similar to the works of Schwichtenberger and Beckmann [24,3], but the reduction we study is in some sense more challenging, because redexes are not destroyed as they are reduced. Moreover, the game semantics setting allows for an extra generality. The present

[2] Note however that in [10], De Bruijn gives an upper bound for his local β-reduction, akin to head linear reduction. The bound is an iterate of the diagonal of an Ackermann-like function!

work also has common points with work by Dal Lago and Laurent [18], in the sense that it uses tools from game semantics to reason on the length of execution. However the approach is very different : their estimate is very precise but uses an information on terms difficult to compute (almost as hard as actually performing execution). Here, we need little information on terms (gathering this information is linear in the size of the term), but our bounds are, in most cases, very rough.

2 Arena Game Semantics

We recall briefly the now usual definitions of arena games, first introduced in [16]. More detailed accounts can be found in [22,13]. We are interested in games with two participants: Opponent (O, the *environment*) and Player (P, the *program*).

2.1 Arenas and Plays

Valid plays are generated by directed graphs called *arenas*, which are semantic versions of *types*. Formally, an **arena** is a structure $A = (M_A, \lambda_A, I_A, \vdash_A)$ where:

- M_A is a set of **moves**,
- $\lambda_A : M_A \to \{O, P\}$ is a polarity function indicating whether a move is an Opponent or Player move (O-move or P-move).
- $I_A \subseteq \lambda_A^{-1}(\{O\})$ is a set of **initial moves**.
- $\vdash_A \subset M_A \times M_A$ is a relation called **enabling**, such that if $m \vdash_A n$, then $\lambda_A(m) \neq \lambda_A(n)$.

In other words, an arena is just a directed bipartite graph. We now define plays as **justified sequences** over A: these are sequences s of moves of A, each non-initial move m in s being equipped with a pointer to an earlier move n in s, satisfying $n \vdash_A m$. In other words, a justified sequence s over A is such that each reversed pointer chain $s_{i_0} \leftarrow s_{i_1} \leftarrow \ldots \leftarrow s_{i_n}$ is a path on A (viewed as a graph). The role of pointers is to allow *reopenings* or *backtracking* in plays. When writing justified sequences, we will often omit the justification information if this does not cause any ambiguity. The symbol $\sqsubseteq$ will denote the prefix ordering on justified sequences, and $s_1 \sqsubseteq^P s_2$ will mean that s_1 is a P-ending prefix of s_2. If s is a justified sequence on A, $|s|$ will denote its length.

Given a justified sequence s on A, it has two subsequences of particular interest: the P-view and O-view. The view for P (resp. O) may be understood as the subsequence of the play where P (resp. O) only sees his own duplications. Practically, the P-view $\ulcorner s \urcorner$ of s is computed by forgetting everything under Opponent's pointers, in the following recursive way:

- $\ulcorner sm \urcorner = \ulcorner s \urcorner m$ if $\lambda_A(m) = P$;
- $\ulcorner sm \urcorner = m$ if $m \in I_A$ and m has no justification pointer;
- $\ulcorner s_1 m s_2 n \urcorner = \ulcorner s \urcorner mn$ if $\lambda_A(n) = O$ and n points to m.

The O-view $\llcorner s \lrcorner$ of s is defined dually, without the special treatment of initial moves[3]. The *legal plays* over A, denoted by $\mathcal{L}_A$, are the justified sequences s on A satisfying the **alternation** condition, *i.e.* that if $tmn \sqsubseteq s$, then $\lambda_A(m) \neq \lambda_A(n)$.

[3] In the terminology of [13], it is the *long* O-view.

2.2 Classes of Strategies

In this subsection, we will present several classes of strategies on arena games that are of interest to us in the present paper. A **strategy** σ on A is a set of even-length legal plays on A, closed under even-length prefix. A strategy from A to B is a strategy $\sigma : A \Rightarrow B$, where $A \Rightarrow B$ is the usual arrow arena defined by $M_{A\Rightarrow B} = M_A + M_B$, $\lambda_{A\Rightarrow B} = [\overline{\lambda_A}, \lambda_B]$ (where $\overline{\lambda_A}$ means λ_A with polarity O/P reversed), $I_{A\Rightarrow B} = I_B$ and $\vdash_{A\Rightarrow B} = \vdash_A + \vdash_B + I_B \times I_A$.

Composition. We define composition of strategies by the usual parallel interaction plus hiding mechanism. If A, B and C are arenas, we define the set of **interactions** $I(A, B, C)$ as the set of justified sequences u over A, B and C such that $u_{\restriction_{A,B}} \in \mathcal{L}_{A\Rightarrow B}$, $u_{\restriction_{B,C}} \in \mathcal{L}_{B\Rightarrow C}$ and $u_{\restriction_{A,C}} \in \mathcal{L}_{A\Rightarrow C}$. Then, if $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$, we define parallel interaction as $\sigma \parallel \tau = \{u \in I(A, B, C) \mid u_{\restriction A,B} \in \sigma \wedge u_{\restriction B,C} \in \tau\}$, composition is then defined as $\sigma;\tau = \{u_{\restriction A,C} \mid u \in \sigma||\tau\}$. Composition is associative and admits copycat strategies as identities.

P-visible strategies. A strategy σ is P**-visible** if each of its moves points to the current P-view. Formally, for all $sab \in \sigma$, b points inside $\ulcorner sa \urcorner$. P-visible strategies are stable under composition, as is proved for instance in [13]. They correspond loosely to functional programs with ground type references [1].

Innocent strategies. The class of *innocent* strategies is central in game semantics, because of their correspondence with purely functional programs (or λ-terms) and of their useful definability properties. A strategy σ is **innocent** if

$$sab \in \sigma \wedge t \in \sigma \wedge ta \in \mathcal{L}_A \wedge \ulcorner sa \urcorner = \ulcorner ta \urcorner \Rightarrow tab \in \sigma$$

Intuitively, an innocent strategy only takes its P-view into account to determine its next move. Indeed, any innocent strategy is characterized by a set of P-views. This observation is very important since P-views can be seen as abstract representations of branches of η-expanded Böhm trees (*a.k.a.* Nakajima trees [23]) : this is the key to the definability process on innocent strategies [16]. It is quite technical to prove that innocent strategies are stable under composition, proofs can be found for instance in [13,5]. Arenas and innocent strategies form a cartesian closed category and are therefore a model of simply typed λ-calculus.

Bounded strategies. A strategy σ is **bounded** if it is P-visible and if the length of its P-views is bounded: formally, there exists $N \in \mathbb{N}$ such that for all $s \in \sigma$, $|\ulcorner s \urcorner| \leq N$. Bounded strategies are stable under composition, as is proved in [6] for the innocent case and in [5] for the general case. This result corresponds loosely to the strong normalisation result on simply-typed λ-calculus. Syntactically, bounded strategies include the interpretation of all terms of a functional programming language without a fixed point operator but with Algol-like ground type references (for details about how reference cells get interpreted as strategies see for instance [1], it is obvious that this interpretation yields a bounded strategy) and arbitrary non determinism. This remark is important since it implies that our results will hold for any program written with these constructs, as long as they do not use recursion or a fixed point operator.

2.3 Size of Strategies and Interactions

Since in this paper we will be interested in the length of interactions, it is sensible to make it precise first what we mean by the *size* of strategies. Let σ be a bounded strategy, its size is defined as

$$|\sigma| = \frac{max_{s \in \sigma} |\ulcorner s \urcorner|}{2}$$

All our analysis on the size of interactions will be based on this notion of size of strategies. Our starting point is the following finiteness result, proved in [6]. We say that an interaction $u \in I(A, B, C)$ is **passive** if the only move by the external Opponent on A, C is the initial move on C, so that the interaction stops as soon as we need additional input from the external Opponent.

Proposition 1. *Let $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ be bounded strategies and let $u \in \sigma || \tau$ be a passive interaction, then u is finite.*

Using this, we can actually deduce the existence of an uniform bound on the length of such $u \in \sigma || \tau$, which only depends on the respective size of σ and τ:

Lemma 1. *For all $n, p \in \mathbb{N}$ there is a lesser $N(n, p) \in \mathbb{N}$ such that for all arenas A, B and C, for all $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ such that $|\sigma| \leq p$ and $|\tau| \leq n$, for all passive $u \in \sigma || \tau$ we have $|u| \leq N(n, p)$.*

Proof. For arenas A, B and C consider the set $T_{A,B,C}$ of all passive interactions $u \in I(A, B, C)$ such that for all $s \sqsubseteq u_{\upharpoonright B,C}$, $|\ulcorner s \urcorner| \leq 2n$ and for all $s \sqsubseteq u_{A,B}$, $|\ulcorner s \urcorner| \leq 2p$. Then, consider the union T of all the $T_{A,B,C}$, our goal here is to find a bound on the length of all elements of T. Consider now the tree structure on T given by the prefix ordering. To make this tree finitely branching, consider the relation $m \cong n \Leftrightarrow depth(m) = depth(n)$ on moves, where $depth(m)$ is the number of pointers required to go from m to an initial move. The tree $T/\cong$ is now finitely branching, but is also well-founded by Proposition 1, therefore it is finite by König's lemma[4]. Let $N(n, p)$ be its maximal depth, it is now obvious that it satisfies the required properties.

We have proved the existence of the uniform bound $N(n, p)$, but in a way that provides no feasible means of estimating $N(n, p)$. The goal of the rest of this paper is to estimate this bound as precisely as possible. As a matter of fact, we will be mainly interested in the "typed" variant $N_d(n, p)$, defined as the maximum length of all possible passive interactions between strategies $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ of respective size p and n, where B has a finite depth $d - 1$.

3 Pointer Structures and Rewriting

We have seen that to prove Lemma 1, we must consider plays up to an equivalence relation $\cong$ which assimilates all moves at the same depth. Indeed, general arenas

[4] Or, more adequately, the fan theorem.

and plays contain information which is useless for us. Following [6], we will here reason on *pointer structures*, which result of considering moves in plays up to $\cong$. Pointer structures are also similar to the *parity pointer functions* of Harmer, Hyland and Melliès [14] and to the *interaction sequences* of Coquand [7]. We will delve here into their combinatorics and extract from them a small rewriting system, whose study is sufficient to characterize their length.

3.1 $N_d(n,p)$ as a Bound for Pointer Structures

Visible pointer structures. In [6], we introduced pointer structures by elementary axioms, independent of the general notions of game semantics. Instead here, we define **pointer structures** as usual alternating plays, but on the particular "pure" arena $I_\omega = \bigsqcup_{n\in\mathbb{N}} I_n$, where $I_0 = \bot$ ($\bot$ is the singleton arena with just one Opponent move) and $I_{n+1} = I_n \Rightarrow \bot$. As we are interested in the interaction between P-visible strategies, we will only consider **visible** pointer structures, where both players point in their corresponding view. Formally, s is visible if for all $s'p \sqsubseteq^P s$, a points inside $\ulcorner s' \urcorner$ and if for all $s'o \sqsubseteq^O s$, o points inside $\llcorner s' \lrcorner$. The **depth** of a visible pointer structure s is the smallest d such that s is a play on I_d. Let us denote by $\mathcal{V}$ the set of all visible pointer structures.

Atomic agents. After forgetting information on plays, let us forget information on strategies. Instead of considering bounded strategies with all their intentional behaviour, we will just keep the data of their size. Pointer structures will then be considered as interactions between the corresponding numbers which will be called **atomic agents**. If n is such a natural number, we define its **trace** as follows, along with the dual notion of **co-trace**:

$$Tr(n) = \{s \in \mathcal{V} \mid \forall s' \sqsubseteq s, |\ulcorner s' \urcorner| \leq 2n\}$$
$$coTr(p) = \{s \in \mathcal{V} \mid \forall s' \sqsubseteq s, |\llcorner s' \lrcorner| \leq 2p + 1\}$$

An **interaction** at depth d between n and p is a visible pointer structure s of depth at most d such that $s \in Tr(n) \cap coTr(p)$. We write $s \in n \star_d p$. These definitions allow to give the following strategy-free equivalent formulation of $N_d(n,p)$.

Lemma 2. *Let n and p be natural numbers and $d \geq 2$, then*

$$N_d(n,p) = max\{|s| \mid s \in n \star_d p\}$$

Proof. Consider the maximal bounded strategies of respective size n and p, defined as $\mathbf{n} = \{s \in I_d \mid \forall s' \sqsubseteq s,\ |\ulcorner s' \urcorner| \leq 2n$ and $\mathbf{p} = \{s \in I_{d-1} \mid \forall s' \sqsubseteq s,\ |\ulcorner s' \urcorner| \leq 2p$. Then pointer structures in $n \star_d p$ are the same as (passive) interactions in $\mathbf{p}||\mathbf{n}$, thus $max\{|s| \mid s \in n \star_d p\} \leq N_d(n,p)$. Reciprocally, if $\sigma : A \Rightarrow B$ has size p and $\tau : B \Rightarrow C$ has size n and if $u \in \sigma||\tau$ is passive, then if u' denotes u where moves are considered up to $\cong$ we have $u' \in \mathbf{p}||\mathbf{n}$ thus $u' \in n \star_d p$ and $N_d(n,p) = max\{|s| \mid s \in n \star_d p\}$.

3.2 Agents

To bound the length of a pointer structure s, our idea is to label each of its moves s_i by an object t, expressing the size that the strategies have left. Let us consider here an analogy between pointer structures and the execution of λ-terms by the KAM[5]. Consider the following three KAM computation steps:

$$(\lambda x.xS) \star T \cdot \pi_0 \rightsquigarrow^3 T \star S^{x \mapsto T} \cdot \pi_0$$

The interaction between two closed terms (with empty environment) leads, after three steps of computation, to the interaction between two *open terms* T and S (where x is free in S), with an environment. By analogy, if s_0 is labelled by the pair (n, p) of interacting "strategies", each move s_i should correspond to an interaction between objects (a, b), where a and b have a tree-like structure which is reminiscent of those of closures[6].

We will call a **pointed visible pointer structure** (pvps) a pair (s, i) where s is a visible pointer structure and $i \leq |s| - 1$ is an arbitrary "starting" move. We adapt the notions of size and depth for pvps, and introduce a notion of *context*.

Definition 1. *Let (s, i) be a pointed visible pointer structure. The* ***residual size*** *of s at i, written* rsize(s, i)*, is defined as follows:*

- *If s_i is an Opponent move, it is* $\max_{s_i \in \ulcorner s_{\leq j} \urcorner} |\ulcorner s_{\leq j} \urcorner| - |\ulcorner s_{\leq i} \urcorner| + 1$
- *If s_i is a Player move, it is* $\max_{s_i \in \llcorner s_{\leq j} \lrcorner} |\llcorner s_{\leq j} \lrcorner| - |\llcorner s_{\leq i} \lrcorner| + 1$

where $s_i \in \ulcorner s_{\leq j} \urcorner$ means that the computation of $\ulcorner s_{\leq j} \urcorner$ reaches[7] s_i. Dually, we have the notion of ***residual co-size*** *of s at i, written* rcosize(s, i)*, defined as follows:*

- *If s_i is an Opponent move, it is* $\max_{s_i \in \llcorner s_{\leq j} \lrcorner} |\llcorner s_{\leq j} \lrcorner| - |\llcorner s_{\leq i} \lrcorner| + 1$
- *Otherwise,* $\max_{s_i \in \ulcorner s_{\leq j} \urcorner} |\ulcorner s_{\leq j} \urcorner| - |\ulcorner s_{\leq i} \urcorner| + 1$

The residual depth *of s at i is the maximal length of a pointer chain in s starting from s_i.*

Definition 2. *Let s be a visible pointer structure. We define the* ***context*** *of (s, i) as:*

- *If s_i is an O-move, the set $\{s_{n_1}, \ldots, s_{n_p}\}$ of O-moves appearing in $\ulcorner s_{<i} \urcorner$,*
- *If s_i is a P-move, the set $\{s_{n_1}, \ldots, s_{n_p}\}$ of P-moves appearing in $\llcorner s_{<i} \lrcorner$.*

In other words it is the set of moves to which s_{i+1} can point whilst abiding to the visibility condition, except s_i. We also need the dual notion of co-context, which contains the moves the other player can point to. The ***co-context*** *of (s, i) is:*

[5] The syntax used here seems natural enough, but is for instance described in [5].

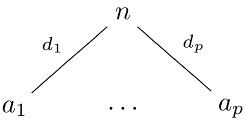
[6] As in the example above, closures are pairs M^σ where M is an open term and σ is an *environment*, *i.e.* a mapping which to each free variable of M associates a closure.

[7] So starting from s_j and following Opponent's pointers eventually reaches s_i.

- *If s_i is an O-move, the set $\{s_{n_1}, \dots, s_{n_p}\}$ of P-moves appearing in $\llcorner s_{<i} \lrcorner$,*
- *If s_i is a P-move, the set $\{s_{n_1}, \dots, s_{n_p}\}$ of O-moves appearing in $\ulcorner s_{<i} \urcorner$.*

Definition 3. *A **general agent** (just called agent for short) is a finite tree, whose nodes and edges are both labelled by natural numbers. If $a_1, \dots, a_p$ are agents and $d_1, \dots, d_p$ are natural numbers, we write:*

$$n[\{d_1\}a_1, \dots, \{d_p\}a_p] =$$



Definition 4 (Trace, co-trace, interaction). *Let us generalize the notion of trace to general agents. The two notions Tr and $coTr$ are defined by mutual recursion, as follows: let $a = n[\{d_1\}a_1, \dots, \{d_p\}a_p]$ be an agent. We say that (s, i) is a **trace** (resp. a **co-trace**) of a, denoted $(s, i) \in Tr(a)$ (resp. $(s, i) \in coTr(a)$) if the following conditions are satisfied:*

- $\mathrm{rsize}(s, i) \leq 2n$ *(resp.* $\mathrm{rcosize}(s, i) \leq 2n + 1$*),*
- *If $\{s_{n_1}, \dots, s_{n_p}\}$ is the context of (s, i) (resp. co-context), then for each $k \in \{1, \dots, p\}$ we have $(s, n_k) \in coTr(a_k)$.*
- *If $\{s_{n_1}, \dots, s_{n_p}\}$ is the context of (s, i) (resp. co-context), then for each $k \in \{1, \dots, p\}$ the residual depth of s at n_k is less than d_k.*

*Then, we define an **interaction** of two agents a and b at depth d as a pair $(s, i) \in Tr(a) \cap coTr(b)$ where the residual depth of s at i is less than d, which we write $(s, i) \in a \star_d b$.*

Notice that we use the same notations Tr, $coTr$ and $\star$ both for natural numbers and general agents. This should not generate any confusion, since the definitions just above coincide with the previous ones in the special case of "atomic", or closed, agents: if n and p are natural numbers, then obviously $s \in n \star_d p$ if and only if $(s, 0) \in n[] \star_d p[]$. Note also that definitions are adapted here to this particular setting where strategies are replaced by natural numbers, however they could be generalized to the usual notion of strategies. An agent would be then a tree of strategies, and a trace of this agent would be a possible interaction between all these strategies. This would be a new approach to the problem of *revealed* or *uncovered* game semantics [12,4], where strategies are not necessarily cut-free.

3.3 Simulation of Visible Pointer Structures

We introduce now the main tool of this paper, a reduction on agents which "simulates" visible pointer structures: if $n[\{d_1\}a_1, \dots, \{d_p\}a_p]$ and b are agents ($n > 0$), we define the non-deterministic reduction relation $\rightsquigarrow$ on triples (a, d, b), where d is a depth (a natural number) and a and b are agents, by the following two cases:

$$(n[\{d_1\}a_1, \dots, \{d_p\}a_p], d, b) \rightsquigarrow (a_i, d_i - 1, (n-1)[\{d_1\}a_1, \dots, \{d_p\}a_p, \{d\}b])$$
$$(n[\{d_1\}a_1, \dots, \{d_p\}a_p], d, b) \rightsquigarrow (b, d - 1, (n-1)[\{d_1\}a_1, \dots, \{d_p\}a_p, \{d\}b])$$

where $i \in \{1, \ldots, p\}$, $d_i > 0$ in the first case and $d > 0$ in the second case. We can now state the following central proposition.

Proposition 2 (Simulation). *Let $(s, i) \in a \star_d b$, then if s_{i+1} is defined, there exists $(a, d, b) \rightsquigarrow (a', d', b')$ such that $(s, i+1) \in a' \star_{d'} b'$.*

Proof. The proof proceeds by a close analysis of where in its P-view (resp. O-view) s_{i+1} can point. If it points to s_i, then the active strategy asks for its argument which corresponds to the second reduction case. If it points to some element s_{n_i} of its context, the active strategy calls the i-th element of its context: this is the first reduction case, putting the subtree a_i in head position. The rest of the proof consists in technical verifications, to check that the new triple (a', d', b') is such that $(s, i+1) \in a' \star_{d'} b'$.

The result above will be sufficient for our purpose. Let us mention in passing that the connection between visible pointer structures and agents is in fact tighter: a reduction chain starting from a triple $(n[], d, p[])$ can also be canonically mapped to a pointed visible pointer structure in $n \star_d p$, and the two translations are inverse of one another. The interested reader is directed to [5].

Before going on to the study of the rewriting rules introduced above, let us give a last simplification. If $a = n[\{d_1\}t_1, \ldots, \{d_p\}t_q]$ and b are agents, then $a \cdot_d b$ will denote the agent obtained by appending b as a new son of the root of a with label d, *i.e.* $n[\{d_1\}t_1, \ldots, \{d_p\}t_q, \{d\}b]$. Consider the following non-deterministic rewriting rule on agents:

$$n[\{d_1\}a_1, \ldots, \{d_p\}a_p] \rightsquigarrow a_i \cdot_{d_i - 1} (n-1)[\{d_1\}a_1, \ldots, \{d_p\}a_p]$$

Both rewriting rules on triples (a, d, b) are actually instances of this reduction, by the isomorphism $(a, d, b) \mapsto a \cdot_d b$. We let the obvious verification to the reader. This is helpful, as all that remains to study is this reduction on agents illustrated in Figure 1. To summarize, if $N(a)$ denotes the length of the longest reduction sequence starting from an agent a, we have the following property.

Proposition 3. *Let $n, p \geq 0$, $d \geq 2$, then $N_d(n, p) \leq N(n[\{d\}p[]]) + 1$.*

Proof. Obvious from the simulation lemma, adding 1 for the initial move which is not accounted for by the reduction on agents. In fact this is an equality, as one can prove using the *reverse* simulation lemma mentioned above. See [5].

4 Length of Interactions

The goal of this section is to study the reduction on agents introduced above, and to estimate its maximal length. We will first provide an upper bound for this length, adapting a method used by Beckmann [3] to estimate the maximal length of reductions on simply typed λ-calculus. We will then discuss the question of lower bounds, and finally describe an application to head linear reduction.

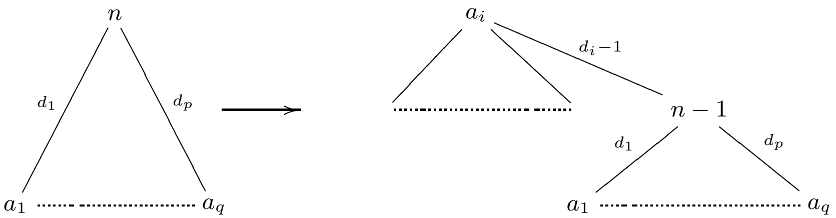


Fig. 1. Rewriting rule on agents

4.1 Upper Bound

We define on agents a predicate $\vdash^{\alpha}_{\rho}$, which introduction rules are compatible both with syntax and reduction.

Definition 5. *The predicate* $\vdash^{\alpha}_{\rho}$ *(where* ρ, α *range over natural numbers) is defined on agents in the following inductive way.*

- Base. $\vdash^{\alpha}_{\rho} 0[\{d_1\}a_1, \dots, \{d_p\}a_p]$
- Red. *Suppose* $a = n[\{d_1\}a_1, \dots, \{d_p\}a_p]$. *Then if for all* a' *such that* $a \rightsquigarrow a'$ *we have* $\vdash^{\alpha}_{\rho} a'$ *and if we also have* $\vdash^{\alpha}_{\rho} (n-1)[\{d_1\}a_1, \dots, \{d_p\}a_p]$, *then* $\vdash^{\alpha+1}_{\rho} a$.
- Cut. *If* $\vdash^{\alpha}_{\rho} a$, $\vdash^{\beta}_{\rho} b$ *and* $d \leq \rho$, *then* $\vdash^{\alpha+\beta}_{\rho} a \cdot_d b$.

By this inductive definition, each proposition $\vdash^{\alpha}_{\rho} a$ is witnessed by a tree using Base, Red and Cut. Red-free trees look like syntax trees, are easy to build but give few information on the reduction, whereas Cut-free trees look like reduction trees, are difficult to build but give very accurate information on the length of reduction. The idea of the proof is then to design an automatic way to turn a Red-free tree to a Cut-free tree, *via* a cut elimination lemma. Let us now give the statement and sketch the proof of the four important lemmas that underlie our reasoning.

A **context-agent** $a()$ is a finite tree whose edges are labelled by natural numbers, and whose nodes are labelled either by natural numbers, or by the variable x, with the constraint that all edges leading to x must be labelled by the same number d; d is called the **type** of x in $a()$. If We denote by $a(b)$ the result of substituting of all occurrences of x in $a()$ by b. We denote by $a(\emptyset)$ the agent obtained by deleting in a all occurrences of x, along with the edges leading to them.

Lemma 3 (Substitution lemma). *If* $\vdash^{\alpha}_{\rho} a(\emptyset)$, $\vdash^{\beta}_{\rho} b$ *and* $d \leq \rho + 1$ *(where* d *is the type of* x *in* a*), then* $\vdash^{\alpha(\beta+1)}_{\rho} a(b)$

Proof. We prove by induction on the tree witness for $\vdash^{\alpha}_{\rho} a(\emptyset)$ that the above property is true for all context-arena $a'()$ such that $a(\emptyset) = a'(\emptyset)$. The way to handle each case is essentially forced by the induction hypothesis.

Lemma 4 (Cut elimination lemma). *Suppose* $\vdash^{\alpha}_{\rho+1} a$. *Then if* $\alpha = 0$, $\vdash^{0}_{\rho} a$. *Otherwise,* $\vdash^{2^{\alpha-1}}_{\rho} a$.

Proof. By induction on the witness for $\vdash^{\alpha}_{\rho+1} a$, using the substitution lemma when the last rule is Cut with a type of $\rho + 1$.

Lemma 5 (Recomposition lemma). *Let* a *be an agent. Then* $\vdash^{max(a)|a|}_{depth(a)} a$, *where* $depth(a)$ *is the maximal label of an edge in* a, $max(a)$ *is the maximal label of a node and* $|a|$ *is the number of nodes.*

Proof. By induction on a.

Lemma 6 (Bound lemma). *Let* a *be an agent, then if* $\vdash^{\alpha}_{0} a$, $N(a) \leq \alpha$.

Proof. The only used rules are Base, Red and Cut with $\rho = 0$. These Cut rules do not add any possible reduction and are easy to eliminate, then the lemma is easily proved by induction on a.

These lemmas are sufficient to give a first upper bound, by iterating the cut elimination lemma starting from the witness tree for $\vdash^{\alpha}_{\rho} a$ generated by the recomposition lemma. However when the type is small, some of the lemmas above can be improved. For instance if $\vdash^{\alpha}_{0} a(\emptyset)$, $\vdash^{\beta}_{0} b$ and the type of x in $a()$ is 1, then $\vdash^{\alpha+\beta}_{0} a(b)$, since once the reduction reaches b it will never enter $a()$ again. Using this we get a "base" cut-elimination lemma, stating that for all a, whenever $\vdash^{\alpha}_{1} a$ then we have actually $\vdash^{\alpha}_{0} a$ instead of $\vdash^{2^{\alpha-1}}_{0} a$. Using this, we prove the following.

Theorem 1 (Upper bound). *Let* $depth(a)$ *denote the highest edge label in* a, $max(a)$ *means the highest node label and* $|a|$ *means the number of nodes of* a. *Then if* $depth(a) \geq 1$ *and* $max(a) \geq 1$ *we have:*

$$N(a) \leq 2^{max(a)|a|-1}_{depth(a)-1}$$

For the particular case when $a = n[\{d\}p[]]$ *and if* $d \geq 2$ *we have:*

$$N_d(n,p) \leq 2^{n(p+1)}_{d-2}$$

Proof. Both proofs are rather direct. For the first part, by the recomposition lemma we have $\vdash^{max(a)|a|}_{depth(a)} a$. It suffices then to apply $depth(a) - 1$ times the cut elimination lemma, then use the "base" cut-elimination lemma to eliminate the remaining cuts. For the second part we reason likewise, but rely on the substitution lemma instead of the recomposition lemma to get $\vdash^{n(p+1)}_{d-1} n[\{d\}p[]]$, which gives $N(n[\{d\}p[]]) \leq 2^{n(p+1)-1}_{d-2}$. But we have $N_d(n,p) \leq N(n[\{d\}p[]]) + 1$ by Proposition 3, which concludes the proof.

Note that whereas the bounds in [3] are asymptotic and give poor quantitative information if instantiated on small types, our bound does provide valuable information on interactions with small depth. For instance, if $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ such that $\text{rsize}(\sigma) = p$, $\text{rsize}(\tau) = n$ and the depth of B is at most 2, then no interaction between σ and τ can be longer than $N_3(n,p) \leq 2^{n(p+1)}$. As we will see below, this can not be significantly improved. In fact, we conjecture that for all $n \geq 1$ and $p \geq 2$, we have $N_3(n,p) = 2\frac{p^n-1}{p-1} + 1$: this was found and machine-checked for all $n + p \leq 17$ thanks to an implementation of agents and their reduction, unfortunately we could not prove its correctness, nor generalize it to higher depths.

4.2 Lower Bound

As argued in the introduction, the upper bound above applies to several programming languages executed by head linear reduction, possibly featuring non determinism and/or ground type references, therefore the fact that we used game semantics to prove it increases its generality. On the other hand, if we try to give the closest possible lower bound for $N_d(n,p)$ using the full power of visible pointer structures, we would get a lower bound without meaning for most languages concerned by the upper bound, since pointer structures have no innocence or determinism requirements[8]. Therefore what makes more sense is to describe a lower bound in the more restricted possible framework, *i.e.* simply typed λ-calculus.

We won't detail the construction much, as the method is standard and does not bring a lot to our analysis. The idea is to define higher types for church integers by $A_0 = \bot$ and $A_{n+1} = A_n \to A_n$. Then, denoting by $\underline{n}_p$ the church integer for n of type A_{p+2}, we define $S_n = \underline{2}_n \underline{2}_{n-1} \dots \underline{2}_0 : A_2$. We apply then S_n to $id_\bot$ to get a term whose head linear reduction chain has at least 2^1_{n+1} steps. In game semantics, $[\![\underline{2}_n]\!]$ has size $n+3$ and all other components have size smaller than $n+2$, the depth of the ambient arena being $n+2$. The function $N_d(n,p)$ being monotonically increasing in all its parameters we have the following inequalities for $3 \leq d \leq min(n-1,p)$, both bounds making sense for all programming languages containing the simply-typed λ-calculus and whose terms can be interpreted as bounded strategies.

$$2^2_{d-2} \leq N_d(n,p) \leq 2^{n(p+1)}_{d-2}$$

Note that from this we can deduce bounds for $N(n,p)$, when we have no information on the depth of the ambient arena. Indeed, we always have $d \leq 2n$ and $d \leq 2p+1$ because a pointer chain in a play is visible by both players. Thus, $N(n,p) = N_{min(2n,2p+1)}(n,p)$.

[8] Our experiments with pointer structures and agents confirmed indeed that the possibility to use non-innocent behaviour does allow significantly longer plays.

4.3 Application to Head Linear Reduction

Earlier works on game semantics [8] suggest that in every games model of a programming language lies a hidden notion of linear reduction, head linear reduction when modelling call-by-name evaluation: this is the foundation for our claim that our game-theoretic result is really about the length of execution in programming languages whose terms can be described as bounded strategies. Of course it requires some work to interface execution in these programming languages to our game-theoretic results, and part of this work has to be redone in each case. To illustrate this, we now describe how to extract from our results a theorem about the length of head linear reduction sequences in simply-typed λ-calculus. For the formal definition of head linear reduction, the reader is directed to [9]. If S is a λ-term then the **spinal height** of S is the quantity $sh(S)$ defined by induction as $sh(x) = 1$, $sh(\lambda x.S) = sh(S)$ and $sh(ST) = max(sh(S), sh(T) + 1)$; when S is a $\beta\eta$-normal form, $sh(S)$ is nothing but the height of its Böhm tree. The **height** of S is the subtly different quantity[9] $h(S)$ defined by $h(x) = 1$, $h(\lambda x.M) = h(M)$ and $h(MN) = max(h(M), h(N)) + 1$. Finally, the **level** of a type $lv(A)$ is defined by $lv(\bot) = 0$ and $lv(A \rightarrow B) = max(lv(A) + 1, lv(B))$ and the **degree** $g(S)$ of a term is the maximal level of the type of all subterms of S.

A **game situation** [5] is the data of λ-terms $S : A_1 \rightarrow \ldots \rightarrow A_p \rightarrow B$ and $T_1 : A_1, \ldots T_p : A_p$ in η-long β-normal form, and we are interested in the term $ST_1 \ldots T_p$. Our game-theoretic results apply immediately to game situations, because of the connection between game-theoretic interaction and head linear reduction [8]: if $N(ST_1 \ldots T_p)$ denotes the length of the head linear reduction chain of $ST_1 \ldots T_p$, then we have $N(ST_1 \ldots T_p) \leq N_d(n, p)$ where d is the depth of the arena corresponding to $A \rightarrow B$, n is the size of $[\![S]\!]$ and p is the maximal size of all of the $[\![T_i]\!]$. But since S and T_i are already in η-long β-normal form, we have $|[\![S]\!]| = sh(S)$ and $|[\![T_i]\!]| = sh(T_i)$. Thus, we conclude that in the case of a game situation we have:

$$N(ST_1 \ldots T_p) \leq 2^{sh(S)(max_i(sh(T_i))+1)}_{max_i lv(A_i)-1}$$

Outside of game situations, it is less obvious to see how our results apply. The more elegant approach would be probably to extend the connection between head linear reduction and game semantics to *revealed* game semantics, which would give the adequate theoretical foundations to associate an agent to any η-long λ-term. Without these tools, we can nonetheless apply the following hack. Suppose we have a λ-term S. The idea is to "delay" all redexes, replacing each redex $(\lambda x.S)T$ of type $A \rightarrow B$ in S with $y_{A,B}(\lambda x.S)T$, where we add a new symbol $y_{A,B} : (A \rightarrow B) \rightarrow A \rightarrow B$ for each pair (A, B). We iterate this operation until we reach a β-normal λ-term S^t, which satisfies $sh(S^t) \leq h(S)$. We then expand S^t to its η-long form $\eta(S^t)$, which satisfies $sh(\eta(S^t)) \leq sh(S^t)+g(S^t) \leq h(S)+g(S)+1$.

[9] One can easily prove that on closed terms, it is always less than the more common notion of height defined as $h(x) = 0$, $h(\lambda x.S) = 1 + h(S)$ and $h(ST) = max(h(S), h(T)) + 1$, for which our upper bound consequently also holds.

We consider now the term $(\lambda y_1 \dots y_p.\eta(S^t))ev_1 \dots ev_p$, where each y_i binds one of the new symbols $y_{A,B}$, and $ev_i : (A \to B) \to A \to B$ is the (η-long form of) the corresponding evaluation λ-term. We recognise here a game situation, whose head linear reduction chain is necessarily longer than for S (we have only added steps due to the delaying of redexes and η-expansion). Using the inequality above for game situations, we conclude:

$$N(S) \leq 2_{g(S)}^{(h(S)+g(S)+1)(g(S)+1)}$$

Of course this bound can very likely be improved, since this approach (delaying the cuts) artificially increases the depth of redexes. If head linear reduction outside of game situations could be formally connected to the reduction of general agents (which would be expected), we believe the height of the tower would be one less. In any case, the complexity of head linear reduction is considerably less (one or two levels less on the tower of exponentials) than the complexity of strong reduction [3], which suggests that the price of strong reduction (w.r.t. head reduction) is largely higher than the price of linearity (w.r.t. usual substitution).

5 Conclusion and Future Work

Applied to head linear reduction on simply typed λ-calculus, our results show that the price of linearity is not as high as one might expect. The bounds remain in $\mathcal{E}^4$, but are also significantly less than those for usual (strong) β-reduction.

A strength of our method is that it is not restricted to λ-calculus; the results should indeed immediately apply as well to similar notions of reduction on other total programming languages. Beyond ground type references and non determinism, there are also games model of call-by-value languages [2] generating pointer structures as well, thus this work should also provide bounds for the corresponding call-by-value linear reduction (*tail* linear reduction?). All the tools used here also can be extended to *non-alternating plays* [20], which suggests that this work could be used to give bounds to the length of reductions in some restricted concurrent languages.

We also believe *agents* are worth studying further. Their combinatorial nature and their connection to execution of programs may prove interesting for the study of higher order systems with restricted complexity, such as *light* linear logics [11]. For instance, proofs typable in light systems may correspond to agents with some restricted behaviours, which would make them a valuable tool for the study of programming languages with implicit complexity.

Acknowledgements. This work was partially supported by the French ANR project CHOCO. The author also would like to thank Fabien Renaud for interesting discussions on related subjects.

References

1. Abramsky, S., McCusker, G.: Linearity, Sharing and State: a Fully Abstract Game Semantics for Idealized Algol with active expressions (1997)
2. Abramsky, S., McCusker, G.: Call-by-value games. In: Nielsen, M., Thomas, W. (eds.) CSL 1997. LNCS, vol. 1414, pp. 1–17. Springer, Heidelberg (1998)
3. Beckmann, A.: Exact bounds for lengths of reductions in typed λ-calculus. Journal of Symbolic Logic 66(3), 1277–1285 (2001)
4. Blum, W.: Thesis fascicle: Local computation of β-reduction. PhD thesis, University of Oxford (2008)
5. Clairambault, P.: Logique et Interaction: une Étude Sémantique de la Totalité. PhD thesis, Université Paris Diderot (2010)
6. Clairambault, P., Harmer, R.: Totality in arena games. Annals of Pure and Applied Logic (2009)
7. Coquand, T.: A semantics of evidence for classical arithmetic. Journal of Symbolic Logic 60(1), 325–337 (1995)
8. Danos, V., Herbelin, H., Regnier, L.: Game semantics and abstract machines. In: 11th IEEE Symposium on Logic in Computer Science, pp. 394–405 (1996)
9. Danos, V., Regnier, L.: How abstract machines implement head linear reduction (2003) (unpublished)
10. de Bruijn, N.G.: Generalizing Automath by means of a lambda-typed lambda calculus. Mathematical Logic and Theoretical Computer Science 106, 71–92 (1987)
11. Girard, J.-Y.: Light linear logic. Inf. Comput. 143(2), 175–204 (1998)
12. Greenland, W.: Game semantics for region analysis. PhD thesis, University of Oxford (2004)
13. Harmer, R.: Innocent game semantics. Lecture notes (2004-2007)
14. Harmer, R., Hyland, M., Melliès, P.-A.: Categorical combinatorics for innocent strategies. In: IEEE Symposium on Logic in Computer Science, pp. 379–388 (2007)
15. Harmer, R., McCusker, G.: A fully abstract game semantics for finite nondeterminism. In: IEEE Symposium on Logic in Computer Science, pp. 422–430 (1999)
16. Hyland, M., Luke Ong, C.-H.: On full abstraction for PCF: I, II and III. Information and Computation 163(2), 285–408 (2000)
17. Krivine, J.-L.: Un interpréteur du λ-calcul (1985) (unpublished)
18. Dal Lago, U., Laurent, O.: Quantitative game semantics for linear logic. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 230–245. Springer, Heidelberg (2008)
19. Laird, J.: Full abstraction for functional languages with control. In: IEEE Symposium on Logic in Computer Science, pp. 58–67 (1997)
20. Laird, J.: A game semantics of the asynchronous π-calculus. In: Jayaraman, K., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 51–65. Springer, Heidelberg (2005)
21. Mascari, G., Pedicini, M.: Head linear reduction and pure proof net extraction. Theoretical Computer Science 135(1), 111–137 (1994)
22. McCusker, G.: Games and Full Abstraction for FPC. Information and Computation 160(1-2), 1–61 (2000)
23. Nakajima, R.: Infinite normal forms for the λ-calculus. In: λ-Calculus and Computer Science Theory, pp. 62–82 (1975)
24. Schwichtenberg, H.: Complexity of normalization in the pure typed lambda-calculus. Studies in Logic and the Foundations of Mathematics 110, 453–457 (1982)

Synchronous Game Semantics via Round Abstraction

Dan R. Ghica* and Mohamed N. Menaa

University of Birmingham, U.K.

Abstract. A synchronous game semantics—one in which several moves may occur simultaneously—is derived from a conventional (sequential) game semantics using a *round abstraction* algorithm. We choose the programming language Syntactic Control of Interference and McCusker's fully abstract relational model as a convenient starting point and derive a synchronous game model first by refining the relational semantics into a trace semantics, then applying a round abstraction to it. We show that the resulting model is sound but not fully abstract. This work is practically motivated by applications to hardware synthesis via game semantics.

1 Introduction

Concurrent computation can be broadly divided into two distinct classes, synchronous or asynchronous, depending on the nature of the communication between processes. The key distinction between the two is that in the former, we must consider the case when two events occur simultaneously whereas in the latter, it is impossible to ascertain that. Asynchronous communication is used when bounds on the time necessary for interaction cannot be guaranteed (e.g. distributed systems) or when time is intentionally abstracted (e.g. concurrent high-level programming languages), whereas synchronous communication is commonly used when time is an essential facet of the system (e.g. safety-critical systems or digital circuits).

One context in which the relation between synchrony and asynchrony has not been studied yet is that of game semantics [1,12]. This would certainly be interesting for foundational reasons, but it is also interesting for practical reasons. Recent work of the first author [5] showed how game semantics can provide a semantics-directed approach to the compilation of higher-order programming languages into digital circuits. Compiling into asynchronous circuits comes naturally, as game models of concurrency are asynchronous [10]. Compilation to synchronous circuits, however, presents substantial technical challenges which must be addressed separately.

The canonical encodings expressed in *loc. cit.* are unsuitable for practical purposes if what is desired is not just a correct encoding, but a correct *low-latency* encoding. In the context of synchronous languages, the concept of time

* Supported by EPSRC Advanced Research Fellowship EP/D070880/1.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 350–364, 2011.

is important as "steps" of synchronised events can be counted. For practical reasons, we are interested in encodings in which the delay, expressed as number of steps, is reduced.

In a seminal paper, Alur and Henzinger describe a general method for reducing latency based on a specification languages called *Reactive Modules* [2]. This class of techniques, called *round abstraction*, allows an arbitrary number of computational steps to be viewed as a single macro-step called a *round*. This is an "abstraction" in the sense of abstract interpretation [4] because timing information about the sequencing of events inside each round is lost. However, if we are to interpret round-abstracted processes as having computational significance they correspond to synchronous processes in which all events in a round are simultaneous. Thus, round abstraction gives us a way to construct synchronous processes by abstracting the behaviour of asynchronous processes.

The original formulation of round abstraction is monolithic and applies to whole systems, not addressing the issue of whether round-abstracted systems interact correctly. This problem was addressed by the authors in a recent paper by providing necessary criteria for the total correctness of round abstraction relative to composition [6].

Correctness of composition is an essential requirement in the formulation of game-semantic models, as is the case for any denotational models. The use of round abstraction in relating asynchronous and synchronous game models should also have the added benefit of allowing the reuse and adaptation of the many existing game models in the synchronous setting, an adaptation which should preserve soundness but not definability. In this paper, we focus on Syntactic Control of Interference (SCI) [17], an affine version of Idealized Algol [18]. Since it has a finite model for any term, while remaining remarkably expressive, SCI plays a special role in our main intended application to hardware compilation.

Contribution. In this paper, we derive a sound *synchronous* game semantics for the prototypical programming language Syntactic Control of Interference (SCI) [17] using the *round abstraction* methodology [2] enriched with total-correctness criteria [6]. This work will provide an initial platform for the game-semantic analysis of synchrony and will have applications to the correct and efficient compilation of higher-level programming languages into hardware via game semantics.

2 Basic Syntactic Control of Interference

2.1 Syntax

The programming language *Syntactic Control of Interference* (SCI) was introduced by Reynolds in order to simplify reasoning about imperative programs by restricting the way procedures interact with their arguments [17]. However, interest in SCI went beyond its original stated reason as it raised several challenging technical issues regarding its type system and semantic model [15]. SCI was first given a fully abstract model using "object-based semantics" [16], although that was only proved later [13].

The semantic properties of SCI make it an interesting programming language for particular applications. On the one hand, SCI is an expressive higher-order imperative ("Algol-like") programming language and its syntactic restrictions rarely impinge on implementing useful algorithms. On the other hand, any term of SCI (with finite data types) can be given a finite-state model [9]. This makes it possible to automatically verify SCI programs using conventional finite-state model checking techniques [7], and also makes it possible to compile SCI programs directly into electronic circuits [5,10].

The types of SCI are given by the grammar $A ::= \mathsf{nat} \mid \mathsf{com} \mid \mathsf{var} \mid A \multimap A$. The typing judgements for terms have form $x_1 : A_1, \ldots, x_n : A_n \vdash M : A$, where x_i are distinct identifiers, A_i and A are types and M a term. We use $\Gamma, \Delta, \ldots$ to range over *contexts*, i.e. the (unordered) list of identifier type assignments above. Well-typed terms are defined by the following rules:

$$\frac{}{x : A \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \multimap B} \qquad \frac{\Delta \vdash M : A \multimap B \qquad \Gamma \vdash N : A}{\Gamma, \Delta \vdash MN : B}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \qquad \frac{\Gamma \vdash M : B}{\Gamma, x : A \vdash M : B}$$

SCI also uses the following constants:

$n : \mathsf{nat}$	natural number constants
$\mathsf{skip} : \mathsf{com}$	no-op
$\circledast : \mathsf{nat} \times \mathsf{nat} \multimap \mathsf{nat}$	arithmetic and arithmetic-logic operators
$; : \mathsf{com} \times A \multimap A$	sequential composition, $A \in \{\mathsf{com}, \mathsf{nat}, \mathsf{var}\}$
$\vert\vert : \mathsf{com} \multimap \mathsf{com} \multimap \mathsf{com}$	parallel command composition
$:= : \mathsf{var} \times \mathsf{nat} \multimap \mathsf{com}$	assignment
$! : \mathsf{var} \multimap \mathsf{nat}$	dereferencing
$\mathsf{while} : \mathsf{nat} \times \mathsf{com} \multimap \mathsf{com}$	iteration
$\mathsf{if} : \mathsf{nat} \times A \times A \multimap A$	selection, $A \in \{\mathsf{com}, \mathsf{nat}, \mathsf{var}\}$
$\mathsf{newvar} : (\mathsf{var} \multimap A) \multimap A$	local variable, $A \in \{\mathsf{com}, \mathsf{nat}\}$.

Note that the typing rules disallow sharing of identifiers between a function and its argument but allow sharing of identifiers in product formation. This means that programs using nested function application $(\cdots f(\cdots f(\cdots)\cdots)\cdots)$ and in particular general recursion do not type. Sequential operators such as arithmetic, composition, assignment, etc. can share arguments and conventional imperative programs, including iterative ones, type correctly. Non-sequential operators ($||$) have a type which prevents sharing of identifiers, hence race conditions, through the type system; note that this also makes it impossible to implement shared-memory concurrency.

As a final note on expressiveness, SCI can be generalised to a richer type system called *Syntactic Control of Concurrency* (SCC), which allows shared-memory concurrency [9], and it was recently shown that almost any recursion-free Concurrent Idealized Algol programs [8], barring pathological examples, can be automatically "serialised" into SCI via SCC [11].

2.2 Relational Semantics

The operational semantics of SCI is the standard one for an Algol-like language, i.e. call-by-name beta reduction with local-state variable manipulation. We will not present it here, but we will present the (fully-abstract) relational semantic model of McCusker [13,14].

A monoid A is a set UA closed under an associative binary operation $\cdot_A$ with identity e_A. The free monoid over set S, written S^*, is the monoid of finite strings of elements of S. Given a monoid A, we write αA for its underlying *alphabet*: the minimal set whose closure under $\cdot_A$ is UA. For a free monoid, this corresponds to the set of free generators.

The model is given in category **MonRel**, which is the dual category of $\mathbf{Mon}_{\mathcal{P}}$, the Kleisli category induced by the powerset monad on the category of monoids and monoid homomorphisms. This category can be described concretely as follows. Objects are monoids and maps $A \to B$ are relations R between the underlying sets UA and UB satisfying

homomorphism: $e_A \, R \, e_B$ and if $a_i \, R \, b_i$, $i = 1, 2$, then $a_1 a_2 \, R \, b_1 b_2$;
identity reflection: if $a \, R \, e_B$ then $a = e_A$;
decomposition: if $a \, R \, b_1 b_2$ then there exist $a_1, a_2 \in A$ such that $a_i \, R \, b_i$ for $i = 1, 2$ and $a = a_1 a_2$.

The definitions of identity and composition are standard. The category has finite products in the case of free monoids, given by $A^* \times B^* = (A + B)^*$, the free monoid over the disjoint union of sets A and B. The category also has an (affine) monoidal structure with the object tensor defined as set product and on morphisms as $R \otimes S : A \otimes C \to B \otimes D$, if $a \, R \, b$ and $c \, S \, d$ then $(a, c) \, R \otimes S \, (b, d)$. The monoid has an associated exponential construction $A \multimap B^*$ given only for free monoids as $(UA \times B)^*$.

A type is interpreted as the free monoid over the set of events at that type:

$$[\![\mathsf{nat}]\!] = \mathbf{N}^*, \quad [\![\mathsf{com}]\!] = \mathbf{1}^*, \quad [\![\mathsf{var}]\!] = [\![\mathsf{com}]\!]^\omega \times [\![\mathsf{nat}]\!] = \{w_n, n \mid n \in \mathbf{N}\}^*,$$
$$[\![A \multimap B]\!] = [\![A]\!] \multimap [\![B]\!] = ((\alpha A)^* \times \alpha B)^*, \; [\![A \times B]\!] = [\![A]\!] \times [\![B]\!] = (\alpha A + \alpha B)^*.$$

For expressions, the only observable event is that of producing a value of that type. For commands, the only event is termination so the set of observable events is the singleton set. For variables, we can either observe reading a certain value or writing a certain value. Note that we can always assume $[\![A]\!]$ and $[\![B]\!]$ are free monoids so that the definition of $\times$ and $\multimap$ make sense; this is clear from the base types. For readability, we refer to the alphabet of $[\![\mathsf{var}]\!]$ as *Var* and to the generators of $[\![\mathsf{com}]\!]^\omega$ as w_n, for some $n \in \mathbf{N}$.

A term $\Gamma \vdash M : A$ is interpreted by a map $[\![M]\!] : [\![A_1]\!] \otimes \cdots \otimes [\![A_n]\!] \to A$. Concretely, this is actually the same as a subset of $(\alpha A_1)^* \times \cdots \times (\alpha A_n)^* \times \alpha A$.

The lambda-calculus terms are interpreted in a canonical way using the monoidal and exponential structures. Constants are interpreted using special morphisms:

$$[\![n : \mathsf{nat}]\!] = \{n\} \subseteq \mathbf{N}$$
$$[\![\mathsf{skip} : \mathsf{com}]\!] = \{*\} \subseteq \mathbf{1}$$
$$[\![\circledast : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}]\!] = \{(\mathrm{inl}(m) \cdot \mathrm{inr}(n), p) \mid m, n, p \in \mathbb{N}, m \circledast n = p\}$$
$$\subseteq (\mathbf{N} + \mathbf{N})^* \times \mathbf{N}$$
$$[\![; : \mathsf{com} \times \mathsf{com} \to \mathsf{com}]\!] = \{(\mathrm{inl}(*) \cdot \mathrm{inr}(*), *)\} \subseteq (\mathbf{1} + \mathbf{1})^* \times \mathbf{1}$$
$$[\![|| : \mathsf{com} \to \mathsf{com} \multimap \mathsf{com}]\!] = \{(*, (*, *))\} \subseteq \mathbf{1}^* \times \mathbf{1}^* \times \mathbf{1}$$
$$[\![:= : \mathsf{var} \times \mathsf{nat} \to \mathsf{com}]\!] = \{(n \cdot w_n, *) \mid n \in \mathbf{N}\} \subseteq (\mathbf{N} + \mathit{Var})^* \times \mathbf{1}$$
$$[\![! : \mathsf{var} \to \mathsf{nat}]\!] = \{(n, n) \mid n \in \mathbf{N}\} \subseteq \mathit{Var} \times \mathbf{N}$$
$$[\![\mathsf{while} : \mathsf{nat} \times \mathsf{com} \to \mathsf{com}]\!] = \{(0 \cdot * \cdot 0 \cdot * \cdots 0 \cdot * \cdot n, *) \mid n \neq 0\} \subseteq (\mathbf{N} + \mathbf{1})^* \times \mathbf{1}$$
$$[\![\mathsf{if} : \mathsf{nat} \times \mathsf{com} \times \mathsf{com} \to \mathsf{com}]\!] = \{(0 \cdot \mathrm{inl}(*), *), (n \cdot \mathrm{inr}(*), *) \mid 0 \neq n \in \mathbf{N}\}$$
$$\subseteq (\mathbf{N} + (\mathbf{1} + \mathbf{1}))^* \times \mathbf{1}$$
$$[\![\mathsf{newvar} : (\mathsf{var} \multimap \mathsf{com}) \to \mathsf{com}]\!] = \{((v, *), *) \mid w_0 \cdot v \in gv(\mathit{Var})\} \subseteq (\mathit{Var} \times \mathbf{1})^* \times \mathbf{1}.$$

In the definition of local variable, we take the set $gv(\mathit{Var}) \subseteq \mathit{Var}^*$ to consist of "good variable" traces, i.e. those traces in which reads and writes follow the causal behaviour of variables. For example, it is legal for a trace in $gv(\mathit{Var})$ to have subsequences such as $w_7 \cdot 7$ or $6 \cdot 6$ but it is not legal to find subsequences such as $w_9 \cdot 5$ or $2 \cdot 3$.

McCusker proves this model to be fully abstract, i.e. equality in the denotational semantics is equivalent to contextual equivalence in the operational semantics [14].

Note that the SCI language of *loc. cit.* has no parallel composition primitive, but it can be defined as $|| : \mathsf{com} \multimap \mathsf{com} \multimap \mathsf{com} \overset{def}{=} \lambda c.\lambda d.c; d$. This is perhaps somewhat strange, and therefore, deserves a brief explanation. The eta-expansion of sequential composition changes the type from $\mathsf{com} \times \mathsf{com} \multimap \mathsf{com}$ to $\mathsf{com} \multimap \mathsf{com} \multimap \mathsf{com}$. The type of this term enforces non-interference between the two arguments, which means that they are not allowed to assign to the same variables. Consequently, the order of execution of the two arguments becomes irrelevant. Therefore, a compiler writer is perfectly entitled to implement a "sequential" composition operator with non-interfering arguments as a parallel composition operator, since the two are observationally equivalent. More precisely, $\lambda c.\lambda d.c; d \cong_{\mathsf{com} \multimap \mathsf{com} \multimap \mathsf{com}} \lambda c.\lambda d.d; c$.

3 Polarised Trace Semantics

McCusker notes that "*in this model, the observed behaviour in each variable is recorded separately; that is, there is no record of how interactions with the various variables are interleaved [...] The models based on game semantics refine the* *present model by breaking each event into two, a start and a finish, and recording the interleaving between actions, thereby overcoming this limitation.*" [14] In this section, we will do precisely that. This additional refinement of the model is required mainly in order to reintroduce the input-output (or, in the game-semantic lingo *Opponent-Proponent*) polarity on traces. Distinguishing between inputs and outputs is essential for our round abstraction to work correctly.

3.1 A Trace Model of Processes

We briefly introduce the trace model of concurrency of [6], which is a slight generalisation of the game models for concurrency [9,8]. A more extensive definition of the model is given in *loc. cit.*

Definition 1 (Signature). *A signature A is a label set together with a labelling function and a causality relation. Formally, it is a triple $\langle L_A, \pi_A, \vdash_A \rangle$ where L_A is a* set of port labels*, $\pi_A : L_A \to \{i, o\}$ maps each label to an* input/output polarity*, $\vdash_A$ is the transitive reduction of a partial order on L_A called* causality*, such that if $a \vdash b$ then $\pi_A(a) \neq \pi_A(b)$.*

Definition 2 (Locally synchronous trace). *A* trace *s over a signature A is a triple $\langle E_s, \preceq_s, \lambda_s \rangle$ where E_s is a finite set of events, $\preceq_s$ is a total pre-order on E_s and $\lambda_s : E_s \to A$ is a function mapping events to labels in A.*

The total pre-order signifies temporal precedence; for an element $e \in E$, if $\lambda(e) = a \in L_A$ we say that e is an *occurrence* of a. It is convenient to define

Definition 3 (Simultaneity). *Given a trace $\langle E, \preceq, \lambda \rangle$ over a signature A, we say that two events are* simultaneous*, written $e_1 \approx e_2$ iff $e_1 \preceq e_2$ and $e_2 \preceq e_1$.*

We will focus on a particular kind of traces which satisfy the following principle:

Definition 4 (Singularity). *Events of a trace $\langle E, \preceq, \lambda \rangle$ over signature A are* singular *iff for any two events $e_1, e_2 \in E$, if $e_1 \approx e_2$ and $\lambda e_1 = \lambda e_2$ then $e_1 = e_2$.*

The set of traces over signature A satisfying singularity is written $\Theta(A)$. An asynchronous trace is a trace where no two events are simultaneous:

Definition 5 (Asynchronous trace). *A trace $\langle E, \preceq, \lambda \rangle$ over signature A is* asynchronous *if $\preceq$ is a total order.*

Another, more technical condition, which also reflects the low-level nature of the systems we model is *serial causation.*

Definition 6 (Serial causation). *In a trace $\langle E, \preceq, \lambda \rangle$ over signature A we say that event $e' \in E$ is the* cause *of $e \in E$, written $e' \curvearrowleft e$, iff $\lambda e' \vdash \lambda e$ and for any e'' such that $e' \preceq e'' \preceq e$, $\lambda e' \nvdash \lambda e$.*

We define the *concatenation* of two traces at the level of rounds, i.e. all events of the second trace come after the events of the first trace.

Definition 7 (Concatenation). *The* concatenation *of two traces $s = \langle E, \preceq_s, \lambda_s \rangle$, $t = \langle F, \preceq_t, \lambda_t \rangle$, denoted by $s \cdot t$, is the trace defined by the triple $\langle E + F, \preceq_s + \preceq_t + (E \times F), \lambda_s + \lambda_t \rangle$.*

A *process* σ over signature A is a prefix-closed, under concatenation, set of locally synchronous traces over A. An *asynchronous process* has to satisfy some further saturation conditions:

Definition 8 (Asynchronous process). *A process σ is asynchronous iff whenever $s \in \sigma$ and $s' \lesssim s$ then $s' \in \sigma$, where $\lesssim$ is the least reflexive and transitive relation such that*

1. *(a) If $\pi e = i$ then $s' = s_0 \cdot e \cdot s_1 \cdot s_2$ and $s = s_0 \cdot s_1 \cdot e \cdot s_2$, or*
   (b) If $\pi e = o$ then $s' = s_0 \cdot s_1 \cdot e \cdot s_2$ and $s = s_0 \cdot e \cdot s_1 \cdot s_2$
2. *There exists a permutation $\phi : E_s \simeq E_{s'}$ so that for any events such that $e_1 \curvearrowleft_s e_2$ we have $\lambda_s e_i = (\lambda_{s'} \circ \phi)(e)$ and $\phi e_1 \curvearrowleft_{s'} \phi e_2$.*

In this paper, we will work with asynchronous processes that are well-behaved in the following sense.

