Second International Workshop on Rewriting Techniques for Program Transformations and Evaluation

WPTE'15, July 2, 2015, Warsaw, Poland

Edited by

Yuki Chiba
Santiago Escobar
Naoki Nishida
David Sabel
Manfred Schmidt-Schauß

OASICS

Editors

Yuki Chiba
School of Information Science
JAIST
chiba@jaist.ac.jp

Santiago Escobar
DSIC
Universitat Politècnica de València
sescobar@dsic.upv.es

Naoki Nishida
Graduate School of Information Science
Nagoya University
nishida@is.nagoya-u.ac.jp

David Sabel
Institute for Informatics
Computer Science and Mathematics Department
Goethe-University Frankfurt am Main
sabel@ki.cs.uni-frankfurt.de

Manfred Schmidt-Schauß
Institute for Informatics
Computer Science and Mathematics Department
Goethe-University Frankfurt am Main
schauss@ki.cs.uni-frankfurt.de

ACM Classification 1998
A.0 Conference proceedings, D.1.3 Concurrent Programming, D.3.1 Formal Definitions and Theory, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages, F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems, I.2.3 Deduction and Theorem Proving

ISBN 978-3-939897-94-1

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-94-1.

Publication date
June, 2015

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/OASIcs.WPTE.2015.i
ISBN /978-3-939897-94-1 **ISSN 2190-6807** **http://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 2190-6807

www.dagstuhl.de/oasics

Contents

Workshop Organization

WPTE 2015 Organizers

Yuki Chiba	JAIST, Japan
Santiago Escobar	Universitat Politècnica de València, Spain
Naoki Nishida	Nagoya University, Japan
David Sabel	Goethe-University Frankfurt am Main, Germany
Manfred Schmidt-Schauß	Goethe-University Frankfurt am Main, Germany

Program Chairs

Santiago Escobar	Universitat Politècnica de València, Spain
Naoki Nishida	Nagoya University, Japan

Program Committee

Takahito Aoto	RIEC, Tohoku University, Japan
Yuki Chiba	JAIST, Japan
Fer-Jan de Vries	University of Leicester, United Kingdom
Santiago Escobar	Universitat Politècnica de València, Spain
Johan Jeuring	Open Universiteit Nederland & Universiteit Utrecht, the Netherlands
Delia Kesner	Université Paris-Diderot, France
Sergueï Lenglet	Université de Lorraine, France
Elena Machkasova	University of Minnesota, Morris, United States
William Mansky	University of Pennsylvania, United States
Joachim Niehren	INRIA Lille, France
Naoki Nishida	Nagoya University, Japan
Kristoffer H. Rose	Two Sigma Investments, LLC, United State
David Sabel	Goethe-University Frankfurt am Main, Germany
Masahiko Sakai	Nagoya University, Japan
Manfred Schmidt-Schauß	Goethe-University Frankfurt am Main, Germany
Janis Voigtländer	University of Bonn, Germany
Johannes Waldmann	HTWK Leipzig, Germany
Harald Zankl	University of Innsbruck, Austria

External Reviewers

Jose Cambronero
Alfons Geser
Peter F. Stadler

Preface

This volume contains the papers presented at the *Second International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2015)* which was held on July 2, 2015 in Warsaw, Poland, and affiliated with the *eighth edition of the International Conference on Rewriting, Deduction, and Programming (RDP 2015).*

Scope of WPTE

Verification and validation of properties of programs, optimizing and compiling programs, and generating programs can benefit from the application of rewriting techniques. Source-level program transformations are used in compilation to simplify and optimize programs, in code refactoring to improve the design of programs; and in software verification and code validation, program transformations are used to translate and/or simplify programs into the forms suitable for specific verification purposes or tests. Those program transformations can be translations from one language into another one, transformations inside a single language, or the change of the evaluation strategy within the same language.

Since rewriting techniques are of great help for studying correctness of program transformations, translations and evaluation, the aim of WPTE is to bring together the researchers working on program transformations, evaluation, and operationally-based programming language semantics, using rewriting methods, in order to share the techniques and recent developments and to exchange ideas to encourage further activation of research in this area. The first WPTE was held in Vienna 2014 during the *Vienna Summer of Logic 2014 (VSL 2014)* as a workshop of the *sixth Federated Logic Conference (FLoC 2014).*

Topics in the scope of WPTE include the correctness of program transformations, optimizations and translations; program transformations for proving termination, confluence and other properties; correctness of evaluation strategies; operational semantics of programs, operationally-based program equivalences such as contextual equivalences and bisimulations; cost-models for reasoning about the optimizing power of transformations and the costs of evaluation; program transformations for verification and theorem proving purposes; translation, simulation, equivalence of programs with different formalisms, and evaluation strategies; program transformations for applying rewriting techniques to programs in specific programming languages; program transformations for program inversions and program synthesis; program transformation and evaluation for Haskell and Rewriting.