Definition 9 (Serial reactivation). *An asynchronous process $\sigma : A \to B$ satisfies* serial reactivation *if for any trace s in σ which records two consecutive initial events, the first event must cause a B-event before the second occurs,*

$$\text{if } e, e', \in E_s \text{ and } \lambda_s(e), \lambda_s(e') \in I_B \text{ and } e \preceq_s e' \text{ then}$$
$$(\exists e'' \in E_s)(\lambda_s(e'') \in L_B \text{ and } e \preceq_s e'' \preceq_s e' \text{ and } e \curvearrowleft_s e'')$$

Lemma 1 (Compositionality of serial reactivation). *If processes $\sigma : A \to B$ and $\tau : B \to C$ satisfy serial reactivation then so does their composition $\sigma;\tau$.*

We define signature $A \to B = \langle L_A + L_B, \overline{\pi}_A + \pi_B, \vdash_A + \vdash_B + I_B \times I_A \rangle$. Let $s \upharpoonright A$ be the trace obtained from s by deleting all events with labels not belonging to L_A. Composition of processes is defined similarly to game semantic composition, by synchronisation followed by hiding.

Definition 10 (Composition). *Let $\sigma : A \to B$ and $\tau : B \to C$ be two processes. Their interaction is $\sigma \lightning \tau = \{t \in \Theta(A + B + C) \mid t \upharpoonright A + B \in \sigma \wedge t \upharpoonright B + C \in \tau\}$. Their composition is $\sigma;\tau : A \to C = \big\{t \upharpoonright A + C \mid t \in \sigma \lightning \tau\big\}$.*

Theorem 1 ([6]). *Processes form a Closed Symmetric Monoidal Category* **Proc**. *Asynchronous processes form a Closed Symmetric Monoidal Category* **AsyProc**.

3.2 A Polarised Trace Model of SCI

The relational model of SCI is a full subcategory of **MonRel**, which we will call **MonRel**$_b$. We define a map G from **MonRel**$_b$ to the category of asynchronous processes **AsyProc**. For each object (free monoid) A^* in **MonRel**$_b$, $G(A^*)$ is a signature in **AsyProc**. We first give direct interpretations to the base types.

Each observable event b at base type is split into two: a *question* b^q and an *answer* b^a, according to the following: $*^q = r, *^a = d, n^q = q, n^a = n, w_n^q = w_n, w_n^a = ok$. Polarity and causality are recovered as follows.

- $G([\![\mathsf{com}]\!]) = \langle \{r, d\}, \{(r, i), (d, o)\}, \{(r, d)\}\rangle$
- $G([\![\mathsf{exp}]\!]) = \langle \{q, n\}, \{(q, i), (n, o)\}, \{(q, n)\}\rangle, n \in \mathbf{N}$
- $G([\![\mathsf{var}]\!]) = \langle\{w_n,ok,q,n\},\{(w_n,i),(ok,o),(q,i),(n,o)\},\{(w_n,ok),(q,n)\}\rangle, n \in \mathbf{N}$

For function types we set $G(A \multimap B)$ to be $G(A) \Rightarrow G(B)$. We also set both $G(A \times B)$ and $G(A \otimes B)$ to be $G(A) \otimes G(B)$.

For morphisms $f : A \to B$ in $\mathbf{MonRel}_b$, we first define a function $\ulcorner - \urcorner$ that maps each element in f to a set of traces. We will call the images produced by $\ulcorner - \urcorner$ trace *interpretations*. In the sequel, we use letters $a, b, \ldots$ to range over $\{*, n, w_n \mid n \in \mathbf{N}\}$. Let $X_A, Y_A \in \alpha A$ and $\overrightarrow{X}_A \in (\alpha A)^*$.

Our definition of interleaving of sequences is standard. $\epsilon \mathbin{|||} \epsilon = \{\epsilon\}$, $\epsilon \mathbin{|||} s = s \mathbin{|||} \epsilon = \{s\}$ and $x{\cdot}s \mathbin{|||} y{\cdot}t = x{\cdot}(s \mathbin{|||} y{\cdot}t) \cup y{\cdot}(x{\cdot}s \mathbin{|||} t)$. We write $|||S = s_1 \mathbin{|||} s_2 \mathbin{|||} \ldots \mathbin{|||} s_k$, $s_i \in S$. For sets of traces S, T, we have $S \mathbin{|||} T = \{s \mathbin{|||} t \mid s \in S \text{ and } t \in T\}$.

We will also make use of the following two auxiliary functions: given a relation element (X, Y)

$$\rho((X, Y)) = \rho(Y), \quad \rho(b) = b$$
$$\lambda((X, Y)) = \{X\} + \lambda(Y), \quad \lambda(b) = \emptyset$$

These are required in order to *deconstruct* the observables in the relational model to build their corresponding traces. Intuitively, ρ will be used to extract the observable event that corresponds to the initial question, whereas λ will collect a set of observables whose interpretations will be interleaved.

We know that each morphism corresponding to a term is a subset of a set of shape $A_1^* \times \cdots \times A_n^* \times B$. Let us first assume that B records a single observable, i.e. an element of αB. Note that B may be of function type. Every element of this subset has shape $(\overrightarrow{X}_{A_1}, \ldots, \overrightarrow{X}_{A_n}, Y_B)$, and is mapped to a set of traces as follows:

$$\ulcorner(\overrightarrow{X}_{A_1}, \ldots, \overrightarrow{X}_{A_n}, Y_B)\urcorner = \rho(Y_B)^q \cdot \left(\mathop{|||}_{i=0}^{n} \ulcorner \overrightarrow{X}_{A_i} \urcorner\right) \mathbin{|||} \left(|||\ulcorner\lambda(Y_B)\urcorner\right) \cdot \rho(Y_B)^a$$

On sequences, the translation map is applied element-wise:

$$\ulcorner \overrightarrow{X} \urcorner = \ulcorner X_1 \cdots X_m \urcorner = \ulcorner X_1 \urcorner \cdots \ulcorner X_m \urcorner.$$

Observables at ground types are interpreted as a sequence of question and answer

$$\ulcorner a \urcorner = a^q \cdot a^a$$

Now we consider the case of elements of shape $(\overrightarrow{X}_{A_1}, \ldots, \overrightarrow{X}_{A_n}, \overrightarrow{Y}_B)$, where B can record a sequence of observables. In **MonRel**, these sequences are built using the monoid operation. This is clear from the *homomorphism* property of **MonRel** in Sec. 2.2: if a relation has elements $(X, Y), (X', Y')$ then it also has element $(X \cdot X', Y \cdot Y')$. As traces, this is interpreted as follows

$$\ulcorner(X \cdot X', Y \cdot Y')\urcorner = \ulcorner(X, Y)\urcorner \cdot \ulcorner(X', Y')\urcorner$$

Note that, by the *decomposition* property, if a relation contains an element whose right projection is a sequence, then the relation must also contain elements whose right projections are members of the sequence. Finally, we show how tensored morphisms are mapped

$$\text{if } x \in f : A \to B \text{ and } y \in g : C \to D \text{ then } \ulcorner x \urcorner \,|||\, \ulcorner y \urcorner \subseteq \ulcorner f \otimes g \urcorner$$

The $\ulcorner - \urcorner$ function is lifted to sets of traces by applying it pointwise and taking the union $\ulcorner R \urcorner = \bigcup_{p \in R} \ulcorner p \urcorner$. This translation maps the special morphisms in **MonRel** representing SCI constants to trace models very similarly to their conventional game semantic interpretations, e.g. [8].

We can now use the above function to define a map from relations to asynchronous processes. For each morphism (relation) f in **MonRel**, $G(f)$ is the function $\ulcorner - \urcorner$ followed by prefix-closing the resulting set. Note that the resulting sets are also saturated by construction under certain event permutation, as required by the asynchronous model. This is because events are interleaved, save for those that are scheduled (i.e. given as a sequence) or causally related. In fact, interleaving was chosen as a default order for non-scheduled events because it is a simple way to ensure saturation.

Theorem 2. $G : \mathbf{MonRel}_b \to \mathbf{AsyProc}$ *is a faithful functor.*

Note. The polarised traces model we obtain from refining McCusker's relational model is not fully abstract as we do not characterise what traces are definable in the syntax. There are other ways of obtaining such models of SCI if full abstraction is not required. Such models were used in previous GoS work, either by defining them directly [5] or by deriving them from the fully abstract SCC model by setting all concurrency bounds to the unit value [10]. The fully abstract model of SCI with passive types [19] could also be used as a starting point. Since we are not aiming for full abstraction the choice between these models is based on a compromise between simplicity of presentation and loss of precision. In that sense, we found McCusker's relational model the most suitable: it is elegant, technically concise and its loss of precision when expressed as polarised traces raises no problems for us.

4 Round Abstraction

This section briefly describes the framework of round abstraction from [6] and introduces several technical definitions which are required in the sequel. This section is interesting only insofar as it details the construction of the synchronous polarised trace semantics which will be presented in the next section and may be skipped without loss of continuity. The reader who is interested in the technical details, along with examples to illustrates the (sometimes complex) technical definitions is advised to refer to the original paper.

Round abstraction was introduced by Alur and Henzinger as part of their specification language *Reactive Modules* [2]. It is a technique that introduces a multiform notion of computational step, allowing arbitrarily many events to be viewed as a discrete *round*. In a synchronous setting, it makes sense to think of the events of the same round as simultaneous, and of round abstraction as a latency-reducing optimisation. We briefly review the key concepts of [6].

Definition 11 (Round abstraction on traces). *Let $s = \langle E_s, \preceq_s, \lambda_s \rangle$, $t = \langle E_t, \preceq_t, \lambda_t \rangle$ be traces. We say that t* is a round abstraction of *s, written $s \sqsubseteq t$, if and only if $\langle E_s, \lambda_s \rangle$ and $\langle E_t, \lambda_t \rangle$ are ϕ-isomorphic and ϕ is monotonic relative to temporal ordering, i.e. for any $e, e' \in E_s$, if $e \preceq_s e'$ then $\phi(e) \preceq_t \phi(e')$.*

Round abstraction is lifted point-wise to processes in two steps. A *partial round abstraction* of a process is a process that only contains round-abstracted traces of the original process. A *total round abstraction* of a process is a process which is a partial round abstraction and all traces in the original process can be recovered. The partial round abstraction is guaranteed not to have junk traces while the total round abstraction is guaranteed not to lose any traces. In *loc. cit.* we show using simple examples that neither partial nor total round abstraction are preserved by process composition and we set up sufficient conditions on round abstractions so that total round abstraction is preserved. The condition on processes is called *compatibility* and on round abstraction is called *receptivity.*

Definition 12 (Compatibility). *Two processes $\sigma_1 : A_1 \to B, \sigma_2 : B \to A_2$ are said to be* compatible*, written $\sigma_1 \asymp \sigma_2$, if and only if for all $v \in \Theta(A_1, B, A_2)$ if $v \upharpoonright A_i, B \in \sigma_i$ and there is a permutation $p \in \Pi(v)$ such that $p \upharpoonright B, A_j \in \sigma_j$ then $v \upharpoonright B, A_j \in \sigma_j$, for $i, j \in \{1, 2\}$, $i \neq j$.*

We now introduce a weaker version of compatibility: instead of requiring asynchronous processes not to deadlock in composition, we instead say that if they do deadlock then their respective round abstractions deadlock in a similar way.

Definition 13 (Post-compatibility). *Round abstractions $\sigma_1' : A_1 \to B, \sigma_2' : B \to A_2$ of asynchronous processes σ_1, σ_2 respectively, are said to be* post-compatible*, written $\sigma_1' \bigcirc \sigma_2'$, iff $\sigma_1 \not\asymp \sigma_2$ (that is, there exist $v \in \Theta(A_1, B, A_2)$ and a permutation $p \in \Pi(v)$ such that $v \upharpoonright A_i, B \in \sigma_i$ and $p \upharpoonright B, A_j \in \sigma_j$ and $v \upharpoonright B, A_j \notin \sigma_j$) implies for all traces $v' \in \sigma_i', p' \in \sigma_j'$ if $v \upharpoonright A_i, B \sqsubseteq v'$ and $p \upharpoonright B, A_j \sqsubseteq p'$ then $v' \upharpoonright B \neq p' \upharpoonright B$, for $i, j \in \{1, 2\}$, $i \neq j$.*

Theorem 3 (Soundness). *For any two* post-compatible *round abstractions $\sigma' : A \to B$ and $\tau' : B \to C$, with original asynchronous processes σ, τ respectively, if $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ then $\sigma; \tau \sqsubseteq \sigma'; \tau'$.*

We now define a particular round abstraction, which fits our framework, and which is applicable to the polarised trace model of SCI. First, we must state the general definition for receptivity, which uses the concepts of total round abstraction and trace fusion.

Definition 14 (Total round abstraction). *Let $\sigma : A \to B$ be an asynchronous process and $\sigma' : A \to B$ be a process. We say that σ' is a* total round abstraction *of σ, written $\sigma \underset{\sim}{\sqsubset} \sigma'$ if and only if $\sigma \sqsubseteq \sigma'$ and for any $s \in \sigma$ there exist $s_0 \in \sigma$, $w \in \Theta(A, B)$ and $s' \in \sigma'$ such that $s_0 \lesssim s$ and $s_0 \cdot w \sqsubseteq s'$.*

Definition 15 (Trace fusion). *The* fusion *of two traces $s = \langle E, \preceq_s, \lambda_s \rangle$, $t = \langle F, \preceq_t, \lambda_t \rangle$, denoted by $s * t$, is the trace defined by the triple $\langle E + F, \preceq', \lambda_s + \lambda_t \rangle$, where $\preceq' = \preceq_s + \preceq_t + E \times F + \mathit{first}(t) \times \mathit{last}(s)$.*

Trace fusion differs from concatenation in that the last round of the first trace and the first round of the second trace are fused into a single round.

Definition 16 ([6]). *Let $\sigma : A$ be an asynchronous process. Process σ' is a* receptive round abstraction *of σ, written $\sigma \underset{\approx}{\sqsubset} \sigma'$, if and only if $\sigma \underset{\sim}{\sqsubset} \sigma'$ and for any distinct inputs i, i_1, i_2 and output o*

1. *if $s_0{\cdot}i_1{\cdot}i_2{\cdot}s_1 \in \sigma$ then there exist traces $s'_0{\bullet}i_1{\cdot}i_2{\bullet}s'_1$ and $s'_0{\bullet}i_1{*}i_2{\bullet}s'_1$ in σ',*
2. *if $s_0{\cdot}o{\cdot}i{\cdot}s_1 \in \sigma$ then there exist traces $s'_0{\bullet}o{\cdot}i{\bullet}s'_1$ and $s'_0{\bullet}o{*}i{\bullet}s'_1$ in σ',*
3. *if $t_0{\cdot}r_0{*}i_1{\cdot}i_2{*}r_1{\cdot}t_1 \in \sigma'$ and $t' = t_0{\cdot}r_0{*}i_1{*}i_2{*}r_1{\cdot}t_1$ is well formed then $t' \in \sigma'$,*
4. *if $t_0{\cdot}r_0{*}o{\cdot}i{*}r_1{\cdot}t_1 \in \sigma'$ and $t' = t_0{\cdot}r_0{*}o{*}i{*}r_1{\cdot}t_1$ is well formed then $t' \in \sigma'$.*

Each instance of $\bullet$ stands for concatenation $\cdot$ or fusion $*$ and $s_k \sqsubseteq s'_k$, $k \in \{0, 1\}$.

We can now define the particular round abstraction we shall use. This definition will take into account the fact that certain sets of labels correspond to types which are not allowed to interfere, courtesy of the type system.

Definition 17 (Noninterference). *We say that a pair of monoids A, B in* **MonRel** *are not interfering with respect to a type C, written $A \bowtie_C B$, whenever $C = A \circledast B$ or A occurs in X and B occurs in Y and $C = X \circledast Y$, $\circledast \in \{\multimap, \otimes\}$.*

Definition 18 (Partially Receptive Round Abstraction). *A process $\sigma' : G(A)$ is a partially receptive round abstraction of asynchronous process $\sigma : G(A)$, if and only if*

1. *if $s_0 \cdot a \cdot b \cdot s_1 \in \sigma$ such that $\lambda(a) \in G(X)$ and $\lambda(b) \in G(Y)$, and $X \not\bowtie_A Y$ then for all $s' \in \sigma'$ such that $s \sqsubseteq s'$ we have $a \not\preceq_{s'} b$.*
2. *in all other cases, σ' behaves like a receptive round abstraction of σ.*

This round abstraction interacts well with process composition.

Lemma 2. *For a pair of relations $f : A \to B$ and $g : B \to C$ in* $\mathbf{MonRel}_b$*, any of their partially receptive round abstractions $G(f)$ and $G(g)$ are post-compatible.*

Lemma 3 (Correctness). *If σ', τ' are respective partially receptive round abstractions of $\sigma : G(A) \to G(B)$ and $\tau : G(B) \to G(C)$ then $\sigma'; \tau'$ is a partially receptive round abstraction of $\sigma; \tau$.*

Lemma 4 (Injectivity). *Let σ', τ' be partially receptive round abstractions of $\sigma : G(A), \tau : G(A)$, respectively. If $\sigma' = \tau'$ then $\sigma = \tau$.*

5 Synchronous Game Semantics

We can now concretely present a synchronous interpretation of basic SCI by applying a particular round abstraction to the trace model obtained via the mapping G. Let us first define an *efficient* partially receptive round abstraction, which reduces the latency of the resulting traces.

Definition 19 (SCI round abstraction). *A process $\sigma' : G(A)$ is a SCI round abstraction of asynchronous process $\sigma : G(A)$ if and only if σ' is a partially receptive round abstraction of σ, and for all traces $s = s_0 \cdot a \cdot b \cdot s_1$ in σ, for all $s' \in \sigma'$ such that $s \sqsubseteq s'$ we have that if a and b are distinct outputs then $a \approx_{s'} b$ and if a is an input and b is an output then $a \approx_{s'} b$. In both cases, s' must be well formed.*

Note that well-formedness implies the traces respect singularity, i.e., making a and b simultaneous is forbidden if it results in a trace with two occurrences of the same label. It also follows from the above definition that SCI round abstraction is a function as the result is unique.

In the sequel, we will write $[\![-]\!]_r$ for the relational semantics, $[\![-]\!]_t$ for the trace semantics induced by G and finally, $[\![-]\!]_s$ for the synchronous semantics resulting from the application of SCI round abstraction.

The types of SCI are interpreted as the signatures resulting from mapping the corresponding monoids in $\mathbf{MonRel}_b$ via G. See Sec. 3.2 for details.

Terms $x_1 : A_1, \ldots, x_n : A_n \vdash M : A$ will be interpreted as a map

$$\bigotimes_{1 \leq i \leq n} [\![A_i]\!]_s \xrightarrow{[\![x_1 : A_1, \ldots, x_n : A_n \vdash M : A]\!]_s} [\![A]\!]_s$$

For the constants of SCI, we obtain the following interpretations where simultaneous events are written in angled brackets, $\bullet$ stands for either concatenation or fusion, and pc for prefix-closure as defined in [6].

$$\begin{aligned}
[\![n : \mathsf{nat}]\!]_s &= pc(\{\langle q, n \rangle\}) \\
[\![\mathsf{skip} : \mathsf{com}]\!]_s &= pc(\{\langle r, d \rangle\}) \\
[\![\circledast : \mathsf{nat}_1 \times \mathsf{nat}_2 \to \mathsf{nat}_3]\!]_s &= pc(\{\langle q_3, q_1 \rangle \bullet n_1 \cdot q_2 \bullet \langle m_2, p_3 \rangle \\
&\qquad\qquad \mid m, n, p \in \mathbb{N}, m \circledast n = p\}) \\
[\![; : \mathsf{com}_1 \times \mathsf{com}_2 \to \mathsf{com}_3]\!]_s &= pc(\{\langle r_3, r_1 \rangle \bullet d_1 \cdot r_2 \bullet \langle d_2, d_3 \rangle\}) \\
[\![! : \mathsf{var}_1 \to \mathsf{nat}_2]\!]_s &= pc(\{\langle q_2, q_1 \rangle \bullet \langle n_1, n_2 \rangle \mid n \in \mathbb{N}\}) \\
[\![:= : \mathsf{var} \times \mathsf{nat}_1 \to \mathsf{com}_2]\!]_s &= pc(\{\langle r_2, q_1 \rangle \bullet n_1 \cdot w_n \bullet \langle ok, d_2 \rangle \mid n \in \mathbb{N}\}) \\
[\![\mathsf{if} : \mathsf{nat} \times \mathsf{com}_1 \times \mathsf{com}_2 \to \mathsf{com}_3]\!]_s &= pc(\{\langle r_3, q \rangle \bullet 0 \cdot r_1 \bullet \langle d_1, d \rangle\}) \\
&\quad \cup pc(\{\langle r_3, q \rangle \bullet n \cdot r_2 \bullet \langle d_2, d \rangle \mid n \neq 0\}) \\
[\![\mathsf{while} : \mathsf{nat} \times \mathsf{com}_1 \to \mathsf{com}_2]\!]_s &= pc(\{\langle r_2, q \rangle \bullet 0 \cdot r_1 \bullet d_1 \cdot (q \bullet 0 \cdot r_1 \bullet d_1)^* \\
&\qquad\qquad \cdot q \bullet n \cdot r_1 \bullet \langle d_1, d_2 \rangle \mid n \neq 0\})
\end{aligned}$$

For parallel composition, the corresponding trace interpretation is given by

$$[\![\mathsf{par}]\!]_t = \{r_3.(r_1.d_1 \mid\mid\mid r_2.d_2).d_3\}$$

Through SCI round abstraction, we obtain the interpretation depicted by the automaton in Fig. 1, where each transition is labelled by simultaneous events.

Next, we discuss the case of the variable allocation primitive. Each element in $[\![\mathsf{var}]\!]_r$ consists of sequences of read and write actions. Through G, each element

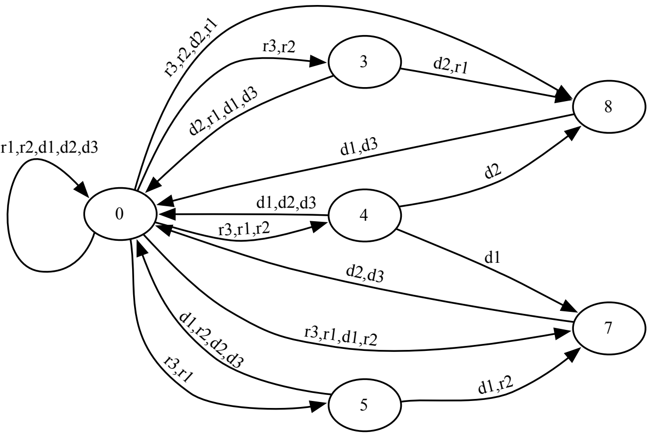


Fig. 1. A synchronous semantics for parallel composition

is mapped to a trace consisting of the corresponding sequence of read and write actions. Since $[\![\mathsf{var}]\!]_r$ is defined as the product of two monoids, one to read and one to write, the actions of these cannot be simultaneous. Good variable traces, representing proper stateful behaviour, have the property that each read action matches the previous write action. Their round abstractions consist of sequences where requests and acknowledgements are simultaneous, e.g.$\cdots \langle w_n, ok\rangle \cdot \langle q, n\rangle \cdot \langle q, n\rangle \cdots$, which we call *synchronous* good variable traces Var_s. The variable allocation primitive is hence given by

$$[\![\mathsf{newvar}_s : (\mathsf{var} \multimap \mathsf{com}_1) \to \mathsf{com}_2]\!] = pc(\{\langle r_2, r_1\rangle \bullet s \bullet \langle d_1, d_2\rangle \mid \langle w_0, ok\rangle \cdot s \in Var_s\})$$

The interpretation of the imperative fragment of the language is defined in the usual way, e.g. for sequential composition

$$[\![M; N]\!]_s = [\![M]\!]_s \otimes [\![N]\!]_s ; [\![;]\!]_s$$

The lambda calculus fragment is interpreted as follows

$$\begin{aligned}
[\![x : G(A) \vdash x : G(A)]\!]_s &= id_{G(A)} \\
[\![\Gamma, x : G(A) \vdash x : G(A)]\!]_s &= proj : [\![\Gamma]\!]_s \otimes [\![G(A)]\!]_s \to [\![G(A)]\!]_s \\
[\![\Gamma \vdash \lambda x^{G(A)}.M : A \multimap B]\!]_s &= \Lambda[\![M]\!] : [\![\Gamma]\!]_s \to ([\![G(A)]\!]_s \Rightarrow [\![G(B)]\!]_s) \\
[\![\Gamma, \Delta \vdash MN : G(B)]\!]_s &= ([\![M]\!]_s \otimes [\![N]\!]_s); eval : [\![\Gamma]\!]_s \otimes [\![\Delta]\!]_s \to [\![G(B)]\!]_s
\end{aligned}$$

Theorem 4 (Equational Soundness). *If $\Gamma \vdash M, N : A$ are terms satisfying $[\![M]\!]_s = [\![N]\!]_s$ then M and N are contextually equivalent.*

Proof. The soundness of the relational model entails if $[\![M]\!]_r = [\![N]\!]_r$ then M and N are contextually equivalent. By Thm. 2, we have if $[\![M]\!]_t = [\![N]\!]_t$ then $[\![M]\!]_r = [\![N]\!]_r$. Moreover, we have, by Lem. 4, that partially receptive round abstraction, of which SCI round abstraction is an instance, is injective; and by Prop. 3, that it is compositional, if $[\![M]\!]_s = [\![N]\!]_s$ then $[\![M]\!]_t = [\![N]\!]_t$. Putting all of the above together, if $[\![M]\!]_s = [\![N]\!]_s$ then M and N are contextually equivalent.

5.1 Discussion

Producing a synchronous game semantics proved to be a surprisingly subtle task which contradicted our initial intuitions. For example, in the definition of sequential composition $[\![; : \mathsf{com}_1 \times \mathsf{com}_2 \to \mathsf{com}_3]\!]_s = pc(\{\langle r_3, r_1\rangle \bullet d_1 \cdot r_2 \bullet \langle d_2, d_3\rangle\})$, a one time step delay is introduced between the Opponent playing d_1, corresponding to the termination of the first argument, and the Proponent playing r_2, initiating execution of the second argument. This is contrary to our initial expectations that the two moves should be simultaneous, because it would result in a lower latency strategy that consists of traces which do not violate singularity. However, it can be easily seen that such an aggressively round abstracted sequential composition would deadlock in a context such as $x := 1; x := 2$ because it would result in simultaneous write requests on the variable x.

Other strategies corresponding to sequential constants have similar such seemingly extraneous “wait” states, for the same reason. Also, other more aggressive naive optimisations can easily end up violating the various requirements for round abstraction to work correctly. Another example would be allowing the synchronous model of the memory cell to handle reads and writes simultaneously. This is of course possible from an implementation point of view but, as we explain in [6], would end up treating statements such as $x := !x + 1$ in a way that is not consistent with the original sequential meaning. Nevertheless, note that the strategy for parallel composition can be aggressively round-abstracted and it does not have to introduce wait steps because the arguments are non-interfering. Our indirect methodology is to be contrasted with the approach taken in languages such as Esterel, which posits a set of computational primitives but at the expense of well know semantic difficulties [3].

Finally, it is important to note that failure of full abstraction is the upshot of the success of round abstraction. In the synchronous model, $[\![\mathsf{skip}; \mathsf{skip}]\!]_s = pc\{r \cdot d\} \neq pc\{\langle r, d\rangle\} = [\![\mathsf{skip}]\!]_s$, because of the delay that sequential composition must introduce. On the other hand, $[\![\mathsf{skip} \,||\, \mathsf{skip}]\!]_s = pc\{\langle r, d\rangle\} = [\![\mathsf{skip}]\!]_s$.

6 Conclusion

We have seen how a sound synchronous game semantics for SCI can be derived using round abstraction. We first introduced a trace interpretation of McCusker’s fully abstract relational model by way of a faithful functor into **AsyProc**. Then, we defined a specific round abstraction, termed *partially-receptive*, that satisfies the correctness criteria outlined in [6] when applied to SCI. The salient feature of the new round abstraction that leads to the correctness and soundness results is that it forbids events corresponding to interfering types from being simultaneous.

GoS will benefit directly from applying these results in the construction of correct compilers to synchronous circuits.

References

1. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. 163(2), 409–470 (2000)
2. Alur, R., Henzinger, T.A.: Reactive Modules. Formal Methods in System Design 15(1), 7–48 (1999)
3. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics and implementation. Sci. Comput. Program. 19(2), 87–152 (1992)
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
5. Ghica, D.R.: Geometry of Synthesis: a structured approach to VLSI design. In: POPL, pp. 363–375 (2007)
6. Ghica, D.R., Menaa, M.N.: On the compositionality of round abstraction. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 417–431. Springer, Heidelberg (2010)
7. Ghica, D.R., Murawski, A.S.: Compositional model extraction for higher-order concurrent programs. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 303–317. Springer, Heidelberg (2006)
8. Ghica, D.R., Murawski, A.S.: Angelic semantics of fine-grained concurrency. Ann. Pure Appl. Logic 151(2-3), 89–114 (2008)
9. Ghica, D.R., Murawski, A.S., Ong, C.-H.L.: Syntactic control of concurrency. Theor. Comput. Sci. 350(2-3), 234–251 (2006)
10. Ghica, D.R., Smith, A.: Geometry of Synthesis II: From games to delay-insensitive circuits. In: MFPS XXVI (2010)
11. Ghica, D.R., Smith, A.: Geometry of Synthesis III: Resource management through type inference. In: POPL (2011)(forthcoming)
12. Hyland, J.M.E., Ong, C.-H.L.: On full abstraction for PCF: I, II, and III. Inf. Comput. 163(2), 285–408 (2000)
13. McCusker, G.: A Fully Abstract Relational Model of Syntactic Control of Interference. In: Bradfield, J.C. (ed.) CSL 2002 and EACSL 2002. LNCS, vol. 2471, pp. 247–261. Springer, Heidelberg (2002)
14. McCusker, G.: A graph model for imperative computation. Logical Methods in Computer Science 6(1) (2010), doi:10.2168/LMCS-6(1:2)2010
15. O'Hearn, P.W., Power, J., Takeyama, M., Tennent, R.D.: Syntactic control of interference revisited. Theor. Comput. Sci. 228(1-2), 211–252 (1999)
16. Reddy, U.S.: Global state considered unnecessary: An introduction to object-based semantics. Lisp and Symbolic Computation 9(1), 7–76 (1996)
17. Reynolds, J.C.: Syntactic control of interference. In: POPL, pp. 39–46 (1978)
18. Reynolds, J.C.: The essence of Algol. In: Proceedings of the 1981 International Symposium on Algorithmic Languages, pp. 345–372. North-Holland, Amsterdam (1981)
19. Wall, M.: Games for Syntactic Control of Interference. PhD thesis, University of Sussex (2004)

Freshness and Name-Restriction in Sets of Traces with Names

Murdoch J. Gabbay and Vincenzo Ciancia

Abstract. We use nominal sets (sets with names and binding) to define a framework for trace semantics with dynamic allocation of resources.

Using novel constructions in nominal sets, including the technical devices of *positive nominal sets* and *maximal planes*, we define notions of capture-avoiding composition and name-restriction on sets of traces with names.

We conclude with an extended version of Kleene algebras which summarises in axiomatic form the relevant properties of the constructions.

1 Introduction

Imagine a process evolving; every so often it may communicate with the outside world. In particular it may generate a new resource (allocate some memory; or perhaps create a cryptographic secret; or perhaps create a new channel name) and communicate that new resource.

Represent resources as *atoms* (set-theorists can think of *urelemente*; process-calculists can think of *names*). Then a model of the behaviour of our process is a trace (finite list of actions) that may contain atoms, and a model of the process is the set of all its possible traces. This is the *trace semantics* of computation [9], widely-used both in theory and application (e.g. the SPIN model-checker [17]).

Thus, a model of behaviour for processes with dynamic allocation, is sets of traces with atoms. But how then to represent dynamic allocation within this framework? After all, if a program outputs ab there is nothing in the string itself to tell us whether a, b, both a and b, or neither a nor b, has been 'created fresh'.

We propose to represent binding using a notion similar to that of nominal abstract syntax [15]; if e.g. ab is a possible trace of the process then the a in that trace is considered α-convertible when for all but finitely many a', $a'b$ is also in the set of possible traces of the process (note there is no binding primitive on the trace *itself*).

We will develop a model of binding based purely on sets of traces, such that:

- There is an operation νa that takes a behaviour (set of traces) X and creates another behaviour (also a set of traces) $\nu a.X$ in which a is α-renameable.
- Union of sets of traces is exactly union of sets.
- There is a notion of capture-avoiding composition $\circ$ such that $(\nu a.X) \circ \nu b.Y = \nu a.\nu b.(X \circ Y)$ if we choose a fresh for Y and b fresh for X.
  Thus $(\nu a.ab) \circ \nu b.b = \nu a.\nu b'.abb'$. Note that $ab \in \nu a.ab$ and $b \in \nu b.b$ but abb is *not* in their composition, because of capture-avoidance.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 365–380, 2011.

Finally we introduce axioms that regulate the behaviour of ν, by suggesting a notion of 'nominal' Kleene algebra [20].

We believe the constructions in this paper would be more generally applicable, so long as the semantics can be represented as sets and sensible notions of composition can be defined in some pointwise manner on their elements (in this paper it is sets of lists and list concatenation; see Definition 5.7).

Technical overview. The meat of this paper is some concrete calculations on nominal sets. The key technical facts are Theorems 3.14 and 3.16 and Proposition 4.6. The key definitions are Definitions 3.7, 4.5, and 5.7. The main theorem is Theorem 6.4.

Nominal sets were developed to represent syntax with binding; see [15] or a recent survey [13]. In this paper we use nominal sets to interpret sets of traces with atoms. The notions of names and free/bound names we use are exactly those from [15]; namely atoms and supporting set/freshness.

To the reader familiar with nominal sets, name-restriction $\nu a.X$ will be no surprise; Proposition 4.6 characterises it as a variation of atoms-abstraction $[a]x$ from [15] (see [13, Definition 3.8 and Lemma 3.13]). Readers familiar with presheaves will recognise this as a sets-based presentation of δ from e.g. [8] or [21]; see [16,7] for further discussion.

The difference, which is significant, is that X and $\nu a.X$ are of the same type (both are sets of traces); our name-restriction is not a monad in the sense of [22], though it does a similar job. More on this in the Conclusions.

Given that behaviour is modelled as 'just sets' and not wrapped up in an explicit abstraction, the challenge is that in composition $X \circ Y$, bound atoms in Y should somehow be detected and renamed to avoid capture with free atoms in X (see Remark 5.8), and vice-versa.

We use *maximal positive planes* as a foundational data structure for a notion of capture-avoiding language composition. Planes (Definition 3.3) are from [12] and were used to model capture-avoiding substitution. Positive planes are new (Definition 3.7), as is the connection with ν (Proposition 4.6). Arguably, planes and positive planes are as interesting as their application in this paper and we expect them to be useful in the future.

We conclude with an axiomatisation in the style of Kleene algebras and a proof of soundness (Section 6).

2 Nominal Preliminaries

More on these constructions in [15] or in a recent survey [13].

Definition 2.1. Fix a countably infinite set $\mathbb{A}$ of **atoms**. We use a *permutative convention* that $a, b, c, \ldots$ range over *distinct* atoms.

A **permutation** π is a bijection on atoms such that $nontriv(\pi) = \{a \mid \pi(a) \neq a\}$ is finite. We write id for the **identity** permutation such that $\mathrm{id}(a) = a$ for all a. We write $\pi' \circ \pi$ for composition, so that $(\pi' \circ \pi)(a) = \pi'(\pi(a))$. We write

$\pi^{\text{-}1}$ for inverse, so that $\pi^{\text{-}1} \circ \pi = \mathrm{id} = \pi \circ \pi^{\text{-}1}$. Write $(a\ b)$ for the **swapping** permutation that maps a to b and b to a and all other c to themselves.

π will range over permutations.

Definition 2.2. If $A \subseteq \mathbb{A}$ define $\mathit{fix}(A) = \{\pi \mid \forall a \in A.\pi(a) = a\}$. A, B, C will range over finite sets of atoms.

Definition 2.3. A **set with a permutation action** is a pair $\mathsf{X} = (|\mathsf{X}|, \cdot)$ of an **underlying set** and a group action of permutations which we write $\pi{\cdot}x$ (so $\mathrm{id}{\cdot}x = x$ and $\pi'{\cdot}(\pi{\cdot}x) = (\pi' \circ \pi){\cdot}x$).

Say finite $A \subseteq \mathbb{A}$ **supports** $x \in |\mathsf{X}|$ when $\pi{\cdot}x = x$ for all $\pi \in \mathit{fix}(A)$. Say $x \in |\mathsf{X}|$ has **finite support** when it is supported by some finite A.

A **nominal set** is a set with a permutation action whose every element has finite support. X will range over nominal sets and x will range over elements $x \in |\mathsf{X}|$ (that is, finitely-supported elements).

Proposition 2.4. *If $A', A \subseteq \mathbb{A}$ are finite and support x then so does $A' \cap A$.*

Definition 2.5. Define $\mathit{supp}(x) = \bigcap\{A \subseteq \mathbb{A} \mid A \text{ finite and support } x\}$. Call $\mathit{supp}(x)$ the **support** of x.
Write $a\#x$ when $a \notin \mathit{supp}(x)$ and call a **fresh for** x.

We know $\mathit{supp}(x)$ is well-defined in Definition 2.5 because by assumption at the end of Definition 2.3, x has some finite supporting set.

Theorem 2.6. *$\mathit{supp}(x)$ supports x.*

Corollary 2.7. *1. If $\pi(a) = a$ for all $a \in \mathit{supp}(x)$ then $\pi{\cdot}x = x$.*
2. If $\pi(a) = \pi'(a)$ for every $a \in \mathit{supp}(x)$ then $\pi{\cdot}x = \pi'{\cdot}x$.
3. $a\#x$ if and only if $\exists b.b\#x \wedge (b\ a){\cdot}x = x$.

Proposition 2.8. *$\mathit{supp}(\pi{\cdot}x) = \pi{\cdot}\mathit{supp}(x)$.*

Definition 2.9. Give $X \subseteq |\mathsf{X}|$ a **pointwise** permutation action given by $\pi{\cdot}X = \{\pi{\cdot}x \mid x \in X\}$.

Define $\mathit{pow}(\mathsf{X})$ to have elements $X \subseteq |\mathsf{X}|$ with the pointwise permutation action and finite support.

It is not hard to use Proposition 2.8 to verify that $\mathit{pow}(\mathsf{X})$ is a nominal set.

3 The Planes of a Set

From now until Section 6 we fix a nominal set X.

x, y, u, v will range over elements of $|\mathsf{X}|$ (i.e. $x, y, u, z{\in}|\mathsf{X}|$) and X, Y, Z, U, and V will range over finitely-supported subsets of $|\mathsf{X}|$ (i.e. $X, Y, Z, U, V{\in}|\mathit{pow}(\mathsf{X})|$).

3.1 Basic Theory of Planes

A subset $X \subseteq |\mathsf{X}|$ can be represented as a not-necessarily-disjoint union of orbits under certain subgroups of the permutation action. A canonical representation can be created using *planes* (Definition 3.3). A plane is an α-equivalence class of an element under simultaneous renaming of one or more of its atoms (cf. Lemma 3.2). For more on planes see [12, Subsection 3.3].

Definition 3.1. Define the **plane** $x\circlearrowleft_A$ by $x\circlearrowleft_A = \{\pi\cdot x \mid \pi \in \mathit{fix}(A)\}$.
We may omit brackets, writing e.g. $x\circlearrowleft_a$ for $x\circlearrowleft_{\{a\}}$ and $x\circlearrowleft_{a,b}$ for $x\circlearrowleft_{\{a,b\}}$.

Lemma 3.2. *If* $\pi \in \mathit{fix}(A)$ *then* $(\pi\cdot x)\circlearrowleft_A = x\circlearrowleft_A$.

Lemma 3.2 expresses α-convertibility for (x, A) as a choice of representative for the plane $x\circlearrowleft_A$, allowing us to rename $\mathit{supp}(x)\backslash A$. A plausible (if long) notation for $x\circlearrowleft_A$ is $\nu(\mathit{supp}(x)\backslash A).\{x\}$ where ν is atoms-restriction (cf. Definition 4.5).

Definition 3.3. Suppose $A \subseteq \mathbb{A}$ is finite. Write $u\circlearrowleft_A \propto Z$ when $u\circlearrowleft_A \subseteq Z$ and for every u' and A', $u\circlearrowleft_A \subseteq u'\circlearrowleft_{A'} \subseteq Z$ implies $u'\circlearrowleft_{A'} = u\circlearrowleft_A$.
We call the plane $u\circlearrowleft_A$ **maximal** in Z.

Example 3.4. 1. $a\circlearrowleft_\varnothing \propto \mathbb{A}$ (this is the only maximal plane in $\mathbb{A}$).
2. $a\circlearrowleft_{\{a\}} \propto \{a\}\cup(\mathbb{A}\times\mathbb{A})$, $(a, a)\circlearrowleft_\varnothing \propto \{a\}\cup(\mathbb{A}\times\mathbb{A})$, and $(a, b)\circlearrowleft_\varnothing \propto \{a\}\cup(\mathbb{A}\times\mathbb{A})$ (there are three maximal planes).
3. $a\circlearrowleft_{\{b\}} \propto \mathbb{A}\backslash\{b\}$ (this is the only maximal plane in $\mathbb{A} \setminus \{b\}$).
4. In Definition 3.3, quantification over A' is necessary. For example: $a\circlearrowleft_{\{a\}} \not\propto \mathbb{A} \setminus \{b\}$, but for no $A' \subsetneq \{a\}$ is $a\circlearrowleft_{A'} \subseteq \mathbb{A} \setminus \{b\}$ (the only choice for A' is $\varnothing$).

Remark 3.5. For a fixed Z and u there may be two distinct subsets $A \subseteq \mathit{supp}(Z)$ and $B \subseteq \mathit{supp}(Z)$ such that $u\circlearrowleft_A \propto Z$ and $u\circlearrowleft_B \propto Z$ (thus; A and B are minimal but not *least*). For example, if $Z = \{(x, y) \mid x = a \vee y = b\}$ and $u = (a, b)$ then $u\circlearrowleft_{\{a\}} \propto Z$ and $u\circlearrowleft_{\{b\}} \propto Z$ but $u\circlearrowleft_\varnothing \not\propto Z$.

Proposition 3.6. $Z = \bigcup\{u\circlearrowleft_A \mid u\circlearrowleft_A \propto Z\}$.

Proof. Planes in Z are ordered by subset inclusion. For each $x \in Z$, take some greatest element above $x\circlearrowleft_{\mathit{supp}(x)}$. Their union is Z.

3.2 Positive Planes

For the definition see Definition 3.7 and Example 3.8.

Planes do not have to be disjoint. This may be a surprise at first, since planes are orbits of an element under a permutation group and we are used to results stating that orbits are either equal or disjoint—but the groups could be different for different planes. For example, $(a, b)\circlearrowleft_{\{a\}}$ and $(a, b)\circlearrowleft_{\{b\}}$ are both positive planes, but $(a, b)\circlearrowleft_{\{a\}} \cap (a, b)\circlearrowleft_{\{b\}}$ is non-empty because it contains (a, b).

Theorem 3.16 expresses that a union of positive planes behaves as a coproduct in one respect: if a plane is included in a union of *positive* planes, then it is included in one of the planes that made up that union.

Thus, the interest of positive planes is that they need not be disjoint, but when we take their union it behaves in some ways like a coproduct. This fails if planes are not positive; see Corollary 3.18 and Example 3.19. The importance of this will become more apparent in the next section.

Definition 3.7. Recall the definition of $x\circlearrowleft_A$ from Definition 3.1 and the definition of $x\circlearrowleft_A \propto X$ from Definition 3.3.
Call $x\circlearrowleft_A$ **positive** when $A \subseteq \mathit{supp}(x)$.
Call X **positive** when every $x\circlearrowleft_A \propto X$ is positive.

Example 3.8. $\{a\} = a\circlearrowleft_{\{a\}}$ is positive. $\mathbb{A} \setminus \{a\} = b\circlearrowleft_{\{a\}}$ is not.

Lemma 3.9. *If $z\circlearrowleft_C \cap x\circlearrowleft_A \neq \varnothing$ and $A \subseteq C$ then $z\circlearrowleft_C \subseteq x\circlearrowleft_A$.*

Proof. Suppose $z' \in z\circlearrowleft_C \cap x\circlearrowleft_A$. Then $z' = \pi_z{\cdot}z = \pi_x{\cdot}x$ for some $\pi_z \in \mathit{fix}(C)$ and $\pi_x \in \mathit{fix}(A)$. Now $A \subseteq C$ so $\mathit{fix}(C) \subseteq \mathit{fix}(A)$. By Lemma 3.2 (twice) $z\circlearrowleft_C = z'\circlearrowleft_C = x\circlearrowleft_C \subseteq x\circlearrowleft_A$.

Proposition 3.10. *Suppose $u\circlearrowleft_A$ and $u'\circlearrowleft_{A'}$ are positive. Suppose $u \in u'\circlearrowleft_{A'}$.*
Then $u\circlearrowleft_A \subseteq u'\circlearrowleft_{A'}$ if and only if $A' \subseteq A$.

Proof. Suppose $u\circlearrowleft_A \subseteq u'\circlearrowleft_{A'}$ and $a \in A' \setminus A$. By assumption $A' \subseteq \mathit{supp}(u')$ so by Proposition 2.8, $a \in \mathit{supp}(v)$ for every $v \in u'\circlearrowleft_{A'}$. Also by assumption $A \subseteq \mathit{supp}(u)$ and so by Proposition 2.8, $a \notin \mathit{supp}((a'\ a){\cdot}u)$ for fresh a' (so $a' \notin A' \cup \mathit{supp}(u)$). Now $(a'\ a) \in \mathit{fix}(A)$ so $(a'\ a){\cdot}u \in u\circlearrowleft_A$, contradicting $u\circlearrowleft_A \subseteq u'\circlearrowleft_{A'}$.

Conversely, if $A' \subseteq A$ then we use Lemma 3.9.

Corollary 3.11. *Suppose $x\circlearrowleft_A \subseteq X$ and $x\circlearrowleft_A$ and X are positive. Then there is $A' \subseteq A$ with $x\circlearrowleft_A \subseteq x\circlearrowleft_{A'} \propto X$.*

Proof. Planes are ordered by subset inclusion, so there exist some x' and A' such that $x\circlearrowleft_A \subseteq x'\circlearrowleft_{A'} \propto X$. By assumption $x'\circlearrowleft_{A'}$ is positive. By Proposition 3.10 $A' \subseteq A$. We use Lemma 3.2 to conclude that $x'\circlearrowleft_{A'} = x\circlearrowleft_{A'}$.

Lemma 3.12. *If $x\circlearrowleft_A$ is positive then $\mathit{supp}(x\circlearrowleft_A) = A$.*

Proof. The left-to-right subset inclusion is from Lemma 3.2. Conversely suppose $a \in A$ and choose b fresh (so $b \notin A \cup \mathit{supp}(x)$). By positivity, $a \in \mathit{supp}(x)$. Using Proposition 2.8 it follows that $(b\ a){\cdot}x \notin x\circlearrowleft_A$, so $(b\ a){\cdot}(x\circlearrowleft_A) \neq x\circlearrowleft_A$. Thus $a \in \mathit{supp}(x\circlearrowleft_A)$.

Lemma 3.13. *If $u\circlearrowleft_A \propto Z$ and Z is positive then $A \subseteq \mathit{supp}(Z)$.*

Proof. Suppose $u\circlearrowleft_A{\subseteq}Z$ and $a{\in}A{\setminus}\mathit{supp}(Z)$. From Theorem 2.6 $(b\,a){\cdot}(u\circlearrowleft_A){\subseteq}Z$ for every $b{\notin}\mathit{supp}(Z)$; by group arguments $u\circlearrowleft_{A\setminus\{a\}}{\subseteq}Z$. Also $((b\,a){\cdot}u)\circlearrowleft_{(b\,a)\cdot A}{\propto}Z$ for every $b{\notin}\mathit{supp}(Z)$. Since $u\circlearrowleft_A$ is positive, so is $u\circlearrowleft_{A\setminus\{a\}}$. By Lemma 3.12 and Proposition 3.10 $u\circlearrowleft_A{\subsetneq}u\circlearrowleft_{A\setminus\{a\}}$, contradicting maximality of $u\circlearrowleft_A$.

Theorem 3.14. *If Z is positive then $supp(Z) = \bigcup\{A \mid u\circlearrowleft_A \propto Z\}$.*

Proof. The right-to-left inclusion is by Lemma 3.13. The left-to-right inclusion then follows from Proposition 3.6 and Lemma 3.12 using [13, Theorem 2.29].[1]

Remark 3.15. Theorem 3.14 fails if we remove the condition of maximality. For instance, $supp(\mathbb{A}) = \varnothing$ and $a\circlearrowleft_{\{a\}}$ is non-maximal in $\mathbb{A}$ for every a, but $supp(a\circlearrowleft_{\{a\}}) = \{a\}$ and $\bigcup\{supp(a\circlearrowleft_{\{a\}} \mid a \in \mathbb{A}\} = \mathbb{A} \neq \varnothing$.

Theorem 3.16. *Suppose I is some indexing set and for each $i \in I$, A_i is a finite set of atoms and x_i is some element. Suppose $x_i\circlearrowleft_{A_i}$ is positive for every $i \in I$ and $\bigcup A_i$ is finite. Then $z\circlearrowleft_C \subseteq \bigcup x_i\circlearrowleft_{A_i}$ implies $z\circlearrowleft_C \subseteq x_i\circlearrowleft_{A_i}$ for some i.*

Proof. We will show that $A_i \subseteq C$ for some $i \in I$ with $z \in x_i\circlearrowleft_{A_i}$; the result follows by Proposition 3.10. Suppose otherwise, so that $\forall i.\, z{\in}x_i\circlearrowleft_{A_i}{\Rightarrow}A_i{\not\subseteq}C$. Choose some π mapping $(\bigcup A_i) \setminus C$ to fresh atoms. For each i there are two possibilities: if $A_i \subseteq C$ then $z \notin x_i\circlearrowleft_{A_i}$ and by Lemma 3.2 $\pi{\cdot}z \notin x_i\circlearrowleft_{A_i}$; if $A_i \not\subseteq C$ then using Proposition 2.8 again $\pi{\cdot}z \notin x_i\circlearrowleft_{A_i}$. Yet by construction $\pi \in fix(C)$ so $\pi{\cdot}z \in z\circlearrowleft_C$, contradicting our assumption that $z\circlearrowleft_C \subseteq \bigcup x_i\circlearrowleft_{A_i}$.

Remark 3.17. Theorem 3.16 ensures that every maximal plane of a union of positive planes is one of the planes of that union.[2] We use this for example in Theorem 5.11 and Lemma 5.13.

Corollary 3.18. *Suppose $x_i\circlearrowleft_{A_i}$ is positive for every $i \in I$, and $\bigcup A_i$ is finite. Then $z\circlearrowleft_C \propto \bigcup x_i\circlearrowleft_{A_i}$ if and only if $z\circlearrowleft_C = x_i\circlearrowleft_{A_i}$ for some i.*

Proof. By Theorem 3.16 $z\circlearrowleft_C \subseteq x_i\circlearrowleft_{A_i}$ for some i. We use maximality of $z\circlearrowleft_C$.

Example 3.19. $U = (\mathbb{A}{\times}\mathbb{A}) \setminus \{(a, b)\}$ is not positive. $U \cup \{(a, b)\} = \mathbb{A} \times \mathbb{A}$; then $(a, b)\circlearrowleft_{\varnothing}$ is maximal in $\mathbb{A} \times \mathbb{A}$ but is not a subset of U or $\{(a, b)\}$. Thus, it is not possible to retrieve from a union of not-necessarily-positive planes a subcollection of planes that make up that union. Extending this paper to the 'negative' case is future work.

Corollary 3.20. *X is positive if and only if $X = \bigcup x_i\circlearrowleft_{A_i}$ for some set of positive $x_i\circlearrowleft_{A_i}$ such that $\bigcup A_i$ is finite.*

Proof. If X is positive the result follows taking $x_i\circlearrowleft_{A_i} \propto X$ and using Theorem 3.14. Conversely suppose $X = \bigcup x_i\circlearrowleft_{A_i}$ for some set of positive $x_i\circlearrowleft_{A_i}$ and suppose $\bigcup A_i$ is finite. We use Corollary 3.18.

[1] Positivity is sufficent but not necessary. It suffices to restrict to $u\circlearrowleft_A$ such that $\neg(supp(u) \subsetneq A)$. This excludes an artefact of representing planes as pairs (u, A) since if $supp(u) \subsetneq A$ and $supp(u) \subsetneq B$ then $u\circlearrowleft_A = u\circlearrowleft_B = u\circlearrowleft_{supp(u)}$. Since we concentrate on positive planes, where this cannot happen, we can ignore this.

[2] Contrast with $\mathbb{A} \setminus \{a\} = \bigcup\{b\circlearrowleft_{\{b\}} \mid b \in \mathbb{A} \setminus \{a\}\} = b\circlearrowleft_{\{a\}} \not\subseteq b\circlearrowleft_{\{b\}}$. Theorem 3.16 is inapplicable because $\bigcup\{supp(b\circlearrowleft_{\{b\}}) \mid b \in \mathbb{A} \setminus \{a\}\}$ is not finite.

4 ν-Restriction on Nominal Sets

We are now ready to define a notion of name-restriction on (positive) sets (Definition 4.5). By Theorem 3.16 the planes of $\nu a.U$ are 'the planes of U, with a taken out of the support of each'. Proposition 4.6 relates that to a notion of 'U, with a taken out of the support of U' which resembles the nominal atoms-abstraction from [15] (see [13, Definition 3.8 and Lemma 3.13] for a proof). The rest of this section proves some useful equalities involving name-restriction.

Definition 4.1. Suppose X is a set. Define $X{\circlearrowright}_A$ and $X{\circlearrowright}^A$ by:

$$\begin{aligned} X{\circlearrowright}_A &= \{\pi{\cdot}x \mid x \in X,\quad \pi \in \mathit{fix}(A)\} \\ X{\circlearrowright}^A &= \{\pi{\cdot}x \mid x \in X,\quad \pi \in \mathit{fix}(\mathit{supp}(X) \setminus A)\} \end{aligned}$$

Write $X{\circlearrowright}^a$ for $X{\circlearrowright}^{\{a\}}$. That is: $X{\circlearrowright}^a = \{\pi{\cdot}x \mid x{\in}X,\ \pi{\in}\mathit{fix}(\mathit{supp}(X){\setminus}\{a\})\}$.

Example 4.2. $(\mathbb{A}{\setminus}\{a\}){\circlearrowright}^a = \mathbb{A}$. For comparison, $(\mathbb{A}{\setminus}\{a\}){\circlearrowleft}_{\varnothing} = \{\mathbb{A}{\setminus}\{x\} | x{\in}\mathbb{A}\}$.

Lemma 4.3. *If $B \cap \mathit{supp}(X) = B' \cap \mathit{supp}(X)$ then $X{\circlearrowright}_B = X{\circlearrowright}_{B'}$.*

Lemma 4.4. *1. Suppose $\mathit{supp}(u) \cap B \subseteq \mathit{supp}(u) \cap A$. Then $u{\circlearrowleft}_A{\circlearrowright}^B = u{\circlearrowleft}_{A\setminus B}$.*
2. Suppose X is positive. Then $X{\circlearrowright}^B = \bigcup\{u{\circlearrowleft}_A{\circlearrowright}^B \mid u{\circlearrowleft}_A \propto X\}$.

Proof. The first part is by routine arguments using part 2 of Corollary 2.7.
For the second part, using Proposition 3.6 we can calculate that

$$X{\circlearrowright}^B = \bigcup\{u{\circlearrowleft}_A{\circlearrowright}_{\mathit{supp}(X)\setminus B} \mid u{\circlearrowleft}_A \propto X\}.$$

By assumption each $u{\circlearrowleft}_A \propto X$ is positive. By Lemma 3.12 $\mathit{supp}(u{\circlearrowleft}_A) = A$. By Lemma 4.3 $u{\circlearrowleft}_A{\circlearrowright}_{\mathit{supp}(X)\setminus B} = u{\circlearrowleft}_A{\circlearrowright}_{A\setminus B}$ and by definition this is equal to $u{\circlearrowleft}_A{\circlearrowright}^B$.

Definition 4.5. Define $\nu a.U = \bigcup\{u{\circlearrowleft}_{A\setminus\{a\}} \mid u{\circlearrowleft}_A \propto U\}$.

Proposition 4.6. *If U is positive then $\nu a.U = U{\circlearrowright}^a$.*

Proof. By Proposition 3.6 and Lemma 4.4.

Lemma 4.7. *If U is positive then so is $\nu a.U$.*

Proof. Suppose U is positive. For every $u{\circlearrowleft}_A \propto U$, by Theorem 3.14 $A \subseteq \mathit{supp}(U)$. The result follows by construction and by Corollary 3.20.

Lemma 4.8. *Suppose $u{\circlearrowleft}_A$ and $u'{\circlearrowleft}_{A'}$ are positive. Suppose $u{\circlearrowleft}_A \subseteq u'{\circlearrowleft}_{A'}$. Then $u{\circlearrowleft}_{A\setminus\{a\}} \subseteq u'{\circlearrowleft}_{A'\setminus\{a\}}$.*

Proof. Suppose $u\circlearrowleft_A \subseteq u'\circlearrowleft_{A'}$. So there exists some $\pi \in \mathit{fix}(A')$ such that $u = \pi{\cdot}u'$. Also, by Proposition 3.10 $A' \subseteq A$.

Consider some $\tau{\cdot}u \in u\circlearrowleft_{A\setminus\{a\}}$ where $\tau \in \mathit{fix}(A \setminus \{a\})$. Then $\tau{\cdot}u = (\tau \circ \pi){\cdot}u'$, and $\tau \circ \pi \in \mathit{fix}(A' \setminus \{a\})$.

Remark 4.9. We illustrate why positivity is necessary in Lemma 4.8. Note that $\{a\} = a\circlearrowleft_{\{a\}} \subseteq b\circlearrowleft_{\{c\}} = \mathbb{A}\setminus\{c\}$ but $\mathbb{A} = a\circlearrowleft_{\varnothing} \not\subseteq b\circlearrowleft_{\{c\}}$. However, $b\circlearrowleft_{\{c\}}$ is not positive because $\{c\} \not\subseteq \mathit{supp}(b) = \{b\}$.

Lemma 4.10. *Suppose U is positive. Then:*

1. *If $u\circlearrowleft_{A'} \propto \nu a.U$ then $a \notin A'$ and either $u\circlearrowleft_{A'} \propto U$ or $u\circlearrowleft_{A'\cup\{a\}} \propto U$.*
2. *If $u\circlearrowleft_A \propto U$ then $u\circlearrowleft_{A\setminus\{a\}} \subseteq \nu a.U$.*

Proof. For part 1, suppose $u\circlearrowleft_{A'} \propto \nu a.U$. By Lemma 4.7 $\nu a.U$ is positive. By Corollary 3.18 $u\circlearrowleft_{A'} = u\circlearrowleft_{A\setminus\{a\}}$ for some $u\circlearrowleft_A \propto U$.

For part 2, if $u\circlearrowleft_A \propto U$ then direct from Definition 4.5 $u\circlearrowleft_{A\setminus\{a\}} \subseteq \nu a.U$.

Proposition 4.11. *For* positive U*,* $\mathit{supp}(\nu a.U) = \mathit{supp}(U) \setminus \{a\}$.

Proof. From Theorem 3.14 and part 1 of Lemma 4.10.

Remark 4.12. Proposition 4.11 is not what it seems. To the reader familiar with nominal techniques it may look just like e.g. [15, Corollary 5.2] which is the corresponding result for $[a]x$.

But consider $U = (\mathbb{A} \times \mathbb{A}) \setminus \{(a, b)\}$. This has three maximal planes: $(c, c)\circlearrowleft_{\varnothing}$, $(c, b)\circlearrowleft_{a}$, and $(a, c)\circlearrowleft_{b}$, and is not positive. Also $\mathit{supp}(U) = \{a, b\}$, yet $\mathit{supp}(\nu a.U) = \varnothing \neq \{b\}$. The reason for this 'discrepancy' is the observation made in the Introduction that $\nu a.U$ and U have the same type, whereas $[a]U$ and U do not. Proposition 4.11 is different, and uses planes.

Corollary 4.13. *Suppose U is positive. Then $\nu a.U = U$ if and only if $a\#U$.*

Proof. Suppose $\nu a.U = U$. By Proposition 4.11 $a\#U$. Conversely if $a\#U$ then by Theorem 3.14 $a \notin A$ for every $x\circlearrowleft_A \propto U$ and $\nu a.U = U$ follows by construction.

Lemma 4.14. $X\circlearrowright^A\circlearrowright^B = X\circlearrowright^{A\cup B}$.

Proof. By routine properties of permutations, and Theorem 2.6.

Corollary 4.15. *Suppose U is positive. Then $\nu a.\nu b.U = \nu b.\nu a.U$.*

Proof. From Proposition 4.6 and Lemma 4.14.

Proposition 4.16. $\pi{\cdot}(X\circlearrowright^A) = (\pi{\cdot}X)\circlearrowright^{\pi\cdot A}$.

As a corollary, if $b\#X$ then $(b\,a){\cdot}(X\circlearrowright^a) = ((b\,a){\cdot}X)\circlearrowright^b$ and $\nu a.X = \nu b.(b\,a){\cdot}X$.

Proof. The first part is by calculations on permutations, or by equivariance [13, Corollary 4.6]. The corollary follows using Proposition 4.11 and Theorem 2.6.

5 Nominal Languages

Definition 5.1. Write $\mathbb{A}^*$ for the set of finite (possibly empty) strings of atoms.

k, l, m will range over elements of $\mathbb{A}^*$. We write kl for the concatenation of k and l. We write $[]$ for the empty string.

$\mathbb{A}^*$ is a nominal set with permutation action $\pi\cdot k = \pi(k_1)\dots\pi(k_n)$ where k_i is the ith element of k. Also, $supp(k) = \{k_1, \dots, k_n\}$ (the atoms in k).

A **nominal language** is a positive subset of $\mathbb{A}^*$. $\mathcal{K}$, $\mathcal{L}$ will range over languages.

An innocuous extension of Definition 5.1 is to use $(\mathbb{A} \cup \Sigma)^*$ for some Σ.

Definition 5.2. The union of two languages $\mathcal{K} \cup \mathcal{L}$ is their sets union.

Proposition 5.3. *Suppose X_i is positive for every $i \in I$. Suppose $\bigcup supp(X_i)$ is finite. Then $\bigcup X_i$ is positive. As a corollary, if $\mathcal{K}$ and $\mathcal{L}$ are languages then so is $\mathcal{K} \cup \mathcal{L}$.*

Proof. From Theorem 3.14 and Corollary 3.20.

Theorem 5.4. *$(\nu a.\mathcal{K}) \cup (\nu a.\mathcal{L}) = \nu a.(\mathcal{K} \cup \mathcal{L})$.*

Proof. Suppose $m\circlearrowleft_C \propto (\nu a.\mathcal{K}) \cup (\nu a.\mathcal{L})$. Using Corollary 3.18 either $m\circlearrowleft_C \propto \nu a.\mathcal{K}$ or $m\circlearrowleft_C \propto \nu a.\mathcal{L}$; suppose without loss of generality $m\circlearrowleft_C \propto \nu a.\mathcal{K}$. By part 1 of Lemma 4.10 $m\circlearrowleft_C = k\circlearrowleft_{A\setminus\{a\}}$ for $k\circlearrowleft_A \propto \mathcal{K}$. Then $m\circlearrowleft_C \subseteq \nu a.(\mathcal{K} \cup \mathcal{L})$ by part 2 of Lemma 4.10. The reverse inclusion follows by similar reasoning.

Definition 5.5. Write $u\circlearrowleft_A \propto^S Z$ when $u\circlearrowleft_A \propto Z$ and $supp(u) \cap S \subseteq A$. We may omit brackets, writing e.g. $u\circlearrowleft_A \propto^a Z$ for $u\circlearrowleft_A \propto^{\{a\}} Z$.

Remark 5.6. $ac\circlearrowleft_a \propto^b \{ax \mid x \in \mathbb{A}\setminus\{a\}\}$ and $ab\circlearrowleft_a \not\propto^b \{ax \mid x \in \mathbb{A}\setminus\{a\}\}$. An analogy with λ-terms: $\lambda c.ac$ avoids name-clash with b whereas $\lambda b.ab$ does not. Think of $u\circlearrowleft_A$ in Definition 5.5 as $\nu b_1.\dots.\nu b_n.\{u\}$ where $\{b_1, \dots, b_n\} = supp(u) \setminus A$. 'Avoiding clash with S' means choosing a representative u such that $b_i \notin S$ for $1 \leq i \leq n$. In Definition 5.7, the superscript $supp(\mathcal{L})$ and $supp(\mathcal{K}) \cup supp(k)$ are *generalised capture-avoidance* conditions. More on this in Remark 5.8.

Definition 5.7. Define **composition** of languages $\mathcal{K} \circ \mathcal{L}$ by:

$$\mathcal{K} \circ \mathcal{L} = \bigcup \{kl\circlearrowleft_{A\cup B} \mid k\circlearrowleft_A \propto^{supp(\mathcal{L})} \mathcal{K},\ \ l\circlearrowleft_B \propto^{supp(\mathcal{K}) \cup supp(k)} \mathcal{L}\}$$

Recall that here, kl denotes list concatenation. In words: $\mathcal{K} \circ \mathcal{L}$ is a capture-avoiding composition of the maximal planes of $\mathcal{K}$ and $\mathcal{L}$. For example, if $\mathcal{K} = \{a\}$ and $\mathcal{L} = \mathbb{A}$ then we can take $k = a$, $A = \{a\}$, $l = b$ and $B = \varnothing$ to calculate that $\mathcal{K} \circ \mathcal{L} = \{ax \mid x \in \mathbb{A}\setminus\{a\}\} = ab\circlearrowleft_a$.

Remark 5.8. The ‘definition’ $\{kl\circlearrowleft_{A\cup B} \mid k\circlearrowleft_A \propto \mathcal{K},\ \ l\circlearrowleft_B \propto \mathcal{L}\}$ would put the string aba in $(\nu b.ab) \circ \nu b.b$. This is undesired behaviour because $\nu b.b$ should avoid clash with the name a free in $\nu b.ab$. Similarly the ‘ordinary’ notion of composition $\{kl \mid k \in \mathcal{K},\ \ l \in \mathcal{L}\}$ does not avoid capture and delivers incorrect results.

Remark 5.9. The mention of $\mathit{supp}(k)$ in $l\circlearrowleft_B \propto^{\mathit{supp}(\mathcal{K})\,\cup\,\mathit{supp}(k)} \mathcal{L}$ tells l to avoid name-clash with atoms used in k; if we wrote $l\circlearrowleft_B \propto^{\mathit{supp}(\mathcal{K})} \mathcal{L}$ then composition would *deallocate* fresh names in k before executing l. Thus, Definition 5.7 does not put abb in $(\nu b.ab) \circ (\nu b.b)$, because of the mention of $\mathit{supp}(k)$.

Lemma 5.10. *$\mathcal{K} \circ \mathcal{L}$ is positive.*

Proof. By assumption $\mathcal{K}$ and $\mathcal{L}$ are positive. Suppose $k\circlearrowleft_A \propto \mathcal{K}$ and $l\circlearrowleft_B \propto \mathcal{L}$. By assumption $A \subseteq \mathit{supp}(k)$ and $B \subseteq \mathit{supp}(l)$ and it is a fact that therefore $A \cup B \subseteq \mathit{supp}(k) \cup \mathit{supp}(l) = \mathit{supp}(kl)$. Furthermore by Theorem 3.14 $A \subseteq \mathit{supp}(\mathcal{K})$ and $B \subseteq \mathit{supp}(\mathcal{L})$. The result follows by Corollary 3.20.

Theorem 5.11. *1. If $a\#\mathcal{L}$ then $(\nu a.\mathcal{K}) \circ \mathcal{L} = \nu a.(\mathcal{K} \circ \mathcal{L})$.*
2. If $b\#\mathcal{K}$ then $\mathcal{K} \circ \nu b.\mathcal{L} = \nu b.(\mathcal{K} \circ \mathcal{L})$.

Proof (Sketch proof). We consider only the first part. Suppose $m\circlearrowleft_C \propto \nu a.(\mathcal{K}\circ\mathcal{L})$. By definition and using part 1 of Lemma 4.10 and Corollary 3.18 there exist $k\circlearrowleft_A \propto^{\mathit{supp}(\mathcal{L})} \mathcal{K}$ and $l\circlearrowleft_B \propto^{\mathit{supp}(\mathcal{K})\,\cup\,\mathit{supp}(k)} \mathcal{L}$ such that $m\circlearrowleft_C = kl\circlearrowleft_{(A\cup B)\setminus\{a\}}$.

By part 1 of Lemma 4.10 $k\circlearrowleft_{A\setminus\{a\}} \propto^{\mathit{supp}(\mathcal{L})\setminus\{a\}} \nu a.\mathcal{K}$. By Theorem 3.14 $a \notin B$ so $(A\cup B)\setminus\{a\} = (A\setminus\{a\})\cup B$. It follows by definition that $kl\circlearrowleft_{(A\cup B)\setminus\{a\}} \subseteq (\nu a.\mathcal{K})\circ\mathcal{L}$.

The reverse inclusion follows by similar reasoning.

Lemma 5.12. *Suppose $\mathcal{K}$ and $\mathcal{L}$ are languages. Suppose $\mathit{supp}(\mathcal{K}) \cup \mathit{supp}(\mathcal{L}) \subseteq C$. Then $\mathcal{K} \circ \mathcal{L} = \{kl\circlearrowleft_{A\cup B} \mid k\circlearrowleft_A \propto^{C} \mathcal{K},\ \ l\circlearrowleft_B \propto^{C\,\cup\,\mathit{supp}(k)} \mathcal{L}\}$.*

Proof. By Proposition 2.8 and Lemma 3.2.

Lemma 5.13. $(\mathcal{K} \circ \mathcal{L}) \circ \mathcal{M} = \mathcal{K} \circ (\mathcal{L} \circ \mathcal{M})$.

Proof (Sketch proof). Set $C = \mathit{supp}(\mathcal{K}) \cup \mathit{supp}(\mathcal{L}) \cup \mathit{supp}(\mathcal{M})$. It is a fact that $\mathit{supp}(\mathcal{K} \circ \mathcal{L}) \subseteq C$ and $\mathit{supp}(\mathcal{L} \circ \mathcal{K}) \subseteq C$.[3]

By Corollary 3.18 and Lemma 5.12 if $n\circlearrowleft_D \propto (\mathcal{K}\circ\mathcal{L})\circ\mathcal{M}$ then $n\circlearrowleft_D = n'm\circlearrowleft_{D'\cup C}$ for some $n'\circlearrowleft_{D'} \propto^{C} \mathcal{K} \circ \mathcal{L}$ and some $m\circlearrowleft_C \propto^{C\,\cup\,\mathit{supp}(n')} M$. Similarly, $n'\circlearrowleft_{D'} = kl\circlearrowleft_{A\cup B}$ for some $k\circlearrowleft_A \propto^{C} \mathcal{K}$ and $l\circlearrowleft_B \propto^{C\,\cup\,\mathit{supp}(k)} \mathcal{L}$.

By renaming k and l appropriately we may assume that $n' = kl$. It is a fact that $\mathit{supp}(n') = \mathit{supp}(k) \cup \mathit{supp}(l)$. It follows that

$$(\mathcal{K}\circ\mathcal{L})\circ M = \bigcup\{klm\circlearrowleft_{A\cup B\cup C} \mid k\circlearrowleft_A \propto^{C} \mathcal{K},\ l\circlearrowleft_B \propto^{C\,\cup\,\mathit{supp}(k)} \mathcal{L},\ m\circlearrowleft_C \propto^{C\,\cup\,\mathit{supp}(k)\,\cup\,\mathit{supp}(l)} \mathcal{M}\}.$$

The result follows.

[3] We can deduce this by direct calculations or by construction and using Theorem 3.14.

Theorem 5.14. *If $\bigcup \mathit{supp}(\mathcal{L}_i)$ is finite then $\mathcal{K} \circ \bigcup \mathcal{L}_i = \bigcup(\mathcal{K} \circ \mathcal{L}_i)$ and $(\bigcup \mathcal{L}_i) \circ \mathcal{K} = \bigcup(\mathcal{L}_i \circ \mathcal{K})$.*

Proof. We consider only the first part; the proof of the second part is similar. Set $G = \mathit{supp}(\mathcal{K}) \cup \bigcup \mathit{supp}(\mathcal{L}_i)$. We prove two subset inclusions.

- *Proof of the left-to-right subset inclusion.* Choose some $k\circlearrowleft_A \propto^G \mathcal{K}$. Suppose $n\circlearrowleft_D \propto^{G\,\cup\,\mathit{supp}(k)} \bigcup \mathcal{L}_i$. From Theorem 3.16 $n\circlearrowleft_D \propto^{G\,\cup\,\mathit{supp}(k)} \mathcal{L}_i$ for some i. By Lemma 5.12 $kn\circlearrowleft_{A\cup D} \subseteq \bigcup(\mathcal{K} \circ \mathcal{L}_i)$.
- *Proof of the right-to-left subset inclusion.* Suppose $n\circlearrowleft_D \propto^G \bigcup(\mathcal{K} \circ \mathcal{L}_i)$. By Lemma 5.12 we may take $n = kl$ and $D = A \cup B$ for some $k\circlearrowleft_A \propto^G \mathcal{K}$ and some i and $l\circlearrowleft_B \propto^{G\,\cup\,\mathit{supp}(k)} \mathcal{L}_i$. Using Corollary 3.11 we can deduce that $l\circlearrowleft_B \subseteq l\circlearrowleft_{B'} \propto^{G\,\cup\,\mathit{supp}(k)} \bigcup \mathcal{L}_i$ for some $B' \subseteq B$. Since $A \cup B' \subseteq A \cup B$ we can conclude that $n\circlearrowleft_D \subseteq kl\circlearrowleft_{A\cup B'} \subseteq \mathcal{K} \circ \bigcup \mathcal{L}_i$.

Definition 5.15. Define $\mathcal{O} = \varnothing$ (the empty set). Define $\mathcal{I} = \{[]\}$ (recall from Definition 5.1 that $[]$ is the empty string).

Definition 5.16. Define $\mathcal{K}^0 = \mathcal{I}$ and $\mathcal{K}^{i+1} = \mathcal{K}^i \circ \mathcal{K}$. Define $\mathcal{K}^* = \bigcup_i \mathcal{K}^i$.

Lemma 5.17. *$\mathcal{O} \circ \mathcal{K} = \mathcal{O} = \mathcal{K} \circ \mathcal{O}$ and $\mathcal{I} \circ \mathcal{K} = \mathcal{K} = \mathcal{K} \circ \mathcal{I}$.*

Theorem 5.18. *$\mathcal{O}$ and $\mathcal{I}$ are languages. Also, the set of languages is closed under $\mathcal{K} \cup \mathcal{L}$, $\nu a.\mathcal{L}$, $\mathcal{K} \circ \mathcal{L}$, and $\mathcal{K}^*$.*

Proof. That $\mathcal{O}$ and $\mathcal{I}$ are languages is easy to verify. The case of $\mathcal{K} \cup \mathcal{L}$ is from Proposition 5.3; that of $\nu a.\mathcal{L}$ is from Lemma 4.7; that of $\mathcal{K} \circ \mathcal{L}$ is from Lemma 5.10.

$\mathit{supp}(\mathcal{K}^i) \subseteq \mathit{supp}(\mathcal{K})$ can be verified by calculations or by equivariance [13, Theorem 4.7]. The case of $\mathcal{K}^*$ follows using Lemma 5.10 and Proposition 5.3.

6 Nominal Kleene Algebra

We can use a nominal algebra style axiomatisation [14] to synthesise what we have proved so far as an extension of Kleene algebras (Definition 6.2 and Theorem 6.4). 'Nominal Kleene algebra' should be read tongue-in-cheek; we have no completeness proof like [20]. This is future work.

Definition 6.1. Call $x \in |\mathsf{X}|$ **equivariant** when $\mathit{supp}(x) = \varnothing$, thus $\forall \pi.\pi{\cdot}x = x$.

Call a function $f \in |\mathsf{X}| \to |\mathsf{Y}|$ **equivariant** when $\pi{\cdot}f(x) = f(\pi{\cdot}x)$ for all $x \in |\mathsf{X}|$ and all permutations π.

Definition 6.2. A **nominal Kleene algebra** is a tuple $\mathbb{X} = (|\mathbb{X}|, +, \cdot, {}^*, 0, 1, \mathcal{V})$ of:

- A **nominal carrier set** $|\mathbb{X}|$.

$$\begin{array}{rcl}
X+(Y+Z)=(X+Y)+Z & & X+Y=Y+X \\
X+0=X & & X+X=X \\
X(YZ)=(XY)Z & & \\
1X=X & & X1=X \\
X(Y+Z)=XY+XZ & & (X+Y)Z=XZ+YZ \\
0X=0 & & X0=0 \\
1+X(X^*)\leq X^* & & 1+X^*X\leq X^* \\
XY\leq Y\Rightarrow X^*Y\leq Y & & YX\leq Y\Rightarrow Y(X^*)\leq Y \\
 & & \\
a\#X\Rightarrow \quad \text{И}a.X=X & & \text{И}a.\text{И}b.X=\text{И}b.\text{И}a.X \\
\text{И}a.X+\text{И}a.Y=\text{И}a.(X+Y) & & a\#Y\Rightarrow \quad (\text{И}a.X)Y=\text{И}a.(XY) \\
b\#X\Rightarrow \quad \text{И}a.X=\text{И}b.(b\ a)\cdot X & & b\#X\Rightarrow \quad X(\text{И}b.Y)=\text{И}b.(XY)
\end{array}$$

Fig. 1. Axioms of nominal Kleene algebra

- Equivariant functions $+$ and $\cdot$ from $||\mathbb{X}||\times||\mathbb{X}||$ to $||\mathbb{X}||$. We usually omit $\cdot$, writing e.g. XY for $X\cdot Y$.
- An equivariant function * from $||\mathbb{X}||$ to $||\mathbb{X}||$.
- Equivariant elements $0\in||\mathbb{X}||$ and $1\in||\mathbb{X}||$.
- An equivariant function И from $\mathbb{A}\times||\mathbb{X}||$ to $||\mathbb{X}||$.

such that for all $X,Y,Z\in||\mathbb{X}||$ and all $a,b\in\mathbb{A}$ the conditions in Figure 1 hold.

The upper axioms are the standard axioms of a Kleene algebra [20]. Here (as standard) we write $r\leq s$ as shorthand for $r+s=s$. Note that these axioms are not purely equational (so Kleene algebra is not, depending on terminology, actually algebraic), and the class of Kleene algebras forms a *quasi-variety.* This will not matter to us in this paper.

The axioms on the lower lines describe behaviour of name-restriction.

Remark 6.3. Note that $\text{И}a.0=0$ and $\text{И}a.1=1$ follow from the axiom $a\#X\Rightarrow\text{И}a.X=X$, because it is a fact that $a\#0$ and $a\#1$.

Theorem 6.4. *Languages form a nominal Kleene algebra if we interpret $+$ as $\cup$, $\cdot$ as $\circ$, И as ν, and 0 and 1 as $\mathcal{O}$ and $\mathcal{I}$ respectively.*

Proof. We consider each axiom in turn:

- It is a fact that $\mathcal{K}\cup(\mathcal{L}\cup\mathcal{M})=(\mathcal{K}\cup\mathcal{L})\cup\mathcal{M}$, and $\mathcal{K}\cup\mathcal{L}=\mathcal{L}\cup\mathcal{K}$. It is also a fact that $\mathcal{K}\cup\mathcal{O}=\mathcal{K}$ and $\mathcal{K}\cup\mathcal{K}=\mathcal{K}$.
- $\mathcal{K}\circ(\mathcal{L}\circ\mathcal{M})=(\mathcal{K}\circ\mathcal{L})\circ\mathcal{M}$ by Lemma 5.13.
- $\mathcal{K}\circ(\mathcal{L}\cup\mathcal{M})=(\mathcal{K}\circ\mathcal{L})\cup(\mathcal{K}\circ\mathcal{M})$ and $(\mathcal{L}\cup\mathcal{M})\circ\mathcal{K}=(\mathcal{L}\circ\mathcal{K})\cup(\mathcal{L}\circ\mathcal{K})$ are by Theorem 5.14.
- $\mathcal{O}\circ\mathcal{K}=\mathcal{O}=\mathcal{K}\circ\mathcal{O}$ and $\mathcal{I}\circ\mathcal{K}=\mathcal{K}=\mathcal{K}\circ\mathcal{I}$ by Lemma 5.17.
- The four axioms for $\mathcal{K}^*$ follow using Theorem 5.14. To use some jargon, our denotation is **$*$-continuous** [19].
- If $a\#\mathcal{K}$ then $\nu a.\mathcal{K}=\mathcal{K}$ is by Corollary 4.13.
- $(\nu a.\mathcal{K})\cup(\nu a.\mathcal{K})=\nu a.(\mathcal{K}\cup\mathcal{L})$ is by Theorem 5.4.
- $(\nu a.\mathcal{K})\circ(\nu a.\mathcal{L})=\nu a.(\mathcal{K}\circ\mathcal{L})$ is by Theorem 5.11.
- $\nu a.\nu b.\mathcal{K}=\nu b.\nu a.\mathcal{K}$ is by Corollary 4.15.
- $b\#\mathcal{K}\Rightarrow\nu a.\mathcal{K}=\nu b.(b\ a)\cdot\mathcal{K}$ is by Proposition 4.16.

7 $\mathcal{L}\circ\mathbb{A}$ Is Equal to $\mathcal{L}\otimes\mathbb{A}$

Nominal sets have an *atoms tensor product* $\mathsf{X}\otimes\mathbb{A}$ ([27] or [13, Definition 9.27]) given by $\mathsf{X}\otimes\mathbb{A}=\{(x,a)\mid x\in\mathsf{X},\ a\in\mathbb{A},\ a\#x\}$. This has an obvious generalisation: $\mathsf{X}\otimes\mathsf{Y}=\{(x,y)\mid x\in\mathsf{X},\ y\in\mathsf{Y},\ \mathit{supp}(x)\cap\mathit{supp}(y)=\varnothing\}$. We can view $\circ$ as another (less obvious) generalisation of $\otimes$, as follows:

Proposition 7.1. *If $\mathit{supp}(\mathcal{K})=\varnothing$ then $\mathcal{K}\circ\mathbb{A}=\mathcal{K}\otimes\mathbb{A}$ and $\mathbb{A}\circ\mathcal{K}=\mathbb{A}\otimes\mathcal{K}$, where we treat $\mathcal{K}$ as a nominal set with underlying set itself.*

Thus, composition of languages generalises $\otimes$. $\text{-}\otimes\mathbb{A}$ is left-adjoint to atoms-abstraction $[\mathbb{A}]\text{-}$ [13, Theorem 9.30]. By Proposition 7.1, so is $\text{-}\circ\mathbb{A}$. It remains to investigate the futher properties of $\text{-}\circ\mathbb{A}$.

8 Conclusions

Name-generation has long been a motivation for nominal techniques.

Odersky in [24] and Pitts and Stark [25] studied name-generation, and this was in the background thinking of the first author's and Pitts's development of Fraenkel-Mostowski/nominal sets. FreshML included a name-generating construct [29] which was a precursor of Fernández and the first author augmenting nominal terms and nominal rewriting explicitly with name generation И$a.t$ [5]; Pitts added a similar construct $\nu a.t$ to system T [26]. The axioms for αa in Figure 1 are of the same family.

A very abstract semantic study of name-generation is the *abstractive functions* considered in [11]. This influenced [12], where much machinery used in this paper was introduced. Abramski *et al.* give a concrete games semantics to the nu-calculus in nominal sets [1]: ideas here and in [12] also appear there, including Definition 3.1 (see e.g. Definition 2.7 of [30]).

There exist denotations for dynamic allocation using atoms-abstraction $[\mathbb{A}]$-, typically written δ in presheaf presentations. Examples are coalgebraic semantics for the π-calculus using δ (see e.g. [6, Subsection 2.2] or [2, Subsection 5.2]), the name-generation monad of FreshML [28, $F_{\langle\langle\mathtt{name}\rangle\rangle T}$, page 38]. We can also include the $X\lceil Y$ construct of nominal games from [1], which is in the same spirit and used in similar ways. In these examples name-generation exists at its own distinct level; in programming terms this corresponds to carrying around an explicit context of known 'fresh' names.

νa (Definition 4.5) is different because it places binding on a level with union $\cup$ and composition $\circ$: a language $\mathcal{L}$ is just a set of traces, not under a monad and not a set of α-equivalence classes of sets of traces. Thus we must work harder because freshness must be 'decrypted', but this buys us an appealingly simple model. A language really *is* just a set—as in the classical case of regular languages, without names and binding. That explicit context of known 'fresh' names is not explicitly necessary in the mathematical models we build.

One can raise the question of decidability of equality and inclusion between (subclasses of) languages, and automata. To consider such questions we need to match the developments of this paper with an automata-theoretic counterpart.

One well-studied notion of finite automaton with names and allocation is *history-dependent (HD-)automata* [23]. The correspondence to coalgebras over presheaves/nominal sets is considered in [3]. Investigation of the languages of HD-automata and the link with *finite-memory automata* [18] has shown that HD-automata are still essentially finite-memory machines [4]. However, the finite-support property of nominal sets corresponds to an idea of 'finitely but unboundedly many'. In FreshML, a type system in [10] first tried to restrict generation of fresh names and later in [28] the programming language appeared without such restrictions but the denotation used a monad to keep track of generated fresh names. Similarly, we would expect acceptors for languages from Section 5 to either impose bounds on support (if they are to be finite), or to be in the style of e.g. pushdown automata.

Most recently, *fresh register automata* have also been proposed, explicitly as an automaton model of names and fresh name generation [31]. It remains to investigate these in connection with this work.

We note in Remark 5.9 a 'deallocating' variant of composition $\mathcal{K} \circ \mathcal{L}$. There is a rich design space here to be studied in future work.

Nominal sets have further structure. We can model when a process omits a name (e.g. 'junk' in the π-calculus; a channel name that is not emitted yet occurs in the syntax of the term) using a freshness constraint: $X_{\#a} = \{x \in X \mid a\#X\}$.

Note that ν is not the $\mathsf{И}$-quantifier introduced by the first author with Pitts in [15]. For instance, $X \subseteq \nu a.X$ is a fact, whereas $\phi(x) \Rightarrow \mathsf{И}x.\phi(x)$ is in general false. It *is* possible to define a version of $\mathsf{И}$ acting on languages, given by $\mathsf{n}a.X = \{x \in X \mid \mathsf{И}b.(b\ a){\cdot}x \in X\}$. We do not believe that n and ν are interdefinable and investigating them is future work.

Our models do not include negation; this is also future work.

Acknowledgements. The authors are grateful to Emilio Tuosto and three anonymous referees. The second author was supported by the VICI grant 639.073.501 of the Netherlands Organization for Scientific Research (NWO).

References

1. Abramsky, S., Ghica, D.R., Murawski, A.S., Ong, C.-H.L., Stark, I.D.B.: Nominal games and full abstraction for the nu-calculus. In: Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004), pp. 150–159. IEEE Computer Society Press, Los Alamitos (2004)
2. Bonsangue, M., Kurz, A.: Pi-calculus in logical form. In: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007), pp. 303–312. IEEE Computer Society Press, Los Alamitos (2007)
3. Ciancia, V., Montanari, U.: Symmetries, local names and dynamic (de)-allocation of names, Information and Computation (2010) (in press)
4. Ciancia, V., Tuosto, E.: A novel class of automata for languages on infinite alphabets, Tech. Report CS-09-003, University of Leicester, UK (2009)

5. Fernández, M., Gabbay, Murdoch J.: Nominal rewriting with name generation: abstraction vs. locality. In: Proceedings of the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP 2005), pp. 47–58. ACM Press, New York (July 2005)
6. Fiore, M., Moggi, E., Sangiorgi, D.: A fully-abstract model for the π-calculus (extended abstract). In: Proceedings of the 11th IEEE Symposium on Logic in Computer Science (LICS 1996), pp. 43–54. IEEE Computer Society Press, Los Alamitos (1996)
7. Fiore, M., Staton, S.: Comparing operational models of name-passing process calculi. Information and Computation 204(4), 524–560 (2006)
8. Fiore, M., Turi, D.: Semantics of name and value passing. In: Proceedings of the 16th IEEE Symposium on Logic in Computer Science (LICS 2001), pp. 93–104. IEEE Computer Society Press, Los Alamitos (2001)
9. Francez, N., Hoare, C.A.R., Lehmann, D.J., de Roever, W.P.: Semantics of nondeterminism, concurrency, and communication. Journal of Computer and System Sciences 19(3), 290–308 (1979)
10. Gabbay, Murdoch J.: A Theory of Inductive Definitions with alpha-Equivalence, Ph.D. thesis, University of Cambridge, UK (March 2001)
11. Gabbay, Murdoch J.: A General Mathematics of Names. Information and Computation 205(7), 982–1011 (2007)
12. Gabbay, Murdoch J.: A study of substitution, using nominal techniques and Fraenkel-Mostowski sets. Theoretical Computer Science 410(12-13), 1159–1189 (2009)
13. Gabbay, Murdoch J.: Foundations of nominal techniques: logic and semantics of variables in abstract syntax. Bulletin of Symbolic Logic (2010) (in press)
14. Gabbay, Murdoch J., Mathijssen, A.: Nominal universal algebra: equational logic with names and binding. Journal of Logic and Computation 19(6), 1455–1508 (2009)
15. Gabbay, Murdoch J., Pitts, A.M.: A New Approach to Abstract Syntax with Variable Binding. Formal Aspects of Computing 13(3-5), 341–363 (2001)
16. Gadducci, F., Miculan, M., Montanari, U.: About permutation algebras (pre)sheaves and named sets. Higher-Order and Symbolic Computation 19(2-3), 283–304 (2006)
17. Holzmann, G.J.: The spin model checker: Primer and reference manual. Addison-Wesley Professional, Reading (September 2003)
18. Kaminski, M., Francez, N.: Finite-memory automata. Theoretical Computer Science 134(2), 329–363 (1994)
19. Kozen, D.: On induction vs. $*$-continuity. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 167–176. Springer, Heidelberg (1982)
20. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Information and Computation 110(2), 366–390 (1994)
21. Miculan, M., Scagnetto, I.: A framework for typed HOAS and semantics. In: Principles and Practice of Declarative Programming, 5th International ACM SIGPLAN Symposium (PPDP 2003), pp. 184–194. ACM, New York (2003)
22. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)
23. Montanari, U., Pistore, M.: π-Calculus, Structured Coalgebras and Minimal HD-Automata. In: Nielsen, M., Rovan, B. (eds.) MFCS 2000. LNCS, vol. 1893, p. 569. Springer, Heidelberg (2000)

24. Odersky, M.: A functional theory of local names. In: Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL 1994), pp. 48–59. ACM Press, New York (1994)
25. Pitts, A.M., Stark, I.D.B.: Observable properties of higher order functions that dynamically create local names, or: What's new? In: Borzyszkowski, A.M., Sokolowski, S. (eds.) MFCS 1993. LNCS, vol. 711, pp. 122–141. Springer, Heidelberg (1993)
26. Pitts, A.M.: Nominal system T. In: Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2010), pp. 159–170. ACM Press, New York (January 2010)
27. Pitts, A.M., Gabbay, Murdoch J.: A Metalanguage for Programming with Bound Names Modulo Renaming. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 230–255. Springer, Heidelberg (2000)
28. Shinwell, M.R., Pitts, A.M.: On a monadic semantics for freshness. Theoretical Computer Science 342(1), 28–55 (2005)
29. Shinwell, M.R., Pitts, A.M., Gabbay, Murdoch J.: FreshML: Programming with Binders Made Simple. In: Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), vol. 38, pp. 263–274. ACM Press, New York (August 2003)
30. Tzevelekos, N.: Nominal game semantics. Ph.D. thesis, Oxford (2008)
31. Tzevelekos, N.: Fresh-register automata. In: Proceedings of the 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2011). ACM Press, New York (January 2011)

Polymorphic Abstract Syntax via Grothendieck Construction

Makoto Hamana

Department of Computer Science, Gunma University, Japan
hamana@cs.gunma-u.ac.jp

Abstract. Abstract syntax with variable binding is known to be characterised as an initial algebra in a presheaf category. This paper extends it to the case of polymorphic typed abstract syntax with binding. We consider two variations, second-order and higher-order polymorphic syntax. The central idea is to apply Fiore's initial algebra characterisation of typed abstract syntax with binding repeatedly, i.e. first to the type structure and secondly to the term structure of polymorphic system. In this process, we use the Grothendieck construction to combine differently staged categories of polymorphic contexts.

1 Introduction

It is well-known that first-order abstract syntax is modelled as an initial algebra [GTW76] in the framework of ordinary universal algebra. Because this algebraic characterisation cleanly captures various important aspects of syntax, such as structural recursion and induction principles, in terms of algebraic notions, it has been extended to more enriched abstract syntax: abstract syntax with *variable binding* [Hof99, FPT99], *simply-typed* abstract syntax with variable binding [Fio02, MS03, TP08], and *dependently-sorted* abstract syntax [Fio08]. These are uniformly modelled in the framework of categorical universal algebra in presheaf categories.

The solid algebraic basis of enriched abstract syntax has produced fruitful applications. The untyped case [FPT99] was applied to characterisations of second-order abstract syntax with metavariables [Ham04, Fio08], higher-order rewriting [Ham05], explicit substitutions [GUH06], and the Fusion calculus [Mic08]. The simply-typed case [Fio02, MS03] was applied to normalisation by evaluation [Fio02], pre-logical predicates [Kat04], simply-typed higher-order rewriting [Ham07], cyclic sharing tree structures [Ham10], and second-order equational logic [FH10].

However, an important extension of abstract syntax remains untouched, namely *polymorphic typed abstract syntax*.

This paper provides the initial algebra characterisation of polymorphic typed abstract syntax with variable binding in a presheaf category. We consider two variations, second-order and higher-order polymorphic syntax. The central idea is to repeatedly apply Fiore's initial algebra characterisation of typed abstract syntax with binding [Fio02] *twice*, i.e. first to the type structure and secondly to the term structure of polymorphic system. In this process, we use the Grothendieck construction to combine differently staged categories of polymorphic contexts, which is a key to defining the category of discourse in our formulation.

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 381–395, 2011.

This characterisation will be a basis of further fruitful research. It is applicable to modern functional programming language such as ML and Haskell. Moreover, it can be a basis of more interesting systems, polymorphic equational logic (along the line of Fiore's programme on synthesis of equational logic [Fio09]), polymorphic higher-order rewriting systems as an extension of untyped [Ham05] and simply-typed [Ham07] higher-order rewriting systems based on algebraic semantics.

Organisation. This paper is organised as follows. We first review the previous algebraic models of abstract syntax with binding in Section 2. We then characterise polymorphic syntax by examining the syntax of system F in Section 3. We further characterise higher-order polymorphic syntax by examining the syntax of system F_ω in Section 4. Finally, in Section 5, we discuss how substitutions on polymorphic syntax can be modelled.

2 Background

2.1 Algebras in $\mathbf{Set}^{\mathbb{F}}$ for Abstract Syntax with Binding

Firstly, we review algebras in a presheaf category $\mathbf{Set}^{\mathbb{F}}$ for modelling untyped abstract syntax with binding by Fiore, Plotkin and Turi [FPT99]. Hofmann [Hof99] also used the same approach to model higher-order abstract syntax. This is the basis of typed abstract syntax in next subsection and polymorphic syntax in §3.

The aim is to model syntax involving variable binding. A typical example is the syntax for untyped λ-terms:

$$\frac{}{x_1,\ldots,x_n \vdash x_i} \qquad \frac{x_1,\ldots,x_n \vdash t \quad x_1,\ldots,x_n \vdash s}{x_1,\ldots,x_n \vdash t@s} \qquad \frac{x_1,\ldots,x_n,x_{n+1} \vdash t}{x_1,\ldots,x_n \vdash \lambda(x_{n+1}.t)}$$

This is seen as abstract syntax generated by three constructors, i.e. the variable former, the application @, and the abstraction λ. The point is that the variable former is a unary and @ is a binary function symbol, but λ is not merely a unary function symbol. It also makes the variable x_{n+1} bound and decreases the context, which is seen as taking the "internal-level abstraction" $(x_{n+1}.t)$ as the argument of the constructor λ.

In order to model this phenomenon of variable binding generally (not only for λ-terms), Fiore et al. took the presheaf category $\mathbf{Set}^{\mathbb{F}}$ to be the universe of discourse, where $\mathbb{F}$ is the category which has finite cardinals $n = \{1,\ldots,n\}$ (n is possibly 0) as objects, and all functions between them as arrows $m \to n$. This is the category of object variables by the method of de Bruijn index/levels (i.e. natural numbers) and their renamings.

Fiore et al. showed that abstract syntax with variable binding is precisely characterised as the initial algebra of suitable endofunctor modelling a signature (e.g. for λ-terms). More precisely, we need the functor $\delta : \mathbf{Set}^{\mathbb{F}} \to \mathbf{Set}^{\mathbb{F}}$ for context extension $(\delta A)(n) = A(n+1)$ for $A \in \mathbf{Set}^{\mathbb{F}}, n \in \mathbb{F}$, and the presheaf $\mathrm{V} \in \mathbf{Set}^{\mathbb{F}}$ of variables defined by $\mathrm{V}(n) = \mathbb{F}(1,n) \cong \{1,\ldots,n\}$. Using these, for example, we can define the endofunctor Σ_λ on $\mathbf{Set}^{\mathbb{F}}$ for abstract syntax of λ-terms by

$$\Sigma_\lambda(A) = \mathrm{V} + A \times A + \delta A$$

where each summand corresponds to the arity of variable former, @ and λ symbols. Then we use ordinary notion of functor-algebras in $\mathbf{Set}^{\mathbb{F}}$. Generally, an endofunctor Σ on $\mathbf{Set}^{\mathbb{F}}$ defined using $+, \times, \delta$ is called *signature functor*, and a Σ-*algebra* is a pair (A, α) consisting of a presheaf A and a map $\alpha : \Sigma A \to A$, called an *algebra structure*. A *homomorphism* of Σ-algebras is a map $\phi : (A, \alpha) \to (B, \beta)$ such that $\phi \circ \alpha = \beta \circ \Sigma\phi$.

The initial Σ_λ-algebra (Λ, in) exists and can be constructed by the method in [SP82]. It can be expressed as the presheaf $\Lambda(n) = \{t \mid x_1, \ldots, x_n \vdash t\}/=_\alpha$ of all terms, where the algebra structure $\mathsf{in} : \Sigma_\lambda\Lambda \to \Lambda$ consists of constructors of λ-terms, i.e. the variable former, @, and λ.

This process is generic with respect to arbitrary signature functor Σ, hence an initial Σ-algebra in $\mathbf{Set}^{\mathbb{F}}$ models abstract syntax with variable binding.

2.2 Algebras in $(\mathbf{Set}^{\mathbb{F}\downarrow U})^U$ for Typed Abstract Syntax with Binding

Algebras in $\mathbf{Set}^{\mathbb{F}}$ lack the treatment of type restrictions on syntax. A typical example of typed abstract syntax with binding is the syntax for *simply-typed λ-terms*:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \sigma \Rightarrow \tau \quad \Gamma \vdash s : \sigma}{\Gamma \vdash t@s : \tau} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda(x\colon \sigma.t) : \sigma \Rightarrow \tau}$$

To model this syntax, we need to model typed contexts $\Gamma = \{x_1 : \tau_1, \cdots, n : \tau_n\}$.

For this, instead of the category $\mathbb{F}$, Fiore [Fio02], and Miculan and Scagnetto [MS03] took the comma category[1] $\mathbb{F}\downarrow U$ for the index category. Now U is the set containing all type names used in syntax. It is similar to a universe used in Martin-Löf type theory (i.e. the set U of all codes of small sets (= types)), hence we call *U type universe* hereafter. In the case of simply typed λ-calculus, we take U to be the set of all simple types generated by base types. The presheaf category $(\mathbf{Set}^{\mathbb{F}\downarrow U})^U$ is now our working category.

The intention of the use of $(\mathbf{Set}^{\mathbb{F}\downarrow U})^U$ is that the inner index $\mathbb{F}\downarrow U$ models contexts, and the outer index U models the target types of judgments. The category $\mathbb{F}\downarrow U$ has objects $\Gamma : n \to U$, which are seen as contexts $\{1 : \tau_1, \cdots, n : \tau_n\}$, and arrows $\rho : \Gamma \to \Gamma'$, which are functions $\rho : n \to n'$ such that $\Gamma = \Gamma' \circ \rho$, i.e. renaming between Γ and Γ'. The coproduct is Γ, Γ' defined by $[\Gamma, \Gamma'] : n + n' \to U$.

All constructions used in the case of $\mathbf{Set}^{\mathbb{F}}$ are smoothly extended to the case of $(\mathbf{Set}^{\mathbb{F}\downarrow U})^U$, which makes modelling typed syntax possible. The context extension $\delta_\tau : \mathbf{Set}^{\mathbb{F}\downarrow U} \to \mathbf{Set}^{\mathbb{F}\downarrow U}$ by a variable of type τ is defined by $(\delta_\tau A)(\Gamma) = A(\Gamma, \tau)$. The presheaf $\mathrm{V} \in (\mathbf{Set}^{\mathbb{F}\downarrow U})^U$ of variables is defined by the Yoneda embedding $\mathrm{V}_\tau = \mathbb{F}\downarrow U(\langle\tau\rangle, -)$, where $\langle\tau\rangle : 1 \to U$ maps $1 \mapsto \tau \in U$. Hence $\mathrm{V}_\tau(\Gamma) \cong \{x \mid x : \tau \in \Gamma\}$, i.e. the set of variables of a certain type τ taken from a context Γ.

For example, the signature functor $\Sigma_{\vec{\lambda}} : (\mathbf{Set}^{\mathbb{F}\downarrow U})^U \to (\mathbf{Set}^{\mathbb{F}\downarrow U})^U$ for abstract syntax of simply-typed λ-terms can be defined by

$$(\Sigma_{\vec{\lambda}} A)_\tau = \mathrm{V}_\tau + \coprod_{\tau' \in U} (A_{\tau' \Rightarrow \tau} \times A_{\tau'}) + \coprod_{\tau_1, \tau_2 \in U} (\tau \equiv \tau_1 \Rightarrow \tau_2) \times (\delta_{\tau_1} A_{\tau_2})$$

[1] In the rigorous notation of comma category, $(\mathbb{F}\downarrow U)$ should be written as $(J_{\mathbb{F}} \downarrow U)$, where $J_{\mathbb{F}} : \mathbb{F} \to \mathbf{Set}$ is the inclusion functor.

Here the binary operator '≡' on types gives a set defined by $(\tau \equiv \tau') \triangleq 1$ (the one point set) if $\tau = \tau'$, $(\tau \equiv \tau') \triangleq 0$ (the empty set) if $\tau \neq \tau'$. This style using '≡' to define a signature functor can be found in [MA09]. Throughout this paper, we use this '≡' operator. The initial $\Sigma_{\vec{\lambda}}$-algebra $(\Lambda^{\rightarrow}, \mathsf{in})$ in $(\mathbf{Set}^{\mathbb{F}\downarrow U})^U$ exists and can be constructed [SP82] as the presheaf of all simply-typed λ-terms

$$(\Lambda^{\rightarrow})_\tau(\Gamma) = \{t \mid \Gamma \vdash t : \tau\} / =_\alpha$$

with algebra structure giving constructors of simply-typed λ terms.

This process is again generic with respect to arbitrary signature functor Σ, hence an initial Σ-algebra in $(\mathbf{Set}^{\mathbb{F}\downarrow U})^U$ models arbitrary typed abstract syntax with variable binding.

Remark. These works do not intend to directly give semantics of λ-calculi. The initial algebras Λ and $\Lambda^{\rightarrow}$ are not models of untyped and typed λ-calculi respectively, because they do not validate the β-axioms. They are the initial models of *abstract syntax* of λ-calculi. Similarly, we do not intend to give models of polymorphic λ-calculi, system F and F_ω in this paper. We give algebraic models of *abstract syntax* of types and terms.

Convention on α-equivalence. In this paper, hereafter we use the method of *de Bruijn levels* [dB72] for representing bound and free variables in a term (and a type, a judgment, etc.). However, keeping de Bruijn level notation strictly (as in [Ham04, Ham05, Ham07]) is sometimes clumsy and hides the essence. Hence, in this paper, at the level of text, we use the usual named notation for terms to avoid clutter. We assume that these actually denote (or are automatically normalised to) de Bruijn level normal forms. For example, when we write $\lambda x.\lambda y.yx$, it actually means $\lambda 1.\lambda 2.21$. Another example is $\alpha_1, \ldots, \alpha_n \vdash \forall\alpha_{n+1}.\tau$ to mean $1, \ldots, n \vdash \forall(n+1.\tau)$. Hence we will drop the explicit quotienting "$/ =_\alpha$" by the α-equivalence in defining a term set hereafter. De Bruijn level notation is different from more well-known de Bruijn *index* notation, and levels are the reverse numbering of variables. See [FPT99] for illustrations of de Bruijn level notation.

3 Second-Order Polymorphic Abstract Syntax

We extend the treatment reviewed in the previous section to the case of second-order polymorphic abstract syntax with variable binding. The leading example of such a syntax is the abstract syntax for Girard and Reynolds' system F. Hence we review its definition.

3.1 System F

Types

$$\tau ::= \alpha \mid b \mid \tau_1 \Rightarrow \tau_2 \mid \forall\alpha.\tau$$

where α ranges over type variables, and b ranges over base types.

Well-formed types

$$\frac{1 \leq i \leq n}{\alpha_1, \ldots, \alpha_n \vdash \alpha_i} \qquad \frac{}{\alpha_1, \ldots, \alpha_n \vdash b}$$

$$\frac{\alpha_1, \ldots, \alpha_n \vdash \sigma \quad \alpha_1, \ldots, \alpha_n \vdash \tau}{\alpha_1, \ldots, \alpha_n \vdash \sigma \Rightarrow \tau} \qquad \frac{\alpha_1, \ldots, \alpha_n, \alpha_{n+1} \vdash \tau}{\alpha_1, \ldots, \alpha_n \vdash \forall \alpha_{n+1}.\, \tau}$$

Well-typed terms

$$\frac{x : \tau \in \Gamma}{\Xi \mid \Gamma \vdash x : \tau} \qquad \frac{\Xi \mid \Gamma, x : \sigma \vdash t : \tau}{\Xi \mid \Gamma \vdash \lambda x : \sigma. t \; : \sigma \Rightarrow \tau} \qquad \frac{\Xi \mid \Gamma \vdash t : \sigma \Rightarrow \tau \quad \Xi \mid \Gamma \vdash s : \sigma}{\Xi \mid \Gamma \vdash t\, s : \tau}$$

$$\frac{\Xi, \alpha \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma \vdash \Lambda \alpha. t : \forall \alpha. \tau} \qquad \frac{\Xi \mid \Gamma \vdash t : \forall \alpha. \tau \quad \Xi \vdash \sigma}{\Xi \mid \Gamma \vdash t\, \sigma : \tau[\alpha := \sigma]}$$

Notes

- $\Xi \mid \Gamma \vdash t : \tau$ is well-formed if $\Xi \vdash \tau_i$ for each $x_i : \tau_i \in \Gamma$, and $\Xi \vdash \tau$.
- $\Xi = \alpha_1, \ldots, \alpha_n$ is a *type context*, i.e., a sequence of type variables.
- $\Gamma = x_1 : \tau_1, \ldots, x_k : \tau_k$ is a *term context*.

We use the formulation that a type context and a term context are explicitly separated in a judgment as $\Xi \mid \Gamma$, where any type variable α appearing in Γ is taken from Ξ. The substitution operation $[- := -]$ on terms and types is the standard capture-avoiding substitution.

3.2 Modelling Syntax of F

First we concentrate on modelling syntax for system F. We generalise it to arbitrary polymorphic abstract syntax later.

The basic idea we take is to use algebras in $(\mathbf{Set}^{\mathbb{F}\downarrow U})^U$ as in §2.2. Now the type universe U is not merely a *set* of all types, since types involve type variables and quantification. This means that U must be given by abstract syntax *with variable binding*. This point of view was also taken in [AHS96].

We use a two-stage approach to model system F terms. Firstly, we construct the universe $\mathbb{T}$ of all system F types as a *presheaf* $\mathbb{T} \in \mathbf{Set}^{\mathbb{F}}$ by an initial algebra in $\mathbf{Set}^{\mathbb{F}}$. Then we move to another presheaf category $\mathbf{Set}^{\int \mathsf{G}}$ (explained later) defined using $\mathbb{T}$, and construct an initial algebra for all well-typed terms in it. We proceed by the following three steps.

(I) Polymorphic types. We follow the method reviewed in §2.1. Let $\mathrm{B} \in \mathbf{Set}^{\mathbb{F}}$ be the constant functor to the set of all base types, and $\mathbb{V}$ the presheaf of type variables defined by $\mathbb{V}(n) = \mathbb{F}(1, n) \cong \{1, \ldots, n\}$. We define the signature functor $\mathrm{F}^{\mathrm{ty}} : \mathbf{Set}^{\mathbb{F}} \to \mathbf{Set}^{\mathbb{F}}$ for system F types by

$$\mathrm{F}^{\mathrm{ty}}(A) = \mathbb{V} + \mathrm{B} + A \times A + \delta A.$$

Each summand corresponds to the arity of type variable, base type, arrow type, and universal type. An initial F^{ty}-algebra exists and can be constructed. We define $\mathbb{T} \in \mathbf{Set}^{\mathbb{F}}$ by the initial F^{ty}-algebra $(\mathbb{T}, \mathsf{in})$ described as the presheaf of all well-formed types:

$$\mathbb{T}(n) = \{\tau \mid \alpha_1, \ldots, \alpha_n \vdash \tau\}.$$

The arrow part $\mathbb{T}(\rho)$ is a renaming action on types using ρ defined by structural recursion [FPT99], i.e., $\mathbb{T}(\rho)(\tau)$ renames each type variable in a type τ by ρ. The algebra structure $\mathsf{in} : \mathrm{F}^{\mathrm{ty}}(\mathbb{T}) \to \mathbb{T}$ consists of constructors of system F types

$$\mathsf{tvar} : \mathbb{V} \to \mathbb{T} \qquad \mathsf{base} : \mathsf{B} \to \mathbb{T} \qquad \mathsf{arrow} : \mathbb{T} \times \mathbb{T} \to \mathbb{T} \qquad \mathsf{forall} : \delta\mathbb{T} \to \mathbb{T}.$$

(II) Contexts. In order to model terms, next we need to choose a presheaf category on some index category using $\mathbb{T}$. We basically follow the style to use $(\mathbf{Set}^{\mathbb{F}\downarrow U})^U \simeq \mathbf{Set}^{(\mathbb{F}\downarrow U)\times U}$ for modelling terms as in §2.2. So we need to choose a suitable type universe U. Now let's try to choose the disjoint union: $U = \coprod_{n\in\mathbb{N}} \mathbb{T}(n)$. But this is imprecise, because in this attempt $\mathbf{Set}^{(\mathbb{F}\downarrow\coprod_{n\in\mathbb{N}} \mathbb{T}(n))\times\coprod_{n\in\mathbb{N}} \mathbb{T}(n)}$, the index n in the left sum on the index category does not synchronize with the index n in the right sum (since each n is locally bound by each sum). These must be equal because

$$n \mid \Gamma \vdash t : \tau \text{ is well-formed} \quad \Leftrightarrow \quad \text{for every } \tau_i \text{ in } \Gamma,\ n \vdash \tau_i \text{ and } n \vdash \tau.$$

Another attempt to use $\mathbf{Set}^{\coprod_{n\in\mathbb{N}}(\mathbb{F}\downarrow\mathbb{T}(n)\times\mathbb{T}(n))}$ is again insufficient because this does not model renaming between two terms in different type contexts n and n'.

The right way to combine all of these $\mathbb{T}(n)$ for the index category is the *Grothendieck construction*. Before going to it, we need to state the following.

Definition 1. (**Categories of context-with-types**) Let $n \in \mathbb{N}$. We use a comma category $\mathbb{F}\downarrow(\mathbb{T}(n))$, where $\mathbb{T}(n)$ is a set. We also regard $\mathbb{T}(n)$ as a discrete category. Then we take the product: *Category* $\mathbb{F}\downarrow(\mathbb{T}(n)) \times \mathbb{T}(n)$

- objects (Γ, τ) where $\Gamma \in \mathbb{F}\downarrow(\mathbb{T}(n)),\ \tau \in \mathbb{T}(n)$
- arrows $\pi : (\Gamma, \tau) \to (\Delta, \tau)$ given by a renaming $\pi : \Gamma \to \Delta$ in $\mathbb{F}\downarrow(\mathbb{T}(n))$.

We use the Grothendieck construction to glue all categories of context-with-types together. We recall the construction [Gro70].

Definition 2. (**Grothendieck**) Given a functor $\mathcal{F} : C^{\mathrm{op}} \to \mathbf{Cat}$, the *Grothendieck construction* of $\mathcal{F}$ is a category $\int \mathcal{F}$ with objects (I, A) where $I \in C$ and $A \in \mathcal{F}(I)$, and arrows $(u, \gamma) : (I, A) \to (J, B)$ where $u : J \to I$ in C^{op} and $\gamma : \mathcal{F}(u)(A) \to B$ in $\mathcal{F}(J)$.

We now define a functor $\mathsf{G} : \mathbb{F}^{\mathrm{op}} \to \mathbf{Cat}$ by

$$\begin{aligned} \mathsf{G}(x) &= \mathbb{F}\downarrow(\mathbb{T}(x)) \times \mathbb{T}(x) \\ \mathsf{G}(f) &= \mathbb{F}\downarrow(\mathbb{T}(f)) \times \mathbb{T}(f) \quad \text{ for } f : x \to y \text{ in } \mathbb{F} \end{aligned}$$

The Grothendieck construction $\displaystyle\int \mathsf{G}$ has

- objects $(n \mid \Gamma \vdash \tau)$, where $n \in \mathbb{F},\ \Gamma \in \mathbb{F}\downarrow(\mathbb{T}(n)),\ \tau \in \mathbb{T}(n)$,
- arrows $(\rho, \pi) : (m \mid \Gamma \vdash \tau) \to (n \mid \Delta \vdash \sigma)$,
  where $\rho : m \to n$ in $\mathbb{F}$ such that $\mathbb{T}(\rho)(\tau) = \sigma$, and
  $\pi : (\mathbb{F}\downarrow\mathbb{T}\rho)\,(\Gamma) \to \Delta$ in $\mathbb{F}\downarrow(\mathbb{T}(n))$.

We now explain why objects and arrows are described as above and their syntactic meaning. If we follow the above definition strictly, an object of $\int\mathsf{G}$ should be $(n, (\Gamma, \tau))$, where $n \in \mathbb{F}$ and $(\Gamma, \tau) \in \mathbb{F}{\downarrow}(\mathbb{T}(n)) \times \mathbb{T}(n)$. We merely use another notation $(n \mid \Gamma \vdash \tau)$ for this triple.

Meaning of arrows. For arrows, the above description is obtained by expanding the definition. An arrow $(\rho, \pi) : (m \mid \Gamma \vdash \tau) \to (n \mid \Delta \vdash \sigma)$ consists of a renaming $\rho : m \to n$ between type contexts, and a renaming $\pi : (\mathbb{F}{\downarrow}\mathbb{T}\rho)\,(\Gamma) \to \Delta$ between term contexts.

Moreover, an arrow (ρ, π) in $\int\mathsf{G}$ can be understood as renaming type and term variables between two judgments having different contexts and result types. To discuss it, we first define the indexed set T of all well-typed terms by

$$\mathrm{T}(\Xi \mid \Gamma \vdash \tau) = \{t \mid (\Xi \mid \Gamma \vdash t : \tau) \text{ is derivable}\}.$$

Let $\rho^\sharp t$ denote renaming each type variable in a term t by ρ, and $\pi^\sharp t$ denotes renaming each term variable in a term t by π. Then, we have an admissible rule for well-typedness of renamed term:

$$\frac{m \mid \Gamma \vdash t : \tau}{n \mid \Delta \vdash \pi^\sharp\rho^\sharp(t) : \mathbb{T}(\rho)(\tau)} \text{ by applying } (\rho, \pi) \text{ in } \int\mathsf{G}$$

This process defines the arrow part of T being a presheaf in $\mathbf{Set}^{\int\mathsf{G}}$

$$\mathrm{T}(\rho, \pi) : \mathrm{T}(m \mid \Gamma \vdash \tau) \to \mathrm{T}(n \mid \Delta \vdash \sigma); \quad t \mapsto \pi^\sharp\rho^\sharp(t)$$

where $\mathbb{T}(\rho)(\tau) = \sigma$. When ρ is the identity $\mathrm{id}_n : n \to n$, then $\mathrm{T}(\mathrm{id}_n, \pi) : \mathrm{T}(n \mid \Gamma \vdash \tau) \to \mathrm{T}(n \mid \Gamma' \vdash \tau)$ is the usual renaming on Γ.

Hence, the Grothendieck construction provides the category of context-with-types and renamings in a mathematically uniform way.

(III) Terms. Now our working category is $\mathbf{Set}^{\int\mathsf{G}}$. As we have seen, system F terms form a presheaf T in $\mathbf{Set}^{\int\mathsf{G}}$. The presheaf $\mathrm{V} \in \mathbf{Set}^{\int\mathsf{G}}$ of variables is defined by

$$\begin{aligned} \mathrm{V}(n \mid \Gamma \vdash \tau) &= (\mathbb{F}{\downarrow}\mathbb{T}(n))(\langle\tau\rangle, \Gamma) \;\cong\; \{x \mid x : \tau \in \Gamma\} \\ \mathrm{V}(\rho, \pi) &= \pi \circ -. \end{aligned}$$

We define the signature functor $\mathrm{F} : \mathbf{Set}^{\int\mathsf{G}} \to \mathbf{Set}^{\int\mathsf{G}}$ for system F terms by

$$\begin{aligned} \mathrm{F}(A)(n \mid \Gamma \vdash \tau) = \;& \mathrm{V}(n \mid \Gamma \vdash \tau) \\ &+ \coprod_{\tau_1, \tau_2 \in \mathbb{T}(n)} (\tau \equiv \tau_1 \Rightarrow \tau_2) \times A(n \mid \Gamma, \tau_1 \vdash \tau_2) \\ &+ \coprod_{\sigma \in \mathbb{T}(n)} A(n \mid \Gamma \vdash \sigma \Rightarrow \tau) \times A(n \mid \Gamma \vdash \sigma) \\ &+ \coprod_{\tau' \in \mathbb{T}(n+1)} (\tau \equiv \forall(\alpha.\tau')) \times A(n+1 \mid \mathsf{wk}(\Gamma) \vdash \tau') \\ &+ \coprod_{\substack{\sigma \in \mathbb{T}(n) \\ \tau' \in \mathbb{T}(n+1)}} (\tau \equiv \tau'[\alpha := \sigma]) \times A(n \mid \Gamma \vdash \forall(\alpha.\tau')). \end{aligned}$$

Each summand corresponds to the arity of variable, abstraction, application, type abstraction, and type application. The weakening $\mathsf{wk} : \mathbb{F}\downarrow\mathbb{T}(n) \to \mathbb{F}\downarrow\mathbb{T}(n+1)$ maps a context under n to the same context but under a weakened $n+1$. The use of "≡" operator is inessential to define the signature functor. It is merely a shortcut way of describing the definition by case analyse, see §3.3 for the general case.