"Program transformation and evaluation for Haskell and Rewriting" is a new topic of this workshop including equational reasoning and other rewriting techniques for program verification and analysis; lambda calculi and type systems for functional programs and higher-order rewrite systems; rewriting of type expressions in the type checker; rewriting of programs by refactoring tools, optimizers, code generators; execution of programs as a form of graph rewriting (terms with sharing); Template Haskell, generally introducing a rewriting-like macro language into the compilation process; rewriting modulo commonly occurring axioms such as associativity, commutativity, and identity element.

WPTE 2015

For WPTE 2015 four regular research papers were accepted out of the submissions. Additionally the program of WPTE contained the following talks which the program committee recommended for presentation:

- Guillaume Madelaine, Cédric Lhoussaine, and Joachim Niehren: *Structural simplification of chemical reaction networks preserving deterministic semantics*
- Naosuke Matsuda: *A simple extension of the Curry-Howard correspondence with intuitionistic lambda rho calculus*
- Koichi Sato, Kentaro Kikuchi, Takahito Aoto, and Yoshihito Toyama: *Context-Moving Transformation for Term Rewriting Systems*

Each submission was reviewed by at least three members of the Program Committee, with the help of three external reviewers. Paper submission, reviewing, and the electronic meeting of the program committee used the great EasyChair system of Andrei Voronkov, which was also indispensable for preparing the WPTE program and collecting the papers for these proceedings.

In addition to the contributed papers, the WPTE program contained an invited talk by Brigitte Pientka with title "*Mechanizing Meta-Theory in Beluga*".

Acknowledgment

We thank our publisher Schloss Dagstuhl–Leibniz-Zentrum für Informatik for publishing our proceedings in the OpenAccess Series in Informatics (OASIcs). In particular we would like to thank Mark Herbstritt and Michael Wagner for their very helpful and always prompt support during production of the OASIcs proceedings.

We thank the organizers of RDP 2015 for hosting our workshop, and for the financial support for our proceedings published via OASIcs. We are particularly indebted to Aleksy Schubert (chair of RDP 2015) and Jacek Chrząszcz for their help in preparing our workshop.

Finally we thank the members of the program committee for their careful reviewing of all submissions and we thank the participants for their valuable contributions.

July 2015

Yuki Chiba
Santiago Escobar
Naoki Nishida
David Sabel
Manfred Schmidt-Schauß

The Collection of all Abstracts of the Talks at WPTE 2015

The aim of this chapter is to document all talks of the “Second International Workshop on Rewriting Techniques for Program Transformations and Evaluation” (WPTE 2015). Hence, this collection contains all abstracts of talks held at WPTE 2015. The abstracts are ordered alphabetically by author names. Further information and e.g. extended abstracts on the talks on work in progress, can also be found in USB flash drives distributed to all participants of RDP 2015.

Head reduction and normalization in a call-by-value lambda-calculus

Author: Giulio Guerrieri

Abstract: Recently, a standardization theorem has been proven for a variant of Plotkin’s call-by-value lambda-calculus extended by means of two commutation rules (sigma-reductions): this result was based on a partitioning between head and internal reductions. We study the head normalization for this call-by-value calculus with sigma-reductions and we relate it to the weak evaluation of original Plotkin’s call-by-value lambda-calculus. We give also a (non-deterministic) normalization strategy for the call-by-value lambda-calculus with sigma-reductions.

Structural simplification of chemical reaction networks preserving deterministic semantics

Authors: Guillaume Madelaine, Cédric Lhoussaine, and Joachim Niehren

Abstract: We study the structural simplification of chemical reaction networks preserving the deterministic kinetics. We aim at finding simplification rules that can eliminate intermediate molecules while preserving the dynamics of all others. The rules should be valid even though the network is plugged into a bigger context. An example is Michaelis-Menten’s simplification rule for enzymatic reactions. In this paper, we present structural simplification rules for reaction networks that can eliminate intermediate molecules at equilibrium, without assuming that all molecules are at equilibrium, i.e. in a steady state. Our simplification rules preserve the deterministic semantics of reaction networks, in all contexts compatible with the equilibrium of the eliminated molecules. We illustrate the simplification on a biological example network from systems biology.

A simple extension of the Curry-Howard correspondence with intuitionistic lambda rho calculus

Author: Naosuke Matsuda

Abstract: In (Fujita et.al., to appear), a natural deduction style proof system called “intuitionistic $\lambda\rho$-calculus” for implicational intuitionistic logic and some reduction rules for the proof system were given. In this paper, we show that the system is easy to treat but has sufficient expressive power to provide a powerful model of computation.