Theorem 3. T *forms an initial* F*-algebra.*

Proof. An initial F-algebra is constructed by the colimit of the ω-chain $0 \to \mathrm{F}0 \to \mathrm{F}^2 0 \to \cdots$ [SP82]. These construction steps correspond to derivations of terms by term forming rules, hence their union T is the colimit. The algebra structure $\mathsf{in} : \mathrm{FT} \to \mathrm{T}$ of the initial algebra is obtained by one-step inference of the term forming rules, i.e., given by the following operations

$$\begin{array}{llllll}
\mathsf{var}_\tau & : \mathrm{V}(n \mid \Gamma \vdash \tau) & \to \mathrm{T}(n \mid \Gamma \vdash \tau) & ;\, x & \mapsto x \\
\mathsf{abs}_{\sigma,\tau} & : \mathrm{T}(n \mid \Gamma, \sigma \vdash \tau) & \to \mathrm{T}(n \mid \Gamma \vdash \sigma \Rightarrow \tau) & ;\, t & \mapsto \lambda x.t \\
\mathsf{app}_{\sigma,\tau} & : \mathrm{T}(n \mid \Gamma \vdash \sigma \Rightarrow \tau) & & & \\
 & \times\, \mathrm{T}(n \mid \Gamma \vdash \sigma)) & \to \mathrm{T}(n \mid \Gamma \vdash \tau) & ;\, s,t & \mapsto s\,t \\
\mathsf{tabs}_{\tau'} & : \mathrm{T}(n+1 \mid \mathsf{wk}(\Gamma) \vdash \tau') & \to \mathrm{T}(n \mid \Gamma \vdash \forall(\alpha.\tau')) & ;\, t & \mapsto \Lambda\alpha.t \\
\mathsf{tapp}_{\sigma,\tau'} & : \mathrm{T}(n \mid \Gamma \vdash \forall(\alpha.\tau')) & \to \mathrm{T}(n \mid \Gamma \vdash \tau'[\alpha := \sigma]) & ;\, t & \mapsto t\,\sigma
\end{array}$$

where $n \in \mathbb{N}$, $\sigma, \tau \in \mathbb{T}(n)$, $\tau' \in \mathbb{T}(n+1)$, and α is actually a de Bruijn level $n+1$. □

3.3 General Signature

Not only for system F, we seek a general framework for polymorphic abstract syntax. Generalising the case of system F, we arrive at the following definition.

Definition 4. A *polymorphic signature* $\Sigma = (\Sigma^{\mathsf{ty}}, \Sigma^{\mathsf{tm}})$ consists of the following data.

- Σ^{ty} for types is a *binding signature* [Acz78, FPT99], i.e. a set of type formers with an arity function $a : \Sigma^{\mathsf{ty}} \to \mathbb{N}^*$ (NB. '*' denotes the Kleene closure). A type former of arity $\langle n_1, \ldots, n_l \rangle$, denoted by $o : \langle n_1, \ldots, n_l \rangle$, has l arguments and binds n_i variables in the i-th argument $(1 \le i \le l)$.
- Let $\mathbb{T} \in \mathbf{Set}^{\mathbb{F}}$ be the free Σ^{ty}-algebra over $\mathbb{V}$, represented by term syntax in de Bruijn levels [FPT99, Ham04].
- Σ^{tm} for terms is a set of function symbols with arities. This is denoted by

$$f : \langle k_1 \rangle (\vec{\sigma}_1)\tau_1, \ldots, \langle k_l \rangle (\vec{\sigma}_l)\tau_l \to \tau$$

  where $n \in \mathbb{N}, k_i \in \mathbb{N}, \vec{\sigma}_i \in \mathbb{T}(n+k_i)^*, \tau_i \in \mathbb{T}(n+k_i), \tau \in \mathbb{T}(n)$, has l arguments, and binds k_i type variables and $|\vec{\sigma}_i|$ variables in the i-th argument $(1 \le i \le l)$. In case of $k_i = 0$ or $|\vec{\sigma}_i| = 0$, each part is omitted. Here, $|-|$ denotes the length of a sequence.

Example 5. The polymorphic signature $\Sigma_{\mathrm{F}} = (\Sigma^{\mathsf{ty}}_{\mathrm{F}}, \Sigma^{\mathsf{tm}}_{\mathrm{F}})$ for system F can be given as follows: $\Sigma^{\mathsf{ty}}_{\mathrm{F}} = \{b : \langle 0 \rangle, \Rightarrow : \langle 0, 0 \rangle, \forall : \langle 1 \rangle\}$, and $\Sigma^{\mathsf{tm}}_{\mathrm{F}}$ is

$$\begin{array}{llll}
\mathsf{abs}_{\sigma,\tau} : (\sigma)\tau \to \sigma \Rightarrow \tau & \quad \mathsf{app}_{\sigma,\tau} & : \sigma \Rightarrow \tau, \sigma & \to \tau \\
\mathsf{tabs}_{\tau'} : \langle 1 \rangle \tau' \to \forall(\alpha.\tau') & \quad \mathsf{tapp}_{\sigma,\tau'} & : \forall(\alpha.\tau') & \to \tau'[\alpha := \sigma]
\end{array}$$

generated by all $n \in \mathbb{N}$, $\sigma, \tau \in \mathbb{T}(n)$, $\tau' \in \mathbb{T}(n+1)$, and α should be regarded as a de Bruijn level $n+1$.

Example 6. If we want to add the let-binding construct, Σ^{tm} has the function symbol $\mathsf{let} : \sigma,\ (\sigma)\tau \to \tau$, where $\sigma, \tau \in \mathbb{T}(n)$.

To a polymorphic signature Σ, we associate the *signature functor* $\Sigma : \mathbf{Set}^{\int \mathsf{G}} \to \mathbf{Set}^{\int \mathsf{G}}$ given by

$$\Sigma A(n \mid \Gamma \vdash \tau) = \coprod_{f:\langle k_1\rangle(\vec{\sigma}_1)\tau_1,\ldots,\langle k_l\rangle(\vec{\sigma}_l)\tau_l \to \tau \in \Sigma^{\mathsf{tm}}} \ \prod_{1 \le i \le l} A(n + k_i \mid \Gamma, \vec{\sigma}_i \vdash \tau_i).$$

3.4 General Syntax Rules

Using a polymorphic signature Σ, we can construct polymorphic terms syntactically and generally. The following construction rules are extracted from the semantic structure we have obtained so far.

Well-formed types

$$\frac{1 \le i \le n}{\alpha_1, \ldots, \alpha_n \vdash \alpha_i} \qquad \frac{\Xi, \vec{\alpha}_1 \vdash \tau_1 \ \ldots \ \Xi, \vec{\alpha}_l \vdash \tau_l}{\Xi \vdash o(\vec{\alpha}_1.\tau_1, \ \ldots, \vec{\alpha}_l.\tau_l\tau)}$$

where $o : \langle n_1, \ldots, n_l\rangle \in \Sigma^{\mathsf{ty}}$, $|\vec{\alpha}_i| = n_i$.

Well-typed terms

$$\frac{x : \tau \in \Gamma}{\Xi \mid \Gamma \vdash x : \tau} \qquad \frac{\Xi, \vec{\alpha}_1 \mid \Gamma, \vec{x}_1 : \vec{\sigma}_1 \vdash t_1 : \tau_1 \quad \cdots \quad \Xi, \vec{\alpha}_l \mid \Gamma, \vec{x}_l : \vec{\sigma}_l \vdash t_l : \tau_l}{\Xi \mid \Gamma \vdash f(\vec{x}_1.t_1, \ldots, \vec{x}_l.t_l) : \tau}$$

where $f : \langle k_1\rangle(\vec{\sigma}_1)\tau_1, \ldots, \langle k_l\rangle(\vec{\sigma}_l)\tau_l \to \tau \in \Sigma^{\mathsf{tm}}$, $|\vec{\alpha}_i| = k_i$.

We define $\mathrm{TV}(\Xi \mid \Gamma \vdash \tau) \triangleq \{t \mid (\Xi \mid \Gamma \vdash t : \tau) \text{ is derivable}\}$, which we call *polymorphic abstract syntax*.

Theorem 7. *Given a polymorphic signature* Σ*,* TV *forms a free* Σ*-algebra over* V.

Proof. Similarly to Thm. 3. Notice that a free Σ-algebra over V is an initial V + Σ-algebra. □

4 Higher-Order Polymorphic Abstract Syntax

We extend the previous algebraic characterisation to the case of higher-order polymorphic abstract syntax with variable binding. The leading example of such a syntax is the abstract syntax for Girard's system F_ω. Hence we review its definition.

4.1 System F_ω

Kinds and types

$$\kappa ::= * \mid \kappa_1 \Rightarrow \kappa_2$$
$$\tau ::= \alpha \mid b \mid \tau_1 \Rightarrow \tau_2 \mid \forall\alpha : \kappa.\tau \mid \lambda\alpha : \kappa.\tau \mid \tau_1\tau_2$$

Well-kinded types

$$\frac{\alpha : \kappa \in \Xi}{\Xi \vdash \alpha : \kappa} \qquad \frac{}{\Xi \vdash b : \kappa} \qquad \frac{\Xi, \alpha : \kappa' \vdash \tau : \kappa}{\Xi \vdash \lambda\alpha : \kappa'.\tau : \kappa' \Rightarrow \kappa}$$

$$\frac{\Xi \vdash \sigma : * \quad \Xi \vdash \tau : *}{\Xi \vdash \sigma \Rightarrow \tau : *} \qquad \frac{\Xi, \alpha : \kappa \vdash \tau : *}{\Xi \vdash \forall\alpha : \kappa.\tau : *} \qquad \frac{\Xi \vdash \sigma : \kappa' \Rightarrow \kappa \quad \Xi \vdash \tau : \kappa'}{\Xi \vdash \sigma\tau : \kappa}$$

Well-typed terms

$$\frac{x : \tau \in \Gamma}{\Xi \mid \Gamma \vdash x : \tau} \qquad \frac{\Xi \mid \Gamma, x : \sigma \vdash t : \tau}{\Xi \mid \Gamma \vdash \lambda x : \tau.t : \sigma \Rightarrow \tau} \qquad \frac{\Xi \mid \Gamma \vdash t : \sigma \Rightarrow \tau \quad \Xi \mid \Gamma \vdash s : \sigma}{\Xi \mid \Gamma \vdash ts : \tau}$$

$$\frac{\Xi, \alpha : \kappa \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma \vdash \Lambda\alpha : \kappa.t : \forall\alpha : \kappa.\tau} \qquad \frac{\Xi \mid \Gamma \vdash t : \forall\alpha : \kappa.\tau \quad \Xi \vdash \sigma : \kappa}{\Xi \mid \Gamma \vdash t\sigma : \tau[\alpha := \sigma]}$$

Notes

- $\Xi \mid \Gamma \vdash t : \tau$ is well-formed if $\Xi \vdash \tau_i : \kappa_i$ for each $x_i : \tau_i \in \Gamma$ and $\Xi \vdash \tau : *$.
- $\Xi = \alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n$ is a sequence of (type variable,kind)-pairs.

4.2 Modelling Syntax of F_ω

First we concentrate on modelling abstract syntax for system F_ω terms. We generalise it to arbitrary higher-order polymorphic abstract syntax later.

The basic idea we take is again to follow the style using algebras in $(\mathbf{Set}^{\mathbb{F}\downarrow U})^U$. In the case of system F, U was given by *untyped* abstract syntax with variable binding since types contain universal quantification. In system F_ω, a type has a kind. This means that U must be given by *typed* (= used for kinding) abstract syntax with variable binding.

We use again a two-stage approach to model system F_ω terms, but the working categories are different from the previous case. Let $\mathcal{K}$ be the set of all kinds (and considered as a discrete category). Firstly, we construct the universe $\mathbb{T}$ of all system F_ω types as a presheaf $\mathbb{T} \in (\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{K}})^{\mathcal{K}}$ by an initial algebra in the category $(\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{K}})^{\mathcal{K}}$. Then we move to another presheaf category $\mathbf{Set}^{\int \mathsf{H}}$ (explained later) defined using $\mathbb{T}$, and construct an initial algebra for all well-typed F_ω terms in it.

(I) Kinded types. Let $\mathrm{B}_\kappa \in \mathbf{Set}^{\mathbb{F}\downarrow\mathcal{K}}$ be the constant functor to the set all base types of kind κ, $\mathbb{V} \in (\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{K}})^{\mathcal{K}}$ the presheaf of kinded type variables defined by $\mathbb{V}_\kappa = \mathbb{F}\!\downarrow\!\mathcal{K}(\kappa, -)$. We define the signature functor $\mathrm{F}^{\mathrm{ty}}_\omega : (\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{K}})^{\mathcal{K}} \to (\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{K}})^{\mathcal{K}}$ for system F_ω types by

$$\begin{aligned}\mathrm{F}^{\mathrm{ty}}_\omega(A)_\kappa = {} & \mathbb{V}_\kappa + \mathrm{B}_\kappa + (\kappa \equiv *) \times (A_* \times A_* + \delta_\kappa A_*) \\ & + \coprod_{\kappa_1,\kappa_2 \in \mathcal{K}} ((\kappa \equiv \kappa_1 \Rightarrow \kappa_2) \times \delta_{\kappa_1} A_{\kappa_2}) + \coprod_{\kappa' \in \mathcal{K}} (A_{\kappa' \Rightarrow \kappa} \times A_{\kappa'}).\end{aligned}$$

Each summand corresponds to the arity of type variable, base type, (arrow type and universal type), type-level λ, and type-level application.

An initial $\mathrm{F}^{\mathrm{ty}}_{\omega}$-algebra exists as in §2.2. We define $\mathbb{T} \in (\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{K}})^{\mathcal{K}}$ by an initial $\mathrm{F}^{\mathrm{ty}}_{\omega}$-algebra $(\mathbb{T}, \mathsf{in})$ described as the presheaf of all well-kinded types[2]

$$\mathbb{T}_{\kappa}(\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n) = \{\tau \mid \alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n \vdash \tau : \kappa\},$$

with algebra structure consisting of constructors

$$\begin{array}{lll} \mathsf{tvar}_{\kappa} : \mathbb{V}_{\kappa} \to \mathbb{T}_{\kappa} & \mathsf{base}_{\kappa} : \mathrm{B}_{\kappa} \to \mathbb{T}_{\kappa} & \mathsf{tyabs}_{\kappa_1,\kappa_2} : \delta_{\kappa_1}\mathbb{T}_{\kappa_2} \to \mathbb{T}_{\kappa_1 \Rightarrow \kappa_2} \\ \mathsf{arrow} : \mathbb{T}_{*} \times \mathbb{T}_{*} \to \mathbb{T}_{*} & \mathsf{forall}_{\kappa} : \delta_{\kappa}\mathbb{T}_{*} \to \mathbb{T}_{*} & \mathsf{tyapp}_{\kappa',\kappa} : \mathbb{T}_{\kappa' \Rightarrow \kappa} \times \mathbb{T}_{\kappa'} \to \mathbb{T}_{\kappa}. \end{array}$$

These six arrows of $\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{K}}$ correspond to the six rules of *Well-kinded types* of F_{ω}.

(II) Contexts. Let $\Xi \in \mathbb{F}\downarrow\mathcal{K}$. We now take $\mathbb{F}\downarrow(\mathbb{T}_{\kappa}\Xi)$ to be the category of contexts. As we have seen, $\mathbb{T}_{\kappa}(\Xi)$ is the set of all types of kind κ under $\Xi = \alpha_1 : \kappa_1, \ldots, \alpha_l : \kappa_l$. An object $\Gamma \in \mathbb{F}\downarrow(\mathbb{T}_{\kappa}\Xi)$ is a map such that

$$\mathbb{F} \ni n \ni \text{a variable } x_i \overset{\Gamma}{\longmapsto} \tau_i \in \mathbb{T}_{\kappa}(\Xi) \qquad \text{i.e. } \Xi \vdash \tau_i : \kappa.$$

Hence, Γ expresses a context $x_1 : \tau_1, \ldots, x_n : \tau_n$, and all types τ_i are of kind κ. The set $\mathbb{T}_{\kappa}(\Xi)$ is also regarded as a discrete category.

(III) Terms. Again, we use the Grothendieck construction to glue all categories of context-with-types together. We define a functor $\mathsf{H} : (\mathbb{F}\downarrow\mathcal{K} \times \mathcal{K})^{\mathsf{op}} \to \mathbf{Cat}$ by

$$\begin{aligned} \mathsf{H}(\Xi, \kappa) &= \mathbb{F}\downarrow(\mathbb{T}_{\kappa}\Xi) \times \mathbb{T}_{\kappa}(\Xi) \\ \mathsf{H}(\rho, \mathrm{id}_{\kappa}) &= \mathbb{F}\downarrow(\mathbb{T}_{\kappa}\rho) \times \mathbb{T}_{\kappa}(\rho) \quad \text{ for } \rho : \Xi \to \Xi' \text{ in } \mathbb{F}\downarrow\mathcal{K}. \end{aligned}$$

The Grothendieck construction $\int \mathsf{H}$ has

- objects $(\Xi \mid \Gamma \vdash \tau : \kappa)$, where $\Xi \in \mathbb{F}\downarrow\mathcal{K}$, $\kappa \in \mathcal{K}$, $\Gamma \in \mathbb{F}\downarrow(\mathbb{T}_{\kappa}\Xi)$, $\tau \in \mathbb{T}_{\kappa}(\Xi)$,
- arrows $(\rho, \pi) : (\Theta \mid \Gamma \vdash \tau : \kappa) \to (\Xi \mid \Delta \vdash \sigma : \kappa)$,
  where $\rho : \Theta \to \Xi$ in $\mathbb{F}\downarrow\mathcal{K}$ such that $\mathbb{T}_{\kappa}(\rho)(\tau) = \sigma$, and
  $\pi : (\mathbb{F}\downarrow\mathbb{T}_{\kappa}\rho)(\Gamma) \to \Delta$ in $\mathbb{F}\downarrow(\mathbb{T}_{\kappa}\Xi)$.

Now our working category is $\mathbf{Set}^{\int \mathsf{H}}$. We define $\mathrm{T}_{\omega} \in \mathbf{Set}^{\int \mathsf{H}}$ of all well-typed terms by

$$\begin{aligned} \mathrm{T}_{\omega}(\Xi \mid \Gamma \vdash \tau : *) &= \{t \mid (\Xi \mid \Gamma \vdash t : \tau) \text{ is derivable}\} \\ \mathrm{T}_{\omega}(\Xi \mid \Gamma \vdash \tau : \kappa) &= \varnothing \qquad \text{if } \kappa \neq * \end{aligned}$$

The second clause is due to that there are no terms of higher-kinded types in F_{ω}. The arrow part is defined similarly to the case of system F.

The presheaf $\mathrm{V} \in \mathbf{Set}^{\int \mathsf{H}}$ of term variables is defined by

$$\begin{aligned} \mathrm{V}(\Xi \mid \Gamma \vdash \tau : \kappa) &= (\mathbb{F}\downarrow(\mathbb{T}_{\kappa}\Xi))(\langle\tau\rangle, \Gamma) \cong \{x \mid x : \tau \in \Gamma\} \\ \mathrm{V}(\rho, \pi) &= \pi \circ -. \end{aligned}$$

[2] Usually, system F_{ω} types are identified modulo type-level β-conversion. Since we only focus on abstract syntax in this paper, we do not treat this process here. This will be uniformly treated within general polymorphic equational logic (cf. the discussion in §5).

We define the signature functor $\mathrm{F}_\omega : \mathbf{Set}^{\int \mathsf{H}} \to \mathbf{Set}^{\int \mathsf{H}}$ for system F_ω terms by

$$\begin{aligned} \mathrm{F}_\omega(A)(\Xi \mid \Gamma \vdash \tau : *) = \; & \mathrm{V}(\Xi \mid \Gamma \vdash \tau : *) \\ + & \coprod_{\tau_1,\tau_2 \in \mathbb{T}_*(\Xi)} (\tau \equiv \tau_1 \Rightarrow \tau_2) \times A(\Xi \mid \Gamma, \tau_1 \vdash \tau_2 : *) \\ + & \coprod_{\sigma \in \mathbb{T}_*(\Xi)} (A(\Xi \mid \Gamma \vdash \sigma \Rightarrow \tau : *) \times A(\Xi \mid \Gamma \vdash \sigma : *)) \\ + & \coprod_{\tau' \in \mathbb{T}_*(\Xi,\, \alpha.\kappa)} (\tau \equiv \forall(\alpha : \kappa.\tau')) \times A(\Xi,\, \alpha : \kappa \mid \mathsf{wk}_{\alpha.\kappa}(\Gamma) \vdash \tau' : *) \\ + & \coprod_{\substack{\sigma \in \mathbb{T}_*(\Xi) \\ \tau' \in \mathbb{T}_*(\Xi,\, \alpha.\kappa)}} (\tau \equiv \tau'[\alpha := \sigma]) \times A(\Xi \mid \Gamma \vdash \forall(\alpha : \kappa.\tau') : *). \end{aligned}$$

The weakening $\mathsf{wk}_{\alpha.\kappa} : \mathbb{F} \downarrow \mathbb{T}(\Xi) \to \mathbb{F} \downarrow \mathbb{T}(\Xi,\, \alpha : \kappa)$ maps a term context under a type context Ξ to the same term context but under a weakened one $(\Xi,\, \alpha : \kappa)$.

Theorem 8. T_ω *forms an initial* F_ω*-algebra.*

4.3 General Signature

Generalising the case of system F_ω, we arrive at the following definition.

Definition 9. A *higher-order polymorphic signature* $\Sigma = (\mathcal{K}, \Sigma^{\mathsf{ty}}, \Sigma^{\mathsf{tm}})$ consists of the following data.

- $\mathcal{K}$ is the set of all kinds.
- Σ^{ty} for types is a *second-order signature* [FH10], i.e. a set of type formers with an arity function $a : \Sigma^{\mathsf{ty}} \to (\mathcal{K}^* \times \mathcal{K})^* \times \mathcal{K}$. A type former with arity, denoted by
$$o : (\vec{\sigma}_1)\tau_1, \ldots, (\vec{\sigma}_l)\tau_l \to \tau$$
has l arguments, and binds $|\vec{\sigma}_i|$ variables of types $\vec{\sigma}_i$ in the i-th argument.
- Let $\mathbb{T} \in (\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{K}})^{\mathcal{K}}$ be the free Σ^{ty}-algebra over $\mathbb{V}$, represented by term syntax. This is the presheaf of all types.
- Σ^{tm} for terms is a set of function symbols with arities. This is denoted by
$$f : \langle\Theta_1\rangle(\vec{\sigma}_1)\tau_1 : \kappa_1, \ldots, \langle\Theta_l\rangle(\vec{\sigma}_l)\tau_l : \kappa_l \;\to\; \tau : \kappa$$
where $\Xi, \Theta_i \in \mathbb{F}\downarrow\mathcal{K}$, $\vec{\sigma}_i \in \mathbb{T}_{\kappa_i}(\Xi, \Theta_i)^*$, $\tau_i \in \mathbb{T}_{\kappa_i}(\Xi, \Theta_i)$, $\tau \in \mathbb{T}_\kappa(\Xi)$, has l arguments, and binds $|\Theta_i|$ type variables (of kind κ_i) and $|\vec{\sigma}_i|$ variables (of types $\vec{\sigma}_i$ that have the same kind κ_i) in the i-th argument ($1 \le i \le l$).

Example 10. The higher-order polymorphic signature $\Sigma_{\mathrm{F}_\omega} = (\mathcal{K}, \Sigma^{\mathsf{ty}}_{\mathrm{F}_\omega}, \Sigma^{\mathsf{tm}}_{\mathrm{F}_\omega})$ for system F_ω is as follows: $\mathcal{K} = \{*\} \cup \{\kappa_1 \Rightarrow \kappa_1 \mid \kappa_1, \kappa_2 \in \mathcal{K}\}$, $\Sigma^{\mathsf{ty}}_{\mathrm{F}_\omega}$ is

$$b : * \qquad \Rightarrow : *, * \to * \quad \forall_\kappa : (\kappa)* \to * \quad \lambda_{\kappa,\kappa'} : (\kappa')\kappa \to \kappa' \Rightarrow \kappa \quad @_{\kappa,\kappa'} : \kappa' \Rightarrow \kappa,\, \kappa' \to \kappa$$

generated by all $\kappa, \kappa' \in \mathcal{K}$, and $\Sigma^{\mathsf{tm}}_{\mathrm{F}_\omega}$ is

$$\begin{array}{llll} \mathsf{abs}_{\sigma,\tau} & : (\sigma)\tau : * \to \sigma \Rightarrow \tau : * & \mathsf{app}_{\sigma,\tau} & : \sigma \Rightarrow \tau : *,\;\; \sigma : * \to \tau : * \\ \mathsf{tabs}_{\tau',\kappa} & : \langle\kappa\rangle\tau' : * \to \forall_\kappa(\alpha.\tau') : * & \mathsf{tapp}_{\sigma,\tau'} & : \forall_\kappa(\alpha.\tau') : * \qquad \to \tau'[\alpha := \sigma] : * \end{array}$$

generated by all $\kappa \in \mathcal{K}$, $\Xi \in \mathbb{F}\downarrow\mathcal{K}$, $\kappa' \in \mathcal{K}$, $\sigma, \tau \in \mathbb{T}_{\kappa'}(\Xi)$, $\tau' \in \mathbb{T}_{\kappa'}(\Xi, \alpha)$.

Example 11. The higher-order polymorphic signature for system F is given by $\Sigma_{\mathrm{F}} = (\{*\}, \Sigma^{\mathsf{ty}}_{\mathrm{F}_\omega} - \{\lambda_{*,*}, @_{*,*}\}), \Sigma^{\mathsf{tm}}_{\mathrm{F}_\omega})$.

To a higher-order polymorphic signature Σ, we associate the *signature functor* $\Sigma : \mathbf{Set}^{\int \mathsf{H}} \to \mathbf{Set}^{\int \mathsf{H}}$ given by

$$\Sigma A(\Xi \mid \Gamma \vdash \tau : \kappa) = \coprod_{f : \langle \Theta_1 \rangle (\vec{\sigma}_1) \tau_1 : \kappa_1, \ldots \to \tau : \kappa \in \Sigma^{\mathsf{tm}}} \prod_{1 \le i \le l} A(\Xi, \Theta_i \mid \Gamma, \vec{\sigma}_i \vdash \tau_i : \kappa_i).$$

4.4 General Syntax Rules

If $\Xi \vdash \tau_i : \kappa$ for all $x_i : \tau_i \in \Gamma$, and $\Xi \vdash \tau : \kappa$, then a term judgment $\Xi \mid \Gamma \vdash t : \tau : \kappa$ is well-formed.

Well-kinded types

$$\frac{1 \le i \le n}{\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n \vdash \alpha_i : \kappa_i} \qquad \frac{\Xi, \vec{\alpha}_1 : \vec{\kappa}'_1 \vdash \tau_1 : \kappa_1 \ \cdots \ \Xi, \vec{\alpha}_l : \vec{\kappa}'_l \vdash \tau_l : \kappa_l}{\Xi \vdash o(\vec{\alpha}_1.\tau_1, \ldots, \vec{\alpha}_l.\tau_l\tau) : \kappa}$$

where $o : (\vec{\kappa}'_1)\kappa_1, \ldots, (\vec{\kappa}'_l)\kappa_l \to \kappa \in \Sigma^{\mathsf{ty}}$.

Well-typed terms

$$\frac{x : \tau \in \Gamma}{\Xi \mid \Gamma \vdash x : \tau : \kappa}$$

$$\frac{\Xi, \Theta_1 \mid \Gamma, \vec{x}_1 : \vec{\tau}_1 \vdash t_1 : \tau_1 : \kappa_1 \quad \cdots \quad \Xi, \Theta_l \mid \Gamma, \vec{x}_l : \vec{\tau}_l \vdash t_l : \tau_l : \kappa_l}{\Xi \mid \Gamma \vdash f(\vec{x}_1.t_1, \ldots, \vec{x}_l.t_l) : \tau : \kappa}$$

where $f : \langle \Theta_1 \rangle (\vec{\sigma}_1)\tau_1 : \kappa_1, \ldots, \langle \Theta_l \rangle (\vec{\sigma}_l)\tau_l : \kappa_l \to \tau : \kappa \in \Sigma^{\mathsf{tm}}$.

We define $\mathrm{TV}(\Xi \mid \Gamma \vdash \tau : \kappa) \triangleq \{t \mid (\Xi \mid \Gamma \vdash t : \tau : \kappa) \text{ is derivable}\}$, which we call *higher-order polymorphic abstract syntax.*

Theorem 12. *Given a higher-order polymorphic signature* Σ*,* TV *forms a free* Σ*-algebra over* V.

5 On Substitutions and Future Work

In this paper, we have focused on abstract syntax. In this final section, we briefly consider the equational axioms of system F and F_ω, and how we can express them in our framework. These remarks pertain to future work on seeking a general equational logic on polymorphic terms.

System F has the axioms:

$$\begin{array}{llcl}
(\beta) & \Xi \mid \Gamma \vdash (\lambda x \colon \sigma.\, t)\, s & = & t[x := s] \ : \tau \\
(\text{type app.}) & \Xi \mid \Gamma \vdash (\Lambda \alpha.t)\, \sigma & = & t[\alpha := \sigma] : \tau[\alpha := \sigma]
\end{array}$$

The terms of the left-hand sides of equations are just elements of the presheaf T of terms. In the right-hand sides and in types, various substitutions are used. We can model these as follows.

Substitution on types: $\tau[\alpha := \sigma]$**.** The category $\mathbf{Set}^{\mathbb{F}}$ has so-called a substitution monoidal structure [FPT99] $(\mathbf{Set}^{\mathbb{F}}, \bullet, \mathbb{V})$, where the monoidal product is given by a coend $(A \bullet B)(n) = \int^{m \in \mathbb{F}} A(m) \times B(n)^m$. The presheaf $\mathbb{T}$ of system F types is a monoid in $(\mathbf{Set}^{\mathbb{F}}, \bullet, \mathrm{V})$, and its multiplication $\mu^{\mathbb{T}} : \mathbb{T} \bullet \mathbb{T} \to \mathbb{T}$ models the substitution operation on types.

Substitution on terms: $t[x := s]$**.** Since both terms t and s are under the same type context Ξ, it suffices to consider the substitution monoidal structure in $(\mathbf{Set}^{\mathbb{F}{\downarrow}(\mathbb{T}(n))})^{\mathbb{T}(n)}$ for each $n = |\Xi| \in \mathbb{N}$. This case is covered by the substitution structure explored in [MS03, FH10], i.e. $(A \bullet B)_\tau(\Gamma) = \int^{\Delta \in \mathbb{F}{\downarrow}(\mathbb{T}(n))} A_\tau(\Delta) \times \prod_{1 \le i \le |\Delta|} B_{\Delta(i)}(\Gamma)$. The presheaf $\mathrm{T}(n \mid - \vdash -) \in (\mathbf{Set}^{\mathbb{F}{\downarrow}(\mathbb{T}(n))})^{\mathbb{T}(n)}$ of system F terms in a fixed type context n is a monoid in it, and its multiplication μ^{T} models the substitution operation on terms.

Substitution of a type for a type variable in a term: $t[\alpha := \sigma]$**.** It can be directly modelled by a map $\mathsf{tsub}_{\sigma,n} : \mathrm{T}(n+1 \mid \Gamma \vdash \tau) \to \mathrm{T}(n \mid \Gamma' \vdash \tau[n+1 := \sigma])$ defined by structural recursion on term that replaces each $n+1$ in a term with $\sigma \in \mathrm{T}(n)$ using $\mu^{\mathbb{T}}$, where $n+1$ is the de Bruijn level of the type variable α. The context Γ' is the one obtained by replacing all $n+1$ with σ in Γ.

Substitution on kinded types: $\tau[\alpha := \sigma]$**.** System F_ω has additionally the axiom

$$(\text{type}\,\beta) \qquad\qquad \Xi \mid \Gamma \vdash (\lambda\alpha\colon \kappa.\, \tau)\,\sigma \quad = \quad \tau[\alpha := \sigma] : \kappa'$$

The substitution in the right-hand side of the equation is modelled using the substitution monoidal structure in $(\mathbf{Set}^{\mathbb{F}{\downarrow}\mathcal{K}})^{\mathcal{K}}$ again following [MS03, FH10]. The presheaf $\mathbb{T}$ of F_ω types is a monoid in $((\mathbf{Set}^{\mathbb{F}{\downarrow}\mathcal{K}})^{\mathcal{K}}, \bullet, \mathbb{V})$, and its multiplication $\mu^{\mathbb{T}} : \mathbb{T} \bullet \mathbb{T} \to \mathbb{T}$ models the substitution operation on kinded types.

On the use of metavariables. When formalising an axiom, e.g. (β), there are actually two different views:

(1) (β) expresses infinitary many axioms generated by all concrete terms s, t.
(2) (β) should be regarded as a single axiom, where each letter "s" and "t" is the symbol of a *metavariable* denoting a concrete term.

Throughout this paper, we have taken the view (1). The view (2) was explored in [Ham04, Ham05, Fio08, FH10]. The presentation of axioms using metavariables is certainly more economical than (1), but technically more involved. This paper focuses on polymorphism in abstract syntax, so, for clarity, we did not go into the issue on metavariables. This should be explored in a future work.

Acknowledgments. I am grateful to Marcelo Fiore for important comments, especially related to the presheaf of variables. I also thank to the anonymous referees for useful comments that have improved the presentation of this paper. This work is supported by the JSPS Grant-in-Aid for Scientific Research (22700004).

References

[Acz78] Aczel, P.: A general Church-Rosser theorem. Technical report, University of Manchester (1978)

[AHS96] Altenkirch, T., Hofmann, M., Streicher, T.: Reduction-free normalisation for a polymorphic system. In: Proc. of LICS 1996, pp. 98–106 (1996)

[dB72] de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae 34, 381–391 (1972)

[FH10] Fiore, M., Hur, C.-K.: Second-order equational logic. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 320–335. Springer, Heidelberg (2010)

[Fio02] Fiore, M.: Semantic analysis of normalisation by evaluation for typed lambda calculus. In: Proc. of PPDP 2002, pp. 26–37. ACM Press, New York (2002)

[Fio08] Fiore, M.: Second-order and dependently-sorted abstract syntax. In: Proc. of LICS 2008, pp. 57–68 (2008)

[Fio09] Fiore, M.: Algebraic meta-theories and synthesis of equational logics (2009), Research Programme

[FPT99] Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding. In: Proc. of LICS 1999, pp. 193–202 (1999)

[Gro70] Grothendieck, A.: Catégories fibrées et descente (exposé VI). In: Grothendieck, A. (ed.) Revêtement Etales et Groupe Fondamental (SGA1). Lecture Notes in Mathematics, vol. 224, pp. 145–194. Springer, Heidelberg (1970)

[GTW76] Goguen, J., Thatcher, J., Wagner, E.: An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM T. J. Watson Research Center (1976)

[GUH06] Ghani, N., Uustalu, T., Hamana, M.: Explicit substitutions and higher-order syntax. Higher-Order and Symbolic Computation 19(2/3), 263–282 (2006)

[Ham04] Hamana, M.: Free Σ-monoids: A higher-order syntax with metavariables. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 348–363. Springer, Heidelberg (2004)

[Ham05] Hamana, M.: Universal algebra for termination of higher-order rewriting. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 135–149. Springer, Heidelberg (2005)

[Ham07] Hamana, M.: Higher-order semantic labelling for inductive datatype systems. In: Proc. of PPDP 2007, pp. 97–108. ACM Press, New York (2007)

[Ham10] Hamana, M.: Initial algebra semantics for cyclic sharing tree structures. Logical Methods in Computer Science 6(3) (2010)

[Hof99] Hofmann, M.: Semantical analysis of higher-order abstract syntax. In: Proc. of LICS 1999, pp. 204–213 (1999)

[Kat04] Katsumata, S.: A generalisation of pre-logical predicates to simply typed formal systems. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 831–845. Springer, Heidelberg (2004)

[MA09] Morris, P., Altenkirch, T.: Indexed containers. In: LICS 2009, pp. 277–285 (2009)

[Mic08] Miculan, M.: A categorical model of the Fusion calculus. In: Proc. of MFPS XXIV. ENTCS, vol. 218, pp. 275–293. Elsevier, Amsterdam (2008)

[MS03] Miculan, M., Scagnetto, I.: A framework for typed HOAS and semantics. In: Proc. of PPDP 2003, pp. 184–194. ACM Press, New York (2003)

[SP82] Smyth, M.B., Plotkin, G.D.: The category-theoretic solution of recursive domain equations. SIAM J. Comput 11(4), 763–783 (1982)

[TP08] Tanaka, M., Power, J.: Category theoretic semantics for typed binding signatures with recursion. Fundam. Inform. 84(2), 221–240 (2008)

Asymptotic Information Leakage under One-Try Attacks*

Michele Boreale[1], Francesca Pampaloni[2], and Michela Paolini[2]

[1] Università di Firenze, Italy
[2] IMT, Lucca, Italy

Abstract. We study the asymptotic behaviour of (a) information leakage and (b) adversary's error probability in information hiding systems modelled as noisy channels. Specifically, we assume the attacker can make a single guess after observing n independent executions of the system, throughout which the secret information is kept fixed. We show that the asymptotic behaviour of quantities (a) and (b) can be determined in a simple way from the channel matrix. Moreover, simple and tight bounds on them as functions of n show that the convergence is exponential. We also discuss feasible methods to evaluate the rate of convergence. Our results cover both the Bayesian case, where a prior probability distribution on the secrets is assumed known to the attacker, and the maximum-likelihood case, where the attacker does not know such distribution. In the Bayesian case, we identify the distributions that maximize the leakage. We consider both the min-entropy setting studied by Smith and the additive form recently proposed by Braun et al., and show the two forms do agree asymptotically. Next, we extend these results to a more sophisticated eavesdropping scenario, where the attacker can perform a (noisy) observation at each state of the computation and the systems are modelled as hidden Markov models.

Keywords: security, quantitative information leakage, information theory, Bayes risk, hidden Markov models.

1 Introduction

In recent years there has been much interest in formal models to reason about quantitative information leakage in computing systems [9,7,3,14,1,21,22]. A general situation is that of a program, protocol or device carrying out computations that depend probabilistically on a secret piece of information, such as a password, the identity of a user or a private key. We collectively designate these as *information hiding systems*, following a terminology established in [7]. During the computation, some observable information related to the secret may be disclosed. This might happen either by design, e.g. if the output of the system is directly related to the secret (think of a password checker denying access), or for reasons depending on the implementation. In the latter case, the

* Work partially supported by the EU funded project ASCENS. Corresponding author: Michele Boreale, Università di Firenze, Dipartimento di Sistemi e Informatica, Viale Morgagni 65, I-50134 Firenze, Italy. E-mail: boreale@dsi.unifi.it

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 396–410, 2011.

observable information may take the form of physical quantities, such as the execution time or the power consumption of the device (think of timing and power attacks on smart cards [12,13]). The observable information released by the system can be exploited by an eavesdropper to reconstruct the secret, or at least to limit the search space. This is all the more true when the eavesdropper is given the ability of observing several executions of the system, thus allowing her/him to mount some kind of statistical attack.

A simple but somehow crucial remark due to Chatzikokolakis et al. [7] is that, for the purpose of quantifying the amount of secret information that is leaked, it is useful to view an information hiding system as a *channel* in the sense of Information Theory: the inputs represent the secret information, the outputs represent the observable information and the two sets are related by a conditional probability matrix. This remark suggests a natural formalization of leakage in terms of Shannon entropy based metrics, like mutual information and capacity. In fact, by a result due to Massey [18], these quantities are strongly related to the resistance of the system against *brute-force* attacks. Specifically, Shannon entropy is related to the average number of questions of the form "is the secret equal to x?" an attacker has to ask an oracle in order to identify the secret *with certainty*. In a recent paper, Smith [21] objects that, even if the number of such questions is very high, the attacker might still have a significant chance of correct guess in just one or very few attempts. Smith demonstrates that *min-entropy* quantities, based on error probability (a.k.a. *Bayes risk*), are more adequate to express leakage in this *one-try* scenario. Whatever the considered attack scenario, brute-force or one-try, the analytic computation of leakage is in general difficult or impossible. Henceforth, a major challenge is being able to give simple and tight bounds on leakage in general, or exact expressions that exploit specific properties of a system (e.g. symmetries in the channel matrix) in some special cases.

In the present paper, we tackle these issues in a scenario of one-try attacks and system re-execution. More precisely, we assume the attacker makes his guess after observing several, say n, independent executions of the system, throughout which the secret information is kept fixed. In real-world situations, re-execution may happen either forced by the attacker (think of an adversary querying several times a smart card), or by design (think of routing paths established repeatedly between a sender and a receiver in anonymity protocols like Crowds [20]). Since the computation is probabilistic, in general the larger the number n of observed executions, the more information will be gained by the attacker. Therefore, it is important to asses the resistance of a system in this scenario.

Our goal is to describe the asymptotic behaviour of the adversary's error probability and of information leakage as n goes to ∞. We show that the asymptotic values of these quantities can be determined in a simple way from the channel matrix. Moreover, we provide simple and tight bounds on error probability and on leakage as functions of n, showing that the convergence is exponential. We also discuss feasible methods for evaluating the rate of convergence. Our results cover both the Bayesian case (MAP rule), where a prior probability distribution on the secrets is assumed known to the attacker, and the maximum-likelihood case (ML rule), where the attacker does not know such distribution. In the Bayesian case, we identify the distributions that maximize leakage.

We consider both the min-entropy leakage studied by Smith [21] and the additive form recently proposed by Braun et al. [6], and show the two forms do agree asymptotically.

We next consider a more sophisticated scenario, where computations of the system may take several steps to terminate, or even not terminate at all. In any case, to each state of the computation there corresponds one (in general, noisy) observation on the part of the attacker. Hence, to each computation there corresponds a sequential *trace* of observations. The attacker may collect multiple such traces, corresponding to multiple independent executions of the system. Like in the simpler scenario, the secret is kept fixed throughout these executions. This set up is well suited to describe situations where the attacker collects information from different sources at different times, like in a coalition of different local eavesdroppers. An instance of this situation in the context of an anonymous routing application will be examined. We formalize this scenario in terms of discrete-time *Hidden Markov Models* [19] and then show that the results established for the simpler scenario carry over to the new one.

Throughout the paper, we illustrate our results with a few examples: the modular exponentiation algorithm used in public-key cryptography, the Crowds anonymity protocol, and onion routing protocols [11] in a network with a fixed topology. Additional examples are provided in [4].

Related work. The last few years have seen a flourishing of research on quantitative models of information leakage. In the context of language-based security, Clark et al. [9] first motivated the use of mutual information to quantify information leakage in a setting of imperative programs. Boreale [3] extended this study to the setting of process calculi, and introduced a notion of rate of leakage. In both cases, the considered systems do not exhibit probabilistic behaviour. Closely related to ours is the work by Chatzikokolakis, Palamidessi and their collaborators. [7] examines information leakage mainly from the point of view of Shannon entropy and capacity, but also contains results on asymptotic error probability, showing that, independently from the input distribution, the ML rule approximates the MAP rule. [8] studies error probability mainly relative to one observation ($n = 1$), but also offers a lower-bound in the case of repeated observations [8, Proposition 7.4]. This lower-bound is generalized by our results. Compositional methods based on process algebras are discussed in [5]; there, the average ML error probability is characterized in terms of MAP error probability under a uniform distribution of inputs. [6] introduces the notion of additive leakage and compares it to the min-entropy based leakage considered by Smith [21], but again in the case of a single observation.

A model of "unknown-message" attacks is considered by Backes and Köpf in [1]. This model is basically equivalent to the information hiding systems considered in [7,8,6] and in the present paper. Backes e Köpf too consider a scenario of repeated independent observations, but from the point of view of Shannon entropy, rather than of error probability. They rely on the information-theoretic method of types to determine the asymptotic behaviour of the considered quantities, as we do in the present paper. An application of their setting to the modular exponentiation algorithm is the subject of [15], where the effect of *bucketing* on security of RSA is examined (see Section 5). This study has recently been extended to the case of one-try attacks by Köpf and Smith in [16]. Earlier, Köpf and Basin had considered a scenario of adaptive chosen-message attacks [14]. They offer an algorithm to compute conditional Shannon entropy in this setting, but not a study of its asymptotic behaviour, which seems very difficult to characterize.

In the context of side-channel cryptanalysis, Standaert et al. propose a framework to reason on side-channel correlation attacks [22]. Both a Shannon entropy based metric and a security metric are considered. This model does not directly compare to ours, since, as we will discuss in Section 5, correlation attacks are inherently known-message – that is, they presuppose the explicit or implicit knowledge of the plaintext on the part of the attacker.

Structure of the paper. The rest of the paper is organized as follows. Section 2 establishes some notations and terminology. Section 3 introduces the model and the quantities that are the object of our study. Section 4 discusses the main results about error probability and leakage. Section 5 illustrates these results with a few examples. Section 6 presents the extension to hidden Markov models. Section 7 contains some concluding remarks. Proofs not reported in this short version for lack of space can be found in the full version [4].

2 Notations and Preliminary Notions

Let $\mathcal{A}$ be a finite nonempty set. A probability distribution on a $\mathcal{A}$ is a function $p : \mathcal{A} \to [0,1]$ such that $\sum_{a\in\mathcal{A}} p(a) = 1$. For any $A \subseteq \mathcal{A}$ we let $p(A)$ denote $\sum_{a\in A} p(a)$. Given $n \geq 0$, we let $p^n : \mathcal{A}^n \to [0,1]$ be the n-th extension of p, defined as $p^n((a_1,\dots,a_n)) \overset{\triangle}{=} \Pi_{i=1}^n p(a_i)$; this is in turn a probability distribution on $\mathcal{A}^n$. For $n = 0$, we set $p^0(\epsilon) = 1$, where ϵ denotes here the empty string. Given two distributions p and q on $\mathcal{A}$, the *Kullback-Leibler (KL) divergence* of p and q is defined as (all the log's are taken with base 2)

$$D(p||q) \overset{\triangle}{=} \sum_{a\in\mathcal{A}} p(a)\cdot \log\frac{p(a)}{q(a)}$$

with the proviso that $0 \cdot \log\frac{0}{q(a)} = 0$ and that $p(a)\cdot\log\frac{p(a)}{0} = +\infty$ if $p(a) > 0$. It can be shown that $D(p||q) \geq 0$, with equality if and only if $p = q$ (*Gibbs inequality*). KL-divergence can be thought of as a sort of distance between p and q, although strictly speaking it is not – it is not symmetric, nor satisfies the triangle inequality.

$\Pr(\cdot)$ will generally denote a probability measure. Given a random variable X taking values in $\mathcal{A}$, we write $X \sim p$ if X is distributed according to p, that is for each $a \in \mathcal{A}$, $\Pr(X = a) = p(a)$.

3 Probability of Error, Leakage, Indistinguishability

An *information hiding system* is a quadruple $\mathcal{H} = (\mathcal{S}, \mathcal{O}, p(\cdot), p(\cdot|\cdot))$, composed by a finite set of *states* $\mathcal{S} = \{s_1, ..., s_m\}$ representing the secret information, a finite set of *observables* $\mathcal{O} = \{o_1, ..., o_l\}$, an a priori probability distribution on $\mathcal{S}$, p, and a *conditional probability matrix*, $p(\cdot|\cdot) \in [0,1]^{\mathcal{S}\times\mathcal{O}}$, where each row sums up to 1. The entry of row s and column o of this matrix will be written as $p(o|s)$, and represents the probability of observing o given that s is the (secret) input of the system. For each s, the s-th row of the matrix is identified with the probability distribution $o \mapsto p(o|s)$ on $\mathcal{O}$, denoted by p_s. The probability distribution p on $\mathcal{S}$ and the conditional probability matrix $p(o|s)$ together induce a probability distribution r on $\mathcal{S}\times\mathcal{O}$ defined as $r(s,o) \triangleq p(s)\cdot p(o|s)$, hence a pair of random variables $(S,O) \sim r$, with S taking values in $\mathcal{S}$ and O taking values in $\mathcal{O}$. Note that $S \sim p$ and, for each s and o s.t. $p(s) > 0$, $\Pr(O = o|S = s) = p(o|s)$.

Let us now discuss the attack scenario. Given any $n \geq 0$, we assume the adversary is a passive eavesdropper that gets to know the observations corresponding to n independent executions of the system, $o^n = (o_1, ..., o_n) \in \mathcal{O}^n$, throughout which the secret state s is kept fixed. Formally, the adversary knows a random vector of observations $O^n = (O_1, ..., O_n)$ such that, for each $i = 1, ..., n$, O_i is distributed like O and the individual O_i are *conditionally independent* given S, that is, the following equality holds true for each $o^n \in \mathcal{O}^n$ and $s \in \mathcal{S}$ s.t. $p(s) > 0$

$$\Pr\left(O^n = (o_1, \ldots, o_n) \,|\, S = s\right) = \Pi_{i=1}^{n} p(o_i|s)\,.$$

We will often abbreviate the right-hand side of the above equation as $p(o^n|s)$. For any n, the attacker strategy is modeled by a function $g : \mathcal{O}^n \to \mathcal{S}$, called *guessing function*: this represents the single guess the attacker is allowed to make about the secret state s, after observing o^n.

Definition 1 (error probability). *Let $g : \mathcal{O}^n \to \mathcal{S}$ be a guessing function. The* probability of error after n observations *(relative to g) is given by $P_e^{(g)}(n) \triangleq 1 - P_{succ}(n)$, where $P_{succ}^{(g)}(n) \triangleq \Pr(g(O^n) = S)$.*

It is well-known (see e.g. [10]) that the optimal strategy for the adversary, that is the one that minimizes the error probability, is the Maximum A Posteriori (MAP) rule, defined below.

Definition 2 (Maximum A Posteriori rule, MAP). *A function $g : \mathcal{O}^n \to \mathcal{S}$ satisfies the* Maximum A Posteriori (MAP) criterion *if for each o^n and s, $g(o^n) = s$ implies $p(o^n|s)$ $p(s) \geq p(o^n|s')p(s')$ for each s'.*

In the above definition, for $n = 0$ one has $o^n = \epsilon$, and it is convenient to stipulate that $p(\epsilon|s) = 1$: that is, with no observations at all, g selects some s maximizing the prior distribution. With this choice, $P_e^{(g)}(0)$ denotes $1 - \max_s p(s)$. It worthwhile to note that, once n and $p(s)$ are fixed, the MAP guessing function is not in general unique. It is readily checked, though, that $P_e(n)$ does *not* depend on the specific MAP function g that is chosen. Hence, throughout the paper we assume w.l.o.g. a fixed guessing function g for each given n and probability distribution $p(s)$. We shall omit the superscript $^{(g)}$, except where this might cause confusion.

Another widely used criterion is *Maximum Likelihood (*ML*)*, which given o^n selects a state s maximizing the likelihood $p(o^n|s)$ among all the states. ML coincides with MAP if the uniform distribution on the states is assumed. ML is practically important because it requires no knowledge of the prior distribution, which is often unknown in security applications. Our main results will also apply to the ML rule (see Remark 2 in the next section).

We now come to information leakage: this is a measure of the information leaked by the system, obtained by comparing the prior and the posterior (to the observations) success probabilities. Indeed, two flavours of this concept naturally arise, depending on how the comparison between the two probabilities is expressed. If one uses subtraction, one gets the additive form of [6], while if one uses the ratio between them, one gets a multiplicative form. In the latter case, one could equivalently consider the difference of the log's, obtaining the *min-entropy* based definition considered by Smith [21].

Definition 3 (Additive and Multiplicative Leakage [6,21]). *The* additive *and* multiplicative leakage *after n observations are defined respectively as* $L_+(n) \overset{\triangle}{=} P_{succ}(n) - \max_s p(s)$ *and* $L_\times(n) \overset{\triangle}{=} \frac{P_{succ}(n)}{\max_s p(s)}$.

In an information hiding system, it may happen that two secret states induce the same distribution on the observables. A common example is that of a degenerate channel matrix modelling a deterministic function $\mathcal{S} \to \mathcal{O}$ with $|\mathcal{O}| < |\mathcal{S}|$. An important role in determining the fundamental security parameters of the system will be played by an indistinguishability equivalence relation over states, which is defined in the following. Recall that, for each $s \in \mathcal{S}$, we let p_s denote the probability distribution $p(\cdot|s)$ on $\mathcal{O}$.

Definition 4 (Indistinguishability). *Given* $s, s' \in \mathcal{S}$, *we let* $s \equiv s'$ *iff* $p_s = p_{s'}$.

Concretely, two states are indistinguishable iff the corresponding rows in the conditional probability matrix are the same. This intuitively says that there is no way for the adversary to tell them apart, no matter how many observations he performs. We stress that this definition does not depend on the prior distribution on states, nor on the number n of observations.

4 Bounds and Asymptotic Behaviour

Let $\mathcal{S}/\equiv$ be $\{C_1, ..., C_K\}$, the set of equivalence classes of $\equiv$. For each $i = 1, ..., K$, let

$$s_i^* \overset{\triangle}{=} \mathrm{argmax}_{s \in C_i} p(s) \quad \text{and} \quad p_i^* \overset{\triangle}{=} p(s_i^*)\,. \tag{1}$$

We assume wlog that $p_i^* > 0$ for each $i = 1, ..., K$ (otherwise all the states in class C_i can be just discarded from the system).

Main results. We shall prove the following bounds and asymptotic behaviour for $P_e(n)$.

Theorem 1. *$P_e(n)$ converges exponentially fast to $1 - \sum_{i=1}^K p_i^*$. More precisely, there is $\epsilon > 0$ s.t.*

$$1 - \textstyle\sum_{i=1}^K p_i^* \;\le\; P_e(n) \;\le\; 1 - (\sum_{i=1}^K p_i^*) \cdot r(n)$$

where $r(n) = 1 - (n+1)^{|\mathcal{O}|} \cdot 2^{-n\epsilon}$. Here, the lower-bound holds true for any n, while the upper-bound holds true for any $n \ge n_0 \overset{\triangle}{=} \epsilon^{-1} \cdot \max_{i,j} \log(\frac{p_i^*}{p_j^*})$. Moreover, ϵ only depends on the rows $p_{s_i^*}$ $(i = 1, ..., K)$ of the conditional probability matrix $p(\cdot|\cdot)$.

Note that in the practically important case of the uniform distribution on states, we have $n_0 = 0$, that is the upper-bound as well holds true for any n. The theorem has a simple interpretation in terms of the attacker's strategy: after infinitely many observations, he can determine the indistinguishability class of the secret, say C_i, and then guess the most likely state in that class, s_i^*. In order to discuss this result, we recall some terminology and a couple of preliminary results from the information-theoretic method of types [10, Ch.11]. Given $n > 0$, a sequence $o^n \in \mathcal{O}^n$ and a symbol $o \in \mathcal{O}$, let us denote by $\mathrm{n}(o, o^n)$ the number of occurrences of o inside o^n. The *type* (or empirical distribution) of o^n is the probability distribution t_{o^n} on $\mathcal{O}$ defined as: $t_{o^n}(o) \triangleq \frac{\mathrm{n}(o,o^n)}{n}$. Let q any probability distribution on $\mathcal{O}$. A *neighborhood* of q is a subset of n-sequences of $\mathcal{O}^n$ whose empirical distribution is close to q. Formally, for each $n \geq 1$ and $\epsilon > 0$

$$U_q^{(n)}(\epsilon) \triangleq \{o^n \in \mathcal{O}^n \mid D(t_{o^n} \| q) \leq \epsilon\} .$$

The essence of the method of types is that (i) there is only a polynomial number of types in n, and that (ii) the probability under q of the set of n-sequences of a given type decreases exponentially with n, at a rate determined by the KL-divergence between q and that type. These considerations are made precise and exploited in the proof of the following lemma, which can be found in [10, Ch.11]. The lemma basically says that the probability that a sequence falls in a neighborhood of q of radius ϵ approaches 1 exponentially fast with n.

Lemma 1. *Let q be a probability distribution on $\mathcal{O}$. Then $q^n(U_q^{(n)}(\epsilon)) \geq 1-(n+1)^{|\mathcal{O}|} \cdot 2^{-n\epsilon}$.*

For any $s \in \mathcal{S}$, we let $A_s^{(n)} \triangleq g^{-1}(s) \subseteq \mathcal{O}^n$ be the *acceptance region* for state s. We note that it is not restrictive to assume that g maps each o^n in one of the K representative elements $s_1^*, ..., s_K^*$ that maximize the prior: indeed, if this were not the case, it would be immediate to build out of g a new MAP function that fulfills this requirement. Thus, from now on we will assume w.l.o.g. that $A_s^{(n)} = \emptyset$ for $s \neq s_1^*, ..., s_K^*$. For the sake of notation, from now on we will denote $U_{p_{s_i^*}}^{(n)}$ as $U_i^{(n)}$ and $A_{s_i^*}^{(n)}$ as $A_i^{(n)}$, for $i = 1, ..., K$. The sets $U_i^{(n)}$ and $A_i^{(n)}$ are related by the following lemma.

Lemma 2. *There is $\epsilon > 0$, not depending on the prior probability on states, such that for each $n \geq n_0$ as defined in Theorem 1 and for each $i = 1, ..., K$, it holds that $U_i^{(n)}(\epsilon) \subseteq A_i^{(n)}$.*

We now come to the proof of the main theorem above.

Proof. (of Theorem 1). We focus equivalently on the probability of success, $P_{succ}(n)$. Under the assumptions on g explained above, we compute as follows

$$\begin{aligned} P_{succ}(n) &= \textstyle\sum_{s\in\mathcal{S}} \Pr(g(O^n) = S \,|\, S = s)p(s) = \sum_{s\in\mathcal{S}} p_s^n(A_s^{(n)})p(s) \\ &= \textstyle\sum_{i=1}^K \underbrace{p_{s_i^*}^n(A_i^{(n)})}_{\leq 1} p_i^* \qquad \leq \qquad \sum_{i=1}^K p_i^* \end{aligned}$$

which implies the lower-bound in the statement. Choose now ϵ as given by Lemma 2. Let $n \geq n_0$. Note that for $n = 0$ the upper-bound holds trivially, as $P_e(0) = 1 - \max_s p(s)$, so assume $n \geq 1$. For each $i = 1, ..., K$ we have

$$p_{s_i^*}^n(A_i^{(n)}) \;\geq\; p_{s_i^*}^n(U_i^{(n)}(\epsilon)) \;\geq\; 1-(n+1)^{|\mathcal{O}|} \cdot 2^{-n\epsilon}$$

where the first inequality comes from Lemma 2 and second one from Lemma 1. In the end, from $P_{succ}(n) = \sum_{i=1}^K p_{s_i^*}^n(A_i^{(n)})p_i^*$, we obtain that for $n \geq n_0$

$$P_{succ}(n) \;\geq\; (\sum_{i=1}^K p_i^*) \cdot (1-(n+1)^{|\mathcal{O}|} \cdot 2^{-n\epsilon})$$

which implies the upper-bound in the statement.

Remark 1. In the expression for $r(n)$, the term $(n+1)^{|\mathcal{O}|}$ is a rather crude upper bound on the number of types of n-sequences. It is possible to replace this term with the expression $\binom{n+|\mathcal{O}|-1}{|\mathcal{O}|-1}$, which is less easy to manipulate analytically, but gives the exact number of types, hence a more accurate upper bound on $P_e(n)$.

The following results show that, asymptotically, the security of the systems is tightly connected to the number of its indistinguishability classes – and in the case of uniform prior distribution *only* depends on this number.

Corollary 1. *If the a priori distribution on $\mathcal{S}$ is uniform, then $P_e(n)$ converges exponentially fast to $1 - \frac{K}{|\mathcal{S}|}$.*

Remark 2 (on the ML *rule).* [5] shows that the probability of error under the ML rule, *averaged* on all distributions, coincides with the probability of error under the MAP rule and the uniform distribution. From Corollary 1 we therefore deduce that the average ML error converges exponentially fast to the value $1 - \frac{K}{|\mathcal{S}|}$ as $n \to \infty$.

We discuss now some consequences of the above results on information leakage. Recall that for $i = 1, ..., K$, we call s_i^* a representative of the indistinguishability class C_i that maximizes the prior distribution $p(s)$ in the class C_i, and let $p_i^* = p(s_i^*)$. Assume w.l.o.g. that $p_1^* = \max_s p(s)$. In what follows, we denote by p_{max} the distribution on $\mathcal{S}$ defined by: $p_{max}(s) = \frac{1}{K}$ if $s \in \{s_1^*, ..., s_K^*\}$ and $p_{max}(s) = 0$ otherwise.

Corollary 2. *1. $L_+(n)$ converges exponentially fast to $\sum_{i=2}^K p_i^*$. This value is maximized by the prior distribution p_{max}, which yields the limit value $1 - \frac{1}{K}$.*

2. $L_\times(n)$ converges exponentially fast to $\frac{\sum_{i=1}^K p_i^*}{p_1^*}$. This value is maximized by the prior distribution p_{max}, which yields the limit value K.

Remark 3. A consequence of Corollary 2(2) is that, in the case of uniform distribution on states, the multiplicative leakage coincides with the number of equivalence classes K. This generalizes a result of [21] for deterministic systems.

In [6] additive and multiplicative leakages are compared in the case of a single observation ($n = 1$). It turns out that, when comparing two systems, the two forms of leakage are in agreement, in the sense that they individuate the same maximum-leaking system w.r.t. a fixed prior distribution on inputs. However, [6] also shows that the two forms disagree as to the distribution on inputs that maximizes leakage w.r.t. a fixed system. This is shown to be the uniform distribution in the case of multiplicative leakage, and a function that uniformly distributes the probability on the set of "corner points" in the case of additive leakage (see [6] for details). Here, we have shown that, despite this difference, additive and multiplicative leakage do agree on the maximizing distribution asymptotically.

Rate of convergence. The quantity ϵ in the statement of Theorem 1 determines how fast the error probability approaches its limit value. Let us call *achievable* any $\epsilon > 0$ for which the upper bound in Theorem 1 holds true for any $n \geq n_0$. The following result gives sufficient and practical conditions for achievability. Let us stress that the achievable rates given by this proposition do not depend on the prior distribution, but only on the relation $\equiv$, and specifically on the minimum norm 1 distance between equivalence classes: the larger this distance, the higher the achievable rates. This result is essentially a re-elaboration on [10, Lemma 11.6.1].

Proposition 1. *Let* $\delta \triangleq \min_{s_i \not\equiv s_j} \|p_{s_i} - p_{s_j}\|_1$. *Then any rate* ϵ *satisfying* $0 < \epsilon < \frac{\delta^2}{16 \ln 2}$ *is achievable. Moreover, if* $p_1^* = p_2^* = \cdots = p_K^*$, *the second inequality can be weakened to* $\epsilon < \frac{\delta^2}{8 \ln 2}$.

The above result prompts the following question. Suppose one somehow ignores the rows of $p(\cdot|\cdot)$ that are close together with each other, and only consider rows that are far from each other: is it then possible to achieve a higher rate of convergence ϵ? The answer is expected to be *yes*, although ignoring some rows might lead to a possibly higher asymptotic error probability. In other word, it should be possible to trade off accuracy in guessing with rate of convergence. This is the content of the next proposition.

Proposition 2. *Let* $\emptyset \neq \mathcal{S}_0 \subseteq \{s_1^*, ..., s_K^*\}$. *Then there is* $\epsilon > 0$ *only depending on the rows* p_s, $s \in \mathcal{S}_0$, *of* $p(\cdot|\cdot)$, *such that for each* $n \geq n_0 \triangleq \epsilon^{-1} \max_{s_i^*, s_j^* \in \mathcal{S}_0} \log(\frac{p_i^*}{p_j^*})$, *it holds true that*

$$P_e(n) \leq 1 - (\sum_{s_j^* \in \mathcal{S}_0} p_j^*) \cdot r(n) \quad \textit{with} \quad r(n) = 1 - (n+1)^{|\mathcal{O}|} \cdot 2^{-n\epsilon} .$$

These concepts are demonstrated in the following example.

An example. Let $\mathcal{S} = \{s_1, s_2, s_3, s_4\}$ and $\mathcal{O} = \{o_1, o_2, o_3\}$. The prior probability distribution on $\mathcal{S}$ is defined by: $p(s_1) = p(s_3) = \frac{1}{2} - 10^{-9}$ and $p(s_2) = p(s_4) = 10^{-9}$. The conditional probability matrix is defined in the table on the right.

Note that $s_1 \equiv s_2$. Applying Theorem 1, we find that, for n sufficiently large, $1 - E \leq P_e(n) \leq 1 - E \cdot r(n)$, where $E = 1 - 10^{-9}$ and $r(n) = 1 - (n+1)^3 \cdot 2^{-n\epsilon}$. Applying Proposition 1, we find that any rate $\epsilon < 3.6067 \times 10^{-11}$ is achievable. Thus the convergence to the value $1 - E = 10^{-9}$ is very slow. One wonders if there is some value $1 - E'$ that is only slightly higher than $1 - E$, but that can be reached much faster. This is indeed the case. Observe that the rows s_3 and s_4 are very close with each other in norm-1 distance: $\|p_{s_3} - p_{s_4}\|_1 = 2 \times 10^{-5}$. We can discard s_4, which has a very small probability, and apply Proposition 2 with $\mathcal{S}_0 = \{s_1, s_3\}$ to get $P_e(n) \leq 1 - E' \cdot r'(n)$, where $E' = \frac{1}{2} - 10^{-9} + \frac{1}{2} - 10^{-9} = 1 - 2 \times 10^{-9}$ and $r'(n) = 1 - (n+1)^3 \cdot 2^{-n\epsilon'}$. The rate ϵ' can be computed by applying the second part of Proposition 1, as $p(s_1) = p(s_3)$. By doing so, we get that any $\epsilon' < 0.18034$ is achievable. This implies that the value $1 - E'$ is approached much faster as n grows. For instance, already after $n = 350$ observations we get that $(1 - E')/P_e(n) > 0.99$.

	o_1	o_2	o_3
s_1	$\frac{1}{2}$	0	$\frac{1}{2}$
s_2	$\frac{1}{2}$	0	$\frac{1}{2}$
s_3	0	$\frac{1}{2}$	$\frac{1}{2}$
s_4	0	$\frac{1}{2} - 10^{-5}$	$\frac{1}{2} + 10^{-5}$

5 Examples

Timing leaks and blinding in modular exponentiation. In the '90's, P. Kocher showed that RSA and other public-key crypto-systems are subject to side-channel attacks exploiting information leaked by implementations of the modular exponentiation algorithm, such as execution time [12] and/or power consumption [13]. Many of these attacks are based on the assumption that the attacker can observe repeated independent execution of the system, throughout which the exponent – the secret key – is kept fixed. Here, we concentrate on timing attacks. *Blinding* [12] was early proposed as a countermeasure to thwart such attacks. The essence of blinding is that exponentiation is performed on a random message unknown to the attacker, rather than on the original message (to be decrypted or digitally signed) known to the attacker. This appears to be sufficient to thwart Kocher's attack, which is of chosen-ciphertext type.

Köpf and Dürmuth [15] have recently quantified the degree of protection provided by blinding when it is enhanced by *bucketing*, a technique by which the algorithm's execution times are adjusted so as to always fall in one of few predefined values. Köpf and Smith have extended this result to the case of one-try attacks and min-entropy [16]. Below, we refine Köpf and Smith's analysis, under a reasonable assumption on the functioning of the algorithm that will be described shortly. We consider an implementation of the modular exponentiation algorithm with blinding, but *no* bucketing. To such an implementation, there corresponds an information hiding system where: $\mathcal{S} = \mathcal{K} = \{0, 1\}^N$ is the set of private keys, i.e. the possible exponents of the algorithm, over which we assume a uniform distribution[1]; $\mathcal{O} = \{t_1, t_2, ...\}$ is the set of possible execution times; $p(t|k)$ is the probability that, depending on the deciphered message, the execution of the algorithm takes times t given that the private key is k. To be more specific about the last point, we assume an underlying set of messages $\mathcal{M}$, with a known prior distribution $p_M(m)$, and a function time : $\mathcal{M}\times\mathcal{S} \to \mathcal{O}$ that yields the duration of the execution of the algorithm when its argument is a given pair (m, k). Then the entries of the probability matrix $p(t|k)$ can be defined thus

$$p(t|k) \overset{\triangle}{=} \textstyle\sum_{m\in\mathcal{M}:\mathrm{time}(m,k)=t} p_M(m)\,.$$

Now, modular exponentiation functions in such a way that at the i-th iteration ($0 \leq i < N$), either a squaring or *both* a squaring and a multiply are performed, depending on whether the i-th bit of the exponent is 0 or 1. Given this functioning, it seems reasonable to assume that, for each m, *the execution time only depends on the number of '1' digits in* k. In other words, we assume that whenever k and k' have the same Hamming weight, $\mathrm{time}(m, k) = \mathrm{time}(m, k')$, for any m. From this assumption and the definition of $p(t|k)$, it follows that whenever k and k' have the same Hamming weight then $p(\cdot|k) = p(\cdot|k')$. So, in the system there are *at most* as many $\equiv$-classes as Hamming weights, that is $N + 1$. The results in Section 4 then allow us to conclude that for any n

$$P_e(n) \geq 1 - \tfrac{N+1}{2^N}\,.$$

For any practical size of the key, say $N = 1024$, this value is ≈ 1. Accordingly, additive and multiplicative leakage satisfy, asymptotically,

$$L_+ \leq \tfrac{N}{2^N} \quad \text{and} \quad L_\times \leq N + 1\,.$$

For any practical size of the key, say $N = 1024$, these upper bounds yield negligible values: $L_+ \approx 0$ and $L_\times \leq 1025$. In the latter case, taking the log we obtain that no more than $\log(1025) = 10.001$ bits of min-entropy are leaked, out of 1024. In conclusion,

[1] In the case of RSA, a negligible fraction of the exponents is ruled out by virtue of number theoretic requirements, so the resulting distribution is not exactly uniform on $\mathcal{S}$. This fact does not substantially affect the significance of our analysis.

under the further assumption on the behaviour of modular exponentiation we made above, blinding alone appears to provide satisfactory guarantees of security against one-try attacks.

Protocol re-execution in Crowds. The Crowds protocol [20] is designed for protecting the identity of the senders of messages in a network where some of the nodes may be corrupted, that is, under the control of an attacker. Omitting a few details, the functioning of the protocol can be described quite simply: the sender first forwards the message to a node of the network chosen at random; at any time, any node holding the message can decide whether to (a) forward in turn the message to another node chosen at random, or (b) submit it to the final destination. The choice between (a) and (b) is made randomly, with alternative (a) being assigned probability p_f (forwarding probability) and alternative (b) probability $1 - p_f$. The rationale here is that, even if a corrupted node C receives the message from a node N (in the Crowds terminology, C *detects* N), C, hence the attacker, cannot decide whether N is the original sender or just a forwarder. In fact, given that N is detected, the probability of N being the true sender is only slightly higher than that of any other node being the true sender. So the attacker is left with a good deal of uncertainty as to the sender's identity. Reiter and Rubin have showed that, depending on p_f, on the fraction of corrupted nodes in the network and on a few other conditions, Crowds offers very good guarantees of anonymity (see [20]).

Chatzikokolakis et al. have recently analyzed Crowds from the point of view of information hiding systems and one-try attacks [7,8]. In their modelling, $\mathcal{S} = \{s_1, ..., s_m\}$ is the set of possible senders (honest nodes), while $\mathcal{O} = \{d_1, ..., d_m\}$ is the set of observables. Here each d_i has the meaning that node s_i has been detected by some corrupted node. The conditional probability matrix is given by

$$p(d_j|s_i) \overset{\triangle}{=} \Pr(s_j\text{is detected}|\ s_i\text{is the true sender and some honest node has been detected})$$

(see [20] for details of the actual computation of these quantities). An example of such a system with $m = 20$ users, borrowed from [8], is given in the table below.

	d_1	d_2	$\cdots$	d_{20}
s_1	0.468	0.028	$\cdots$	0.028
s_2	0.028	0.468	$\cdots$	0.028
$\vdots$			$\vdots$	
s_{20}	0.028	0.028	$\cdots$	0.468

The interesting case for us is that of re-execution, in which the protocol is executed several times, either forced by the attacker himself (e.g. by having corrupted nodes suppress messages) or by some external factor, and the sender is kept fixed through the various executions. This implies the attacker collects a sequence of observations $o^n = (o_1, ..., o_n) \in \mathcal{O}^n$, for some n. The repeated executions are assumed to be independent, hence we are precisely in the setting considered in this paper. This case is also considered in [8], which gives lower bounds for the error probability holding for any n. Our results in Section 4 generalize those in [8] by providing both lower- and upper- bounds converging exponentially fast to the asymptotic error probability. As an example, for the system in the table above, we have $P_e(n) \to 0$, independently of the prior distribution on the senders. An achievable convergence rate, estimated with the method of Proposition 1, is $\epsilon \approx 0.13965$. This implies that already after observing $n = 1000$ re-executions the probability of error is, using the refined bound given in Remark 1, < 0.01.

It is worth to stress that protocol re-execution is normally prevented in Crowds for the very reason that it decreases anonymity, although it may be necessary in some cases. See the discussion on static vs. dynamic paths in [20].

6 Sequential Observations and Hidden Markov Models

We consider in this section an attack scenario where to each state of the computation there corresponds one observation on the part of the attacker. Hence, to each computation of the system there corresponds a sequential *trace* of observations. Discrete-time Hidden Markov Models [19] provide a convenient setting to formally model such systems, which we may designate as *sequential* information hiding system.

Definitions. Let $\mathcal{S}$ and $\mathcal{O}$ be finite sets of states and observations, respectively. A (discrete-time, homogeneous) *Hidden Markov Model (*HMM*)* with states in $\mathcal{S}$ and observations in $\mathcal{O}$ is a a pair of random processes $\langle\, (S_i)_{i\geq 1}\,,\, (O_i)_{i\geq 1}\,\rangle$, such that, for each $t \geq 1$

- S_t and O_t are random variables taking values in $\mathcal{S}$ and $\mathcal{O}$, respectively; and,
- the following equalities hold true (whenever the involved conditional probabilities are defined):

$$\Pr(S_{t+1} = s_{t+1}|S_t = s_t, O_t = o_t, ..., S_1 = s_1, O_1 = o_1) = \Pr(S_{t+1} = s_{t+1}|S_t = s_t) \quad (2)$$

$$\Pr(O_t = o_t|S_t = s_t, S_{t-1} = s_{t-1}, O_{t-1} = o_{t-1}, ..., S_1 = s_1, O_1 = o_1) = \Pr(O_t = o_t|S_t = s_t) \quad (3)$$

 Moreover, the value of the above probabilities does not depend on the index t, but only on s_t, s_{t+1} and o_t.

Equation (2) says that the state at time $t + 1$ only depends on the state at time t, that is $(S_i)_{i\geq 1}$ forms a Markov chain. Equation (3) says that the observation at time t only depends on the state at time t. A consequence of this equation is that the state at time $t + 1$ is independent from the observation at time t, given the state at time t, that is

$$\Pr(O_t = o_t, S_{t+1} = s_{t+1}|S_t = s_t) \;=\; \Pr(O_t = o_t|S_t = s_t) \cdot \Pr(S_{t+1} = s_{t+1}|S_t = s_t)\,. \quad (4)$$

Assume now $\mathcal{S} = \{s_1, ..., s_m\}$ and $\mathcal{O} = \{o_1, ..., o_l\}$. A finite-state HMM on $\mathcal{S}$ and $\mathcal{O}$ is completely specified by, hence can be identified with, a triple (π, F, G) such that:

- $\pi \in \mathbb{R}^{1\times m}$ is a row-vector representing the prior distribution on $\mathcal{S}$, that is $\pi(i) = p(S_1 = s_i)$ for each $1 \leq i \leq m$;
- $F \in \mathbb{R}^{m\times m}$ is a matrix such that $F(i, j)$ is the probability of transition from s_i to s_j, for $1 \leq i, j \leq m$;
- $G \in \mathbb{R}^{m\times l}$ is a matrix such that $G(i, j)$ is the probability of observing o_j at state s_i, for $1 \leq i \leq m$ and $1 \leq j \leq l$.

In our scenario, a Bayesian attacker targets the first state of the computation, that is the value of S_1. We are interested in analyzing the attacker's probability of error after observing n traces of length t, corresponding to n conditionally independent executions of the system up to and including time t, as both n and t go to $+\infty$. This we define in the following. Let σ range over the set of observation traces, that is $\mathcal{O}^*$. For any $\sigma = o_1 \cdots o_t$ $(t \geq 0)$ and $s \in S$, define

$$p(\sigma\,|\,s) \overset{\triangle}{=} \Pr(O_1 = o_1, O_2 = o_2, ..., O_t = o_t\,|\,S_1 = s)$$

with the proviso that $p(\epsilon \mid s) \triangleq 1$. We note that for any fixed $t \geq 0$ and $s \in \mathcal{S}$, $p(\sigma|s)$ defines a probability distribution as σ ranges over $\mathcal{O}^t$, the set of traces of length t, or *t-traces*. In other words, for *any fixed t*, we have an information hiding system in the sense of Section 3, with $\mathcal{S}$ as a set of states, $\mathcal{O}^t$ as a set of observables, a conditional probability matrix $p(\sigma|s)$ ($s \in \mathcal{S}, \sigma \in \mathcal{O}^t$) and π as a prior distribution on states. Call $\mathcal{H}^{(t)}$ this system. We have the following error probabilities of interest ($t \geq 0$):

$$P_e^{(t)}(n) \triangleq \text{ probability of error after } n \text{ observations (of } t\text{-traces) in } \mathcal{H}^{(t)} \tag{5}$$

$$P_e^{(t)} \triangleq \lim_{n\to\infty} P_e^{(t)}(n) \tag{6}$$

$$P_e \triangleq \lim_{t\to\infty} P_e^{(t)} . \tag{7}$$

We will show in the next paragraph that the above two limits exist and are easy to compute. Correspondingly, we have the information leakage quantities of interest (here $P_{succ} = 1 - P_e$): $L_+^{(t)}(n) \triangleq P_{succ}^{(t)}(n) - \max_s \pi(s)$ $\quad L_+^{(t)} \triangleq P_{succ}^{(t)} - \max_s \pi(s)$ $\quad L_+ \triangleq P_{succ} - \max_s \pi(s)$. Multiplicative leakages are defined similarly.

Results. That the limit (6) exists is an immediate consequence of Theorem 1 applied to $\mathcal{H}^{(t)}$. Indeed, let us denote by $\equiv^{(t)}$ the indistinguishability relation on states for $\mathcal{H}^{(t)}$, that is, explicitly $s \equiv^{(t)} s'$ iff for each $\sigma \in \mathcal{O}^t : p(\sigma|s) = p(\sigma|s')$.. Let $C_1^{(t)}, ..., C_{K_t}^{(t)}$ be the equivalence classes of $\equiv^{(t)}$ and let $p_i^{*(t)} \triangleq \max_{s \in C_i^{(t)}} \pi(s)$. Then we have by Theorem 1 that $P_e^{(t)} = 1 - \sum_{i=1}^{K_t} p_i^{*(t)}$. Note that, for any fixed t, Corollary 2 carries over to $\mathcal{H}^{(t)}$. We now consider the case $t \to \infty$. We introduce the following fundamental relation.

Definition 5 (Indistinguishability for HMM). *The* indistinguishability relation *on a* HMM *is defined as* $\equiv \triangleq \bigcap_{t\geq 0} \equiv^{(t)}$. *Equivalently,* $s \equiv s'$ *iff for every* $\sigma \in \mathcal{O}^*$, $p(\sigma|s) = p(\sigma|s')$.

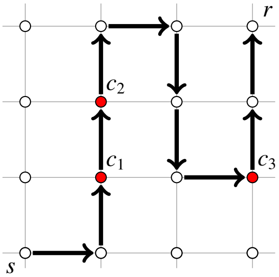
It is immediate to check that $\equiv$ is an equivalence relation. Let $C_1, ..., C_K$ be its equivalence classes and let $p_i^* \triangleq \max_{s \in C_i} \pi(s)$, for $i = 1, ..., K$.

Proposition 3. *The limit (7) is given by* $P_e = 1 - \sum_{i=1}^{K} p_i^*$.

The actual computation of P_e, and of the corresponding information leakage quantities, is therefore reduced to the computation of $\equiv$. Below, we show that this computation can indeed be performed quite efficiently. We do so by using some elementary linear algebra. Let us introduce some additional notation. We define the transition matrices $M_{o_k} \in \mathbb{R}^{m\times m}$, for any $o_k \in \mathcal{O}$, as follows: $M_{o_k}(i, j) \triangleq \Pr(S_{t+1} = s_j, O_t = o_k | S_t = s_i) = F(i, j) \cdot G(i, k)$, where the last equality is justified by equation (4). For any $\sigma = o_1 \cdots o_t$, we let M_σ denote $M_{o_1} \times \cdots \times M_{o_t}$. Finally, we let $e_i \in \mathbb{R}^{1\times m}$ denote the row vector with 1 in the i-th position and 0 elsewhere and let $e \triangleq \sum_{i=1}^{m} e_i$ denote the everywhere 1 vector. We say a row vector v is *orthogonal* to a set of column vectors U, written $v \perp U$, if $vu = 0$ for each $u \in U$.

Theorem 2. *Let B be a basis of the (finite-dimensional) sub-space of* $\mathbb{R}^{m\times 1}$ *spanned by* $\bigcup_{\sigma \in \mathcal{O}^*} \{M_\sigma e^T\}$. *For* $s_i, s_j \in \mathcal{S}$, $s_i \equiv s_j$ *iff* $(e_i - e_j) \perp B$.

A basis B of span$(\bigcup_\sigma \{M_\sigma e^T\})$ can be expressed as $B = \{M_\sigma e^T \mid \sigma \in \mathcal{F}\}$ for a suitable finite, prefix-closed $\mathcal{F} \subseteq \mathcal{O}^*$. More precisely, B can be computed by a procedure that starts with the set $B := \{e^T\}$ and iteratively updates B by joining in the vectors $M_{o\cdot\sigma}e^T = M_o \cdot (M_\sigma e^T)$, with $M_\sigma e^T \in B$ and $o \in \mathcal{O}$, that are linearly independent from the vectors already present in B, until no other vector can be joined in. This procedure must terminate in a number of steps $\leq m$. The set of strings $\mathcal{F}$ can be computed alongside with B. In the full version of the paper we also discuss a method to compute the rate of convergence to P_e.



$\sigma = * * c_1 c_2 * * * * c_3 * *$

An example: hiding routing information. We outline a simple anonymity protocol in the vein of onion routing [11]. The protocol aims at protecting the identity of the sender *and* of the receiver of a transaction in a network where some of the nodes are compromised by local eavesdroppers. The routing paths are established randomly. The local eavesdroppers have limited observation capabilities and, perhaps because of encryption, can only tell whether, at any discrete time step, the compromised node is holding a message in the target transaction, or not. Assume the topology of the network is specified by a nonempty graph $\mathcal{G} = (V, E)$. For each node $v \in V$, we let $N(v)$ denote the set of neighbours of v, that is the set of nodes u for which an arc $\{v, u\}$ in E exists; $N(v)$ is always assumed nonempty. Let $C \subseteq V$ represent a subset of corrupted nodes. We let $\mathcal{S} \triangleq V \times V$ be the set of states of the system and $\mathcal{O} \triangleq C \cup \{*\}$ be the set of observables. State $(s, r) \in \mathcal{S}$ means the message is hold by s and that r is the final receiver. Observation $c \in C$ means that the message is presently hold by the node c, while $*$ means no observation other than the elapse of a discrete time unit. What the attacker can observe are therefore traces σ like in the picture above. The exact definition of the transition and observation matrices F and G of the HMM, as well as the outcome of several experiments conducted with this simple model, are reported in the full version of the paper [4].

7 Conclusion and Further Work

We have characterized the asymptotic behaviour of error probability, and information leakage in terms of indistinguishability in a scenario of one-try attacks after repeated independent, noisy observations. We have first examined the case in which each execution gives rise to a single observation, then extended our results to the case where each state traversed during an execution induces one observation.

In the future, we would like to systematically characterize achievable rates of convergence given an error probability threshold, thus generalizing Proposition 1. It would also be natural to generalize the present one-try scenario to the case of k-tries attack, for $k \geq 2$. Experiments and simulations with realistic anonymity protocols may be useful to asses at a practical level the theoretical results of our study. For example, we believe that HMM's are relevant to security in peer-to-peer overlays. We would also like to clarify the relationship of our model with the notion of probabilistic *opacity* [2], and with the huge amount of work existing on *covert channels* (see e.g. [17] and references therein).

References

1. Backes, M., Köpf, B.: Formally Bounding the Side-Channel Leakage in Unknown-Message Attacks. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 517–532. Springer, Heidelberg (2008)
2. Bérard, B., Mullins, J., Sassolas, M.: Quantifying Opacity. In: Proc. of QEST 2010, pp. 263–272. IEEE Society, Los Alamitos (2010)
3. Boreale, M.: Quantifying information leakage in process calculi. Information and Computation 207(6), 699–725 (2009)
4. Boreale, M., Pampaloni, F., Paolini, M.: Asymptotic information leakage under one-try attacks, Full version of the present paper http://rap.dsi.unifi.it/~boreale/Asympt.pdf
5. Braun, C., Chatzikokolakis, K., Palamidessi, C.: Compositional Methods for Information-Hiding. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 443–457. Springer, Heidelberg (2008)
6. Braun, C., Chatzikokolakis, K., Palamidessi, C.: Quantitative Notions of Leakage for One-try Attacks. In: Proc. of MFPS 2009. Electr. Notes Theor. Comput. Sci, vol. 249, pp. 75–91 (2009)
7. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: Anonymity protocols as noisy channels. Information and Computation 206(2-4), 378–401 (2008)
8. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: On the Bayes risk in information-hiding protocols. Journal of Computer Security 16(5), 531–571 (2008)
9. Clark, D., Hunt, S., Malacaria, P.: Quantitative Analysis of the Leakage of Confidential Data. Electr. Notes Theor. Comput. Sci. 59(3) (2001)
10. Cover, T.M., Thomas, J.A.: Elements of Information Theory, 2/e. John Wiley & Sons, Chichester (2006)
11. Goldschlag, D.M., Reed, M.G., Syverson, P.F.: Anonymous Connections and Onion Routing. IEEE Journal on Selected Areas in Communication, Special Issue on Copyright and Privacy Protection (1998)
12. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
13. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
14. Köpf, B., Basin, D.A.: An information-theoretic model for adaptive side-channel attacks. In: ACM Conference on Computer and Communications Security 2007, pp. 286–296 (2007)
15. Köpf, B., Dürmuth, M.: A Provably Secure and Efficient Countermeasure against Timing Attacks. In: CSF 2009, pp. 324–335 (2009)
16. Köpf, B., Smith, G.: Vulnerability Bounds and Leakage Resilience of Blinded Cryptography under Timing Attacks. In: CSF 2010, pp. 44–56 (2010)
17. Mantel, H., Sudbrock, H.: Information-Theoretic Modeling and Analysis of Interrupt-Related Covert Channels. In: Degano, P., Guttman, J., Martinelli, F. (eds.) FAST 2008. LNCS, vol. 5491, pp. 67–81. Springer, Heidelberg (2009)
18. Massey, J.L.: Guessing and Entropy. In: Proc. 1994 IEEE Symposium on Information Theory (ISIT 1994), vol. 204 (1994)
19. Rabiner, L.R.: A tutorial on Hidden Markov Models and selected applications in speech recognition. Proc. of the IEEE 77(2), 257–286 (1989)
20. Reiter, M.K., Rubin, A.D.: Crowds: Anonymity for Web Transactions. ACM Trans. Inf. Syst. Secur. 1(1), 66–92 (1998)
21. Smith, G.: On the Foundations of Quantitative Information Flow. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 288–302. Springer, Heidelberg (2009)
22. Standaert, F.-X., Malkin, T.G., Yung, M.: A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 443–461. Springer, Heidelberg (2009)

A Trace-Based View on Operating Guidelines

Christian Stahl[1] and Walter Vogler[2]

[1] Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, The Netherlands
c.stahl@tue.nl

[2] Institut für Informatik, Universität Augsburg, Germany
vogler@informatik.uni-augsburg.de

Abstract. *Operating guidelines* have been introduced to characterize all controllers for a given service S. A controller of S is a service that interacts with S without deadlocking. An operating guideline of S can be used to decide whether S *refines* another service. It is a special-purpose structure to describe the behavior of service S from the perspective of its controllers rather than from the perspective of S.

This paper provides a more conceptual understanding of operating guidelines from the perspective of a traditional concurrency semantics: a *trace-based semantics*. As benefits, we get an easier characterization of service refinement, and prove that this is a fully abstract precongruence.

1 Introduction

Service-oriented computing (SOC) [14] is a computing paradigm that has found growing acceptance in industry. A service-oriented system is a distributed system that is aggregated from building blocks, called *services*. A service encapsulates a certain functionality and offers it to other services over a well-defined interface. The distributed nature of service-oriented systems requires services to interact with each other via *asynchronous* message passing.

Service orientation supports the shift from monolithic systems to systems in which multiple organizations are involved, and it allows faster changes than in monolithic systems. The latter requires a notion of *service replaceability*. Replacing one service with another one has to preserve correctness of the overall system (*compositionality*). Because organizations usually do not know the services involved in the system, replacement must be independent of a service context; that is, for every context a service correctly interacts with, the replacement must correctly interact with as well. We refer to such a context as a *controller*.

A minimal requirement for the correctness of a system is the absence of deadlocks — that is, of unsuccessful termination. The replacement (or refinement) relation in the context of deadlock freedom has been formalized by the *accordance preorder* in [15]. The current decision procedure uses that, for finite-state services with bounded buffers, the set of controllers has a finite representation, the *operating guideline* [8] of the service; technically, it is an automaton annotated with Boolean formulae. Deciding accordance of two services is reduced to

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 411–425, 2011.

checking that one of the corresponding operating guidelines simulates the other and that the corresponding formulae of related states imply each other.

This characterization of accordance differs from existing refinement relations, because the special-purpose operating guideline describes the service behavior from the perspective of a controller and has an unusual nature. This makes operating guidelines more difficult to understand. In addition, it raises the question how the semantics of an operating guideline and hence the accordance preorder are related to existing refinement notions in literature. Another open question is whether accordance is a precongruence w.r.t. composition.

This paper answers these questions. First, we show that the behavior of a service can be characterized by four trace sets — somewhat similar to standard or completed traces, and to catastrophic traces in the spirit of divergence traces [2]. This semantics naturally gives rise to a refinement relation, which we prove to coincide with (a slight modification of) accordance. Like in [8,15], we use Petri nets to model finite state services; however, as a further improvement, we consider a more general notion of finite state service than in [8,15,9].

Second, our modified accordance relation is more uniform than the original accordance relation. In particular, the semantics of a composition can be determined from the semantics of the components, which proves that this relation is a precongruence w.r.t. composition. The latter can also be phrased as a full abstractness result. We further sketch how accordance can be decided in practice.

Third, we show explicitly how (automaton representations of) the new semantics can be translated back and forth into operating guidelines. This allows us to take a different view on operating guidelines.

Section 2 gives some background, Sect. 3 introduces an accordance relation for infinite state services and characterizes its semantics using traces. In Sect. 4, we present our results concerning accordance for finite state services. Section 5 compares operating guidelines and our trace-based semantics. We close with a discussion of related work and a conclusion.

2 Preliminaries

As a basic model, we use place/transition Petri nets extended with final markings and transition labels. For sets A and B, $A \uplus B$ denotes the disjoint union; writing $A \uplus B$ expresses the implicit assumption that A and B are disjoint.

Definition 1 (net). A *net* $N = (P, T, F, m_N, \Omega)$ consists of a finite set P of *places*, a finite set T of *transitions* such that P and T are disjoint, a *flow relation* $F \subseteq (P \times T) \cup (T \times P)$, an *initial marking* m_N, where a marking is a mapping $m : P \to \mathbb{N}$, and a set Ω of *final markings.*

A *labeled net* is a net N together with an *alphabet* Σ of actions and a *labeling function* $l : T \to \Sigma \uplus \{\tau\}$, where τ represents an invisible, internal action.

Introducing N implicitly introduces its components P, T, F, m_N, Ω; the same applies to N', N_1, etc. and their components $P', T', \ldots$, and $P_1, T_1, \ldots$, resp. — and it also applies to other structures later on. We use the standard graphical representation, writing transition labels $\neq \tau$ into the respective boxes. The *preset* of $x \in P \cup T$ is ${}^\bullet x = \{y \mid (y, x) \in F\}$ and the *postset* is $x^\bullet = \{y \mid (x, y) \in F\}$. We interpret pre- and postsets as multisets when used in operations with multisets.

For a word $w \in \Sigma_1^*$ and $\Sigma_2 \subseteq \Sigma_1$, $w|_{\Sigma_2}$ denotes the projection of w to the subalphabet Σ_2. With $v \sqsubseteq w$ we denote that v is a *prefix* of word w, and $cont(L) = \{w \in \Sigma^* \mid \exists v \in L : v \sqsubseteq w\}$ is the set of all *continuations* of language $L \subseteq \Sigma^*$.

A marking is a multiset over P; for example, $[p_1, 2p_2]$ denotes a marking m with $m(p_1) = 1, m(p_2) = 2$, and $m(p) = 0$ for $p \in P \backslash \{p_1, p_2\}$. We define $+$ and $-$ for the sum and the difference of two markings and $=, <, >, \leq, \geq$ for comparison of markings in the standard way. A marking is changed by *firing* a transition. A transition t is *enabled* at a marking m, denoted by $m \xrightarrow{t}$, if $m \geq {}^\bullet t$. Then, t can *fire*, reaching a marking m' (denoted by $m \xrightarrow{t} m'$), where $m' = m - {}^\bullet t + t^\bullet$.

The behavior of N can be extended to sequences: $m_1 \xrightarrow{t_1} m_2 \xrightarrow{t_2} \ldots m_k$ is a *run* of N if $m_i \xrightarrow{t_i} m_{i+1}$, for all $0 < i < k$. A marking m' is *reachable from* a marking m if there exists a (possibly empty) run $m_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{k-1}} m_k$ with $m = m_1$ and $m' = m_k$; for $w = t_1 \ldots t_{k_1}$ we also write $m_1 \xrightarrow{w} m_k$. Marking m' is *reachable* if $m_N = m$. In the case of labeled nets, we lift runs to traces: If $m_1 \xrightarrow{w} m_k$ and v is obtained from w by replacing each transition by its label and removing all τ labels, we write $m_1 \overset{v}{\Longrightarrow} m_k$ and call v a *trace* whenever $m_1 = m_N$. A marking m is *stable* if it does not enable a τ-labeled transition. The *reachability graph* $RG(N)$ *of* N has the reachable markings as its nodes and a t-labeled edge from m to m' whenever $m \xrightarrow{t} m'$. In the case of a labeled net, each edge label t is replaced by $l(t)$.

Finally, we introduce two properties of nets. A net N is *b-bounded* for $b \in \mathbb{N}$ if $m(p) \leq b$ for every reachable marking m and $p \in P$; N is bounded if it is b-bounded for some $b \in \mathbb{N}$. A reachable marking $m \notin \Omega$ of N is a *deadlock* if no transition $t \in T$ of N is enabled at m.

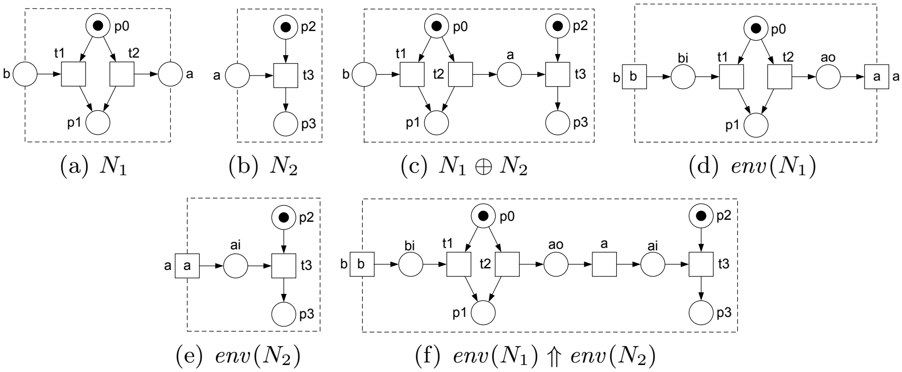
2.1 Open Nets

Like in [8,15], we model services as *open nets* [16,8], thereby restricting ourselves (like in [8,15]) to the communication protocol of a service. An open net extends a net by an interface. An interface consists of two disjoint sets of input and output places corresponding to asynchronous input and output channels. In the initial marking and the final markings, interface places are not marked. An input place has an empty preset, whereas an output place has an empty postset.

Definition 2 (open net). An *open net* N is a tuple $(P, T, F, m_N, I, O, \Omega)$ with

- $(P \uplus I \uplus O, T, F, m_N, \Omega)$ is a net;
- for all $p \in I \uplus O$: $m_N(p) = 0$ and for all $m \in \Omega$: $m(p) = 0$;
- the set I of *input places* satisfies ${}^\bullet p = \emptyset$ for all $p \in I$; and
- the set O of *output places* satisfies $p^\bullet = \emptyset$ for all $p \in O$.

Two open nets are *interface equivalent* if they have the same sets of input and output places. If $I = O = \emptyset$, then N is a *closed net*. The net that results from removing the interface places and their adjacent arcs from N is *inner*(N).



(a) N_1 (b) N_2 (c) $N_1 \oplus N_2$ (d) $env(N_1)$

(e) $env(N_2)$ (f) $env(N_1) \Uparrow env(N_2)$

Fig. 1. Two open nets, their environments, and their composition

Graphically, we represent an open net like a net with a dashed frame around it. The interface places are depicted on the frame.

For the composition of open nets, we assume that the sets of transitions are pairwise disjoint and that no internal place of an open net is a place of any other open net. The interfaces intentionally overlap. We require that all communication is *bilateral* and *directed*; that is, every shared place p has only one open net that sends into p and one open net that receives from p. We refer to open nets that fulfill these properties as *composable*. We compose two composable open nets N_1 and N_2 by merging those interface places they have in common and turn these places into internal places.

Definition 3 (open net composition). Open nets N_1 and N_2 are *composable* if $(P_1 \uplus T_1 \uplus I_1 \uplus O_1) \cap (P_2 \uplus T_2 \uplus I_2 \uplus O_2) = (I_1 \cap O_2) \uplus (I_2 \cap O_1)$. The *composition* of two composable open nets N_1 and N_2 is the open net $N_1 \oplus N_2 = (P, T, F, m_N, I, O, \Omega)$, where

- $P = P_1 \uplus P_2 \uplus (I_1 \cap O_2) \uplus (I_2 \cap O_1)$; $T = T_1 \uplus T_2$; $F = F_1 \uplus F_2$;
- $m_N = m_{N_1} + m_{N_2}$, extending m_{N_i} with $m_{N_i}(p) = 0 \; \forall p \in P_{3-i}$, $i = 1, 2$;
- $I = (I_1 \uplus I_2) \setminus (O_1 \uplus O_2)$; $O = (O_1 \uplus O_2) \setminus (I_1 \uplus I_2)$; and
- $\Omega = \{m_1 + m_2 \mid m_1 \in \Omega_1 \wedge m_2 \in \Omega_2\}$.

Example 1. Figure 1 shows two composable open nets N_1 and N_2 and their composition $N_1 \oplus N_2$. The composition is still an open net.

2.2 Environments

To give an open net N a trace-based semantics, we consider its environment $env(N)$ similarly as in [16]. The net $env(N)$ can be constructed from N by adding to each interface place $p \in I$ ($p \in O$) a p-labeled transition p in $env(N)$ and renaming place p by p^i (p^o). This way, the asynchronous interface of N is translated into a buffered synchronous interface described by the transition labels of $env(N)$.

Definition 4 (open net environment). The *environment of an open net N* is the labeled net $env(N) = (P \uplus P^I \uplus P^O, T \uplus I \uplus O, F', m_N, \Omega, I \uplus O, l')$, where

- $P^I = \{p^i \mid p \in I\}; \qquad P^O = \{p^o \mid p \in O\};$
- $F' = \quad ((P \cup T) \times (T \cup P)) \cap F$
  $\cup \{(p^i, t) \mid p \in I, t \in T, (p,t) \in F\} \cup \{(t, p^o) \mid p \in O, t \in T, (t,p) \in F\}$
  $\cup \{(p^o, p) \mid p \in O\} \cup \{(p, p^i) \mid p \in I\}; and$
- $l'(t) = \begin{cases} \tau, & t \in T \\ t, & t \in I \cup O. \end{cases}$

The language of N is defined by $L(N) = \{w \in (I \uplus O)^* \mid m_{env(N)} \xRightarrow{w}\}$.

To compose environments of composable open nets, we define a parallel composition $\Uparrow$, where, for each action a that the components have in common, each a-labeled transition of one component is synchronized with each a-labeled transition of the other; afterward a is hidden. Because a itself is the only a-labeled transition, the definition can be simplified compared to a more general setting, for instance in [16].

Definition 5 (parallel composition). Let $env(N_1) = (P_1, T_1, F_1, m_{N_1}, \Omega_1, \Sigma_1, l_1)$ and $env(N_2) = (P_2, T_2, F_2, m_{N_2}, \Omega_2, \Sigma_2, l_2)$ be the environments of composable open nets N_1 and N_2. The *parallel composition* of $env(N_1)$ and $env(N_2)$ is defined by the labeled net $env(N_1) \Uparrow env(N_2) = (P, T, F, m_N, \Omega, \Sigma, l)$, where

- $P = P_1 \uplus P_2; \qquad T = T_1 \cup T_2;$
- $F = F_1 \uplus F_2; \qquad m_N = m_{N_1} + m_{N_2};$
- $\Omega = \{m_1 + m_2 \mid m_1 \in \Omega_1 \wedge m_2 \in \Omega_2\}; \qquad \Sigma = (\Sigma_1 \cup \Sigma_2) \setminus (\Sigma_1 \cap \Sigma_2)$; and
- $l(t) = \begin{cases} l_1(t), & t \in (T_1 \setminus T_2) \\ l_2(t), & t \in (T_2 \setminus T_1) \\ \tau, & \text{otherwise.} \end{cases}$

Example 2. Figure 1 depicts environment $env(N_1)$ and $env(N_2)$ of open nets N_1 and N_2 and their parallel composition $env(N_1) \Uparrow env(N_2)$. $L(N_1) = b^* + b^*ab^*$ and $L(N_2) = a^*$.

To describe the behavior of compositions, we define parallel compositions of words and languages; operator $\|$ synchronizes common actions, operator $\Uparrow$ also hides them. Observe that, in $env(N_1) \Uparrow env(N_2)$, just common transitions are merged; operator $\|$ is needed to relate the respective transition sequences.

Definition 6. Given alphabets Σ_1, Σ_2, $\Sigma = (\Sigma_1 \cup \Sigma_2) \setminus (\Sigma_1 \cap \Sigma_2)$, words $w_1 \in \Sigma_1^*$ and $w_2 \in \Sigma_2^*$, and languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, we define

- $w_1 \| w_2 = \{w \in (\Sigma_1 \cup \Sigma_2)^* \mid w|_{\Sigma_1} = w_1 \wedge w|_{\Sigma_2} = w_2\};$
- $w_1 \Uparrow w_2 = \{w|_\Sigma \mid w \in w_1 \| w_2\};$
- $L_1 \| L_2 = \bigcup\{w_1 \| w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$; and
- $L_1 \Uparrow L_2 = \bigcup\{w_1 \Uparrow w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$.

3 Trace Semantics for Deadlock Freedom

This section considers the intuitive idea that the communication of a user with a service is successful if it only stops in case of successful termination; that is, the composition is in a final state. Such a user can also be seen as a *controller* of the service. When refining an open net *Spec*, understood as a service specification, to an implementing open net *Impl*, we require that every satisfied user of *Spec* should also be satisfied with *Impl*. Our aim is to characterize this refinement relation without considering users or controllers explicitly, and to show that the relation is a precongruence for $\oplus$; that is, it supports compositional reasoning.

Definition 7 (controller, accordance). An open net C is a *controller* of an open net N if the composition $N \oplus C$ is a closed net without deadlocks.

Let *Spec* and *Impl* be interface equivalent open nets. Open net *Impl accords with* open net *Spec*, denoted by $\mathit{Impl} \sqsubseteq_{acc} \mathit{Spec}$, if every controller C of *Spec* is also a controller of *Impl*.

Accordance has been defined in [15] for finite state services; that is, the composition of N and a controller must be a bounded net. The unbounded case is interesting in its own right, and it is simpler in theory and prepares the approach for the bounded case.

3.1 Stop-Dead Semantics

Our new semantics of an open net N considers two sets of traces for its environment $\mathit{env}(N)$. A *stop-trace* records a run of $\mathit{env}(N)$ that ends in a marking m enabling transitions of I only; it is a *dead-trace* if in addition m is not a final marking.

Definition 8 (stop-dead (sd) semantics). Let N be an open net. Marking m of $\mathit{env}(N)$ is a *stop except for inputs* if $\forall t : m \xrightarrow{t}$ implies $t \in I$. Define

- $\mathit{stop}(N) = \{w \mid m_{\mathit{env}(N)} \overset{w}{\Longrightarrow} m$ and m is a stop except for inputs$\}$ and
- $\mathit{dead}(N) = \{w \mid m_{\mathit{env}(N)} \overset{w}{\Longrightarrow} m \notin \Omega$ and m is a stop except for inputs$\}$.

With this semantics, we can characterize when a closed net has a deadlock.

Proposition 9. *A closed net N has a deadlock iff $\mathit{dead}(N) = \{\epsilon\}$.*

We study now how the sd-semantics of two open nets can be combined to obtain the sd-semantics of their composition. For the rest of this section, we fix two composable open nets $N_1 = (P_1, T_1, F_1, m_{N_1}, I_1, O_1, \Omega_1)$ and $N_2 = (P_2, T_2, F_2, m_{N_2}, I_2, O_2, \Omega_2)$, and we define $E = \mathit{env}(N_1) \Uparrow \mathit{env}(N_2)$. Note that $\mathit{env}(N_1 \oplus N_2)$ and E have the same places except for the $p \in (I_1 \cap O_2) \cup (I_2 \cap O_1)$ in $\mathit{env}(N_1 \oplus N_2)$ and the corresponding p^i, p^o in E.

Theorem 10 (sd-semantics for open net composition). *For composable open nets N_1 and N_2, we have*

1. $L(N_1 \oplus N_2) = L(N_1) \Uparrow L(N_2)$*;*
2. $\mathit{stop}(N_1 \oplus N_2) = \mathit{stop}(N_1) \Uparrow \mathit{stop}(N_2)$*; and*
3. $\mathit{dead}(N_1 \oplus N_2) = (\mathit{dead}(N_1) \Uparrow \mathit{stop}(N_2)) \cup (\mathit{stop}(N_1) \Uparrow \mathit{dead}(N_2))$*.*

3.2 Coincidence with Accordance

We define a refinement relation between open nets based on their sd-semantics. From Theorem 10, it is preserved under $\oplus$; that is, it is a precongruence.

Definition 11 (sd-refinement). For any two interface equivalent open nets *Spec* and *Impl*, *Impl sd-refines Spec*, denoted by $\mathit{Impl} \sqsubseteq_{sd} \mathit{Spec}$, iff $\mathit{dead}(\mathit{Impl}) \subseteq \mathit{dead}(\mathit{Spec}) \wedge \mathit{stop}(\mathit{Impl}) \subseteq \mathit{stop}(\mathit{Spec})$.

Theorem 12 (precongruence). *Relation $\sqsubseteq_{sd}$ is a precongruence for $\oplus$.*

Next, we prove that the notion of accordance coincides with sd-refinement, and get a precongruence result as an easy corollary.

Theorem 13 (accordance and sd-refinement coincide). *For interface equivalent open nets Spec and Impl, we have $\mathit{Impl} \sqsubseteq_{acc} \mathit{Spec} \Leftrightarrow \mathit{Impl} \sqsubseteq_{sd} \mathit{Spec}$.*

Corollary 14. *Accordance is a precongruence for composition operator $\oplus$.*

4 Trace Semantics for Deadlock Freedom and Boundedness

In this section, we restrict ourselves to finite state services — more precisely, a composition with a controller must be b-bounded for some $b \in \mathbb{N} \setminus \{0\}$. The motivation for this restriction is that we model the control flow of service compositions and assume a (finite) capacity of the message channels. To ensure boundedness in the composition, controllers must not send a message if there are already b messages in the respective message channel, but they must also avoid bound violations on interior places. Technically, parameter b enforces that the composition has a finite number of reachable states. Pragmatically, it could either represent a reasonable buffer size in the middleware — for example, the result of a static analysis of the communication behavior of a service — or be chosen sufficiently large.

Definition 15 (b-controllability). Let $b \in \mathbb{N} \setminus \{0\}$. An open net C *b-controls* an open net N if the composition $N \oplus C$ is closed, deadlock-free, and b-bounded.

Different from [8], we also prescribe bound b for the interior of services and controllers. Instead, only boundedness with possibly varying bounds is required in [8]. That this requirement is weaker is of little relevance[1], but it can create problems, because interface places turn into interior places of a composition. More importantly, boundedness is required in [8] for the inner of services and controllers on their own — that is, with unlimited input tokens in particular. We are more general, because we allow services (and users) that have an unbounded

[1] In principle, one can determine the state space of such a bounded open net and transform it into an automaton (such automata are studied in [8], for example) and, thus, into an open net which has even a 1-bounded inner.

inner; we only require that they are b-bounded in suitable contexts. Thus, we are more optimistic in the sense of [3], because we assume that the controller will steer the service away from bound violations.

We lift accordance for open nets to b-accordance by comparing b-controllers only. In [15], 1-accordance has been defined for controllers as defined in [8].

Definition 16 (b-accordance). For interface equivalent open nets *Spec* and *Impl*, *Impl b-accords with Spec*, denoted by $Impl \sqsubseteq^b_{acc} Spec$, if $\forall C : C$ b-controls *Spec* implies C b-controls *Impl*.

In the remainder of this section, we extend the sd-semantics of open nets taking into account the bound. With the new semantics, we define a refinement relation between open nets and prove that it coincides with b-accordance.

4.1 Bounded Stop-Dead Semantics

The idea is to specify a set $bd_b(N)$ containing all traces of $env(N)$ that lead to a marking violating bound b. We refer to such a trace v as a *bound-violator*. As we want to forbid such a trace, a continuation w of v is automatically forbidden as well; thus, it is irrelevant whether w is a bound-violator or can be performed at all. Technically, this abstraction from continuations is achieved by including all continuations of bound-violators in $bd_b(N)$. For the same reason, $bd_b(N)$ is contained in all other components of our semantics. This is similar to the treatment of divergence traces in failures semantics [2] and also of $U(N)$ in [16, Def. 3.3.2] — for a setting with synchronous communication in both cases.

Definition 17 (bounded stop-dead (bsd-) semantics). Let $b \in \mathbb{N} \setminus \{0\}$. For an open net $N = (P, T, F, m_N, I, O, \Omega)$ with environment $env(N)$, we call a trace v a *bound$_b$-violator of* N if $m_{env(N)} \overset{v}{\Longrightarrow} m \wedge \exists p \in P \cup I \cup O : m(p) > b$.

- $bd_b(N) = \{w \in (I \cup O)^* \mid \exists v \sqsubseteq w : v \text{ is a bound}_b\text{-violator}\}$,
- $L_b(N) = L(N) \cup bd_b(N)$,
- $stop_b(N) = stop(N) \cup bd_b(N)$, and
- $dead_b(N) = dead(N) \cup bd_b(N)$.

Just as $stop(N_1)$ is needed to determine $dead(N_1 \oplus N_2)$, $L_b(N_1)$ will be needed to determine $bd_b(N_1 \oplus N_2)$.

We characterize the bsd-semantics of the composition of two open nets based on their bsd-semantics, enforcing continuation closure of $bd_b(N_1 \oplus N_2)$ explicitly.

Theorem 18 (bsd-semantics for open net composition). *For composable open nets N_1 and N_2 we have*

1. $bd_b(N_1 \oplus N_2) = cont((bd_b(N_1) \Uparrow L_b(N_2)) \cup (L_b(N_1) \Uparrow bd_b(N_2)))$
2. $L_b(N_1 \oplus N_2) = (L_b(N_1) \Uparrow L_b(N_2)) \cup bd_b(N_1 \oplus N_2)$
3. $stop_b(N_1 \oplus N_2) = (stop_b(N_1) \Uparrow stop_b(N_2)) \cup bd_b(N_1 \oplus N_2)$
4. $dead_b(N_1 \oplus N_2)$
$$= (stop_b(N_1) \Uparrow dead_b(N_2)) \cup (dead_b(N_1) \Uparrow stop_b(N_2)) \cup bd_b(N_1 \oplus N_2)$$

4.2 Coincidence with b-Accordance and Nonredundancy

Analogously to sd-refinement, we define bsd-refinement and conclude from Theorem 18(1) to (4) that it is preserved under the composition operator $\oplus$.

Definition 19 (bsd-refinement). For interface equivalent open nets *Spec* and *Impl* and a bound $b \in \mathbb{N} \setminus \{0\}$, *Impl bsd*$_b$*-refines Spec*, denoted by $\mathit{Impl} \sqsubseteq_{bsd,b} \mathit{Spec}$, if $L_b(\mathit{Impl}) \subseteq L_b(\mathit{Spec}) \wedge \mathit{dead}_b(\mathit{Impl}) \subseteq \mathit{dead}_b(\mathit{Spec}) \wedge \mathit{stop}_b(\mathit{Impl}) \subseteq \mathit{stop}_b(\mathit{Spec}) \wedge \mathit{bd}_b(\mathit{Impl}) \subseteq \mathit{bd}_b(\mathit{Spec})$.

Theorem 20 (precongruence). *For each $b \in \mathbb{N} \setminus \{0\}$, bsd$_b$-refinement is a precongruence for composition operator $\oplus$.*

Each refinement relation b-accordance (see Definition 16) coincides with bsd$_b$-refinement, which implies a novel precongruence result as a corollary. The main point for the latter result is that we have a more uniform approach compared to [8], where in Definition 15 the interface places of N and C play a special role in $N \oplus C$ although they are ordinary places of this closed net. In fact, b-accordance, as introduced in [15], is no precongruence for our composition operator $\oplus$. Corollary 22 could also be proved directly by a standard argument, as presented in a very general setting in [13].

Theorem 21 ($\sqsubseteq_{acc,b}$ and $\sqsubseteq_{bsd,b}$ coincide). *For interface equivalent open nets Spec and Impl and a bound $b \in \mathbb{N} \setminus \{0\}$ holds $\mathit{Impl} \sqsubseteq_{acc,b} \mathit{Spec} \Leftrightarrow \mathit{Impl} \sqsubseteq_{bsd,b} \mathit{Spec}$.*

Corollary 22. *For each $b \in \mathbb{N} \setminus \{0\}$, b-accordance is a precongruence for $\oplus$.*

As the bsd-semantics consists of four overlapping components, the question arises whether each of them is really needed. In particular, one might suspect that the set L_b can be obtained from the other three sets by taking prefixes. To answer this question, one can construct examples showing that none of the components can be deduced from the others.

4.3 Full Abstractness

We can present the results of Sect. 4.2 also as a *full abstractness* result [11]; that is, b-accordance is the coarsest precongruence for open nets w.r.t. a suitable basic relation $\sqsubseteq_b$ and composition operator $\oplus$. It is somewhat unusual that this basic relation is only defined for closed nets: $\mathit{Impl} \sqsubseteq_b \mathit{Spec}$ says that *Impl* must represent a 'good' communication if *Spec* does.

Definition 23 (basic relation). For closed open nets *Spec* and *Impl* and a bound $b \in \mathbb{N} \setminus \{0\}$, define the *basic relation* $\mathit{Impl} \sqsubseteq_b \mathit{Spec}$ iff whenever *Spec* cannot deadlock and does not violate bound b so does *Impl*.

Relation $\sqsubseteq_b^c$ is the *coarsest precongruence* for $\oplus$ such that for all closed *Impl* and *Spec*: $\mathit{Impl} \sqsubseteq_b^c \mathit{Spec}$ implies $\mathit{Impl} \sqsubseteq_b \mathit{Spec}$.

Theorem 24 ($\sqsubseteq_{acc,b}$ is fully abstract). *For interface equivalent open nets Spec and Impl and a bound $b \in \mathbb{N} \setminus \{0\}$ holds $\mathit{Impl} \sqsubseteq_{acc,b} \mathit{Spec} \Leftrightarrow \mathit{Impl} \sqsubseteq_b^c \mathit{Spec}$.*

4.4 Deciding bsd-Refinement

To decide bsd-refinement between two open nets, four language inclusions must be checked. We discuss a decision procedure for bsd-refinement, also to prepare the comparison to the standard approach for deciding b-accordance (see [15]), which is based on operating guidelines [8].

For deciding the four language inclusions, we build one automaton with four types of final states. To do this, we construct the reachability graph of the labeled net $env(N)$ under consideration, but stop the construction whenever we reach a marking m that violates the bound b. As any bounded net has only finitely many reachable markings, this guarantees finiteness of our construction. Each such marking m gets a loop for each visible label from $I \cup O$. The initial state of the automaton is the initial marking $m_{env(N)}$.

This way, the automaton has $bd_b(N)$ as language if we designate each m that violates b as final state. If we additionally take those markings m as final states that are stops except for inputs and not in the set Ω of final markings, then the language is $dead_b(N)$. If we add all markings m as final states that are stops except for inputs, then the language is $stop_b(N)$. Finally, letting all states be final, the language is $L_b(N)$.

If we label the final states added in the four stages with $0, 1, 2, 3$, then, for each state, we need two additional bits for encoding the four languages. We use

- 0 to describe the language $bd_b(N)$;
- 0 and 1 to describe the language $dead_b(N)$;
- $0, 1$, and 2 to describe the language $stop_b(N)$; and
- $0, 1, 2$, and 3 to describe the language $L_b(N)$.

For checking language inclusion, one makes at least the automaton for the larger language deterministic (by constructing the powerset automaton) and constructs the (unique minimal) simulation [12] between the two automata. Practically, this powerset construction often leads to smaller automata. In our case, it suffices to apply this construction once — although we have four languages. A similar construction is used when building a lexer, for instance. Here, we refer to a version of the powerset construction that also removes τ-transitions — that is, ϵ-transitions according to automata theory.

Each set of states, as state of the powerset automaton, is a *node* and is labeled with the minimum label of its states. As a further simplification, one can identify all nodes with label 0 into one node U. This identification is integrated into the powerset construction. We refer to the resulting automaton as $BSD_b(N)$.

To decide bsd-refinement of two open nets *Spec* and *Impl*, we calculate the minimal simulation ϱ of $BSD_b(Impl)$ by $BSD_b(Spec)$ and check the labels of related states as stated next; one could also consider the minimal simulation of the modified reachability graph of *Impl*, as described previously, by $BSD_b(Spec)$.

Theorem 25 (deciding bsd-refinement). *For interface equivalent open nets Spec and Impl and a bound $b \in \mathbb{N} \setminus \{0\}$, we have Impl $\sqsubseteq_{bsd,b}$ Spec iff the minimal simulation of $BSD_b(Impl)$ by $BSD_b(Spec)$ exists and relates a node of $BSD_b(Impl)$ with label i only to nodes of $BSD_b(Spec)$ with label $j \leq i$.*

5 Comparing bsd-Semantics and Operating Guidelines

We compare the bsd-semantics of Definition 17 with the notion of an operating guideline [8] for open nets N. An operating guideline $OG_b(N)$ of a service N describes how a *user* should successfully communicate with N; technically, it characterizes the possibly infinite set of b-controllers of N in a finite manner. Because a b-controller of N provides suitable inputs for N and accepts its outputs, $OG_b(N)$ is similar to $BSD_b(N)$ but with inputs and outputs interchanged.

The operating guidelines of two open nets *Spec* and *Impl* can be used to decide that *Impl* b-accords with *Spec* [15]. Thus, $OG_b(N)$ and $BSD_b(N)$ should have the same semantic content. As their details are rather different, we clarify their relation by showing how to translate $OG_b(N)$ and $BSD_b(N)$ into each other. These translations do not make use of the states contained in the nodes, but the correctness proof for the translations does; for example, the node labels are defined on the basis of theses states.

Operating guidelines have been defined only for services N with b-bounded interface for which $inner(N)$ is bounded. As these two bounds are not necessarily the same, we restrict the translation to services N where $inner(N)$ is b-bounded.

5.1 Deriving an Operating Guideline from the bsd-Semantics

Both $OG_b(N)$ and $BSD_b(N)$ are finite automata that can be distinguished by their graph structure and annotations.

Graph structure. There are two differences in the graph structure of $OG_b(N)$ and $BSD_b(N)$. First, a b-controller may be able to accept inputs that N will never send. As a consequence, $OG_b(N)$ contains an additional node, the *empty node* (which does not contain any state). For each output $x \in O$ of N and each node without an outgoing x-labeled edge in $BSD_b(N)$, there is a respective edge leading to the empty node, and this node has loops for all possible visible actions from $I \cup O$. Because $BSD_b(N)$ represents the behavior of N rather than the behavior of its b-controllers, the empty node does not exist in $BSD_b(N)$. To ease the presentation, we ignore the empty node in the following.

Second, node U, which identifies all nodes with label 0 in $BSD_b(N)$, is not present in $OG_b(N)$. Because $OG_b(N)$ characterizes all b-controllers, possible bound violations will not happen and, hence, do not need to be stored. It is easy to omit U from $BSD_b(N)$, as it is identified by label 0. Vice versa, it is also possible to add U to $OG_b(N)$ by adding, for each input $a \in I$ of N to each node of $OG_b(N)$ without an outgoing a-labeled edge, such an edge leading to U. In addition, node U has like the empty node loops for all visible actions. Note that the empty node and node U are characterized in their respective automaton by having a loop for a respective output action, leading to behavior that certainly violates any bound b.

Annotations. Instead of the remaining node labels $1, 2, 3$ in $BSD_b(N)$, each node Q of $OG_b(N)$ is annotated with a Boolean formula $\phi(Q)$. The propositional atoms of ϕ are $I \cup O \cup \{\mathit{final}\}$. A b-controller cannot know which state q of a node Q net $\mathit{env}(N)$ might be in, but it has to avoid a deadlock and a bound violation in any case; the formula $\phi(Q)$ describes how to do this. Nonstable states have an internal transition and, thus, are not deadlocks; all internal transitions remain in the same node. As a consequence, $\phi(Q)$ is a *conjunction indexed by all stable states* $q \in Q$. Every conjunct is a disjunction of the following propositional atoms: *final* if q is a final state, $a \in I$ if $Q \xrightarrow{a}$, and $x \in O_N$ if $q \xrightarrow{x}$. Hence, the formulae are in conjunctive normal form (CNF) without negation.

To construct $\phi(Q)$ from $\mathit{BSD}_b(N)$ (up to equivalence), we need a procedure to which we refer to as *1,2-DFS*. This procedure is something like a depth-first search through the automaton starting from node Q. It considers only x-labeled edges with $x \in O_N$ and backtracks if (and only if) there is no further such edge or when encountering a 1,2-node (i. e., a node labeled 1 or 2); that is, visited states are not marked as visited in this procedure. When a 1,2-node is encountered, the path in the stack is called a *maximal path constructed during 1,2-DFS*. One can see that 1,2-DFS never encounters node U, because edges leading to U are input edges of N. Also note that 1,2-DFS cannot run into a cycle; if $Q \xrightarrow{x} Q'$, then Q' consists of all q' such that $\exists q \in Q : \; q \xrightarrow{x} q'$; in each case, the markings q and q' coincide except that $q'(x^o) = q(x^o) - 1$.

As the formula $\phi(Q)$ depends on the stable states $q \in Q$, we first deduce whether node Q contains any stable state.

Lemma 26. *A node Q of $\mathit{BSD}_b(N)$ contains a stable state iff 1,2-DFS starting from Q encounters a 1,2-node.*

So far, we considered nodes Q that contain a stable state. If node Q does not contain a stable state, then $\phi(Q)$ is the empty conjunction which is equal to true. In this case, node Q has label 3.

Lemma 27. *Q does not contain a stable state iff $\phi(Q)$ is the empty conjunction iff $\phi(Q)$ is true. If these three statements hold, then Q has label 3.*

For nodes without a stable state, we know by Lemmata 26 and 27 how to translate label and formula into each other, and we can *assume for the following* that $\phi(Q)$ has a conjunct. Q can have label 1, 2, or 3; the next two lemmata characterize the nodes with label 1 and 2, respectively.

By definition of formula $\phi(Q)$, all conjuncts have those inputs a of N as literals such that there is an a-labeled edge leaving Q. We factor out the disjunction of these inputs from $\phi(Q)$, and it remains to construct the remaining disjunct which is a CNF C. If this *remaining CNF C* has a conjunct false, then C is equivalent to false and we can reconstruct $\phi(Q)$ from the edges leaving Q. In this case, Q is labeled 1.

Lemma 28. *The remaining CNF C of some node Q has a conjunct false iff Q is labeled 1.*

We are left with considering nodes that are not labeled 1 and whose remaining CNF C is not equivalent to false. For such nodes, we determine whether there is some conjunct containing a literal *final*.

Lemma 29. *Assuming that node Q is not labeled 1, the remaining CNF C of Q has a conjunct containing final iff Q is labeled 2. Then the respective conjunct equals the literal final.*

Finally, we determine the remaining conjuncts for which it suffices to find at least the minimal ones regarding each conjunct as the set of its literals. These are sets of edge labels of the maximal paths being constructed during the 1,2-DFS.

Lemma 30. *Let C be the remaining CNF of a node Q. The sets of edge labels of the maximal paths being constructed during 1,2-DFS are conjuncts of C. Each conjunct of C that is not false or equal to final contains the set of edge labels of some maximal path constructed during 1,2-DFS.*

5.2 Deriving bsd-Semantics from Operating Guidelines

The previous results also show how to derive $BSD_b(N)$ from $OG_b(N)$; we collect the respective observations. First, we add the 0-labeled node U, as previously described, and we remove the empty node. For the other nodes, we run through our preceding considerations for constructing ϕ.

If $\phi(Q)$ is true, then there is no stable state $q \in Q$ and the label of Q is, therefore, 3 (see Lemma 27).

Otherwise, if $\phi(Q)$ is equivalent to the disjunction of those inputs a of N such that there is an a-labeled edge leaving Q, then the remaining CNF is false and the label of Q is 1 (see Lemma 28).

Otherwise, the remaining CNF C is not equal to false and we have to check whether *final* is a conjunct of C. By Lemma 29, this is the case if and only if the label of Q is 2. (To check semantically whether *final* is a conjunct without relying on the precise form of ϕ of C, we assign true to all outputs of N and false to all inputs and to *final*.) Then $\phi(Q)$ evaluates to false if and only if *final* is a conjunct. To all remaining nodes we assign label 3.

5.3 Accordance Check with Operating Guidelines

For open nets *Spec* and *Impl*, with operating guidelines $OG_b(\mathit{Spec})$ and $OG_b(\mathit{Impl})$, we have that $\mathit{Impl} \sqsubseteq_{acc,b} \mathit{Spec}$ if and only if there exists a minimal simulation ϱ of $OG_b(\mathit{Spec})$ by $OG_b(\mathit{Impl})$ and, for each pair of nodes $(Q, Q') \in \varrho$, $\phi(Q)$ implies $\phi(Q')$ is a tautology [15].

Interestingly, this check shows that the simulation relation is the other way around compared to the bsd-setting. Checking b-accordance involves repeated checks of implications between the annotations, whereas in the bsd-setting (see Theorem 25), we simply compare numbers $0, 1, 2, 3$ in constant time.

6 Conclusions

We presented a novel semantics for open nets assuming the absence of deadlocks as a minimal correctness criterion. The semantics consists of four sets of traces.

The set $bd_b(N)$ collects catastrophic traces due to bound violation; these traces modify all components just as divergence traces in failures semantics [2]. The sets $stop_b(N)$ and $dead_b(N)$ are successful and unsuccessful completed traces; because we are in an asynchronous setting, they can be continued with inputs; that is, they are quiescent as in I/O automata [10]. The fourth component, L_b, is just the language of the net N, comprising all traces.

We proved that our semantics can be translated back and forth into operating guidelines, and we derived the bsd-refinement relation from it, which coincides with (a slight modification of the) accordance relation. In addition, we proved bsd-refinement to be a fully abstract precongruence.

The bsd-semantics is related to the recently proposed notion of a reduced operating guideline [9]. The idea is to distinguish three disjoint subsets of nodes in an operating guideline such that the Boolean formulae can be derived from this information. The argumentation that leads to this representation has similarities to our arguments for translating $OG_b(N)$ into $BSD_b(N)$. However, in [9] responsiveness is considered: in addition to deadlock freedom, a b-controller in [9] also has to guarantee that in each state of the composition either a final state can be reached or a message is exchanged.

Different precongruences for asynchronously communicating processes have been studied. The introduced bsd-refinement is closely related to previous work of Vogler [16], closest to P-deadlock refinement. As a difference, in the setting of Vogler [16], the interface is not separated into input and output places, interface places may be unbounded (like in the sd-semantics), and the notion of a deadlock ignores τ-loops: a marking m is a deadlock if $m \overset{w}{\Longrightarrow}$ implies $w = \epsilon$. This leads to a much more complicated characterization.

Testing equivalence for asynchronous CCS is considered in [1], but it is an asymmetric notion focusing on the controller (i. e., the test) and not on the service; in contrast, the defining condition in Definitions 7 and 15 is symmetric in C and N. Avoidance of deadlocks is not so essential, and in fact τ-loops are not helpful or even a catastrophe; a process may receive the messages it has sent.

Stuck-free conformance [6] is a precongruence that excludes deadlocks. In contrast to bsd-refinement, it is based on a variation of failures semantics rather than traces. In [6], it has been proved that stable failures refinement [2] does not imply stuck-free conformance. In contrast, it has already been argued in a shared-variable setting in [4], that refusal sets are not needed when working with asynchronous communication.

In the area of SOC, the subcontract preorder [7] is an asymmetric notion also focusing on the tests. It is based on synchronous communication and only requires that the controller never gets stuck. In [7], it has been proved that the subcontract preorder coincides with must-testing [5] — that is, with stable failures refinement (for finitely branching processes without divergences).

Our goal was to provide a trace-based view on operating guidelines. In ongoing work, we investigate how the bsd-semantics of a service S can be used to check whether a service is a controller of S. In addition, we want to gain a better understanding which transformations bsd-refine a service, because these transformations allow us to construct replaceable services. We will further study how our semantics needs to be adjusted if we consider responsiveness [9] rather than deadlock freedom. Here, we are particularly interested in the relation of the resulting semantics and the notion of a reduced operating guideline [9].

References

1. Boreale, M., De Nicola, R., Pugliese, R.: Trace and testing equivalence on asynchronous processes. Inf. Comput. 172(2), 139–164 (2002)
2. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM 31(3), 560–599 (1984)
3. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC / SIGSOFT FSE 2001, pp. 109–120 (2001)
4. de Boer, F.S., Kok, J.N., Palamidessi, C., Rutten, J.J.M.M.: The failure of failures in a paradigm for asynchronous communication. In: Groote, J.F., Baeten, J.C.M. (eds.) CONCUR 1991. LNCS, vol. 527, pp. 111–126. Springer, Heidelberg (1991)
5. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. 34, 83–133 (1984)
6. Fournet, C., Hoare, T., Rajamani, S.K., Rehof, J.: Stuck-Free Conformance. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 242–254. Springer, Heidelberg (2004)
7. Laneve, C., Padovani, L.: The must preorder revisited. In: Caires, L., Li, L. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 212–225. Springer, Heidelberg (2007)
8. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 321–341. Springer, Heidelberg (2007)
9. Lohmann, N., Wolf, K.: Compact representations and efficient algorithms for operating guidelines. Fundam. Inform. (2010) (accepted for publication in January 2010)
10. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
11. Milner, R.: Fully abstract models of typed *lambda*-calculi. Theor. Comput. Sci. 4(1), 1–22 (1977)
12. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc., Englewood Cliffs (1989)
13. Mooij, A.J., Voorhoeve, M.: Proof techniques for adapter generation. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 207–223. Springer, Heidelberg (2009)
14. Papazoglou, M.P.: Web Services: Principles and Technology. Pearson, London (2007)
15. Stahl, C., Massuthe, P., Bretschneider, J.: Deciding substitutability of services with operating guidelines. In: Jensen, K., van der Aalst, W.M.P. (eds.) Transactions on Petri Nets and Other Models of Concurrency II. LNCS, vol. 5460, pp. 172–191. Springer, Heidelberg (2009)
16. Vogler, W.: Modular Construction and Partial Order Semantics of Petri Nets. LNCS, vol. 625. Springer, Heidelberg (1992)

HTML Validation of Context-Free Languages

Anders Møller* and Mathias Schwarz*

Aarhus University, Denmark
{amoeller,schwarz}@cs.au.dk

Abstract. We present an algorithm that generalizes HTML validation of individual documents to work on context-free sets of documents. Together with a program analysis that soundly approximates the output of Java Servlets and JSP web applications as context-free languages, we obtain a method for statically checking that such web applications never produce invalid HTML at runtime. Experiments with our prototype implementation demonstrate that the approach is useful: On 6 open source web applications consisting of a total of 104 pages, our tool finds 64 errors in less than a second per page, with 0 false positives. It produces detailed error messages that help the programmer locate the sources of the errors. After manually correcting the errors reported by the tool, the soundness of the analysis ensures that no more validity errors exist in the applications.

1 Introduction

An HTML document is *valid* if it syntactically conforms to a DTD for one of the versions of HTML. Since the HTML specifications only prescribe the meaning of valid documents, invalid HTML documents are often rendered differently, depending on which browser is used [1]. For this reason, careful HTML document authors validate their documents, for example using the validation tool provided by W3C[1]. An increasing number of HTML documents are, however, produced dynamically by programs running on web servers. It is well known that errors caught early in development are cheaper to fix. Our goal is to develop a program analysis that can check statically, that is, at the time programs are written, that they will never produce invalid HTML when running. We want this analysis to be *sound*, in the sense that whenever it claims that the given program has this property that is in fact the case, *precise* meaning that it does not overwhelm the user with spurious warnings about potential invalidity problems, and *efficient* such that it can analyze non-trivial applications with modest time and space resources. Furthermore, all warning messages being produced must be *useful* toward guiding the programmer to the source of the potential errors.

The task can be divided into two challenges: 1) Web applications typically generate HTML either by printing page fragments as strings to an output stream

* Supported by The Danish Research Council for Technology and Production, grant no. 274-07-0488.

[1] http://validator.w3.org

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 426–440, 2011.

(as in e.g. Java Servlets) or with template systems (as e.g. JSP, PHP, or ASP). In any case, the analysis front-end must extract a formal description of the set of possible outputs of the application, for example in the form of a context-free grammar. 2) The analysis back-end must analyze this formal description of the output to check that all strings that it represents are valid HTML. Several existing techniques follow this pattern, although considering XHTML instead of HTML [8,6]. In practice, however, many web applications output HTML data, not XHTML data, and the existing techniques – with the exception of the work by Nishiyama and Minamide [11], which we discuss in Section 2 – do not work for HTML.

The key differences between HTML and XHTML are that the former allows certain tags to be omitted, for example the start tags `<html>` and `<tbody>` and the end tags `</html>` and `</p>`, and that it uses tag inclusions and exclusions, for example to forbid deep nesting of `a` elements. This extra flexibility of HTML is precisely what makes it popular, compared to its XML variant XHTML. On the other hand, this flexibility means that the process of checking *well-formedness*, i.e. that a document defines a proper tree structure, cannot be separated from the process of checking *validity*, i.e. that the tree structure satisfies the requirements of the DTD.

In this paper, we present an algorithm that, given as input a context-free grammar G and an SGML DTD D (one of the DTDs that exist for the different versions of HTML[2]), checks whether every string in the language of G is valid according to D, written $\mathcal{L}(G) \subseteq \mathcal{L}(D)$. The key idea in our approach is a generalization of a core algorithm for SGML parsing [4,14] to work on context-free sets of documents rather than individual documents.

1.1 Outline of the Paper

The paper is organized as follows. We first give an overview of related approaches in Section 2. In Section 3 we then present a formal model of SGML/HTML validation that captures the essence of the features that distinguish it from XML/XHTML validation. Based on this model, in Section 4 we present our generalization for validating context-free sets of documents. We have implemented the algorithm together with an analysis front-end for Java Servlets and JSP, which constitute a widely used platform for server-based web application development. (Due to the limited space we focus on the back-end in this paper.) In Section 5, we report on experiments on a range of open source web applications. Our results show that the algorithm is fast and able to pinpoint programming errors. After manually correcting the errors based on the messages generated by the tool, the analysis is able to prove that the output will always be valid HTML when the applications are executed.

[2] The HTML 5 language currently under development will likely evoke renewed interest in HTML. Although it technically does not use SGML, its syntax closely resembles that of the earlier versions.

```
<%@ page import="java.util.*, org.example" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<html><head><meta name="description" content="Joke Collection">
  <title>Jokes</title>
  <%! List<Joke> js = Jokes.get();%>
  <body><table>
    <tr><th>Question<th>Punch line</tr>
      <% if (js.size() > 0) {
        request.setParameter("Jokes", js); %>
        <c:forEach items="${Jokes}" var="joke">
          <tr><td><c:out value="${joke.question}"/>
            <td><c:out value="${joke.punchline}"/></tr>
        </c:forEach>
      <% } else {
        out.print("<td>No more jokes</tr>");
      } %>
  </table></body>
</html>
```

Fig. 1. A JSP page that uses the JSTL tag library and embedded Java code. The example takes advantage of SGML features such as tag omission and inclusions.

1.2 Example

Figure 1 shows an example of a JSP program that outputs a dynamically generated table from a list of data using a combination of many of the JSP and SGML features that appear in typical applications. The `meta` element is not part of the content model of `head`, but it is allowed by an SGML inclusion rule. The `body` element contains a table where both the start and the end tag of the `tbody` element are omitted, and a parser needs to insert those to validate a generated document. Similarly, all `td` and `th` end tags are omitted. The contents of the table are generated by a combination of tags from JSP Standard Tag Library, embedded Java code that prints to the output stream, and ordinary JSP template code.

The static analysis that we present is able to soundly check that the output from such code is always valid according to e.g. the HTML 4.01 Transitional specification.

2 Related Work

Previous work on reasoning about programs that dynamically generate semi-structured data has focused on XML [9], not SGML, despite the fact that the SGML language HTML remains widely used. (Since XML languages are essentially the subclass of SGML languages that do not use the tag omission and exception features, our algorithm also works for XML.) Most closely related to our approach is the work by Minamide et al. [7,8,11] and Kirkegaard and Møller [6].

In [7] context-free grammars are derived from PHP programs. From such a grammar, sample documents are derived and processed by an ordinary HTML or XHTML validator. Unless the nesting depth of the elements in the generated documents is bounded, this approach is unsound as it may miss errors. Later, an alternative grammar analysis was suggested for soundly validating dynamically generated XML data [8]. That algorithm relies on the theory of balanced grammars over an alphabet of tag names, which does not easily generalize to handle the tag omission and inclusion/exclusion features that exist in HTML. The approach in [6] is comparable to [8], however considering the more fine-grained alphabet of individual Unicode characters instead of entire tag names and using XML graphs for representing sets of XML documents.

Yet another grammar analysis algorithm is presented by Nishiyama and Minamide [11]. They define a subclass of SGML DTDs that includes HTML and shows a translation into regular hedge grammars, such that the validation problem reduces to checking inclusion of a context-free language in a regular language. That approach has some limitations, however: 1) it does not support start tag omission, although that feature of SGML is used in HTML (e.g. `tbody` and `head`); 2) the exclusion feature is handled by a transformation of the DTD that may lead to an exponential blow-up prohibiting practical use; and 3) the inclusion feature is not supported. The alternative approach we suggest overcomes all these limitations.

The abstract parsing algorithm by Doh et al. [3] and the grammar-based analysis by Thiemann [12] are also based on the idea of generalizing existing parsing algorithms. The approach in [3] relies on abstract interpretation with a domain of LR(k) parse stacks constructed from an LR(k) grammar for XHTML, and [12] is based on Earley's parsing algorithm. By instead using SGML parsing as a starting point, we avoid the abstraction and we handle the special features of HTML: Given a context-free grammar describing the output of a program, our algorithm for checking that all derivable strings are valid HTML is both sound and complete.

3 Parsing HTML Documents

Although HTML is based on the SGML standard [4] it uses only a small subset of the features of the full standard. SGML languages are formally described using the DTD language (not to confuse with the DTD language for XML). Such a description provides a formal description for the parser on how a document is parsed from its textual form into a tree structure. Specifically, in SGML both start and end tags may be omitted if 1) allowed by the DTD, and 2) the omission does not result in ambiguities in the parsing of the document. The DTD description provides the content models, that is, the allowed children of each element, as deterministic regular expressions over sequences of elements. Furthermore special exceptions, called inclusions and exclusions, are possible for allowing additional element children or disallowing nesting of certain elements. An inclusion rule permits elements anywhere in the descendant tree even if not allowed by the content model expressions. Conversely, an exclusion rule prohibits elements, overriding the content model expressions and inclusions.

Consider a small example DTD:

```
<!ELEMENT inventory - - (item*) +(note)>
<!ELEMENT item - O (#PCDATA)>
<!ELEMENT note - O (#PCDATA)>
```

In each element declaration, `O` means "optional" and `-` means "required", for the start tag and the end tag, respectively. This DTD declares an element `inventory` where the start and end tags are both required. (Following the usual SGML terminology, an *element* generally consists of a start tag and its matching end tag, although certain tags may be omitted in the textual representation of the documents.) The content model of `inventory` allows a sequence of `item` elements as children in the document tree. In addition, `note` is included such that `note` elements may be descendants of `inventory` elements even though they are not allowed directly in the content models of the descendants. The second line declares an element `item` that requires a start tag but allows omission of the end tag. The content model of `item` allows only text (PCDATA) and no child elements in the document tree. Finally, the element `note` is also declared with end tag omission and PCDATA content. An example of a valid document for this DTD is the following:

```
<inventory><item>gadget<item>widget</inventory>
```

The parser inserts the omitted end tags for `item` to obtain the following document, which is valid according to the DTD content models for `inventory` and `item`:

```
<inventory><item>gadget</item><item>widget</item></inventory>
```

Because of the inclusion of `note` elements in the declaration of `inventory`, the following document is also parsed as a valid instance:

```
<inventory><item>gadget<note>new</note><item>widget</inventory>
```

SGML is similar to XML but it has looser requirements on the syntax of the input documents. For the features used by HTML, the only relevant differences are that XML does not support tag omissions nor content model exceptions.

We consider only DTDs that are acyclic:

Definition 1. *An SGML DTD is* acyclic *if it satisfies the following requirement: For elements that allow end tag omissions there must be a bound on the possible depth of the direct nesting of those elements. That is, if we create a directed graph where the nodes correspond to the declared elements whose end tags may be omitted and there is an edge from a node A to a node B if the content model of A contains B, then there must be no cycles in this graph.*

This requirement also exists in Nishiyama and Minamide's approach [11], and it is fulfilled by all versions of the HTML DTD. Contrary to their approach we do not impose any further restrictions and our algorithm thus works for all the HTML DTDs without any limitations or rewritings.

3.1 A Model of HTML Parsing

As our algorithm is a generalization of the traditional SGML parsing algorithm we first present a formal description of the essence of that algorithm. We base our description on the work by Warmer and van Egmond [14]. The algorithm provides the basis for explaining our main contribution in the next section.

We abstract away from SGML features such as text (i.e. PCDATA), comments, and attributes. These features are straightforward to add subsequently. Furthermore, a lexing phase allows us to consider strings over the alphabet of start and end tags, written `<`a`>` and `</`a`>`, respectively, for every element a declared in the DTD. (This lexing phase is far from trivial; our implementation is based on the technique used in [6], and we omit the details here due to the limited space.) More formally, we consider strings over the alphabet $\Sigma = \{\texttt{<}a\texttt{>} \mid a \in \mathcal{E}\} \cup \{\texttt{</}a\texttt{>} \mid a \in \mathcal{E}\}$ where $\mathcal{E}$ is the set of declared element names in the DTD. We assume that `root` $\in \mathcal{E}$ is a pseudo-element representing the root node of the document, with a content model that accepts a single element of any kind (or, one specific, such as `html` for HTML). The sets of included and excluded elements of an element $a \in \mathcal{E}$ are denoted I_a and E_a, respectively.

For simplicity, we represent all content models together as one finite-state automaton [5] defined as follows:

Definition 2. *A* content model automaton *for a DTD D is a tuple $(Q, \mathcal{E}, [q_a]_{a\in\mathcal{E}}, F, \delta)$ where Q is a set of states, its alphabet is $\mathcal{E}$ as defined above, $[q_a]_{a\in\mathcal{E}}$ is a family of initial states (one for each declared element), $F \subseteq Q$ is a set of accept states and $\delta : Q \times \Sigma \hookrightarrow Q$ is a partial transition function (with $\perp$ representing undefined).*

Following the requirement from the SGML standard that content models must be unambiguous, this content model automaton can be assumed to be deterministic by construction. Also, we assume that all states in the automaton can reach some accept state. Each state in the automaton uniquely corresponds to a position in a content model expression in D.

SGML documents are parsed in a single left-to-right scan with a look-ahead of 1. The state of the parser is represented by a *context stack*. The set of possible contexts is $\mathcal{H} = \mathcal{E} \times Q \times \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\mathcal{E})$. ($\mathcal{P}(\mathcal{E})$ denotes the powerset of $\mathcal{E}$.) We refer to the context $c_n = (a, q, \iota, \eta)$ at the top of a stack $c_1 \cdots c_n \in \mathcal{H}^*$ as the *current context*, and a, q, ι, and η are then the *current element*, the *current state*, the *current inclusions*, and the *current exclusions*, respectively. An element b is *permitted* in the current context (a, q, ι, η) if $\delta(q, b) \neq \perp$. We refer to a tag a just below another tag b in the context stack as b's *parent*. We say that $\textsc{OmitStart}(a, q)$ holds if the start tag of a elements may be omitted according to D when the current state is q, and, similarly, $\textsc{OmitEnd}(a, q)$ holds if the end tag of a elements may be omitted in state q. (The precise rules defining $\textsc{OmitStart}$ and $\textsc{OmitEnd}$ from D are quite complicated; we refer to [4,14] for the details.) The current inclusions and exclusions reflect the sets of included and excluded elements, respectively. These two sets can in principle be determined

```
function $Parse_D(p \in \mathcal{H}^*,\ x \in \Sigma^*)$ :
 if $|x| = 0$ then
    // reached end of input
    return $p$
 else if $|p| = 0$ then
    // empty stack error
    return $\bigcirc$
 let $p_1 \cdots p_{n-1} \cdot (a_n, s_n, \iota_n, \eta_n) = p$
 let $x_1 \cdots x_m = x$
 if $x_1 = \texttt{<}a\texttt{>}\ \wedge\ a \notin \eta_n$ for some $a \in \mathcal{E}$ then
    // reading a non-excluded start tag
    if $\delta(s_n, a) \neq \bot$ then
       // the start tag is permitted by the content model, push onto stack and proceed
       return $Parse_D\big(p_1 \cdots p_{n-1} \cdot (a_n, \delta(s_n, a), \iota_n, \eta_n) \cdot (a, q_a, \iota_n \cup I_a, \eta_n \cup E_a),\ x_2 \cdots x_m\big)$
    else if $a \in \iota_n$ then
       // the start tag is permitted by inclusion, push onto stack and proceed
       return $Parse_D\big(p_1 \cdots p_n \cdot (a, q_a, \emptyset, \eta_n \cup E_a),\ x_2 \cdots x_m\big)$
 else if $x_1 = \texttt{</}a\texttt{>} \wedge a = a_n \wedge s_n \in F$ for some $a \in \mathcal{E}$ then
    // reading an end tag that is permitted, pop from stack and proceed
    return $Parse_D\big(p_1 \cdots p_{n-1},\ x_2 \cdots x_m\big)$
 else if $\textsc{OmitEnd}(a_n, s_n)$ then
    // insert omitted end tag, then retry
    return $Parse_D(p,\ \texttt{</}a_n\texttt{>} \cdot x)$
 else if $\exists a' \in \mathcal{E} : \textsc{OmitStart}(a', s_n)$ then
    // insert omitted start tag, then retry
    return $Parse_D(p,\ \texttt{<}a'\texttt{>} \cdot x)$
 else
    // parse error
    return $\lightning$
```

Fig. 2. The $Parse_D$ function for checking validity of a given document

from the element names appearing in the context stack, but we maintain them in each context for reasons that will become clear in Section 4.

Informally, when encountering a start tag <a> that is permitted in the current context, its content automaton state is modified accordingly, and a new context is pushed onto the stack. When an end tag </a> is encountered, the current context is popped off the stack if it matches the element name a.

An end tag may be omitted only if it is followed by either the end tag of another open element or a start tag that is not allowed at this place. A start tag may be omitted only if omission does not cause an ambiguity during parsing. These conditions, which define OmitEnd and OmitStart, can be determined from the current state and either the next tag in the input or the current element on the stack, respectively, without considering the rest of the parse stack and input. Moreover, OmitStart has the property that no more than $|\mathcal{E}|$ omitted start tags can be inserted before the next tag from the input is consumed.

Our formalization of SGML parsing is expressed as the function $Parse_D : \mathcal{H}^* \times \Sigma^* \to \big(\mathcal{H}^* \cup \{\lightning, \bigcirc\}\big)$ shown in Figure 2. The result $\bigcirc$ arises if an end tag is encountered while the stack is empty, and $\lightning$ represents other kinds of parse errors. In this algorithm, OmitEnd and OmitStart allow us to abstract away from the precise rules for tag omission, to keep the presentation simple. The algorithm captures an essential property of SGML parsing: a substring $x \in \Sigma^*$ of a document is parsed relative to a parse stack $p \in \mathcal{H}^*$ as defined above, and it outputs a new parse stack or one of the error indicators $\bigcirc$ and $\lightning$. We distinguish between the two kinds of errors for reasons that become clear in Section 4.

With this, we can define validity of a document relative to the DTD D:

Definition 3. *A string $x \in \Sigma^*$ is a* valid document *if*

$$Parse_D\big((\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset), x\big) = (\mathsf{root}, q, \emptyset, \emptyset)$$

for some $q \in F$.

The $Parse_D$ function has some interesting properties that we shall need in Section 4:

Observation 4. *Notice that the $Parse_D$ function either returns directly or via a tail call to itself. Let $(p^1, x^1), (p^2, x^2), \ldots$ be the sequence of parameters to $Parse_D$ that appear if executing $Parse_D(p^1, x^1)$ for some $p^1 \in \mathcal{H}^*, x^1 \in \Sigma^*$. Now, because the DTD is acyclic, for all $i = 1, 2, \ldots$ we have $|x^{i+|\mathcal{E}|}| < |x^i|$, that is, after at most $|\mathcal{E}|$ recursive calls, one more input symbol is consumed. Moreover, in each step in the recursion sequence, the decisions made depend only on the current context and the next input symbol.*

4 Parsing Context-Free Sets of Documents

We now show that the parsing algorithm described in the previous section can be generalized to work for *sets* of documents, or more precisely, context-free languages over the alphabet Σ. The resulting algorithm determines whether or not all strings in a given language are valid according to a given DTD. The languages are represented as context-free grammars that are constructed by the analysis front-end from the programs being analyzed.

The definitions of context-free grammars and their languages are standard:

Definition 5. *A* context-free grammar *(CFG) is a tuple $G = (N, \Sigma, P, S)$ where N is the set of nonterminal symbols, Σ is the alphabet (of start and end tag symbols, as in Section 3.1), P is the set of productions of the form $A \rightarrow r$ where $A \in N$, $r \in (\Sigma \cup N)^*$, and S is the start nonterminal. The* language *of G is $\mathcal{L}(G) = \{x \in \Sigma^* \mid S \Rightarrow^* x\}$ where $\Rightarrow^*$ is the reflexive transitive closure of the derivation relation $\Rightarrow$ defined by $u_1 A u_2 \Rightarrow u_1 r u_2$ whenever $u_1, u_2 \in (\Sigma \cup N)^*$ and $A \rightarrow r \in P$.*

Definition 6. *A CFG G is* valid *if x is valid for every $x \in \mathcal{L}(G)$.*

To simplify the presentation we will assume that G is in Chomsky normal form, so that all productions are of the form $A \rightarrow s$ or $A \rightarrow A'A''$ where $s \in \Sigma$ and $A, A', A'' \in N$, and that there are no useless nonterminals. It is well-known how to transform an arbitrary CFG to this form [5]. We can disregard the empty string since that is never valid for any DTD, and the empty language is trivially valid.

The idea behind the generalization of the parse algorithm is to find out for every occurrence of an alphabet symbol s in the given CFG which context stacks may appear when encountering s during parsing of a string. The context stacks may of course be unbounded in general. However, because of Observation 4 *we only need to keep track of a bounded size top (i.e. a postfix) of each context stack*, and hence a bounded number of context stacks, at every point in the grammar.

4.1 Generating Constraints

To make the idea more concrete, we define a family of *context functions*, one for each nonterminal $A \in N$. Each is a partial function that takes as input a context stack and returns a set of context stacks:

$$\mathcal{C}_A : \mathcal{H}^* \hookrightarrow \mathcal{P}(\mathcal{H}^*)$$

Informally, the domain of $\mathcal{C}_A$ consists of the context stacks that appear during parsing when entering a substring derived from A, and the co-domain similarly consists of the context stacks that appear immediately after the substring has been parsed. Formally, assume $x \in \mathcal{L}(G)$ such that $S \Rightarrow^* u_1 A u_2 \Rightarrow^* u_1 y u_2 = x$, that is, the nonterminal A is used in the derivation of x, and y is the substring derived from A. The domain $dom(\mathcal{C}_A)$ then contains the context stack p that arises after parsing of u_1, that is, $p = Parse_D\big((\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset), u_1\big) \in dom(\mathcal{C}_A)$. Similarly, $\mathcal{C}_A(p)$ contains the context stack that arises after parsing of $u_1 y$, that is, $Parse_D\big((\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset), u_1 y\big) = Parse_D(p, y) \in \mathcal{C}_A(p)$ if $p \notin \{\lightning, \bigcirc\}$. As explained in detail below, we truncate the context stacks and only store the top of the stacks in these sets. To obtain an efficient algorithm, we truncate as much as possible and exploit the fact that $Parse_D$ returns $\bigcirc$ if a too short context stack is given.

The context functions are defined from the DTD as a solution to the set of constraints defined by the following three rules:

§1 Following Definition 3, parsing starts with the initial context stack at the start nonterminal S and must end in a valid final stack:

$$\mathcal{C}_S(\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset) \subseteq \{(\mathsf{root}, q, \emptyset, \emptyset) \mid q \in F\}$$

§2 For every production of the form $A \to s$ in P where $s \in \Sigma$, the context function for A respects the $Parse_D$ function, which must not return $\lightning$ or $\bigcirc$:

$$\forall p \in dom(\mathcal{C}_A) : p' \notin \{\lightning, \bigcirc\} \;\wedge\; p' \in \mathcal{C}_A(p) \text{ where } p' = Parse_D(p, s)$$

§3 For every production of the form $A \to A'A''$ in P, the entry context stacks of A are also entry context stacks for A', the exit context stacks for A' are also entry context stacks for A'', and the exit context stacks for A'' are also exit context stacks for A. However, we allow the context stacks to be truncated when propagated from one nonterminal to the next:

$$\begin{aligned}\forall p \in dom(\mathcal{C}_A) : \exists p_1, p_2 : p = p_1 \cdot p_2 \;\wedge\; p_2 \in dom(\mathcal{C}_{A'}) \wedge \\ \forall p_2' \in \mathcal{C}_{A'}(p_2) : \exists t_1, t_2 : p_1 \cdot p_2' = t_1 \cdot t_2 \;\wedge\; t_2 \in dom(\mathcal{C}_{A''}) \wedge \\ \forall t_2' \in \mathcal{C}_{A''}(t_2) \Rightarrow t_1 \cdot t_2' \in \mathcal{C}_A(p)\end{aligned}$$

Note that rule §3 permits the context stacks to be truncated; on the other hand, rule §2 ensures that the stacks are not truncated too much since that would lead to the error value $\bigcirc$.

Theorem 7. *There exists a solution to the constraints defined by the rules above for a grammar G if and only if G is valid.*

Proof. *See the technical report [10].*

4.2 Solving Constraints

It is relatively simple to construct an algorithm that searches for a solution to the collection of constraints generated from a CFG by the rules defined in Section 4.1. Figure 3 shows the pseudo-code for such an algorithm, $\mathit{ParseCFG}_D$. We write $w\,\textsc{defs}\,A$ for $w \in P$, $A \in N$ if A appears on the left-hand side of w, and $w\,\textsc{uses}\,A$ if A appears on the right-hand side of w. The solution being constructed is represented by the family of context functions, denoted $[\mathcal{C}_A]_{A \in N}$.

The idea in the algorithm is to search for a solution by truncating the context stacks as much as possible, iteratively trying longer context stacks, until the special error value $\bigcirc$ no longer appears. The algorithm initializes $[\mathcal{C}_A]_{A \in N}$ on line 6 and iteratively on lines 9–58 extends these functions to build a solution. The worklist W (a queue, without duplicates) consists of productions that need to be processed because the domains of the context functions of their left-hand-side nonterminals have changed. The function Δ maintains for each nonterminal a set of context stacks that are known to lead to $\bigcirc$.

Each production in the worklist of the form $A \to s$ is parsed according to rule §2 on lines 14– 26, relative to each context stack p in $\mathit{dom}(\mathcal{C}_A)$. If this results in $\bigcirc$, the corresponding context stack is added to $\Delta(A)$, and all productions that use A are added to the worklist to make sure that the information that the context stack was too short is propagated back to those productions. If a parse error $\lightning$ occurs (line 20), the algorithm terminates with a failure. If the parsing is successful (line 23), the resulting context stack p' is added to $\mathcal{C}_A$.

For a production of two nonterminals, $A \to A'A''$, we proceed according to rule §3. For each context stack p in $\mathit{dom}(\mathcal{C}_A)$ on line 29 we pick the smallest possible postfix p_2 of p that is not in $\Delta(A')$ and propagate this to $\mathcal{C}_{A'}$. If no such postfix exists, we know that p is too short, so we update $\Delta(A)$ and W as before. Otherwise, we repeat the process (line 37) to propagate the resulting context stack through A'' and further to $\mathcal{C}_A$ (line 46).

Finally, on line 57 we check that rule §1 is satisfied.

Theorem 8. *The $\mathit{ParseCFG}_D$ algorithm always terminates, and it terminates successfully if and only if a solution exists to the constraints from Section 4.1 for the given CFG.*
(We leave a proof of this theorem as future work.)

Corollary 9. *Combining Theorem 7 and Theorem 8, we see that $\mathit{ParseCFG}_D$ always terminates, and it terminates successfully if and only if the given CFG is valid.*

4.3 Example

As an example of a normalized grammar, consider $G_{ul} = (N, \Sigma, P, S)$ where $N = \{A_1, A_2, A_3, A_4, A_5, A_6\}$, $\Sigma = \{\texttt{<ul>}, \texttt{</ul>}, \texttt{<li>}, \texttt{</li>}\}$, $S = A_1$, and P consists of the following productions:

```
function $\mathit{ParseCFG}_D(N, \Sigma, P, S)$ :
  declare $W \subseteq P,\ [\mathcal{C}_A]_{A \in N} : \mathcal{H}^* \hookrightarrow \mathcal{P}(\mathcal{H}^*),\ \Delta : N \to \mathcal{P}(\mathcal{H}^*)$
  // initialize worklist and context functions
  $W := [\, w \in P \mid w \text{ DEFS } S \,]$
  for all $A \in N,\ p \in \mathcal{H}^*$ do
    $\mathcal{C}_A(p) := \begin{cases} \emptyset & \text{if } A = S \ \wedge\ p = (\mathsf{root}, q_{\mathsf{root},\emptyset,\emptyset}) \\ \bot & \text{otherwise} \end{cases}$
    $\Delta(A) := \emptyset$
  // iterate until fixpoint
  while $W \neq \emptyset$ do
    remove the next production $A \to r$ from $W$
    for all $p \in \mathit{dom}(\mathcal{C}_A)$ do
      if $A \to r$ is of the form $A \to s$ where $s \in \Sigma$ then
        // rule §2
        let $p' = \mathit{Parse}_D(p, s)$
        if $p' = \bigcirc$ then
          // record that entry context stack p is too short for A
          $\Delta(A) := \Delta(A) \cup \{p\}$
          $\mathcal{C}_A(p) := \bot$
          for all $w \in P$ where $w \text{ USES } A$ add $w$ to $W$
        else if $p' = \lightning$ then
          // fail right away
          fail
        else if $p' \notin \mathcal{C}_A(p)$ then
          // add new final context stack p' for A
          $\mathcal{C}_A(p) := \mathcal{C}_A(p) \cup \{p'\}$
          for all $w \in P$ where $w \text{ USES } A$ add $w$ to $W$
      else if $A \to r$ is of the form $A \to A'A''$ where $A', A'' \in N$ then
        // rule §3
        let $p_2$ be the smallest string such that $p = p_1 \cdot p_2$ and $p_2 \notin \Delta(A')$
        if no such $p_2$ exists then
          // record that entry context stack p is too short for A
          $\Delta(A) := \Delta(A) \cup \{p\}$
          $\mathcal{C}_A(p) := \bot$
          for all $w \in P$ where $w \text{ USES } A$ add $w$ to $W$
        else if $p_2 \in \mathit{dom}(\mathcal{C}_{A'})$ then
          for all $p_2' \in \mathcal{C}_{A'}(p_2)$ do
            let $t_2$ be the smallest string such that $p_1 \cdot p_2' = t_1 \cdot t_2$ and $t_2 \notin \Delta(A'')$
            if no such $t_2$ exists then
              // record that entry context stack p is too short for A
              $\Delta(A) := \Delta(A) \cup \{p\}$
              $\mathcal{C}_A(p) := \bot$
              for all $w \in P$ where $w \text{ USES } A$ add $w$ to $W$
            else if $t_2 \in \mathit{dom}(\mathcal{C}_{A''})$ then
              if $\{t_1 \cdot t_2' \mid t_2' \in \mathcal{C}_{A''}(t_2)\} \not\subseteq \mathcal{C}_A(p)$ then
                // add new final context stacks for A
                $\mathcal{C}_A(p) := \mathcal{C}_A(p) \cup \{t_1 \cdot t_2' \mid t_2' \in \mathcal{C}_{A''}(t_2)\}$
                for all $w \in P$ where $w \text{ USES } A$ add $w$ to $W$
            else
              // add new entry context stack t_2 for A''
              $\mathcal{C}_{A''}(t_2) := \emptyset$
              for all $w \in P$ where $w \text{ DEFS } A''$ add $w$ to $W$
        else
          // add new entry context stack p_2 for A'
          $\mathcal{C}_{A'}(p_2) := \emptyset$
          for all $w \in P$ where $w \text{ DEFS } A'$ add $w$ to $W$
    // rule §1
    if $\mathcal{C}_S(\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset) \not\subseteq \{(\mathsf{root}, q, \emptyset, \emptyset) \mid q \in F\}$ then
      fail
  return $[\mathcal{C}_A]_{A \in N}$
```

Fig. 3. The $\mathit{ParseCFG}_D$ algorithm for solving the parse constraints for a given CFG

$$\begin{array}{ll}
A_1 \rightarrow A_5\ A_2 & A_2 \rightarrow A_6\ A_3 \\
A_3 \rightarrow \texttt{</ul>} & A_4 \rightarrow \texttt{<li>} \\
A_5 \rightarrow \texttt{<ul>} & A_6 \rightarrow \texttt{<li>} \\
A_6 \rightarrow A_4\ A_1 &
\end{array}$$

The language generated by G_{ul} consists of documents that have a `ul` root element containing a single `li` element that in turn contains zero or one `ul` element. The grammar can thus generate deeply nested `ul` and `li` elements, and truncation of context stacks is therefore crucial for the $\mathit{ParseCFG}_D$ algorithm to terminate. Notice that all `</li>` end tags are omitted in the documents.

We wish to ensure that the strings generated from G_{ul} are valid relative to the following DTD, which mimics a very small fraction of the HTML DTD for unordered lists:

```
<!ELEMENT ul - - (li*)>
<!ELEMENT li - O (ul*)>
```

For this combination of a CFG and a DTD, the $\mathit{ParseCFG}_D$ algorithm produces the following solution to the constraints:

	$\mathcal{C}$
A_1	$(\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset) \mapsto \{(\mathsf{root}, q, \emptyset, \emptyset)\}$ $(\texttt{li}, q_{li}, \emptyset, \emptyset) \mapsto \{(\texttt{li}, q_{li}, \emptyset, \emptyset)\}$
A_2	$(\texttt{ul}, q_{ul}, \emptyset, \emptyset) \mapsto \{\epsilon\}$
A_3	$(\texttt{ul}, q_{ul}, \emptyset, \emptyset) \cdot (\texttt{li}, q_{li}, \emptyset, \emptyset) \mapsto \{\epsilon\}$
A_4	$(\texttt{ul}, q_{ul}, \emptyset, \emptyset) \mapsto \{(\texttt{ul}, q_{ul}, \emptyset, \emptyset) \cdot (\texttt{li}, q_{li}, \emptyset, \emptyset)\}$
A_5	$(\texttt{li}, q_{li}, \emptyset, \emptyset) \mapsto \{(\texttt{li}, q_{li}, \emptyset, \emptyset) \cdot (\texttt{ul}, q_{ul}, \emptyset, \emptyset)\}$ $(\mathsf{root}, q_{\mathsf{root}}, \emptyset, \emptyset) \mapsto \{(\mathsf{root}, q, \emptyset, \emptyset) \cdot (\texttt{ul}, q_{ul}, \emptyset, \emptyset)\}$
A_6	$(\texttt{ul}, q_{ul}, \emptyset, \emptyset) \mapsto \{(\texttt{ul}, q_{ul}, \emptyset, \emptyset) \cdot (\texttt{li}, q_{li}, \emptyset, \emptyset)\}$

Although the context stacks may grow arbitrarily when parsing individual documents with Parse_D, the truncation trick ensures that $\mathit{ParseCFG}_D$ terminates and succeeds in capturing the relevant top-most parts of the context stacks.

5 Experimental Results

We have implemented the algorithm from Section 4.2 in Java, together with an analysis front-end for constructing CFGs that soundly approximate the output of web applications written with Java Servlets and JSP. The front-end follows the structure described in [6], extended with specialized support for JSP, and builds on Soot [13] and the Java String Analyzer [2]. (We omit a more detailed explanation of this front-end, due to the limited space.)

The purpose of the prototype implementation is to obtain preliminary answers to the following research questions:

- What is the typical analysis time for a Servlet/JSP page, and how is the analysis time affected by the absence or presence of validity errors?
- What is the precision of the analysis in terms of false positives?
- Are the warnings produced by the tool useful to locate the sources of the errors?

We have run the analysis on six open source programs found on the web. The programs range from simple one man projects, such as the JSP Chat application (JSP Chat[3]), the official J2EE tutorial Servlet and JSP examples (J2EE Bookstore 1 and 2[4]) to the widely used blogging framework Pebble[5], which included dozens of pages and features. We have also included the largest example from a book on JSTL (JSTL Book ex.[6]) and an application named JPivot[7]. The tests have been performed on a 2.4 GHz Core i5 laptop with 4GB RAM running OS X. As DTD, we use HTML 4.01 Transitional.

Figure 4 summarizes the results. For each program, it shows the number of JSP pages, the time it takes to run the whole analysis on all pages (excluding the time used by Soot), the time spent in the CFG parser algorithm, the number of warnings from the analyzer, and the number of false positives determined by manual inspection of the analyzed source code.

The tool currently has two limitations, which we expect to remedy with a modest additional implementation effort. First, validation of attributes is currently not supported. Second, the implementation can track a validity error to the place in the generated Java code where the invalid element is generated, but not all the way back to the JSP source in the case of JSP pages.

In some cases when an unknown value is inserted into the output without escaping special XML characters (for example, by using the `out` tag from JSTL), the front-end is unable to reason about the language of that value. This may for instance happen when the value is read from disk or input at runtime. The analysis will in such cases issue an additional warning, which is not included in the count in Figure 4, and treat the unknown value as a special alphabet symbol and continue analyzing the grammar. In practice, there are typically a few such symbols per page. While they may be indications of cross site scripting vulnerabilities, there may also be invariants in the program ensuring that there is no problem at runtime.

The typical analysis time for a single JSP page is around 200-600 ms. As can be seen from the table, only a small fraction of the time is spent on parsing the CFG. The worklist algorithm typically requires between 1 and 100 iterations for each JSP page, which means that each nonterminal is visited between 1 and 10 times.

Validity errors were found in all the applications. The following is an example of a warning generated by the tool on the JSP Chat application:

[3] http://www.web-tech-india.com/software/jsp_chat.php

[4] http://download.oracle.com/javaee/5/tutorial/doc/bnaey.html

[5] http://pebble.sourceforge.net/

[6] http://www.manning.com/bayern/

[7] http://jpivot.sourceforge.net/

Program	Pages	Time	CFG Parser time	Warnings	False positives
Pebble[3]	61	24.0 s	369 ms	32	0
J2EE Bookstore 1[4]	5	6.7 s	93 ms	5	0
J2EE Bookstore 2[4]	7	9.0 s	<1 ms	7	0
JPivot[5]	3	2.8 s	8 ms	2	0
JSP Chat[6]	14	6.8 s	100 ms	12	0
JSTL Book ex.[7]	14	4.9 s	24 ms	6	0

Fig. 4. Analysis times and results for various open source web applications written in Java Servlets and JSP

```
ERROR: Invalid string printed in
  dk.brics.servletvalidator.jsp.generated.editInfo_jsp on line 94:
Start tag INPUT not allowed in TBODY
Parse context is [root HTML BODY DIV CENTER FORM TABLE TBODY]
```

This warning indicates that the programmer forgot both a `tr` start tag and a `td` start tag in which the `input` element would be allowed, causing the `input` tag to appear directly inside the `tbody` element. This may very well lead to browsers rendering the page differently.

The reason that all JSP pages of the J2EE Bookstore applications are invalid it that there is an unmatched `</center>` tag and a nonstandard `<comment>` tag in a header used by all pages. After removing these two tags, only one page of this application is (correctly) rejected by the analysis. While Pebble seems to be programmed with the goal of only outputting valid HTML, the general problem in this web application is that the `table`, `ul`, and `tr` elements require non-empty contents, which is not always respected by Pebble. Furthermore, several more serious errors, such as forgotten `td` tags, exist in the application. The JSP Chat application is written in JSP but makes heavy use of embedded Java code. The tool is able to analyze it precisely enough to find several errors that are mostly due to unobvious (but feasible) flow in the program.

Based on the warnings generated by the tool, we managed to manually correct all the errors within a few hours without any prior knowledge of the applications. After running the analysis again, no more warnings were produced. This second round of analysis took essentially the same time as before the errors were corrected. Since the analysis is sound, we can trust that the applications after the corrections cannot output invalid HTML.

6 Conclusion

We have presented an algorithm for validating context-free sets of documents relative to an HTML DTD. The key idea – to generalize a parsing algorithm for SGML to work on grammars instead of concrete documents – has lead to an approach that smoothly handles the intricate features of HTML, in particular tag omissions and exceptions. Preliminary experiments with our prototype implementation indicate that the approach is sufficiently efficient and precise to function as a practically useful tool during development of web applications. In future work, we plan to improve the tool to accommodate for attributes and to trace error messages all the way back to the JSP source (which is tricky because of the JSP tag file mechanism) and to perform a more extensive evaluation.

References

1. Chen, S., Hong, D., Shen, V.Y.: An experimental study on validation problems with existing HTML webpages. In: Proc. International Conference on Internet Computing, ICOMP 2005 (June 2005)
2. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
3. Doh, K.-G., Kim, H., Schmidt, D.A.: Abstract parsing: Static analysis of dynamically generated string output using LR-parsing technology. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 256–272. Springer, Heidelberg (2009)
4. Goldfarb, C.F.: The SGML Handbook. Oxford University Press, Oxford (1991)
5. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
6. Kirkegaard, C., Møller, A.: Static analysis for java servlets and JSP. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 336–352. Springer, Heidelberg (2006)
7. Minamide, Y.: Static approximation of dynamically generated Web pages. In: Proc. 14th International Conference on World Wide Web, WWW 2005, pp. 432–441. ACM, New York (May 2005)
8. Minamide, Y., Tozawa, A.: XML validation for context-free grammars. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 357–373. Springer, Heidelberg (2006)
9. Møller, A., Schwartzbach, M.I.: The design space of type checkers for XML transformation languages. In: Eiter, T., Libkin, L. (eds.) ICDT 2005. LNCS, vol. 3363, pp. 17–36. Springer, Heidelberg (2005)
10. Møller, A., Schwarz, M.: HTML validation of context-free languages. Technical report, Department of Computer Science, Aarhus University (2011), `http://cs.au.dk/~amoeller/papers/htmlcfg/`
11. Nishiyama, T., Minamide, Y.: A translation from the HTML DTD into a regular hedge grammar. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 122–131. Springer, Heidelberg (2008)
12. Thiemann, P.: Grammar-based analysis of string expressions. In: Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, TLDI 2005 (2005)
13. Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot – a Java optimization framework. In: Proc. IBM Centre for Advanced Studies Conference, CASCON 1999. IBM (November 1999)
14. Warmer, J., van Egmond, S.: The implementation of the Amsterdam SGML parser. Electronic Publishing 2(2), 65–90 (1988)

On the Power of Cliques in the Parameterized Verification of Ad Hoc Networks

Giorgio Delzanno[1], Arnaud Sangnier[2], and Gianluigi Zavattaro[3]

[1] University of Genova, Italy
[2] LIAFA, University Paris 7, CNRS, France
[3] University of Bologna, INRIA Focus Team, Italy

Abstract. We study decision problems for parameterized verification of protocols for ad hoc networks. The problem we consider is control state reachability for networks of arbitrary size. We restrict our analysis to topologies that approximate the notion of bounded diameter often used in ad hoc networks for optimizing broadcast communication. We show that restricting to graphs with bounded diameter is not sufficient to make control state reachability decidable, but the problem turns out to be decidable when considering an additionally restricted class of graphs that still includes cliques. Although decidable, the problem is already Ackermann-hard over clique graphs.

1 Introduction

Ad hoc networks consist of wireless hosts that, in the absence of a fixed infrastructure, communicate sending broadcast messages. In this context protocols are typically supposed to work independently from the communication topology and from the size (number of nodes) of the network. As suggested in [3], the *control state reachability problem* (or *coverability problem*) seems a particularly adequate formalization of parameterized verification problems for ad hoc networks. A network is represented in [3] as a graph in which nodes are individual processes and edges represent communication links. Each node executes an instance of the same protocol. A protocol is described by a finite state communicating automaton. The control state reachability problem consists in checking whether there exists an initial configuration that can evolve into a configuration in which at least one node is in a given error state. Since the size and the topology of the initial configuration is not fixed a priori, the state-space to be explored is in general infinite. As proved in [3], control state reachability is undecidable if no restrictions are considered for the possible initial configurations. As in other communication models [12,20], finding interesting classes of network topologies for which verification is, at least theoretically, possible is an important research problem.

Moving along this line, in this paper we consider networks in which the underlying topology is in between the class of *cliques* and the strictly larger class of *bounded diameter graphs*. Cliques represent the best possible topology for minimizing the number of hops needed for diffusing data. Furthermore, control state

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 441–455, 2011.

reachability in clique graphs reduces to coverability in a Broadcast Protocol (with unstructured configurations), a problem proved to be decidable in [7].

Our first result is negative. Indeed, we prove that control state reachability is undecidable for networks in which configurations have diameter bounded by a value k such that $k > 1$ (notice that connected graphs with diameter 1 corresponds to cliques). We investigate then further restrictions having in mind the constraint that they must allow at least cliques of arbitrary order. By using an original well-quasi ordering result, we prove that coverability becomes decidable when considering a class of graphs in which the corresponding maximal cliques are connected by paths of bounded length. Furthermore, by exploiting a recent result of Schnoebelen [18] and a reduction to coverability in reset nets, we show that the resulting decision procedure is Ackermann-hard. Interestingly, this complexity result already holds in the subclass of clique topologies. Finally, we introduce a unicast mechanism inspired by rendezvous communication in other concurrency models. Having the two mechanisms in the same model allows us to compare them, with complexity measures, with respect to the coverability problem. Specifically, coverability for unicast communication is easier than for selective broadcast. Indeed, it turns out to be in EXPSPACE for unrestricted graphs. To the best of our knowledge, this discrimination result is novel compared to the existing literature on concurrency models with selective broadcast and unicast communication.

Related Work. Model checking has been applied to verify protocols for ad hoc networks with a fixed number of nodes in [8,19]. A possibly non-terminating procedure for the verification of routing protocols in ad hoc networks of arbitrary size is described in [17]. In [3] we have introduced and studied the (repeated) control state reachability problem described in the introduction for the ad hoc network model of [19]. Specifically, we have shown that the problem is undecidable when the topology is unrestricted and that it becomes decidable when the initial network has a topology taken from the class of graphs with bounded paths (the maximal length of a path is bounded by a constant). However this class does not include cliques of arbitrary order. In contrast, we extend here the decidability result to a larger class of graphs, and we investigate the problem for graphs with bounded diameter. Graphs with bounded paths have also been considered in verification problems with point-to-point (unicast in the ad hoc setting) communication in [12,16,20].

Due to lack of space, omitted proofs can be found in [4].

2 Preliminaries on Graphs

In this section we assume that Q is a finite set of elements. A *Q-labeled undirected graph* (shortly Q-graph or graph) is a tuple $G = (V, E, L)$, where V is a finite set of *vertices* (sometimes called *nodes*), and $E \subseteq V \times V$ is a finite set of *edges*, and $L : V \rightarrow Q$ is a labeling function. We consider here undirected graphs, i.e., such that $\langle u, v\rangle \in E$ iff $\langle v, u\rangle \in E$. We denote by $\mathcal{G}_Q$ the set of Q-graphs. For an edge $\langle u, v\rangle \in E$, u and v are called its *endpoints* and we say that u and v are adjacent vertices. For a node u we call *vicinity* the set of its adjacent nodes (neighbors). Given a vertex $v \in V$, the *degree* of v is the size of the set $\{u \in V \mid \langle v, u \rangle \in E\}$. The degree of a graph is the maximum degree of its vertices. We will sometimes denote $L(G)$ the set $L(V)$ (which is a subset of Q). A *path* π in a graph is a finite sequence $v_1, v_2, \ldots, v_m$ of vertices such that for $1 \leq i \leq m-1$, $\langle v_i, v_{i+1} \rangle \in E$ and the integer $m-1$ (i.e. its number of edges) is called the length of the path π, denoted by $|\pi|$. A path $\pi = v_1, \ldots, v_m$ is simple if for all $1 \leq i, j \leq m$ with $i \neq j$, $v_i \neq v_j$, in other words each vertex of the graph occurs at most once in π. A *cycle* is a path $\pi = v_1, \ldots, v_m$ such that $v_1 = v_m$. A graph $G = \langle V, E, L \rangle$ is *connected* if for all $u, v \in V$ with $u \neq v$, there exists a path from u to v in G. A *clique* in an undirected graph $G = \langle V, E, L \rangle$ is a subset $C \subseteq V$ of vertices, such that for every $u, v \in C$ with $u \neq v$, $\langle u, v \rangle \in E$. A clique C is said to be *maximal* if there exists no vertex $u \in V \setminus C$ such that $C \cup \{u\}$ is a clique. If the entire set of nodes V is a clique, we say that G is a clique. A *bipartite Q-graph* is a tuple $\langle V_1, V_2, E, L \rangle$ such that $\langle V_1 \cup V_2, E, L \rangle$ is a Q-graph, $V_1 \cap V_2 = \emptyset$ and $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$.

The *diameter* of a graph $G = \langle V, E, L \rangle$ is the length of the *longest shortest simple path* between any two vertices of G. Hence, the diameter of a clique is always one. We also need to define some graph orderings. Given two graphs $G = \langle V, E, L \rangle$ and $G' = \langle V', E', L' \rangle$, G is in the *subgraph* relation with G', written $G \preceq_s G'$, whenever there exists an injection $f : V \rightarrow V'$ such that, for every $v, v' \in V$, if $\langle v, v' \rangle \in E$, then $\langle f(v), f(v') \rangle \in E'$ and for every $v \in V$, $L(v) = L'(f(v))$. Furthermore, G is in the *induced subgraph* relation with G', written $G \preceq_i G'$, whenever there exists an injection $f : V \rightarrow V'$ such that, for every $v, v' \in V$, $\langle v, v' \rangle \in E$ if and only if $\langle f(v), f(v') \rangle \in E'$ and for every $v \in V$, $L(v) = L'(f(v))$. As an example, a path with three nodes is a subgraph, but not an induced subgraph, of a ring of the same order. Finally, we recall the notion of *well-quasi-ordering* (wqo for short). A quasi order $(A, \leq)$ is a wqo if for every infinite sequence of elements $a_1, a_2, \ldots, a_i, \ldots$ in A, there exist two indices $i < j$ s.t. $a_i \leq a_j$. Examples of wqo's are the sub-multiset relation, and both the subgraph and the induced subgraph relation over graphs with simple paths of bounded length [5].

3 Ad Hoc Networks

In our model of ad hoc networks a configuration is simply a graph and we assume that each node of the graph is a process that runs a common predefined protocol. A protocol is defined by a communicating automaton with a finite set Q of control states. Communication is achieved via selective broadcast. The effect of a broadcast is in fact local to the vicinity of the sender. The initial configuration is any graph in which all the nodes are in an initial control state. Remark that even if Q is finite, there are infinitely many possible initial configurations. We next formalize the above intuition.

Individual Behavior. The protocol run by each node is defined via a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$, where Q is a finite set of control states, Σ is a finite alphabet,

$R \subseteq Q \times (\{\tau\} \cup \{!!a, ??a \mid a \in \Sigma\}) \times Q$ is the transition relation, and $Q_0 \subseteq Q$ is a set of initial control states. The label τ represents the capability of performing an internal action, and the label $!!a$ ($??a$) represents the capability of broadcasting (receiving) a message $a \in \Sigma$.

Network Semantics. An AHN associated to $\mathcal{P}$ is defined via a transition system $\mathcal{A}_{\mathcal{P}} = \langle \mathcal{C}, \Rightarrow, \mathcal{C}_0 \rangle$, where $\mathcal{C} = \mathcal{G}_Q$ (undirected graphs with labels in Q) is the set of configurations, $\mathcal{C}_0 = \mathcal{G}_{Q_0}$ (undirected graphs with labels in Q_0) is the subset of initial configurations, and $\Rightarrow \subseteq \mathcal{C} \times \mathcal{C}$ is the transition relation defined next. For $u \in V$, we first define the set $R_a(u) = \{q \in Q \mid \langle L(u), ??a, q \rangle \in R\}$ that contains states that can be reached from the state $L(u)$ upon reception of message a. For $G = \langle V, E, L \rangle$ and $G' = \langle V', E', L' \rangle$, $G \Rightarrow G'$ holds iff G and G' have the same underlying structure, i.e., $V = V'$ and $E = E'$, and one of the following conditions on L and L' holds:

- $\exists v \in V$ s.t. $(L(v), \tau, L'(v)) \in R$, and $L(u) = L'(u)$ for all u in $V \setminus \{v\}$;
- $\exists v \in V$ s.t. $(L(v), !!a, L'(v)) \in R$ and for every $u \in V \setminus \{v\}$
  - if $\langle v, u \rangle \in E$ and $R_a(u) \neq \emptyset$ (reception of a in u is enabled), then $L'(u) \in R_a(u)$.
  - $L(u) = L'(u)$, otherwise.

An execution is a sequence $G_0 G_1 \ldots$ such that $G_0 \in \mathcal{G}_{Q_0}$ and $G_i \Rightarrow G_{i+1}$ for $i \geq 0$. We use $\Rightarrow^*$ to denote the reflexive and transitive closure of $\Rightarrow$.

Observe that a broadcast message a sent by v is delivered only to the subset of neighbors interested in it. Such a neighbor u updates its state with a new state taken from $R(u)$. All the other nodes (including neighbors not interested in a) simply ignore the message. Also notice that the topology is static, i.e., the set of nodes and edges remain unchanged during a run.

Finally, for a set of Q-graphs $\mathcal{T} \subseteq \mathcal{G}_Q$, the AHN $A_{\mathcal{P}}^{\mathcal{T}}$ restricted to $\mathcal{T}$ is defined by the transition system $\langle \mathcal{C} \cap \mathcal{T}, \Rightarrow_{\mathcal{T}}, \mathcal{C}_0 \cap \mathcal{T} \rangle$ where the relation $\Rightarrow_{\mathcal{T}}$ is the restriction of $\Rightarrow$ to $(\mathcal{C} \cap \mathcal{T}) \times (\mathcal{C} \cap \mathcal{T})$.

Decision Problem

Given a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$ with an associated AHN $\mathcal{A}_{\mathcal{P}} = \langle \mathcal{C}, \Rightarrow, \mathcal{C}_0 \rangle$, we define the *control state reachability* (COVER) as follows:

> Given a control state $q \in Q$, does there exist $G \in \mathcal{C}_0$ and $G' \in \mathcal{C}$ such that $q \in L(G')$ and $G \Rightarrow^* G'$?

Control state reachability is strictly related to parameterized verification of safety properties. The input control state q can be seen as an error state for the execution of the protocol in some node of the network. If the answer to COVER is yes, then there exists a sufficient number of processes, all executing the same protocol, and an initial topology from which we can generate a configuration in which the error is exposed. Under this perspective, COVER can be viewed as instance of a parameterized verification problem.

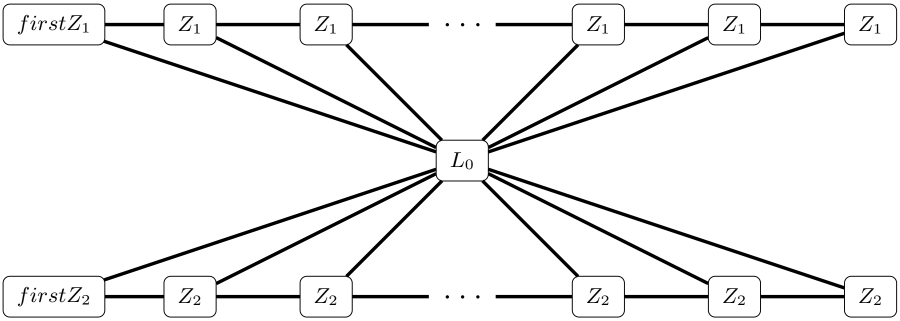


Fig. 1. Butterfly-shaped induced subgraph needed to simulate a Minsky machine

4 Configurations with Bounded Diameter

As mentioned in the introduction, COVER is undecidable for configurations with unrestricted topology [3]. The problem becomes decidable when configurations are restricted to graphs with *k-bounded paths* (BP_k) for any $k \geq 0$. k-bounded path graphs are graphs in which there exist no simple path with length strictly greater than k. The class BP_k is infinite for any $k > 0$. As an example, with $k = 2$ it includes star-shaped graphs of any order.

Unfortunately, restricting protocol analysis to configurations in BP_k seems to have a limited application in a communication model with selective broadcast. Indeed, we first observe that BP_k does not include the class K consisting of clique graphs of any order. Cliques however are appealing for at least two reasons. First, they represent the best possible scenario for optimizing broadcast communication (one broadcast to reach all nodes). Second, when restricting configurations only to graphs in the class K, COVER can be reduced to coverability in a Broadcast Protocol, i.e., in a model in which configurations are multisets of processes defined by communicating automata [6]. Coverability is decidable in Broadcast Protocols in [7]. For these reasons, in this paper we investige COVER in restricted classes of graphs that at least include the class K. The first class we consider is that of graphs with bounded diameter. Fixed $k > 0$, a graph G has a k-bounded diameter if and only if its diameter is smaller than or equal to k. Observe that for every $k > 0$, clique graphs belong to the class of graphs with a diameter bounded by k. Furthermore, given $k > 0$ the class BP_k is included in the class of graphs with a diameter bounded by k. Graphs with k-bounded diameter coincide with the so called k-clusters used in partitioning algorithm for ad hoc networks [9]. Thus, this class is of particular relevance for the analysis of selective broadcast communication. Intuitively, the diameter corresponds to the minimal number of broadcasts (hops) needed to send a message to all nodes connected by a path with the sender.

The COVER problem restricted to configurations with k-bounded diameter turns out to be undecidable for $k > 1$. Indeed, we show next that AHNs working over this class of configurations can be used to simulate the behavior of a deterministic Minsky machine. A deterministic Minsky machine manipulates two integer variables c_1 and c_2, which are called counters, and it is composed of a finite set of instructions. Instructions are of the form (1) $L : c_i := c_i + 1;$ $\texttt{goto}$ L' or (2) L : $\texttt{if}$ $c_i = 0$ $\texttt{then goto}$ L' $\texttt{else}$ $c_i := c_i - 1;$ $\texttt{goto}$ L'' where $i \in \{1, 2\}$ and L, L', L'' are labels preceding each instruction. There is also a special halting label L_F. The halting problem consists in deciding whether or not the execution that starts from L_0, with both counters set to 0, reaches L_F. The halting problem for deterministic two counter machines is undecidable [14]. The encoding is built in two steps.

We first need to run a protocol that terminates successfully only when the projection of the configuration on an appropriate set of control state is a sort of butterfly (see Figure 1) consisting of two lists (to represent the counters) and in which all nodes in the lists are connected to a monitor node (to represent the program counter).

To reach such a configuration, we use a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$ with $\{L_0, firstZ_1, Z_1, firstZ_2, Z_2, error\} \subseteq Q$ such that if G_0 is an initial configuration in $\mathcal{A}_{\mathcal{P}} = \langle \mathcal{C}, \Rightarrow, \mathcal{C}_0 \rangle$ and if $G_0 \Rightarrow^* G$ for a configuration $G = \langle V, E, L \rangle$ verifying $L_0 \in L(V)$, then the graph $\theta = \langle V', E', L' \rangle$ represented in Figure 1 is an induced subgraph of G. Furthermore, all vertices $v \in V \setminus V'$ adjacent to a vertex of θ in G are labeled with *error*. We also have that all the nodes labeled with L_0 in G are connected as in θ (when abstracting away the nodes in state *error*). In these graphs θ of Figure 1, the number of nodes is guessed nondeterministically and represents the maximum value reached by the counters during the simulation. Thus, the number of Z_1 and Z_2 can be different. However, there is at least one node labeled Z_1 and one labeled Z_2. The diameter of θ is equal to 2 no matters how many nodes there are labelled with Z_1 or Z_2. The protocol for the first step is described in detail in [4].

Once the configuration is in the desired form, the second step consists in the simulation of the instructions of the encoded Minsky machine. The protocol for this step is shown in Figure 2 (as far as the simulation of the counters is concerned) and 3 (for the simulation of the instructions). More precisely, we build a process $\mathcal{P}'$ obtained by completing the process $\mathcal{P}$ with the processes shown on the Figures 2 and 3.

The simulation works as follows: first if the Minsky machine is at the line labelled with L and m is the value of the first counter (the same reasoning holds for the second counter) then in the corresponding configuration of the AHN there is one node labelled with L which is neighbor of $m - 1$ nodes labelled by NZ_1 and if $m > 0$ this node has also a neighbor labelled by $firstNZ_i$, if $m = 0$ then this same neighbor is labelled by $firstZ_i$. To simulate an increment of the form $L_1 : c_i := c_i + 1;$ $\texttt{goto}$ L_2, the node of the AHN labelled with L_1 sends an inc_i and the unique node labelled by $nextZ_i$ receives it, acknowledges it by sending an $ackinc_i$ and updates its unique neighbor labelled by Z_i to $nextZ_i$. The decrement works in the same manner except that if the value of the counter c_i is equal to 0 the node labelled with a label of the Minsky machine receives a $zero_i$ otherwise it receives a dec_i. We have then that in $\mathcal{P}'$ there is an execution from an initial configuration which reaches a configuration where at least one

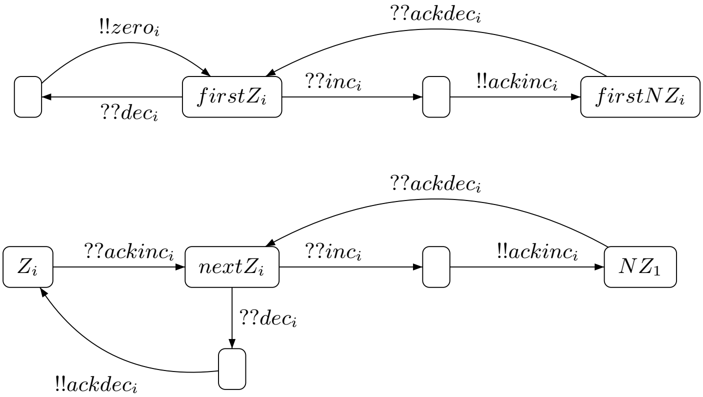


Fig. 2. Simulation of the instructions for counter c_i

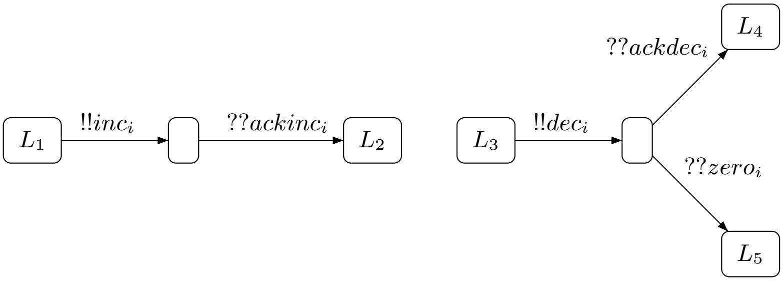


Fig. 3. Encoding (L_1 : $c_i := c_i + 1$; `goto` L_2) and (L_3 : `if` $c_i = 0$ `then` `goto` L_5 `else` $c_i := c_i - 1$; `goto` L_4)

node is labelled by L_F if and only if the corresponding Minsky machine halts. This allows us to deduce:

Theorem 1. *For $k > 1$,* COVER *restricted to configurations with k-bounded diameter is undecidable.*

Note that if we restrict our attention to graphs with a diameter bounded by 1, the above encoding does not work anymore. The class of graphs with diameter 1 corresponds to the set of clique graphs and, as said above, COVER turns out to be decidable when restricting to clique topologies.

Bounded diameter and bounded degree. From a non trivial result on bounded diameter graphs [11], we obtain an interesting decidable subclass. Indeed, in [11] the authors show that, given two integers $k, d > 0$, the number of graphs whose diameter is smaller than k and whose degree (max number of neighbors) is smaller than d is finite. The Moore bound $M(k, d) = (k(k-1)^d-2)/(k-2)$ is an upper bound for the size of the largest undirected graph in such a class. The following property then holds.

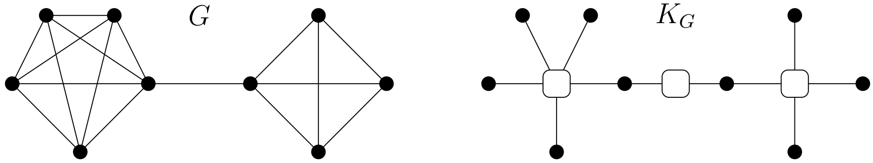


Fig. 4. A graph G and its associated clique graph K_G

Theorem 2. *For fixed $k, d > 0$ and given a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$,* COVER *restricted to configurations with k-bounded diameter and d-bounded degree is in* PSPACE *in the size of P.*

Proof. From [11], it follows that the number of possible configurations is finite. Thus, COVER is decidable. Since k and d are two fixed constants, the constant $N = M(k,d)$ gives us an upper bound on the number of nodes of the largest graph to be considered. Notice that we only need polynomial space in the size of P to store a graph with size smaller or equal than N. To solve the COVER problem, we define a non deterministic algorithm that first guesses the initial graph G_0 and then explore all possible successor configurations in search for an error state. Since the topology never changes, in the worst case we have to consider all possible relabelings of the initial graph. Thus, the size of the state space is bounded by $|Q|^N$ and it is still polynomial in the size of P.

5 Maximal Clique Graphs with Bounded Paths

In this section we prove decidability for COVER restricted to the class of graphs we call BPC_n (n-Bounded Path maximal Cliques graphs). BPC_n contains both n-bounded path graphs and any clique graph, while being strictly contained in the class of graphs with k-bounded diameter. The class is defined on top of the notion of *maximal clique graphs* associated to a configuration.

Definition 1. *Given a connected undirected graph $G = \langle V, E, L \rangle$ and $\bullet \notin L(V)$, the* maximal clique graph *K_G is the bipartite graph $\langle X, W, E', L' \rangle$ in which*

- *$X = V$;*
- *$W \subseteq 2^V$ is the set of maximal cliques of G;*
- *For $v \in V, w \in W$, $\langle v, w \rangle \in E'$ iff $v \in w$;*
- *$L'(v) = L(v)$ for $v \in V$, and $L'(w) = \bullet$ for $w \in W$.*

Note that for each connected graph G there exists a unique maximal clique graph K_G. An example of construction is given by Figure 4. One can also easily prove that if G is a clique graph then in K_G there is no path of length strictly greater than 3. Furthermore, from the maximality of the cliques in W if two nodes $v_1, v_2 \in V$ are connected both to w_1 and $w_2 \in W$, then w_1 and w_2 are distinct cliques. We use the notation $v_1 \sim_w v_2$ to denote that v_1, v_2 belong to the same clique w.

Definition 2. *For $n \geq 1$, the class BPC_n consists of the set of configurations whose associate maximal clique graph has n-bounded paths (i.e. the length of the simple paths of K_G is at most n).*

Let us now study the properties of this class of graphs. We first introduce the following ordering on BPC_n graphs.

Definition 3. *Assume $G_1 = \langle V_1, E_1, L_1 \rangle$ with $K_{G_1} = \langle X_1, W_1, E'_1, L'_1 \rangle$, and $G_2 = \langle V_2, E_2, L_2 \rangle$ with $K_{G_2} = \langle X_2, W_2, E'_2, L'_2 \rangle$ with G_1 and G_2 both connected graphs. Then, $G_1 \sqsubseteq G_2$ iff there exist two injections $f : X_1 \to X_2$ and $g : W_1 \to W_2$, such that*

i. for every $v \in X_1$, and $C \in W_1$, $v \in C$ iff $f(v) \in g(C)$;
ii. for every $v_1, v_2 \in X_1$, and $C \in W_2$, if $f(v_1) \sim_C f(v_2)$, then there exists $C' \in W_1$ s.t. $f(v_1) \sim_{g(C')} f(v_2)$;
iii. for every $v \in X_1$, $L'_1(v) = L'_2(f(v))$;
iv. for every $C \in W_1$, $L'_1(C) = L'_2(g(C))$.

The first condition ensures that (dis)connected nodes remain (dis)connected inside the image of g. Indeed, from i it follows that, for every $v_1, v_2 \in X_1$, and $C \in W_1$, $v_1 \sim_C v_2$ iff $f(v_1) \sim_{g(C)} f(v_2)$. The second condition ensures that disconnected nodes remain disconnected outside the image of g.

By condition i in the definition of $\sqsubseteq$, we have that $G_1 \sqsubseteq G_2$ (via f and g) implies that K_{G_1} is in the induced subgraph relation with K_{G_2} (via $f \cup g$). Furthermore, we also have the following property:

Lemma 1. *$G_1 \sqsubseteq G_2$ iff $G_1 \preceq_i G_2$ (G_1 is an induced subgraph of G_2).*

We are now interested in the property of being wqo for the above defined graph orderings. Lemma 1 shows that the ordering $\sqsubseteq$, defined on the maximal clique graph, is equivalent to the induced subgraph ordering on the original graphs. It is well known that the induced subgraph relation is not a wqo for generic graphs (e.g. consider the infinite sequence of rings of increasing size). There are however interesting classes of graphs for which the induced subgraph ordering is wqo. For instance, induced subgraphs is a wqo for the class of graphs for which the length of simple paths is bounded by a constant (bounded path graphs). This result is known as Ding's Theorem [5]. Now observe that given $n \geq 1$ the class BPC_n we are interested in contains cliques of arbitrary order and it also strictly contains the class of $n/2$-bounded path graphs. Interestingly, Ding's result can be extended to the BPC_n class for every $n \geq 1$.

Lemma 2. *For any $n \geq 1$, $(BPC_n, \sqsubseteq)$ is a well-quasi ordering.*

The proof, given in [4], follows Ding's induction method and exploits a decomposition property of bounded path graphs due to Robertson and Seymour.

Given a subset $S \subseteq BPC_n$, we now define its *upward closure* $S \uparrow = \{G' \in BPC_n \mid G \in S \text{ and } G \sqsubseteq G'\}$, i.e., $S \uparrow$ is the set of configurations generated by those in S via $\sqsubseteq$. A set $S \subseteq BPC_n$ is an *upward closed set* w.r.t. to $(BPC_n, \sqsubseteq)$ if $S \uparrow = S$. Since $(BPC_n, \sqsubseteq)$ is a wqo, we obtain that every set of configurations that is upward closed w.r.t. $(BPC_n, \sqsubseteq)$ has a finite basis, i.e., it can be finitely represented by a finite number of graphs. We can exploit this property to define a decision procedure for the coverability problem. For this purpose, we apply the methodology proposed in [1]. The first property we need is that the transition relation induced by our model is compatible with $\sqsubseteq$.

Lemma 3. *Fixed $n \geq 1$, for every $G_1, G_2, G_1' \in BPC_n$ such that $G_1 \Rightarrow_{BPC_n} G_2$ and $G_1 \sqsubseteq G_1'$, there exists $G_2' \in BPC_n$ such that $G_1' \Rightarrow_{BPC_n} G_2'$ and $G_2 \sqsubseteq G_2'$.*

For a fixed $n \geq 1$, monotonicity ensures that if S is an upward closed set of configurations, then the set of predecessors of S according to $\Rightarrow$, defined as $pre(S) = \{G \mid G \Rightarrow_{BPC_n} G' \text{ and } G' \in S\}$, is still upward closed. Furthermore, we can effectively compute a finite representation of $S \cup pre(S)$.

Lemma 4. *Given a finite basis B of an upward closed set $S \subseteq BPC_n$, there exists an algorithm to compute a finite basis B' of $S \cup pre(S)$ s.t. $S \cup pre(S) = B' \uparrow$.*

This allows us to state the main theorem of this section.

Theorem 3. *Given $n \geq 1$,* COVER *restricted to BPC_n configurations is decidable.*

Proof. It follows from Lemmas 2, 3, and 4 and from the general properties of well structured transition systems [1,2,10]. □

6 Ackermann-Hardness of COVER in BPC_n

In the previous section we have proved that, despite COVER is undecidable for AHNs, it becomes decidable when imposing the configurations to be in BPC_n and this for every $n \geq 1$. We prove here, that even if decidable, this problem is not primitive recursive. The proof is by reduction from the coverability problem for reset nets, which is known to be an Ackermann-hard problem [18].

A reset net RN is a tuple $\langle P, T, \boldsymbol{m_0} \rangle$ such that P is a finite set of places, T is a finite set of transitions, and $\boldsymbol{m_0}$ is a marking, i.e. a mapping from P to $\mathbb{N}$ that defines the initial number of tokens in each place of the net. A transition $t \in T$ is defined by a mapping ${}^\bullet t$ (preset) from P to $\mathbb{N}$, a mapping ${}^\bullet t$ (postset), and by a set of reset arcs $t\!\downarrow \subseteq P$. A configuration is a marking $\boldsymbol{m}$. Transition t is enabled at marking $\boldsymbol{m}$ iff ${}^\bullet t(p) \leq \boldsymbol{m}(p)$ for each $p \in P$. Firing t at $\boldsymbol{m}$ leads to a new marking $\boldsymbol{m'}$ defined as $\boldsymbol{m'}(p) = \boldsymbol{m}(p) -^\bullet t(p) + t^\bullet(p)$ if $p \notin t\!\downarrow$, and $\boldsymbol{m'}(p) = 0$ otherwise. We assume that if $p \in t\!\downarrow$ then $t^\bullet(p) = 0$. A marking $\boldsymbol{m}$ is reachable from $\boldsymbol{m_0}$ if it is possible to produce it after firing finitely many times transitions in T. Given a reset net $\langle P, T, \boldsymbol{m_0} \rangle$ and a marking $\boldsymbol{m}$, the coverability problem consists in checking for the existence of a reachable marking $\boldsymbol{m'}$ such that $\boldsymbol{m'}(p) \geq \boldsymbol{m}(p)$ for every $p \in P$. In [18] it is proved that the coverability problem for reset nets is Ackermann-hard.

We start by showing a linear reduction of the coverability problem for reset nets to COVER for the class of AHNs with clique topologies, denoted with K. Note

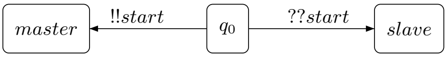


Fig. 5. The *initialization* phase

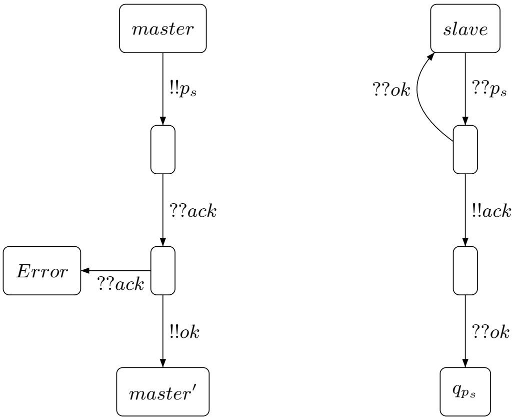


Fig. 6. Generating the initial marking with only one token in p_s

that K corresponds to BPC_n with $2 \leq n < 4$. Then, we show how to generalize the presented reduction to AHNs with topologies in BPC_n, with $n \geq 4$.

Let $RN = \langle P, T, \boldsymbol{m_0} \rangle$ be a reset net, and let $\boldsymbol{m}$ be a marking. We construct a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$ with a control state $q \in Q$ such that $\boldsymbol{m}$ is coverable in RN iff the control state q is reachable in $A^K_{\mathcal{P}}$. We assume, without loss of generality, that both $\boldsymbol{m_0}$ and $\boldsymbol{m}$ contain only one token, i.e. there exist two places p_s and p_e such that $\boldsymbol{m}_0(p_s) = 1$ (resp. $\boldsymbol{m}(p_e) = 1$) and $\boldsymbol{m_0}(p) = 0$ (resp. $\boldsymbol{m}(p) = 0$) for every $p \neq p_s$ (resp. $p \neq p_e$).

We now describe the corresponding process definition $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$. We define $Q_0 = \{q_0\}$, i.e. all the processes are initially in the state q_0. At the beginning the processes perform a simple protocol (depicted in Fig. 5) that elects one node as the *master*, and the other nodes become *slaves.*

The master will control the simulation of the reset net, while the slaves will be used to represent tokens in the net markings. Namely, when a slave process is in the state $q_p \in Q$, it represents one token in the place $p \in P$. For instance, in order to represent the initial marking it is necessary for the master to move one slave in the state q_{p_s}. This is achieved by the protocol in Fig. 6.

Note that the protocol can deadlock in two possible cases: either when there is no slave node, or two of them reply with the *ack* message before the master closes the protocol with the *ok* message. If the master completes the protocol by entering in the state *master*′, exactly one slave moved to the state q_{p_s}.

At this stage, the simulation of the net transitions starts. The master in state *master*′ nondeterministically selects one of the possible transitions t, with $^{\bullet}t = \{p_1, \ldots, p_n\}$, $t \downarrow = \{p'_1, \ldots, p'_m\}$, and $t^{\bullet} = \{p''_1, \ldots, p''_l\}$, and it starts its

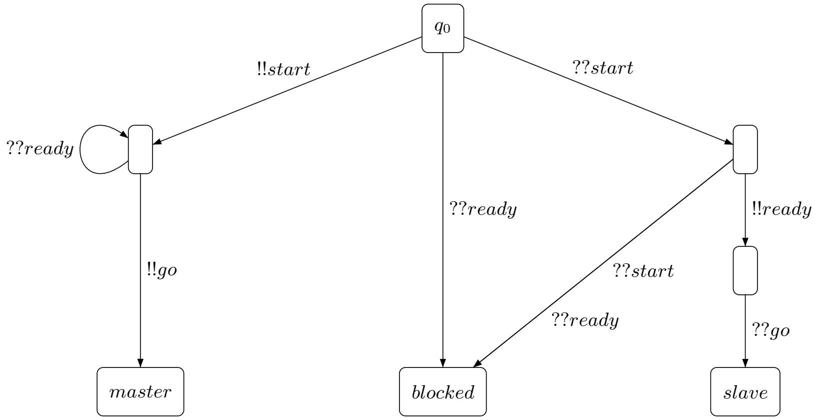


Fig. 7. The *initialization* phase for BPC_n with $n \geq 4$

simulation by performing the following protocol. It first tries to consume the tokens in the preset $^\bullet t$ by performing in sequence protocols similar to the one in Fig. 6: in this case, it moves processes from the states q_{p_i} to *slave* to simulate the consumption of tokens in the places p_i. After, the reset actions are performed simply by emitting the messages $resetp'_i$, whose effect is to move all nodes in the states $q_{p'_i}$ to the *slave* state. Finally, by performing in sequence the same protocol of Fig. 6, it simulates the production of tokens in the places p''_i.

Lemma 5. *Given a marking* $\boldsymbol{m}$ *containing only one token in* p_e*, we have that* $\boldsymbol{m}$ *can be covered in RN iff* $A^K_{\mathcal{P}}$ *satisfies* COVER *for the state* q_{p_e}*.*

Proof. The *if* part follows from the fact that every ad hoc network in $A^K_{\mathcal{P}}$ correctly reproduces computations of the reset net (it simply introduces deadlocks that are not relevant as far as the coverability problem is concerned). The *only-if* part is a consequence of the fact that every finite computation of the reset net can be simulated by at least one ad hoc network in $A^K_{\mathcal{P}}$ having a sufficient number of nodes. □

It is easy to see that the above construction does not work for topologies different from the clique. For instance, in the topologies in BPC_n with $n \geq 4$, there are nodes belonging to two distinct maximal cliques. If such nodes are connected to two distinct masters, they could generate interferences among them. In order to cope with this problem, we build another process $\mathcal{P}'$ obtained by replacing the trivial initialization protocol of Fig. 5 with the most sophisticated one depicted in Fig. 7.

After the execution of this initialization protocol, we have the guarantee that *slave* processes do not generate interferences between two distinct *master* nodes. In fact, at the end of the protocol we have the guarantee that every *slave* node is connected to exactly one *master*, and all of its other neighbors are *blocked*.

Lemma 6. *Given a marking $\boldsymbol{m}$ containing only one token in p_e and $n \geq 4$, we have that $\boldsymbol{m}$ can be covered in RN iff $A_{\mathcal{P}'}^{BPC_n}$ satisfies* COVER *for the state q_{p_e}.*

We can conclude with the main result of this section.

Theorem 4. *For every $n \geq 2$,* COVER *restricted to BPC_n configurations is non-primitive recursive.*

Proof. The result follows from the Ackermann-hardness of reset nets [18], and from Lemma 5 and Lemma 6. It is sufficient to note that K coincides with BPC_n with $2 \leq n < 4$, and to observe that $\mathcal{P}$ is obtained in linear time from the reset net RN.

7 Broadcast vs. Unicast Communication

Although broadcast communication is specifically devised for networks in which nodes have no complete knowledge of the surrounding topology, unicast (point-to-point) communication is often provided, e.g., to exchange information after the acquisition of the information on the vicinity of a node. We investigate here the relationship between the coverability problem for unicast and broadcast communication. Specifically, we show that the two problems can be kept separated (i.e. the problem is more difficult for broadcast) in all the classes of graphs studied in [3] and in the present paper.

For this analysis we first introduce two primitives $!a$ and $?a$ for unicast communication. As in CCS [13], when a process sends a message a (action $!a$) it synchronizes with only one process that is in a state in which it is ready to receive a (action $?a$). The receiving process is nondeterministically chosen among those ready to receive a. For unicast communication the definition of a process $\mathcal{P} = \langle Q, \Sigma, R, Q_0 \rangle$ is modified in the component R that is now a subset of $Q \times (\{\tau\} \cup \{!a, ?a\} \mid a \in \Sigma) \times Q$. The operational semantics is obtained via a transition relation $\Rightarrow$ defined as follows: given two configurations $G = \langle V, E, L \rangle$ and $G' = \langle V', E', L' \rangle$, we have $G \Rightarrow G'$ iff G and G' have the same underlying structure, i.e., $V = V'$ and $E = E'$, and, in addition to the conditions for τ of Section 3, the following condition on L and L' defines a case in which a transition may take place:

- there exists $v \neq w \in V$ such that $(L(v), !a, L'(v))$ and $(L(w), ?a, L'(w))$ are both in R, and $L(u) = L'(u)$ for all u in $V \setminus \{v, w\}$.

For the sake of clarity, in the rest of the section we name AHN^b the model with broadcast (and no unicast) and AHN^u the model with unicast (and no broadcast).

The coverability problem for AHN^u can be reduced to the corresponding problem for AHN^b. Indeed, unicast communication can be simulated via broadcast messages via a protocol like the one in Figure 6. The encoding introduces deadlocks that are not relevant as far as the COVER problem is concerned. The following theorem then holds.

Theorem 5. *The control state reachability problem for AHN*^{u} *is in* EXPSPACE*.*

Proof. We first show that we can restrict our attention to clique graphs only. Indeed, given a state q, if there exist G_0 and G_1 with n nodes s.t. $G_0 \Rightarrow_G^* G_1$ and q is a label in G_1, then there exist two cliques K_0 and K_1 with order n s.t. $K_0 \Rightarrow_G^* K_1$ and q is a label in K_1. This property follows from the observation that for any graph G with $n' \leq n$ nodes, there exists a clique graph with n nodes such that $G \preceq_s K_n$.

Now let K_0 be the clique such that $G_0 \preceq_s K_0$. Since $G_0 \Rightarrow_G^* G_1$, by exploiting the monotonicity of unicast communication w.r.t. subgraph ordering, we have that there exists K_1 s.t. $K_0 \Rightarrow_G^* K_1$ and q is a label in K_1. We observe that control state reachability in the class of clique graphs can be reduced to coverability in a Petri net in which each place corresponds to a state in Q. The initial marking is produced by firing transitions that produce a nondeterministically chosen number of tokens in the places in Q_0. For each unicast communication step involving a pair of nodes in state q and q', we add a transition with q and q' in the preset, and the corresponding target states in the postset.

It follows then from classical results on Petri nets [15] that we can use an EXPSPACE decision procedure for deciding COVER for AHN^u.

From this property and from the undecidability result for coverability in AHN^b (for unrestricted topologies), we observe that there cannot be any recursive encoding of coverability in AHN^b into the corresponding problem in AHN^u. Furthermore, in the case of cliques with bounded paths, from Theorem 4, we have that there is no primitive recursive encoding of coverability in AHN^b. This way we separate the difficulty of the coverability problem in the two types of communication schemes.

8 Conclusions

In this paper we have extended the decidability result for verification of ad hoc networks with bounded path topology presented in [3] to a larger and more interesting class of graphs. The new class consists of topologies in which the corresponding maximal cliques are connected by paths of bounded length. This class of graphs is contained in the class of graphs with bounded diameter, for which control state reachability turns out to be undecidable. Furthermore, it contains the class of clique graphs for which control state reachability is proved to be Ackermann-hard. In the paper we have also compared the expressive power of broadcast and unicast communication with respect to the control state reachability problem. As a future work, we plan to study decidability issues in presence of communication and node failure and to consider extensions of the ad hoc network model with features like timing information and structured messages.

References

1. Abdulla, P.A., Čerāns, C., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: LICS 1996, pp. 313–321. IEEE Computer Society, Los Alamitos (1996)
2. Abdulla, P.A., Čerāns, C., Jonsson, B., Tsay, Y.-K.: Algorithmic analysis of programs with well quasi-ordered domains. Inf. Comput. 160(1-2), 109–127 (2000)
3. Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized Verification of Ad Hoc Networks. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 313–327. Springer, Heidelberg (2010)
4. Delzanno, G., Sangnier, A., Zavattaro, G.: On the Power of Cliques in the Parameterized Verification of Ad Hoc Networks. Technical Report DISI-TR-11-01, DISI-University of Genova (2011)
5. Ding, G.: Subgraphs and well quasi ordering. J. of Graph Theory 16(5), 489–502 (1992)
6. Emerson, E.A., Namjoshi, K.S.: On model checking for non-deterministic infinite-state systems. In: LICS 1998, pp. 70–80. IEEE Computer Society, Los Alamitos (1998)
7. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: LICS 1999, pp. 352–359. IEEE Computer Society, Los Alamitos (1999)
8. Fehnker, A., van Hoesel, L., Mader, A.: Modelling and verification of the LMAC protocol for wireless sensor networks. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 253–272. Springer, Heidelberg (2007)
9. Fernandess, Y., Malkhi, D.: K-clustering in wireless ad hoc networks. In: POMC 2002, pp. 31–37. ACM, New York (2002)
10. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoret. Comp. Sci. 256(1-2), 63–92 (2001)
11. Hoffman, A., Singleton, R.: On Moore graphs with diameter 2 and 3. IBM J. Res. Develop. 4, 497–504 (1960)
12. Meyer, R.: On boundedness in depth in the pi-calculus. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) IFIP TCS 2008. IFIP, vol. 273, pp. 477–489. Springer, Heidelberg (2008)
13. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
14. Minsky, M.: Computation: finite and infinite machines. Prentice-Hall, Englewood Cliffs (1967)
15. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theoret. Comp. Sci. 6, 223–231 (1978)
16. Rosa-Velardo, F.: Depth boundedness in multiset rewriting systems with name binding. In: Kučera, A., Potapov, I. (eds.) RP 2010. LNCS, vol. 6227, pp. 161–175. Springer, Heidelberg (2010)
17. Saksena, M., Wibling, O., Jonsson, B.: Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 18–32. Springer, Heidelberg (2008)
18. Schnoebelen, P.: Revisiting Ackermann-Hardness for Lossy Counter Machines and Reset Petri Nets. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 616–628. Springer, Heidelberg (2010)
19. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: Query-Based Model Checking of Ad Hoc Network Protocols. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 603–619. Springer, Heidelberg (2009)
20. Wies, T., Zufferey, D., Henzinger, T.A.: Forward analysis of depth-bounded processes. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 94–108. Springer, Heidelberg (2010)

The Reduced Product of Abstract Domains and the Combination of Decision Procedures

Patrick Cousot [2,3], Radhia Cousot [3,1], and Laurent Mauborgne [3,4]

[1] Centre National de la Recherche Scientifique, Paris
[2] Courant Institute of Mathematical Sciences, New York University
[3] École Normale Supérieure & Inria, Paris
[4] Instituto Madrileño de Estudios Avanzados, Madrid

Abstract. The algebraic/model theoretic design of static analyzers uses abstract domains based on representations of properties and pre-calculated property transformers. It is very efficient. The logical/proof theoretic approach uses SMT solvers and computation on-the-fly of property transformers. It is very expressive. We propose a combination of the two approaches to reach the sweet spot best adapted to a specific application domain in the precision/cost spectrum. The proposed combination uses an iterated reduction to combine abstractions. The key observation is that the Nelson-Oppen procedure which decides satisfiability in a combination of logical theories by exchanging equalities and disequalities computes a reduced product (after the state is enhanced with some new "observations" corresponding to alien terms). By abandoning restrictions ensuring completeness (such as disjointness, convexity, stably-infiniteness or shininess, etc) we can even broaden the application scope of logical abstractions for static analysis (which is incomplete anyway). We also introduce a semantics based on multiple interpretations to deal with the soundness of that combinations on a formal basis.

1 Introduction

Recent progress in SMT solvers and theorem provers as used in program verification [2] has been recently exploited for static analysis by abstract interpretation [3, 4] using logical abstract domains [21, 10]. This approach hardly scales up and is based on a mathematical program semantics quite different from the implementation (such as integers instead of modular arithmetics). Static analyzers such as Astrée [1] which are based on algebraic abstractions of the machine semantics do not have such efficiency and soundness limitations. However their expressivity is limited by that of their abstract domains. It is therefore interesting to combine algebraic and logical abstract interpretations to get the best of both worlds i.e. scalability, expressivity, natural interface with the end-user using logical formulæ, and soundness with respect to the machine semantics. The proposed combination is based on the understanding of the Nelson-Oppen procedure [15] as an iterated observation reduced product.

After some syntax, we recall *multi-interpreted* semantics [5], a necessary mean to describe the soundness and relative precision of the *logical abstract domains* defined in sect. 2.8. Next section, we introduce *observational semantics*, which is a new construction generalizing static analysis practices and necessary to describe the first step of

M. Hofmann (Ed.): FOSSACS 2011, LNCS 6604, pp. 456–472, 2011.

the Nelson-Oppen procedure in the abstract interpretation framework. Sect. 4 recalls the notion of reduced product and introduces the iterated reduced product, with new incompleteness results on that approach. Then sect. 5 is focused on the Nelson-Oppen procedure and the links with abstract interpretation. Finally, sect. 6 develops new methods to combine classical abstract interpretation and theorem provers.

2 Syntax and Semantics of Programs

In this section, we recall the notions introduced in [5] necessary to deal with classical abstract domains and logical abstract domains in a common semantic framework.

2.1 Syntax

We define a *signature* as a tuple $\Sigma = \langle \mathbb{x},\ \mathbb{f},\ \mathbb{p} \rangle$ such that the sets $\mathtt{x} \in \mathbb{x}$ of variables, $\mathtt{f} \in \mathbb{f} = \bigcup_{n \geqslant 0} \mathbb{f}^n$ of function symbols ($\mathtt{c} \in \mathbb{f}^0$ are constants), and $\mathtt{p} \in \mathbb{p} \triangleq \bigcup_{n \geqslant 0} \mathbb{p}^n$ of predicate symbols are mutually disjoints. The *terms* $t \in \mathbb{T}(\Sigma) ::= \mathtt{x} \mid \mathtt{c} \mid \mathtt{f}(t_1, \ldots, t_n)$. Conjunctive clauses $\varphi, \psi, \ldots \in \mathbb{C}(\Sigma) ::= a \mid \varphi \wedge \varphi$ are quantifier-free formulæ in simple conjunctive normal form. First-order logic formulæ $\Psi, \Phi, \ldots \in \mathbb{F}(\Sigma) ::= a \mid \neg\Psi \mid \Psi \wedge \Psi \mid \exists\, \mathtt{x} : \Psi$ may be quantified. Finally programs of the programming language on a given signature Σ are built out of basic expressions $e \in \mathbb{E}(\Sigma) \triangleq \mathbb{T}(\Sigma) \cup \mathbb{A}(\Sigma)$ and imperative commands $C \in \mathbb{L}(\Sigma)$ including assignments and tests $C ::= \mathtt{x} := e \mid \varphi$. Tests appear in conditionals and loops whose syntax, as well as that of programs, is irrelevant.

2.2 Interpretations

An *interpretation* I for a signature Σ is a pair $\langle I_{\mathcal{V}},\ I_{\gamma} \rangle$ such that $I_{\mathcal{V}}$ is a non-empty set of values and I_{γ} interprets constants, functions (into $I_{\mathcal{V}}$) and predicates (into Boolean values $\mathcal{B} \triangleq \{\mathit{false}, \mathit{true}\}$). Let $\mathfrak{I}(\Sigma)$ be the class of all such interpretations. In a given interpretation $I \in \mathfrak{I}(\Sigma)$, an environment $\eta \in \mathcal{R}_I^{\Sigma} \triangleq \mathbb{x} \to I_{\mathcal{V}}$ is a function from variables to values. An interpretation I and an environment $\eta \in \mathcal{R}_I^{\Sigma}$ *satisfy* a formula Ψ, written $I \models_{\eta} \Psi$, in the following way:

$$\begin{array}{rcl} I \models_{\eta} a \triangleq [\![a]\!]_I\eta & \quad & I \models_{\eta} \Psi \wedge \Psi' \triangleq (I \models_{\eta} \Psi) \wedge (I \models_{\eta} \Psi') \\ I \models_{\eta} \neg\Psi \triangleq \neg(I \models_{\eta} \Psi) & \quad & I \models_{\eta} \exists\, \mathtt{x} : \Psi \triangleq \exists\, v \in I_{\mathcal{V}} :\ I \models_{\eta[\mathtt{x} \leftarrow v]} \Psi\,^{1} \end{array}$$

where the value $[\![a]\!]_I\eta \in \mathcal{B}$ of an atomic formula $a \in \mathbb{A}(\Sigma)$ in environment $\eta \in \mathcal{R}_I^{\Sigma}$ is

$$\begin{array}{lcl} [\![\mathtt{ff}]\!]_I\eta \triangleq \mathit{false} & \qquad & [\![\mathtt{p}(t_1, \ldots, t_n)]\!]_I\eta \triangleq I_{\gamma}(\mathtt{p})([\![t_1]\!]_I\eta, \ldots, [\![t_n]\!]_I\eta), \quad n \geqslant 1 \\ [\![\neg a]\!]_I\eta \triangleq \neg[\![a]\!]_I\eta, & & \text{where } \neg\mathit{true} = \mathit{false},\ \neg\mathit{false} = \mathit{true} \end{array}$$

and the value $[\![t]\!]_I\eta \in I_{\mathcal{V}}$ of the term $t \in \mathbb{T}(\Sigma)$ in environment $\eta \in \mathcal{R}_I^{\Sigma}$ is

$$[\![\mathtt{x}]\!]_I\eta \triangleq \eta(\mathtt{x}) \qquad [\![\mathtt{c}]\!]_I\eta \triangleq I_{\gamma}(\mathtt{c}) \qquad [\![\mathtt{f}(t_1, \ldots, t_n)]\!]_I\eta \triangleq I_{\gamma}(\mathtt{f})([\![t_1]\!]_I\eta, \ldots, [\![t_n]\!]_I\eta)\,.$$

In addition, in first-order logics with equality the interpretation of equality is always $I \models_{\eta} t_1 = t_2 \triangleq [\![t_1]\!]_I\eta =_I [\![t_2]\!]_I\eta$ where $=_I$ is the unique reflexive, symmetric, and transitive relation on $I_{\mathcal{V}}$ encoded by its characteristic function.

[1] $\eta[\mathtt{x} \leftarrow v]$ is the assignment of v to $\mathtt{x}$ in η where and $\eta[\mathtt{x} \leftarrow v](\mathtt{y}) \triangleq \eta(\mathtt{y})$ when $\mathtt{x} \neq \mathtt{y}$.

2.3 Multi-interpreted Program Semantics

A *multi-interpreted semantics* [5] assigns meanings to a program $\mathtt{P}$ in the context of a set of interpretations $\mathcal{I} \subseteq \boldsymbol{\mathfrak{I}}(\Sigma)$ for the program signature Σ. For example, integers can have a mathematical interpretation or a modular interpretation on machines. Then a program property in $\mathfrak{P}^{\Sigma}_{\mathcal{I}}$ provides for each interpretation in $\mathcal{I}$, a set of environments for variables $\mathbb{x}$ satisfying that property in that interpretation.

$$\begin{array}{lcll} \mathfrak{R}^{\Sigma}_{\mathcal{I}} & \triangleq & \left\{\langle I,\, \eta\rangle \,\middle|\, I \in \mathcal{I} \wedge \eta \in \mathcal{R}^{\Sigma}_{I}\right\} & \text{multi-interpreted environments} \\ \mathfrak{P}^{\Sigma}_{\mathcal{I}} & \triangleq & \wp(\mathfrak{R}^{\Sigma}_{\mathcal{I}}) & \text{multi-interpreted properties.} \end{array}$$

The multi-interpreted concrete semantics $C^{\Sigma}_{\mathcal{I}}[\![\mathtt{P}]\!] \in \mathfrak{P}^{\Sigma}_{\mathcal{I}}$ of a program $\mathtt{P}$ in the context of multi-interpretations $\mathcal{I}$ is assumed to be defined in least fixpoint form $C^{\Sigma}_{\mathcal{I}}[\![\mathtt{P}]\!] \triangleq \mathbf{lfp}^{\subseteq} F^{\Sigma}_{\mathcal{I}}[\![\mathtt{P}]\!]$ where the concrete transformer $F^{\Sigma}_{\mathcal{I}}[\![\mathtt{P}]\!] \in \mathfrak{P}^{\Sigma}_{\mathcal{I}} \xrightarrow{\nearrow} \mathfrak{P}^{\Sigma}_{\mathcal{I}}$ is assumed to be increasing[2]. Since $\langle \mathfrak{P}^{\Sigma}_{\mathcal{I}},\, \subseteq,\, \emptyset,\, \mathfrak{R}^{\Sigma}_{\mathcal{I}},\, \cup,\, \cap\rangle$ is a complete lattice, $\mathbf{lfp}^{\subseteq} F^{\Sigma}_{\mathcal{I}}[\![\mathtt{P}]\!]$ does exist by Tarski's fixpoint theorem. The transformer $F^{\Sigma}_{\mathcal{I}}[\![\mathtt{P}]\!]$ is defined by structural induction on the program $\mathtt{P}$ in terms of the complete lattice operations and the following local transformers for the

$$\begin{array}{ll} \text{assignment postcondition} & \mathsf{f}_{\mathcal{I}}[\![\mathtt{x} := e]\!]P \triangleq \left\{\langle I,\, \eta[\mathtt{x} \leftarrow [\![e]\!]_I\eta]\rangle \,\middle|\, I \in \mathcal{I} \wedge \langle I,\, \eta\rangle \in P\right\} \\ \text{assignment precondition} & \mathsf{b}_{\mathcal{I}}[\![\mathtt{x} := e]\!]P \triangleq \left\{\langle I,\, \eta\rangle \,\middle|\, I \in \mathcal{I} \wedge \langle I,\, \eta[\mathtt{x} \leftarrow [\![e]\!]_I\eta]\rangle \in P\right\} \\ \text{and tests} & \qquad \mathsf{p}_{\mathcal{I}}[\![\varphi]\!]P \triangleq \left\{\langle I,\, \eta\rangle \in P \,\middle|\, I \in \mathcal{I} \wedge [\![\varphi]\!]_I\eta = \mathit{true}\right\}\ . \end{array}$$

To recover the usual concrete semantics, let $\mathfrak{I} \in \boldsymbol{\mathfrak{I}}(\Sigma)$ be the *program standard interpretation* e.g. as defined explicitly by a standard or implicitly by a compiler, linker, loader, operating system and network of machines. Then the standard concrete semantics is $\mathcal{I} = \{\mathfrak{I}\}$. The reason why we consider multi-interpretations is that it is the natural setting for the logical abstract domains which are valid up to a theory (Sect. 2.7), which can have many different interpretations.

2.4 Algebraic Abstract Domains

We let $\langle A^{\Sigma}_{\mathcal{I}},\, \sqsubseteq,\, \top,\, \sqcup,\, \ldots,\, \bar{\mathsf{f}}, \bar{\mathsf{p}},\, \ldots\rangle$ be an *abstract domain* abstracting multi-interpreted properties in $\mathfrak{P}^{\Sigma}_{\mathcal{I}}$ for signature Σ and multi-interpretations $\mathcal{I}$ with partial ordering $\sqsubseteq$. Pre-orders are assumed to be quotiented by the preorder equivalence so $A^{\Sigma}_{\mathcal{I}}$ is a poset but may-be not a complete lattice nor a cpo. The meaning of the abstract properties is defined by an increasing *concretization* function $\gamma^{\Sigma}_{\mathcal{I}} \in A^{\Sigma}_{\mathcal{I}} \xrightarrow{\nearrow} \mathfrak{P}^{\Sigma}_{\mathcal{I}}$. In case of existence of a best abstraction, we use a *Galois connection* $\langle \mathfrak{P}^{\Sigma}_{\mathcal{I}},\, \subseteq\rangle \underset{\alpha^{\Sigma}_{\mathcal{I}}}{\overset{\gamma^{\Sigma}_{\mathcal{I}}}{\leftrightarrows}} \langle A^{\Sigma}_{\mathcal{I}},\, \sqsubseteq\rangle$ [4].

The soundness of abstract domains $\langle A^{\Sigma}_{\mathcal{I}},\, \sqsubseteq\rangle$, is defined, for all $\overline{P}, \overline{Q} \in A^{\Sigma}_{\mathcal{I}}$, as

$$\begin{array}{llll} (\overline{P} \sqsubseteq \overline{Q}) \Rightarrow (\gamma^{\Sigma}_{\mathcal{I}}(\overline{P}) \subseteq \gamma^{\Sigma}_{\mathcal{I}}(\overline{Q})) & \text{implication} & \gamma^{\Sigma}_{\mathcal{I}}(\top) = \left\{\langle I,\, \eta\rangle \,\middle|\, I \in \mathcal{I} \wedge \eta \in \mathcal{R}^{\Sigma}_{I}\right\} & \text{supremum} \\ \gamma^{\Sigma}_{\mathcal{I}}(\overline{P} \sqcup \overline{Q}) \supseteq (\gamma^{\Sigma}_{\mathcal{I}}(\overline{P}) \cup \gamma^{\Sigma}_{\mathcal{I}}(\overline{Q})) & \text{join} & \ldots & \end{array}$$

[2] f is increasing (or monotone) if $x \preceq y$ implies $f(x) \sqsubseteq f(y)$, written $f \in \langle P,\, \preceq\rangle \xrightarrow{\nearrow} \langle Q,\, \sqsubseteq\rangle$.

The concrete least fixpoint semantics $C^{\Sigma}_{\mathcal{I}}[\![\mathtt{P}]\!]$ of a program $\mathtt{P}$ in Sect. 2.3 may have no correspondent in the abstract e.g. because the abstract domain $\langle A^{\Sigma}_{\mathcal{I}}, \sqsubseteq\rangle$ is not a cpo so that the abstract transformer has no least fixpoint, even maybe no fixpoint. In that case, we can define the abstract semantics $\overline{C}^{\Sigma}_{\mathcal{I}}[\![\mathtt{P}]\!] \in \wp(A^{\Sigma}_{\mathcal{I}})$ as the set of abstract inductive invariants for an abstract transformer $\overline{F}^{\Sigma}_{\mathcal{I}}[\![\mathtt{P}]\!] \in A^{\Sigma}_{\mathcal{I}} \to A^{\Sigma}_{\mathcal{I}}$ of $\mathtt{P}$.

$$\overline{C}^{\Sigma}_{\mathcal{I}}[\![\mathtt{P}]\!] \triangleq \left\{\overline{P} \in A^{\Sigma}_{\mathcal{I}} \;\middle|\; \overline{F}^{\Sigma}_{\mathcal{I}}[\![\mathtt{P}]\!](\overline{P}) \sqsubseteq \overline{P}\right\} \quad \text{postfixpoint semantics.}$$

In practice, only one abstract postfixpoint needs to be computed (while the abstract semantics defines all possible ones). Such an abstract postfixpoint can be computed e.g. by elimination or iteratively from the infimum using widenings and narrowings [3].

In the concrete semantics the least fixpoint is, by Tarski's theorem, an equivalent representation of the set of concrete postfixpoints.

2.5 Soundness and Completeness of Abstract Semantics

The abstract semantics $\overline{C}[\![\mathtt{P}]\!] \in A$ is *sound* with respect to a concrete semantics $C[\![\mathtt{P}]\!]$ of a program $\mathtt{P}$ for concretization γ whenever $\forall \overline{P} \in A : (\exists\, \overline{C} \in \overline{C}[\![\mathtt{P}]\!] : \overline{C} \sqsubseteq \overline{P}) \Rightarrow (C[\![\mathtt{P}]\!] \subseteq \gamma(\overline{P}))$. It is *complete* whenever $\forall \overline{P} \in A : (C[\![\mathtt{P}]\!] \subseteq \gamma(\overline{P})) \Rightarrow (\exists\, \overline{C} \in \overline{C}[\![\mathtt{P}]\!] : \overline{C} \sqsubseteq \overline{P})$. When the concrete semantics is defined in fixpoint form $C[\![\mathtt{P}]\!] \triangleq \mathbf{lfp}^{\subseteq} F[\![\mathtt{P}]\!]$ and the abstract semantics in postfixpoints, the soundness of the abstract semantics follows from the soundness conditions of the abstraction in Sect. 2.4 and the soundness of the abstract transformer $\forall \overline{P} \in A : F[\![\mathtt{P}]\!] \circ \gamma(\overline{P}) \subseteq \gamma \circ \overline{F}[\![\mathtt{P}]\!](\overline{P})$ [3, 4]. If the concrete semantics is also defined in postfixpoint form, then the soundness condition becomes

$$\forall \overline{P} \in A : (\exists\, \overline{C} \in \overline{C}[\![\mathtt{P}]\!] : \overline{C} \sqsubseteq \overline{P}) \Rightarrow (\exists\, C \in C[\![\mathtt{P}]\!] : C \subseteq \gamma(\overline{P}))\,.$$

Moreover, the composition of sound abstractions is necessarily sound.

The soundness of $\overline{F}[\![\mathtt{P}]\!]$ can usually be proved by induction on the syntactical structure of the program $\mathtt{P}$ using local soundness conditions.

$$\begin{array}{ll}
\gamma(\bar{\mathsf{f}}[\![\mathtt{x} := t]\!]\overline{P}) \supseteq \mathsf{f}_{\mathcal{I}}[\![\mathtt{x} := t]\!]\gamma(\overline{P}) & \text{assignment postcondition} \\
\gamma(\bar{\mathsf{b}}[\![\mathtt{x} := t]\!]\overline{P}) \supseteq \mathsf{b}_{\mathcal{I}}[\![\mathtt{x} := t]\!]\gamma(\overline{P}) & \text{assignment precondition} \\
\gamma(\bar{\mathsf{p}}[\![\varphi]\!]\overline{P}) \supseteq \mathsf{p}_{\mathcal{I}}[\![\varphi]\!]\gamma(\overline{P}) & \text{test.}
\end{array}$$

2.6 Abstractions between Multi-interpretations

The natural ordering to express abstraction (or precision) on multi-interpreted semantics is the subset ordering, which gives a complete lattice structure to the set of multi-interpreted properties: a property P_2 is more abstract than P_1 when $P_1 \subset P_2$, meaning that P_2 allows more behaviors for some interpretations, and maybe that it allows new interpretations. Following that ordering $\langle \mathfrak{P}^{\Sigma}_{\mathcal{I}}, \subseteq\rangle$, we can express systematic abstractions of the multi-interpreted semantics.

If we can only compute properties on the standard interpretation $\mathfrak{I}$ then we can approximate a multi-interpreted program saying that we know the possible behaviors when the interpretation is $\mathfrak{I}$ and we know nothing (so all properties are possible) for the other interpretations of the program. On the other hand, if we analyze a program that can only have one possible interpretation with a multi-interpreted property, then we are doing an abstraction in the sense that we add more behaviors and forget the actual property that should be associated with the program by the standard semantics. So, in general, we have two sets of interpretations, one is $\boldsymbol{\mathcal{I}}$, the context of interpretations for the program and the other one is $\boldsymbol{\mathcal{I}}^\sharp$, the set of interpretations used in the analysis. The correspondance between the two is a Galois connection.

Lemma 1. $\langle \mathfrak{P}^\Sigma_{\mathcal{I}}, \subseteq\rangle \underset{\alpha^\Sigma_{\mathcal{I}\to\mathcal{I}^\sharp}}{\overset{\gamma^\Sigma_{\mathcal{I}^\sharp\to\mathcal{I}}}{\leftrightarrows}} \langle \mathfrak{P}^\Sigma_{\mathcal{I}^\sharp}, \subseteq\rangle$ *with* $\gamma^\Sigma_{\mathcal{I}^\sharp\to\mathcal{I}}(P^\sharp) \triangleq \left\{\langle I, \eta\rangle \in \mathfrak{R}^\Sigma_{\mathcal{I}} \;\middle|\; I \in \mathcal{I}^\sharp \Rightarrow \langle I, \eta\rangle \in P^\sharp\right\}$ *and* $\alpha^\Sigma_{\mathcal{I}\to\mathcal{I}^\sharp}(P) \triangleq P \cap \mathfrak{R}^\Sigma_{\mathcal{I}^\sharp}$. □

Note that if the intersection of $\mathcal{I}^\sharp$ and $\mathcal{I}$ is empty then the abstraction is trivially $\emptyset$ for all properties, and if $\mathcal{I} \subseteq \mathcal{I}^\sharp$ then the abstraction is the identity.

Observe that $\mathsf{f}_{\mathcal{I}^\sharp}[\![\mathtt{x} := e]\!]$ and $\mathsf{f}_{\mathcal{I}}[\![\mathtt{x} := e]\!]$ have exactly the same definition. However, the corresponding fixpoint semantics do differ when $\mathcal{I}^\sharp \neq \mathcal{I}$ and $\mathcal{I} \not\subseteq \mathcal{I}^\sharp$ since $\langle \mathfrak{P}^\Sigma_{\mathcal{I}^\sharp}, \subseteq\rangle \neq \langle \mathfrak{P}^\Sigma_{\mathcal{I}}, \subseteq\rangle$. We have soundness.

Lemma 2. $f_{\mathcal{I}}[\![\mathtt{x} := e]\!] \circ \gamma^\Sigma_{\mathcal{I}^\sharp\to\mathcal{I}}(P^\sharp) = \gamma^\Sigma_{\mathcal{I}^\sharp\to\mathcal{I}} \circ \mathsf{f}_{\mathcal{I}^\sharp}[\![\mathtt{x} := e]\!](P^\sharp)$, *and similarly for the other transformers.* □

2.7 Theories and Models

The set $\mathtt{x}_\Psi$ of *free variables* of a formula $\Psi \in \mathbb{F}(\Sigma)$ is defined inductively as the set of variables in the formula which are not in the scope of an existential quantifier. A *sentence* of $\mathbb{F}(\Sigma)$ is a formula with no free variable, $\mathbb{S}(\Sigma) \triangleq \{\Psi \in \mathbb{F}(\Sigma) \mid \mathtt{x}_\Psi = \emptyset\}$. A *theory* $\mathcal{T} \in \wp(\mathbb{S}(\Sigma))$ is a set of sentences (called the *theorems* of the theory). The set of predicate and function symbols that appear in at least one sentence of a theory $\mathcal{T}$ should be contained in the *signature* $\mathbb{S}(\mathcal{T}) \dot{\subseteq} \Sigma$ of theory $\mathcal{T}$.

The idea of theories is to restrict the possible meanings of functions and predicates in order to reason under these hypotheses. The meanings which are allowed are the meanings which make the sentences of the theory true.

An interpretation $I \in \mathfrak{I}(\Sigma)$ is said to be a *model* of $\Psi \in \mathbb{F}(\Sigma)$ when $\exists \eta : I \models_\eta \Psi$ (i.e. I makes Ψ true). An interpretation is a *model* of a theory $\mathcal{T}$ if and only if it is a model of all the theorems of the theory (i.e. makes true all theorems of the theory). The class of all models of a theory $\mathcal{T}$ is

$$\mathfrak{M}(\mathcal{T}) \triangleq \{I \in \mathfrak{I}(\mathbb{S}(\mathcal{T})) \mid \forall \Psi \in \mathcal{T} : \exists \eta : I \models_\eta \Psi\} = \{I \in \mathfrak{I}(\mathbb{S}(\mathcal{T})) \mid \forall \Psi \in \mathcal{T} : \forall \eta : I \models_\eta \Psi\}$$

since if Ψ is a sentence and if there is an I and an η such that $I \models_\eta \Psi$, then for all η', $I \models_{\eta'} \Psi$.

Quite often, the set of sentences of a theory is not defined by extension, but using a (generally finite or enumerable) set of axioms which generates the set of theorems of the theory by implication. A theory is said to be *deductive* if and only if it is closed by deduction, that is all the theorems that are true on all models of the theory are in the theory.

This notion of models gives a natural way of approximating sets of interpretations by a theory: a set of interpretations $\mathcal{I}$ can be approximated by any theory $\mathcal{T}$ such that

$\mathcal{I} \subseteq \mathfrak{M}(\mathcal{T})$. Notice, though, that because the lattice of sentences of a theory is not complete, there is no best abstraction in general[3].

2.8 Logical Abstract Domains

Given a theory $\mathcal{T}$ over Σ, a *logical abstract domain* is an abstract domain $\langle A^{\Sigma}_{\mathcal{T}}, \sqsubseteq, \top, \sqcup, \ldots, \bar{\mathsf{f}}, \bar{\mathsf{p}}, \ldots\rangle$ such that $A^{\Sigma}_{\mathcal{T}} \subseteq \mathbb{F}(\Sigma)$, $\sqsubseteq \triangleq \Rightarrow$, $\top \triangleq \mathtt{tt}$, $\sqcup \triangleq \vee$, etc, and the concretization is $\gamma^{\Sigma}_{\mathcal{T}}(\Psi) \triangleq \left\{\langle I, \eta\rangle \,\middle|\, I \in \mathfrak{M}(\mathcal{T}) \text{ and } I \models_{\eta} \Psi\right\}$. Note that a logical abstract domain is a special case of algebraic abstract domain over a multi-interpretation.

Remark that there might be no finite formula in the language $\mathbb{F}(\Sigma)$ of the theory $\mathcal{T}$ to encode a best abstraction in which case there is no Galois connection. In any case soundness can be formalized by a concretization function as in Sect. 2.4. Moreover, in presence of infinite ascending chains of finite first-order formulæ (e.g. $(\mathtt{x} = 0) \Rightarrow (\mathtt{x} = 0 \vee \mathtt{x} = 1) \Rightarrow \ldots \Rightarrow \bigvee_{i=1}^{n} \mathtt{x} = i \Rightarrow \ldots$) and descending chains of finite formulæ (e.g. $(\mathtt{x} \neq -1) \Leftarrow (\mathtt{x} \neq -1 \wedge \mathtt{x} \neq -2) \Leftarrow \ldots \Leftarrow \bigwedge_{i=1}^{n} \mathtt{x} \neq -i \Leftarrow \ldots$) with no finite first-order formula to express their limits, the fixpoint may not exist. Hence the fixpoint semantics in the style of Sect. 2.3 is not well-defined in the abstract. However, following Sect. 2.4, we can define the abstract semantics as the set of abstract inductive invariants for an increasing abstract transformer of program P.

3 Observational Semantics

Besides values of program variables, the concrete semantics may also observe values of auxiliary variables or values of functions over program variables. Whereas such cases can be described in the general setting above (e.g. by inclusion of the auxiliary variables as program variables), it is more convenient to explicitly define the observables of the program semantics.

3.1 Observable Properties of Multi-interpreted Programs

The signature $\Sigma = \langle \mathbb{x}, \mathbb{f}, \mathbb{p}\rangle$ of multiple interpretations $\mathcal{I} \in \wp(\mathfrak{I}(\Sigma))$ is decomposed into a program signature $\Sigma_{\mathrm{P}} = \langle \mathbb{x}_{\mathrm{P}}, \mathbb{f}, \mathbb{p}\rangle$ over program variables $\mathtt{x} \in \mathbb{x}_{\mathrm{P}} \subseteq \mathbb{x}$ and an observable signature $\Sigma_{O} = \langle \mathbb{x}_{O}, \mathbb{f}, \mathbb{p}\rangle$ over observable identifiers $x \in \mathbb{x}_{O} \subseteq \mathbb{x}$. So we now have

program variables	observable variables	
$\eta \in \mathcal{R}^{\Sigma_{\mathrm{P}}}_{I} \triangleq \mathbb{x}_{\mathrm{P}} \to I_{\mathcal{V}}$	$\zeta \in \mathcal{R}^{\Sigma_{O}}_{I} \triangleq \mathbb{x}_{O} \to I_{\mathcal{V}}$	program environments
$\mathfrak{R}^{\Sigma_{\mathrm{P}}}_{\mathcal{I}} \triangleq \left\{\langle I, \eta\rangle \,\middle|\, I \in \mathcal{I} \wedge \eta \in \mathcal{R}^{\Sigma_{\mathrm{P}}}_{I}\right\}$	$\mathfrak{R}^{\Sigma_{O}}_{\mathcal{I}} \triangleq \left\{\langle I, \zeta\rangle \,\middle|\, I \in \mathcal{I} \wedge \zeta \in \mathcal{R}^{\Sigma_{O}}_{I}\right\}$	multi-interpreted environments
$\mathfrak{P}^{\Sigma_{\mathrm{P}}}_{\mathcal{I}} \triangleq \wp(\mathfrak{R}^{\Sigma_{\mathrm{P}}}_{\mathcal{I}})$	$\mathfrak{P}^{\Sigma_{O}}_{\mathcal{I}} \triangleq \wp(\mathfrak{R}^{\Sigma_{O}}_{\mathcal{I}})$	multi-interpreted properties

[3] If $\mathfrak{I}$ interprets programs over the natural numbers there is no enumerable first-order theory characterizing this interpretation (by Gödel first incompleteness theorem), so the poset has no best abstraction of $\{\mathfrak{I}\}$.

We name observables by identifiers (which, in particular, can be variable identifiers). Observables are functions from values of program variables to values $v \in I_{\mathcal{V}}$ (for interpretation $I \in \mathfrak{I}(\Sigma)$).

$$\begin{array}{llll} \omega_I & \in \mathcal{O}_I^{\Sigma_{\mathrm{P}}} \triangleq \mathcal{R}_I^{\Sigma_{\mathrm{P}}} \to I_{\mathcal{V}} & & \text{observables (for } I \in \mathcal{I}) \\ \Omega_I & \in \mathbb{x}_O \to \mathcal{O}_I^{\Sigma_{\mathrm{P}}} & & \text{observable naming.} \end{array}$$

Whereas a concrete *program semantics* is relative to $\mathfrak{P}_{\mathcal{I}}^{\Sigma_{\mathrm{P}}}$, the *observational semantics* is relative to $\mathfrak{P}_{\mathcal{I}}^{\Sigma_O}$ and both can be specified in fixpoint or in postfixpoint form.

Example 1 (Memory model). In the memory model of [14], a 32 bits unsigned/positive integer variable $\mathtt{x}$ can be encoded by its constituent bytes $\langle x_3, x_2, x_1, x_0\rangle$ so that, for little endianness, $\eta(\mathtt{x}) = \Omega_I(x_3)\eta \times 2^{24} + \Omega_I(x_2)\eta \times 2^{16} + \Omega_I(x_1)\eta \times 2^{8} + \Omega_I(x_0)\eta$. □

Given a program property $P \in \mathfrak{P}_{\mathcal{I}}^{\Sigma_{\mathrm{P}}}$, the corresponding observable property is

$$\alpha_{\mathcal{I}}^{\Omega}(P) \triangleq \left\{ \langle I,\ \lambda x \bullet \Omega_I(x)\eta\rangle \in \mathfrak{R}_{\mathcal{I}}^{\Sigma_O} \;\middle|\; \langle I,\ \eta\rangle \in P \right\} .$$

The value of the observable named x is therefore $\Omega_I(x)\eta$ where the values of program variables are given by η. Conversely, given an observable property $Q \in \mathfrak{P}_{\mathcal{I}}^{\Sigma_O}$, the corresponding program property is

$$\gamma_{\mathcal{I}}^{\Omega}(Q) \triangleq \left\{ \langle I,\ \eta\rangle \in \mathfrak{R}_{\mathcal{I}}^{\Sigma_{\mathrm{P}}} \;\middle|\; \langle I,\ \lambda x \bullet \Omega_I(x)\eta\rangle \in Q \right\} .$$

We have a Galois connection between the program and observable properties.

Theorem 1. $\langle \mathfrak{P}_{\mathcal{I}}^{\Sigma_{\mathrm{P}}},\ \subseteq\rangle \underset{\alpha_{\mathcal{I}}^{\Omega}}{\overset{\gamma_{\mathcal{I}}^{\Omega}}{\leftrightarrows}} \langle \mathfrak{P}_{\mathcal{I}}^{\Sigma_O},\ \subseteq\rangle$. □

3.2 Soundness of the Abstraction of Observable Properties

The *observational abstraction* will be of observable properties in $\mathfrak{P}_{\mathcal{I}}^{\Sigma_O}$ so with concretization $\gamma_{\mathcal{I}}^{\Sigma_O} \in A_{\mathcal{I}}^{\Sigma_O} \to \mathfrak{P}_{\mathcal{I}}^{\Sigma_O}$ where $A_{\mathcal{I}}^{\Sigma_O}$ is the abstract domain. The classical direct abstraction of program properties in $\mathfrak{P}_{\mathcal{I}}^{\Sigma_{\mathrm{P}}}$ will be the particular case where $\mathbb{x}_O = \mathbb{x}_{\mathrm{P}}$ and $\lambda \mathtt{x} \bullet \Omega_I(\mathtt{x})$ is the identity. The program properties corresponding to observable Ω_I are given by $\gamma_{\mathcal{I}}^{\Omega,\mathrm{P}} \in A_{\mathcal{I}}^{\Sigma_O} \mapsto \mathfrak{P}_{\mathcal{I}}^{\Sigma_{\mathrm{P}}}$ such that

$$\gamma_{\mathcal{I}}^{\Omega,\mathrm{P}} \triangleq \gamma_{\mathcal{I}}^{\Omega} \circ \gamma_{\mathcal{I}}^{\Sigma_O} = \lambda \overline{P} \bullet \left\{ \langle I,\ \eta\rangle \in \mathfrak{R}_{\mathcal{I}}^{\Sigma_{\mathrm{P}}} \;\middle|\; \langle I,\ \lambda x \bullet \Omega_I(x)\eta\rangle \in \gamma_{\mathcal{I}}^{\Sigma_O}(\overline{P}) \right\}.$$

Under the observational semantics, soundness conditions remain unchanged, but they must be proved with respect to $\gamma_{\mathcal{I}}^{\Omega,\mathrm{P}}$, not $\gamma_{\mathcal{I}}^{\Sigma_O}$. So the soundness conditions on transformers become slightly different. For example the soundness condition on the assignment abstract postcondition $\bar{\mathsf{f}}[\![\mathtt{x} := e]\!]$ becomes:

Lemma 3. $\gamma_{\mathcal{I}}^{\Omega,\mathrm{P}}(\bar{f}[\![\mathtt{x}\ :=\ e]\!]\overline{P}) \supseteq f_{\mathcal{I}}[\![\mathtt{x}\ :=\ e]\!](\gamma_{\mathcal{I}}^{\Omega,\mathrm{P}}(\overline{P}))$ *and similarly for the other transformers.*

3.3 Observational Extension

It can sometimes be useful to extend an abstract property $\overline{P}$ for observables Ω with a new observable ω named x. For example, this was useful for intervals in [6]. We will write $\mathsf{extend}_{(x,\omega)}\left(\overline{P}\right)$ for the extension of $\overline{P}$ with observable ω for observable identifier x.

Example 2. Let $A_{\mathbb{x}_O}$ be the abstract domain mapping observable identifiers $\mathtt{x} \in \mathbb{x}_O$ to an interval of values [3]. Assume that intervals of program variables are observable, that is $\mathbb{x}_\mathrm{P} \subseteq \mathbb{x}_O$ and let $\mathtt{x} \in \mathbb{x}_\mathrm{P}$ be a program variables for which we want to observe the square $\mathtt{x}^2$ so $\omega_I \triangleq [\![\mathtt{x}^2]\!]_I$. Let $\mathtt{x2} \notin \mathbb{x}_O$ be a fresh name for this observable. This extension of observable properties with a new observable $\mathsf{extend}_{(\mathtt{x2},[\![\mathtt{x}^2]\!])} \in A_{\mathbb{x}_O} \to A_{\mathbb{x}_O \cup \{\mathtt{x2}\}}$ can be defined as

$$\mathsf{extend}_{(\mathtt{x2},[\![\mathtt{x}^2]\!])}\left(\overline{P}\right) \triangleq \boldsymbol{\lambda}\, x \in \mathbb{x}_O \cup \{\mathtt{x2}\} \bullet (\, x \neq \mathtt{x2}\ \boldsymbol{?}\ \overline{P}(x) : \overline{P}(\mathtt{x}) \otimes \overline{P}(\mathtt{x})\,)$$

(where $\otimes$ is the product of intervals) is sound. □

The extension operation is assumed to be defined so that its semantics satisfies the following soundness condition

$$\gamma_{\mathcal{I}}^{\lambda I \bullet\ \lambda y \bullet y = x\,?\,\omega_I\ :\ \Omega_I(y),\,\mathrm{P}}\left(\mathsf{extend}_{(x,\omega)}\left(\overline{P}\right)\right) \supseteq \gamma_{\mathcal{I}}^{\Omega,\mathrm{P}}\left(\overline{P}\right) .$$

The introduction of auxiliary variables to name alien terms in logical abstract domains is an observational extension of the domains.

Lemma 4. *For the logical abstract domain* $A \triangleq \mathbb{F}(\Sigma)$ *with* $\gamma_{\mathcal{I}}^{\Sigma_O}(\Psi) \triangleq \left\{\langle I, \eta\rangle \,\middle|\, I \in \mathcal{I} \wedge I \models_\eta \Psi\right\}$,

$$\mathsf{extend}_{(x,[\![e]\!])}(\Psi[x \leftarrow e]) \triangleq \exists x : (x = e \wedge \Psi) \quad \textit{is sound.}$$ □

This extension operation can also be used for vectors of fresh variables and vectors of observables in the natural way.

4 Iterated Reduction and Reduced Product

A reduction makes a property more precise in the abstract without changing its concrete meaning. By iterating this reduction, one can improve the precision of a static analysis without altering its soundness. A case of iterated reduction was proposed by [8] following [4].

Definition 1 (Reduction). *Let* $\langle A, \sqsubseteq\rangle$ *be a poset which is an abstract domain with concretization* $\gamma \in A \xrightarrow{\nearrow} C$ *where* $\langle C, \subseteq\rangle$ *is the concrete domain. A* reduction *is* $\rho \in A \to A$ *which is reductive that is* $\forall \overline{P} \in A : \rho(\overline{P}) \sqsubseteq \overline{P}$ *and sound in that* $\forall \overline{P} \in A : \gamma(\rho(\overline{P})) = \gamma(\overline{P})$. *The* iterates *of the reduction are* $\rho^0 \triangleq \boldsymbol{\lambda}\,\overline{P} \bullet \overline{P}$, $\rho^{\lambda+1} = \rho(\rho^\lambda)$ *for successor ordinals and* $\rho^\lambda = \bigsqcap_{\beta<\lambda} \rho^\beta$ *for limit ordinals. The iterates are* well-defined *when the greatest lower bounds* $\bigsqcap$ *(glb) do exist in the poset* $\langle A, \sqsubseteq\rangle$. □

Theorem 2 (Iterated reduction). *Given a sound reduction* ρ, *for all ordinals* λ, ρ^λ *is a sound reduction. If the iterates of* ρ *from* $\overline{P}$ *are well-defined then their limit* $\rho^*(\overline{P})$ *exists. We have* $\forall \beta < \lambda : \rho^*(\overline{P}) \sqsubseteq \rho^\lambda(\overline{P}) \sqsubseteq \rho^\beta(\overline{P}) \sqsubseteq \overline{P}$. *If* γ *is the upper adjoint of a Galois connection then* ρ^* *is a sound reduction. If* ρ *is increasing then* $\rho^* = \boldsymbol{\lambda}\,\overline{P} \bullet \mathbf{gfp}^{\sqsubseteq}_{\overline{P}}\rho$ *is the greatest fixpoint (gfp) of* ρ *less than or equal to* $\overline{P}$. □

The reduced product is defined as follows [4].

Definition 2 (Reduced product). *Let* $\langle A_i,\ \sqsubseteq_i\rangle$, $i \in \Delta$, Δ *finite, be abstract domains with increasing concretization* $\gamma_i \in A_i \to \mathfrak{P}^{\Sigma o}_{\boldsymbol{I}}$. *Their* Cartesian product *is* $\langle \boldsymbol{A},\ \sqsubseteq\rangle$ *where* $\boldsymbol{A} \triangleq \bigtimes_{i\in\Delta} A_i$, $(\boldsymbol{P} \sqsubseteq \boldsymbol{Q}) \triangleq \bigwedge_{i\in\Delta}(\boldsymbol{P}_i \sqsubseteq_i \boldsymbol{Q}_i)$ *and* $\boldsymbol{\gamma} \in \bigtimes_{i\in\Delta} A_i \to \mathfrak{P}^{\Sigma o}_{\boldsymbol{I}}$ *is* $\boldsymbol{\gamma}(\boldsymbol{P}) \triangleq \bigcap_{i\in\Delta} \gamma_i(\boldsymbol{P}_i)$. *In particular the* product $\langle A_i{\times}A_j, \sqsubseteq_{ij}\rangle$ *is such that* $\langle x, y\rangle \sqsubseteq_{ij} \langle x', y'\rangle \triangleq (x \sqsubseteq_i x') \wedge (y \sqsubseteq_j y')$ *and* $\gamma_{ij}(\langle x,\ y\rangle) \triangleq \gamma_i(x) \cap \gamma_j(y)$.

Their reduced product *is* $\langle (\bigtimes_{i\in\Delta} A_i) /_{\equiv},\ \sqsubseteq\rangle$ *where* $(\boldsymbol{P} \equiv \boldsymbol{Q}) \triangleq (\boldsymbol{\gamma}(\boldsymbol{P}) = \boldsymbol{\gamma}(\boldsymbol{Q}))$ *and* $\boldsymbol{\gamma}$ *as well as* $\sqsubseteq$ *are naturally extended to the equivalence classes* $[\boldsymbol{P}]/_{\equiv}$, $\boldsymbol{P} \in \boldsymbol{A}$, *of* $\equiv$. □

The simple cartesian product can be a representation for the reduced product, but if we just apply abstract transformers componentwise, then we obtain the same result as running analyses with each abstract domain independently. We can obtain much more precise results if we try to compute precise abstract values for each abstract domain, while staying in the same class of the reduced product. Computing such values is naturally a reduction.

Implementations of the most precise reduction (if it exists) can hardly be modular since in general adding a new abstract domain to increase precision implies that the reduced product must be completely redesigned. On the contrary, the pairwise iterated product reduction below, is more modular, in that the introduction of a new abstract domain only requires defining the reduction with the other existing abstract domains.

Definition 3 (Iterated pairwise reduction). *For* $i, j \in \Delta$, $i \neq j$, *let* $\rho_{ij} \in \langle A_i \times A_j,\ \sqsubseteq_{ij}\rangle \mapsto \langle A_i{\times}A_j, \sqsubseteq_{ij}\rangle$ *be pairwise reductions (so that* $\forall \langle x, y\rangle \in A_i{\times}A_j : \rho_{ij}(\langle x, y\rangle) \sqsubseteq_{ij} \langle x,\ y\rangle$*, preferably lower closure operators i.e. reductive, increasing and idempotent). Define the pairwise reductions* $\boldsymbol{\rho}_{ij} \in \langle \boldsymbol{A},\ \sqsubseteq\rangle \mapsto \langle \boldsymbol{A},\ \sqsubseteq\rangle$ *of the Cartesian product as*

$$\boldsymbol{\rho}_{ij}(\boldsymbol{P}) \triangleq \mathit{let}\ \langle \boldsymbol{P}'_i,\ \boldsymbol{P}'_j\rangle \triangleq \rho_{ij}(\langle \boldsymbol{P}_i,\ \boldsymbol{P}_j\rangle)\ \mathit{in}\ \boldsymbol{P}[i \leftarrow \boldsymbol{P}'_i][j \leftarrow \boldsymbol{P}'_j]$$

where $\boldsymbol{P}[i \leftarrow x]_i = x$ *and* $\boldsymbol{P}[i \leftarrow x]_j = \boldsymbol{P}_j$ *when* $i \neq j$. *Define the iterated pairwise reductions* $\boldsymbol{\rho}^n$, $\boldsymbol{\rho}^* \in \langle \boldsymbol{A},\ \sqsubseteq\rangle \mapsto \langle \boldsymbol{A},\ \sqsubseteq\rangle$, $n \geqslant 0$ *of the Cartesian product as in Def. 1 for*

$$\boldsymbol{\rho} \triangleq \bigcirc_{\substack{i,j\,\in\,\Delta,\\ i\neq j}} \boldsymbol{\rho}_{ij} \tag{1}$$

where $\bigcirc_{i=1}^{n} f_i \triangleq f_{\pi_1} \circ \ldots \circ f_{\pi_n}$ *is the function composition for some arbitrary permutation* π *of* $[1, n]$. □

The pairwise reductions $\boldsymbol{\rho}_{ij}$ and the iterated ones $\boldsymbol{\rho}^n$, $n \geqslant 0$ as well as their closure $\boldsymbol{\rho}^\star$, if any, are sound over-approximations of the reduced product in that

Theorem 3. *Under the hypotheses of Def. 1 and assuming the limit of the iterated reductions is well defined, the reductions are such that* $\forall \boldsymbol{P} \in \boldsymbol{A} : \forall \lambda : \boldsymbol{\rho}^\star(\boldsymbol{P}) \sqsubseteq \boldsymbol{\rho}^\lambda(\boldsymbol{P}) \sqsubseteq \boldsymbol{\rho}_{ij}(\boldsymbol{P}) \sqsubseteq \boldsymbol{P}$, $i, j \in \Delta$, $i \neq j$ *and sound since* $\boldsymbol{\rho}^\lambda(\boldsymbol{P}), \boldsymbol{\rho}_{ij}(\boldsymbol{P}), \boldsymbol{P} \in [\boldsymbol{P}]/_{\equiv}$ *and if* γ *preserves lower bounds then* $\boldsymbol{\rho}^\star(\boldsymbol{P}) \in [\boldsymbol{P}]/_{\equiv}$. □

The following theorem proves that the iterated reduction may not be as precise as the reduced product, a fact underestimated in the literature. It is nevertheless easier to implement.

Theorem 4. *In general* $\boldsymbol{\rho}^\star(\boldsymbol{P})$ *may not be a minimal element of the reduced product class* $[\boldsymbol{P}]/_{\equiv}$ *(in which case* $\exists \boldsymbol{Q} \in [\boldsymbol{P}]/_{\equiv} : \boldsymbol{Q} \sqsubset \boldsymbol{\rho}^\star(\boldsymbol{P})$ *).* □

Sufficient conditions exist for the iterated pairwise reduction to be a total reduction to the reduced product.

Theorem 5. *If the $\langle A_i, \sqsubseteq_i, \sqcup_i \rangle$, $i \in \varDelta$ are complete lattices, the ρ_{ij}, $i, j \in \varDelta$, $i \neq j$, are lower closure operators, and $\forall \boldsymbol{P}, \boldsymbol{Q} : \big(\boldsymbol{\gamma}(\boldsymbol{P}) \subseteq \boldsymbol{\gamma}(\boldsymbol{Q})\big) \Rightarrow \big(\exists n \geqslant 0 : \big(\dot{\bigsqcap}_{\substack{i,j \in \varDelta,\\ i \neq j}} \boldsymbol{\rho}_{ij}\big)^n (\boldsymbol{P}) \sqsubseteq \boldsymbol{Q}\big)$ then $\forall \boldsymbol{P} : \boldsymbol{\rho}^\star(\boldsymbol{P})$ is the minimum of the class $\boldsymbol{P}/_\equiv$.* □

4.1 Observational Reduced Product

The observational reduced product of abstract domains $\langle A_i, \sqsubseteq_i \rangle$, $i \in \varDelta$ consists in introducing observables to increase the precision of the Cartesian product. We will write ${}^{\varOmega}\bigtimes_{i \in \varDelta} A_i$ for the *observational Cartesian product* with observables named by $\varOmega$. It can be seen as the application of the extension operator of Sect. 3 followed by a Cartesian product $\bigtimes_{i \in \varDelta} A_i$. This operation is not very fruitful, as the shared observables will not bring much information. But used in conjunction with an iterated reduction, it can give very precise results since information about the observables can bring additional reductions.

Definition 4 (Observational reduced product). *For all $i \in \varDelta$, let $\langle {}^iA_{\mathcal{I}}^{\Sigma_{\mathbb{O}}}, {}^i\sqsubseteq \rangle$, $\langle {}^iA_{\mathcal{I}}^{\Sigma_{\mathbb{O}'}}, {}^i\sqsubseteq' \rangle$ be abstract domains, $\varOmega'$ be the new observables, and ${}^i\mathsf{extend}_{\varOmega'} \in {}^iA_{\mathcal{I}}^{\Sigma_{\mathbb{O}}} \to {}^iA_{\mathcal{I}}^{\Sigma_{\mathbb{O}'}}$ be sound extensions in the sense that ${}^i\boldsymbol{\gamma}_{\mathcal{I}}^{\varOmega',\mathrm{P}}\left({}^i\mathsf{extend}_{\varOmega'}\left(\overline{P}\right)\right) \supseteq {}^i\boldsymbol{\gamma}_{\mathcal{I}}^{\varOmega,\mathrm{P}}\left(\overline{P}\right)$.*

The observational cartesian product *is* ${}^{\varOmega'}\bigtimes_{i \in \varDelta} {}^iA_{\mathcal{I}}^{\Sigma_{\mathbb{O}}} \triangleq \bigtimes_{i \in \varDelta} {}^i\mathsf{extend}_{\varOmega'}\left({}^iA_{\mathcal{I}}^{\Sigma_{\mathbb{O}}}\right)$ *and the* observational reduced product *is* $\langle \left({}^{\varOmega}\bigtimes_{i \in \varDelta} A_i\right)/_\equiv, \sqsubseteq \rangle$. □

5 The Nelson-Oppen Combination Procedure

The Nelson-Oppen procedure which decides satisfiability in a combination of logical theories by exchanging equalities and disequalities is shown to consist in computing a reduced product after the state is enhanced with some new "observations" corresponding to alien terms.

5.1 Formula Purification

Formula purification in the Nelson-Oppen theory combination procedure. Given disjoint deductive theories $\mathcal{T}_i$ in $\mathbb{F}(\Sigma_i)$, $\Sigma_i \dot{\subseteq} \Sigma$ with equality and decision procedures sat_i for satisfiability of quantifier-free conjunctive formulæ $\varphi_i \in \mathbb{C}(\Sigma_i)$, $i = 1, \ldots, n$, the Nelson-Oppen combination procedure [15] decides the satisfiability of a quantifier-free conjunctive formula $\varphi \in \mathbb{C}(\bigcup_{i=1}^n \Sigma_i)$ in theory $\mathcal{T} = \bigcup_{i=1}^n \mathcal{T}_i$ such that $\mathfrak{M}(\mathcal{T}) = \bigcap_{i=1}^n \mathfrak{M}(\mathcal{T}_i)$.

The first "purification" phase [18, Sect. 2] of the Nelson-Oppen combination procedure consists in repeating the replacement of (all occurrences of) an alien subterm $t \in \mathbb{T}(\Sigma_i) \setminus \mathbb{x}$ of a subformula $\psi[t] \notin \mathbb{C}(\Sigma_i)$ (including equality or inequality predicates $\psi[t] = (t = t')$ or $(t' = t)$) of φ by a fresh variable $x \in \mathbb{x}$ and introducing the equation

$x = t$ (i.e. $\varphi[\psi[t]]$ is replaced by $\varphi[\psi[x]] \wedge x = t$ and the replacement is recursively applied to $\varphi[\psi[x]]$ and $x = t$). Upon termination, the quantifier-free conjunctive formula φ is transformed into a formula φ' of the form

$$\varphi' = \exists\, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n : \bigwedge_{i=1}^{n} \varphi_i \quad \text{where} \quad \varphi_i = \varphi'_i \wedge \bigwedge_{x_i \in \boldsymbol{x}_i} x_i = t_{x_i},$$

$\boldsymbol{x} \triangleq \bigcup_{i=1}^{n} \boldsymbol{x}_i$ is the set of auxiliary variables $x_i \in \boldsymbol{x}_i$ introduced by the purification, each $t_{x_i} \in \mathbb{T}(\Sigma_i)$ is an alien subterm of φ renamed as $x_i \in \mathbb{x}$ and each φ'_i (hence each φ_i) is a quantifier-free conjunctive formula in $\mathbb{C}(\Sigma^i_O)$. We have $\varphi \Leftrightarrow \bigwedge_{i=1}^{n} \varphi'_i[x_i \leftarrow t_{x_i}]_{x_i \in \boldsymbol{x}_i}$ so φ and φ' are equisatisfiable.

Example 3 (Formula purification). Assume $f \in \mathbb{f}_1$ and $g \in \mathbb{f}_2$. $\varphi = (g(\mathtt{x}) = f(g(g(\mathtt{x}))))$ $\to (\exists\, y : y = f(g(y)) \wedge y = g(\mathtt{x})) \to (\exists\, y : \exists\, z : y = f(z) \wedge y = g(\mathtt{x}) \wedge z = g(y)) \to$ $(\exists\, y : \exists\, z : \varphi_1 \wedge \varphi_2) = \varphi'$ where $\varphi_1 = (y = f(z))$ and $\varphi_2 = (y = g(\mathtt{x}) \wedge z = g(y))$. □

In case of non-disjoint theories $\mathcal{T}_i$, $i = 1, ..., n$, purification is still possible, by considering the worst case (so as to purify any subterm of theories $\mathcal{T}_i$ or $\mathcal{T}_j$ occurring in a term of theories $\mathcal{T}_i$ or $\mathcal{T}_j$). The reason the Nelson-Oppen purification requires disjointness of theory signatures is that otherwise they can share more than equalities and cardinality, a sufficient reason for the procedure to be incomplete. Nevertheless, the purification procedure remains sound for non-disjoint theories, which can be exploited for static analysis, as shown below.

The Nelson-Oppen purification as an observational cartesian product. Let the observable identifiers be the free variables of $\varphi \in \mathbb{C}(\Sigma)$, $\mathbb{x}_{\mathrm{P}} = \mathtt{x}_\varphi$ plus the fresh auxiliary variables $\boldsymbol{x}$ introduced by the purification $\mathbb{x}_O = \mathbb{x}_{\mathrm{P}} \cup \boldsymbol{x}$. Let Σ_{P} and Σ_O be the corresponding signatures of Σ. Given an interpretation $I \in \mathcal{I}$, with values $I_{\mathcal{V}}$, the observable naming $\Omega^{\varphi}_I \in \mathbb{x}_O \to \mathcal{R}^{\Sigma_{\mathrm{P}}}_I \to I_{\mathcal{V}}$ is such that

$$\begin{aligned} \Omega^{\varphi}_I(x)\eta &\triangleq \eta(x) \quad \text{when} \quad x \in \mathbb{x}_{\mathrm{P}}, \\ &\triangleq [\![t_x]\!]\eta \quad \text{when} \quad x \in \boldsymbol{x} \quad . \end{aligned}$$

From a model-theoretic point of view, the purification of $\varphi \in A$ into $\langle \varphi_1, \ldots, \varphi_n \rangle$ can be considered as an abstraction of the program properties in $\mathfrak{P}^{\Sigma_O}_{\boldsymbol{I}}$ abstracted by φ to observable properties in $\mathfrak{R}^{\Sigma_O}_{\boldsymbol{I}}$ themselves abstracted to the observational cartesian product ${}^{\Omega^\varphi}\bigtimes_{i \in \Delta} {}^{i}A^{\Sigma_O}_{\boldsymbol{I}}$ where the component abstract domains are $\langle {}^{i}A^{\Sigma_O}_{\boldsymbol{I}}, \sqsubseteq_i \rangle \triangleq \langle \mathbb{C}(\Sigma^i_O),$ $\Rightarrow \rangle$ with concretizations ${}^{i}\gamma^{\Sigma_O}_{\boldsymbol{I}} \in \mathbb{C}(\Sigma^i_O) \to {}^{i}\mathfrak{P}^{\Sigma_O}_{\boldsymbol{I}}$ and ${}^{i}\gamma^{\Sigma_O}_{\boldsymbol{I}}(\varphi) \triangleq \left\{ \langle I, \eta \rangle \in \mathfrak{R}^{\Sigma_O}_{\boldsymbol{I}} \,\middle|\, I \in \mathfrak{M}(\mathcal{T}_i) \wedge I \models_\eta \varphi \right\}$, $i = 1, \ldots, n$. This follows from the fact that the concretization is the same.

Theorem 6. $\gamma^{\mathrm{P}}_{\boldsymbol{I}}(\varphi') = \boldsymbol{\gamma}^{\Omega^\varphi, \mathrm{P}}_{\boldsymbol{I}}\left({}^{\Omega^\varphi}\bigtimes_{i=1}^{n} \varphi'_i \right)$. □

After purification, the components of the observational cartesian product are not yet the most precise ones.

5.2 Formula Reduction

Formula reduction in the Nelson-Oppen theory combination procedure. After purification, the Nelson-Oppen combination procedure [15] includes a reduction phase where all variable equalities $x = y$ and inequalities $x \neq y$ deducible from one component φ_i in its theory $\mathcal{T}_i$ are propagated to all components φ_j.The decision procedure for $\mathcal{T}_i$ is used to determine all possible disjunctions of conjunctions of (in)equalities that are implied by φ_i. These are determined by exhaustively trying all possibilities in the nondeterministic version of the procedure or by an incremental construction in the deterministic version, which is more efficient for convex theories [18]. The reduction is iterated until no new disjunction of (in)equalities is found.

The Nelson-Oppen reduction as an iterated fixpoint reduction of the product. Let $\mathfrak{E}(S)$ be the set of all equivalence relations on S. Define the pairwise reduction $\rho_{ij}(\varphi_i, \varphi_j) \triangleq \langle \varphi_i \wedge E_{ij} \wedge E_{ji},\ \varphi_j \wedge E_{ji} \wedge E_{ij} \rangle$ where

$$\mathfrak{eq}(E) \triangleq \bigvee_{\equiv \in E} \left(\bigwedge_{x \equiv y} x = y \wedge \bigwedge_{x \not\equiv y} x \neq y \right) \text{and } E_{ij} \triangleq \bigwedge \left\{ \mathfrak{eq}(E) \;\middle|\; E \subseteq \mathfrak{E}(\mathbf{x}_{\varphi_i} \cap \mathbf{x}_{\varphi_j}) \wedge \varphi_i \Rightarrow \mathfrak{eq}(E) \right\} .$$

The Nelson-Oppen reduction of φ purified into ${}^{\Omega^\varphi}\bigtimes_{i=1}^{n} \varphi'_i$ consists in computing the iterated pairwise reduction $\boldsymbol{\rho}^* \left({}^{\Omega^\varphi}\bigtimes_{i=1}^{n} \varphi'_i \right)$.

Example 4. Let $\varphi_1 \triangleq (x = \mathtt{a} \vee x = \mathtt{b}) \wedge y = \mathtt{a} \wedge z = \mathtt{b}$ and $\varphi_2 \triangleq \mathtt{f}(x) \neq \mathtt{f}(y) \wedge \mathtt{f}(x) \neq \mathtt{f}(z)$ so that $\varphi \triangleq \varphi_1 \wedge \varphi_2$ is purified. We have $E_{12} \triangleq (x = y) \vee (x = z)$ and $E_{21} \triangleq (x \neq y) \wedge (x \neq z)$ so that $\boldsymbol{\rho}^* (\varphi) = \mathit{ff}$. □

Example 5. A classical example showing that the Nelson-Oppen reduction may not be as precise as the reduced product is given by [18, p. 11] where $\varphi_1 \triangleq \mathtt{f}(x) \neq \mathtt{f}(y)$ in the theory of Booleans admitting models of cardinality at most 2 and $\varphi_2 \triangleq \mathtt{g}(x) \neq \mathtt{g}(z) \wedge \mathtt{g}(y) \neq \mathtt{g}(z)$ in a disjoint theory admitting models of any cardinality so that $\varphi = \varphi_1 \wedge \varphi_2$ is purified. The reduction yields $\varphi \wedge x \neq y \wedge x \neq z \wedge y \wedge z$ and not ff since the cardinality information is not propagated whereas it would be propagated by the reduced product which is defined at the interpretation level. Therefore the pairwise reduction ought to be refined to include cardinality information, as proposed by [20]. □

Formula reduction and the reduced product. A formula over a set of theories is equivalent to its purification, so that to find an invariant or to check that a formula is invariant, we could first purify it and then proceed with the computation of the transformer of the program. This would lead to the same result as simply using one mixed formula if the reduction is total at each step of the computation. Such a process would be unnecessarily expensive if decision procedures could handle arbitrary formulæ. But this is not the case actually: most of the time, they cannot deal with quantifiers, and assignments introduce existential quantifiers which have to be approximated. Such approximations have to be redesigned for each set of formulæ. Using a reduced product of formulæ on base theories allows reusing the approximations on each theory (as in [12], even if the authors didn't recognize the reduced product). In that way, a reduced product of logical abstract domains will provide a modular approach to invariant proofs.

5.3 Formula Satisfiability

After purification and reduction, the Nelson-Oppen combination procedure [15] includes a decision phase to decide satisfiability of the formula by testing the satisfiability of its purified components. This phase can also be performed during the program static analysis since an unsatisfiability result means unreachability encoded by $\mathbb{ff}$. The satisfiability decision can also be used as an approximation to check for a postfixpoint and that the specification is satisfied.

For briefness, we have concentrated in this paper on the Nelson-Oppen combination procedure [15] but Shostak combination procedure [17] can be handled in exactly the same way. The idea of iterated reduction also applies to theorem proving [13].

6 Reduced Product of Logical and Algebraic Abstract Domains

6.1 Combining Logical and Algebraic Abstract Domains

Static analyzers such as Astrée [1] and Clousot [7] are based on an iterated pairwise reduction of a product of abstract domains over-approximating their reduced product. Since logical abstract domains as combined by the Nelson-Oppen combination procedure are indeed an iterated pairwise reduction of a product of abstract domains over-approximating their reduced product, as shown in Sect. 5.2, the design of abstract interpreters based on an approximation of the reduced product can use both logical and algebraic abstract domains.

An advantage of using a product of abstract domains with iterated reductions is that the reduction mechanism can be implemented once for all while the addition of a new abstract domain to improve precision essentially requires the addition of a reduction with the other existing abstract domains when necessary.

Notice that the Nelson-Oppen procedure and its followers aim at so-called ”soundness” and refutation completeness (for the reduction to $\mathbb{ff}$). In the theorem prover community, ”soundness” here means that if the procedure answers no, then the formula is not satisfiable. In program analysis we have a slightly different notion, where soundness means that whatever the answer, it is correct, and that would mean that if the procedure here answers yes, then the formula is satisfiable. This notion of soundness, when the only answers are yes it is satisfiable or no it is not, is equivalent to the old ”soundness” plus completeness. This is obtained by restricting the applicability of the procedure e.g. to stably-infinite theories [18] or other similar hypotheses on interpretations [20] to ensure that models of the various theories all have the same cardinalities, and additionaly by requiring that the theories are disjoint to avoid having to reduce on other properties than [dis]equality. In absence of such applicability restrictions, one can retain unsatisfiability if one component formula is unsatisfiability and abandon satisfiability if all component formula are satisfiable in favor of “unknown”, which yields reductions that are sound although potentially not optimal.

So the classical restrictions on the Nelson-Oppen procedure unnecessarily restrict its applicability to static analysis. Lifting them yields reductions that may not be optimal but preserves the soundness of the analyses which have to be imprecise anyway by undecidability. Hence, abandoning refutation completeness hypotheses, broaden the applicability of SMT solvers to static analysis. Many SMT solvers already contain lots of sound, but incomplete, heuristics hence no longer insist on refutational completeness.

Example 6. As a simple example, consider the combination of the logical domain of Presburger arithmetics (where the multiplication is inexpressible) and the domain of sign analysis (which is complete for multiplication). The abstraction of a first-order formula to a formula of Presburger arithmetics is by abstraction to a subsignature eliminating all terms of the signature not in the subsignature:

$$\begin{array}{rlll}
\alpha_\Sigma(\mathtt{x}) & \triangleq \mathtt{x} & & \\
\alpha_\Sigma(\mathtt{f}(t_1,\ldots,t_n)) & \triangleq\ ?, & \mathtt{f} \notin \Sigma \vee \exists i \in [1,n] : \alpha_\Sigma(t_i) = ? & \\
 & \triangleq \mathtt{f}(t_1,\ldots,t_n), & & \text{otherwise} \\
\alpha_\Sigma(\mathtt{ff}) & \triangleq \mathtt{ff} & & \\
\alpha_\Sigma(\mathsf{p}(t_1,\ldots,t_n)) & \triangleq \mathtt{tt}, & \mathsf{p} \notin \Sigma \vee \exists i \in [1,n] : \alpha_\Sigma(t_i) = ?, & \text{in positive position} \\
 & \triangleq \mathtt{ff}, & \mathsf{p} \notin \Sigma \vee \exists i \in [1,n] : \alpha_\Sigma(t_i) = ?, & \text{in negative position} \\
 & \triangleq \mathsf{p}(t_1,\ldots,t_n), & & \text{otherwise}
\end{array}$$

$$\alpha_\Sigma(\neg\Psi) \triangleq \neg\alpha_\Sigma(\Psi) \qquad \alpha_\Sigma(\Psi \wedge \Psi') \triangleq \alpha_\Sigma(\Psi) \wedge \alpha_\Sigma(\Psi')) \qquad \alpha_\Sigma(\exists\,\mathtt{x} : \Psi) \triangleq \exists\,\mathtt{x} : \alpha_\Sigma(\Psi)\,.$$

The abstract transformers for Presburger arithmetics become simply $\mathsf{f}_{\mathcal{P}}[\![\mathtt{x} := e]\!]P \triangleq \alpha_{\Sigma_{\mathcal{P}}}(\exists\, x' : P[\mathtt{x} \leftarrow x'] \wedge \mathtt{x} = e[\mathtt{x} \leftarrow x'])$, $\mathsf{p}_{\mathcal{P}}[\![\varphi]\!]P \triangleq \alpha_{\Sigma_{\mathcal{P}}}(P \wedge \varphi)$, etc, where $\Sigma_{\mathcal{P}}$ is the signature of Presburger arithmetics.

The reduction of the Presburger arithmetics logical abstract domain by the sign algebraic abstract domain is given by the concretization function for signs.

$$E_{ij}(\overline{\eta}) \triangleq \bigwedge_{\mathtt{x}\in\mathsf{dom}(\overline{\eta})} \gamma(\mathtt{x}, \overline{\eta}(\mathtt{x})) \quad \text{where} \quad \gamma(\mathtt{x}, \mathsf{pos0}) \triangleq (\mathtt{x} \geqslant 0), \quad \gamma(\mathtt{x}, \mathsf{pos}) \triangleq (\mathtt{x} > 0), \quad \text{etc.}$$

Assume the precondition $\langle P(\mathtt{x}),\ \mathtt{x} : \top\rangle$ holds, then after the assignment $\mathtt{x} := \mathtt{x} \times \mathtt{x}$, the post condition $\langle \exists\, x' : P(x') \wedge \mathtt{x} = x' \times x',\ \mathtt{x} : \mathsf{pos0}\rangle$ holds, which must be abstracted by $\alpha_{\Sigma_{\mathcal{P}}}$ to the Presburger arithmetics logical abstract domain that is $\langle \exists\, x' : P(x'), \mathtt{x} : \mathsf{pos0}\rangle$. The reduction reduces the postcondition to $\langle \exists\, x' : P(x') \wedge x \geq 0,\ \mathtt{x} : \mathsf{pos0}\rangle$.

Symmetrically, the sign abstract domain may benefit from equality information. For example, if the sign of x is unknown then it would remain unknown after the code `y := x; x := x * y` whereas knowing that x = y is enough to conclude that x is positive.

Of course the same result could be achieved by encoding by hand the Presburger arithmetics transformer for the assignment to cope with this case and other similar ones. Here the same result is achieved by the reduction without specific programming effort for each possible particular case. □

6.2 Program Purification

Whereas the reduced product proceeds componentwise, logical abstract domains often combine all these components into the single formula of their conjunction which is then globally propagated by property transformers before being purified again into components by the Nelson-Oppen procedure. These successive abstractions by purification and concretization by conjunction can be avoided when implementing the logical abstract domain as an iterated reduction of the product of the component and program purification, as defined below. The observational semantics is then naturally implemented by a program transformation.

Given disjoint signatures $\langle \mathbb{f}_i,\ \mathbb{p}_i\rangle$, $i = 1, ..., n$, the purification of a program P over $\mathbb{C}(\mathbb{x}, \bigcup_{i=1}^n \mathbb{f}_i, \bigcup_{i=1}^n \mathbb{p}_i)$ consists in purifying the terms t in its assignments $\mathtt{x} := e$ and the clauses in simple conjunctive normal form φ appearing in conditional or iteration tests. A term $t \in \mathbb{T}(\mathbb{x}, \bigcup_{i=1}^{n} \mathbb{f}_i)$ not reduced to a variable is said "*to have type i*" when it is of the form $\mathtt{c} \in \mathbb{f}_i^0$ or $\mathtt{f}(t_1, \ldots, t_n)$ with $\mathtt{f} \in \mathbb{f}_i^n$. As a side note, one may observe that this could very well be equivalent to using the variable and term types in a typed language.

The purification of an assignment $\mathtt{x} := e[t]$ where term e has type i and the alien subterm t has type j, $j \neq i$ consists in replacing this assignment by $x = t; \mathtt{x} := e[x]$ where $x \in \mathbb{x}$ is a fresh variable, $e[x]$ is obtained from $e[t]$ by replacing all occurrences of the alien subterm t by the fresh variable x in e, and in recursively applying the replacement to $x = t$ and $\mathtt{x} := e[x]$ until no alien subterm is left.

An atomic formula $a \in \mathbb{A}(\mathbb{x}, \bigcup_{i=1}^{n} \mathbb{f}_i, \bigcup_{i=1}^{n} \mathbb{p}_i)$ not reduced to $\mathtt{false}$ is said *to have type i* when it is of the form $\mathtt{p}(t_1, \ldots, t_n)$ with $\mathtt{p} \in \mathbb{p}_i^n$ or $t_1 = t_2$ and t_1 has type i or $\mathtt{x} = t_2$ and t_2 has type i. Then we can purify an assignment $\mathtt{x} := a[t]$ exactly in the same way as with terms. Finaly the purification of a clause in a test consists in replacing each atomic subformula a of the clause by a fresh variable and introducing assignments $x := a$ before the test and in recursively purifying the assignments $x := a$.

Example 7. Assume that $f \in \mathbb{f}_1$ and $g \in \mathbb{f}_2$. The purification is

$\mathtt{if}\ (g(\mathtt{w}) = f(g(g(\mathtt{w}))))\ \mathtt{then} \ldots$

→ $x := (g(\mathtt{w}) = f(g(g(\mathtt{w})))); \mathtt{if}\ x\ \mathtt{then} \ldots$

→ $y := g(\mathtt{w}); x := (y = f(g(y))); \mathtt{if}\ x\ \mathtt{then} \ldots$ $\wr g(\mathtt{w})$ has type 2 and $f(g(g(\mathtt{w})))$ has type $1 \int$

→ $y := g(\mathtt{w}); z := g(y); x := (y = f(z)); \mathtt{if}\ x\ \mathtt{then} \ldots$

$\wr (y = f(g(y)))$ has type 1 and $g(y)$ has type $2 \,. \int$ □

After purification all program terms and clauses are pure in that no term of a theory has a subterm in a different theory or a clause containing terms of different theories. So all term assignments $x := e$ (or atomic formulæ $x := a$) have $t \in \mathbb{T}(\Sigma_O^i)$ for some $i \in [1, n]$ and all clauses in tests are Boolean expressions written using only variables, ¬ and ∧.

We let the observable identifiers $\mathbb{x}_O = \mathbb{x}_P \cup \boldsymbol{x}$ be the program variables $\mathbb{x}_P$ plus the fresh auxiliary variables $x \in \boldsymbol{x}$ introduced by the purification with assignments $x := e_x$. Given an interpretation I, with values $I_{\mathcal{V}}$, the observable naming $\Omega_I \in \mathbb{x}_O \mapsto (\mathbb{x}_P \mapsto I_{\mathcal{V}}) \mapsto I_{\mathcal{V}}$ is

$$\begin{aligned} \Omega_I(x)\eta &\triangleq \eta(x) && \text{when} \quad x \in \mathbb{x}_P \\ &\triangleq [\![e_x]\!]\eta && \text{when} \quad x \in \boldsymbol{x} \,. \end{aligned}$$

This program transformation provides a simple implementation of the observational product of Def. 4. Moreover, the logical abstract domains no longer need to perform purification.

Theorem 7. *A static analysis of the transformed program with a (reduced/iteratively reduced) product of logical abstract domains only involves purified formulæ hence can be performed componentwise (with reduction) without changing the observational semantics.* □

Purification can also be performed for non-disjoint theories, but this requires using as many variables as the number of theories that contain the expression $\mathtt{e}$ in their language, so that we can use existentials and remain precise by asserting the equality between thoses variables.

7 Related Work

SMT solvers have been used in abstract interpretation, e.g. to implement specific logical abstract domains such as uninterpreted functions [11] or to automatically design transformers in presence of a best abstraction [16].

Contrary to the logical abstract interpretation framework developed by [12, 21, 10] we do not assume that the behavior of the program is described by formulæ in the same theory as the theory of the logical abstract domain, which offers no soundness guarantee, but instead we give the semantics of the logical abstract domains with respect to a set of possible semantics which includes the possibility of a sound combination of a mathematical semantics and a machine semantics, which is hard to achieve in SMT solvers without breaking down their performances (e.g. by encoding modular arithmetics in integer arithmetics or encoding floats either bitwise or with reals and roundings). So, our approach allows the description of the abstraction mechanism, comparisons of logical abstract domains, and to provide proofs of soundness on a formal basis.

Specific combinations of theories have been proposed for static analysis such as linear arithmetic and uninterpreted functions [12], universally quantified formulæ over theories such as linear arithmetic and uninterpreted functions [10] or the combination of a shape analysis with a numerical analysis [9][4]. The framework that we propose to combine algebraic and logical abstract domains can be used to design static analyzers incrementally, with minimal efforts to include new abstractions to improve precision either globally for the whole program analysis or locally, e.g. to prove loop invariants provided by the end user.

8 Conclusion

We have proposed a new design method of static analyzers based on the reduced product or its approximation by the iterated reduction of the product to combine algebraic and logical abstract domains. This is for invariance inference but is also applicable to invariant verification. The key points were to consider an observational semantics with multiple interpretations and the understanding of the Nelson-Oppen theory combination procedure [15] and its followers, as well as consequence finding in structured theories [13], as an iterated reduction of the product of theories so that algebraic and logical abstract domains can be symmetrically combined in a product either reduced or with iterated reduction. The interest of the (reduced) product in logical abstract interpretation is that the analysis for each theory can be separated, even when they are not disjoint, thus allowing for an effective use of dedicated SMT solvers for each of the components.

Finally, having shown the similarity and complementarity of analysis by abstract interpretation and program proofs by theorem provers and SMT solvers, we hope that our framework will allow reuse and cooperations between developments in both communities.

Acknowledgments. We thank D. Jovanović and A. Podelski for help and comments.

[4] These approaches can be formalized as observational reduced products.

References

[1] Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. In: Infotech@Aerospace, pp. 2010–3385 (2010)

[2] Bradley, A.R., Manna, Z.: The Calculus of Computation, Decision procedures with Applications to Verification. Springer, Heidelberg (2007)

[3] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4^{th} POPL, pp. 238–252 (1977)

[4] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6^{th} POPL, pp. 269–282 (1979)

[5] Cousot, P., Cousot, R., Mauborgne, L.: Logical Abstract Domains and Interpretations. In: Nanz, S. (ed.) The Future of Engineering. Springer, Heidelberg (2010)

[6] Elder, M., Gopan, D., Reps, T.: View-Augmented Abstractions. In: 2^{nd} NSAD, ENTCS (2010)

[7] Ferrara, P., Logozzo, F., Fähndrich, M.: Safer unsafe code in.NET. In: OOPSLA, pp. 329–346 (2008)

[8] Granger, P.: Improving the results of static analyses of programs by local decreasing iterations. In: Shyamasundar, R.K. (ed.) FSTTCS 1992. LNCS, vol. 652, pp. 68–79. Springer, Heidelberg (1992)

[9] Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: 36^{th} POPL, pp. 239–251 (2009)

[10] Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: 35th POPL, pp. 235–246 (2008)

[11] Gulwani, S., Necula, G.C.: Path-sensitive analysis for linear arithmetic and uninterpreted functions. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 328–343. Springer, Heidelberg (2004)

[12] Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: PLDI, pp. 376–386 (2006)

[13] McIlraith, S.A., Amir, E.: Theorem proving with structured theories. In: IJCAI, pp. 624–634 (2001)

[14] Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: LCTES, pp. 54–63 (2006)

[15] Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. TOPLAS 1(2), 245–257 (1979)

[16] Reps, T.W., Sagiv, S., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004)

[17] Shankar, N., Rueß, H.: Combining shostak theories. In: Tison, S. (ed.) RTA 2002. LNCS, vol. 2378, pp. 1–18. Springer, Heidelberg (2002)

[18] Tinelli, C., Harandi, M.T.: A new correctness proof of the Nelson–Oppen combination procedure. In: Frontiers of Combining Systems, pp. 103–120. Kluwer Academic Publishers, Dordrecht (1996)

[19] Tinelli, C., Ringeissen, C.: Unions of non-disjoint theories and combinations of satisfiability procedures. Theor. Comput. Sci. 290(1), 291–353 (2003)

[20] Tinelli, P., Zarba, C.G.: Combining non-stably infinite theories. Electr. Notes Theor. Comput. Sci. 86(1) (2003)

[21] Tiwari, A., Gulwani, S.: Logical interpretation: Static program analysis using theorem proving. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 147–166. Springer, Heidelberg (2007)

Author Index