Towards Modelling Actor-Based Concurrency in Term Rewriting

Authors: Adrián Palacios and Germán Vidal

Abstract: In this work, we introduce a scheme for modelling actor systems within sequential term rewriting. In our proposal, a TRS consists of the union of three components: the functional part (which is specific of a system), a set of rules for reducing concurrent actions, and a set of rules for defining a particular scheduling policy. A key ingredient of our approach is that concurrent systems are modelled by terms in which concurrent actions can never occur inside user-defined function calls. This assumption greatly simplifies the definition of the semantics for concurrent actions, since no term traversal will be needed. We prove that these systems are well defined in the sense that concurrent actions can always be reduced.

Our approach can be used as a basis for modelling actor-based concurrent programs, which can then be analyzed using existing techniques for term rewrite systems.

Mechanizing Meta-Theory in Beluga

Author: Brigitte Pientka

Abstract: Mechanizing formal systems, given via axioms and inference rules, together with proofs about them plays an important role in establishing trust in formal developments. In this talk, I will survey the proof environment Beluga. To specify formal systems and represent derivations within them, Beluga provides a sophisticated infrastructure based on the logical framework LF; in particular, its infrastructure not only supports modelling binders via binders in LF, but extends and generalizes LF with first-class contexts to abstract over a set of assumptions, contextual objects to model derivations that depend on assumptions, and first-class simultaneous substitutions to relate contexts. These extensions allow us to directly support key and common concepts that frequently arise when describing formal systems and derivations within them.

To reason about formal systems, Beluga provides a dependently typed functional language for implementing inductive proofs about derivations as recursive functions on contextual objects following the Curry-Howard isomorphism. Recently, the Beluga system has also been extended with a totality checker which guarantees that recursive programs are well-founded and correspond to inductive proofs and an interactive program development environment to support incremental proof / program construction. Taken together these extensions enable direct and compact mechanizations. To demonstrate Beluga's strength, we develop a weak normalization proof using logical relations. The Beluga system together with examples is available from `http://complogic.cs.mcgill.ca/beluga/`.

Observing Success in the Pi-Calculus

Authors: David Sabel and Manfred Schmidt-Schauß

Abstract: A contextual semantics – defined in terms of successful termination and may- and should-convergence – is analyzed in the synchronous pi-calculus with replication and a constant Stop to denote success. The contextual ordering is analyzed, some nontrivial process equivalences are proved, and proof tools for showing contextual equivalences are provided. Among them are a context lemma and new notions of sound applicative similarities for may- and should-convergence. A further result is that contextual equivalence in the pi-calculus with Stop conservatively extends barbed testing equivalence in the (Stop-free) pi-calculus and thus results on contextual equivalence can be transferred to the (Stop-free) pi-calculus with barbed testing equivalence.

Context-Moving Transformation for Term Rewriting Systems

Authors: Koichi Sato, Kentaro Kikuchi, Takahito Aoto, and Yoshihito Toyama

Abstract: Proofs by induction are often incompatible with tail-recursive definitions as the accumulator changes in the course of unfolding the definitions. Context-moving (Giesl, 2000) for functional programs transforms tail-recursive programs into non tail-recursive ones which are more suitable for verification. In this work, we formulate a context-moving transformation for term rewriting systems, and prove the correctness with respect to both eager evaluation semantics and initial algebra semantics under some conditions on the programs to be transformed.

Formalizing Bialgebraic Semantics in PVS 6.0

Authors: Sjaak Smetsers, Ken Madlener, and Marko van Eekelen

Abstract: Both operational and denotational semantics are prominent approaches for reasoning about properties of programs and programming languages. In the categorical framework developed by Turi and Plotkin both styles of semantics are unified using a single, syntax independent format, known as GSOS, in which the operational rules of a language are specified. From this format, the operational and denotational semantics are derived. The approach of Turi and Plotkin is based on the categorical notion of bialgebras. In this paper we specify this work in the theorem prover *PVS*, and prove the adequacy theorem of this formalization. One of our goals is to investigate whether *PVS* is adequately suited for formalizing metatheory. Indeed, our experiments show that the original categorical framework can be formalized conveniently. Additionally, we present a GSOS specification for the simple imperative programming language *While*, and execute the derived semantics for a small example program.

List of Authors

Takahito Aoto
RIEC
Tohoku University, Japan

Giulio Guerrieri
Laboratoire PPS, UMR 7126
Université Paris Diderot, France

Kentaro Kikuchi
RIEC
Tohoku University, Japan

Cédric Lhoussaine
CRIStAL, UMR 9189
University of Lille, France

Guillaume Madelaine
CRIStAL, UMR 9189
University of Lille, France

Ken Madlener
Institute for Computing and Information Sciences
Radboud University Nijmegen, the Netherlands

Naosuke Matsuda
Department of Mathematical and Computing Sciences
Tokyo Institute of Technology, Japan

Joachim Niehren
CRIStAL, UMR 9189
INRIA Lille, France

Adrián Palacios
DSIC
Universitat Politècnica de València, Spain

Brigitte Pientka
School of Computer Science
McGill University, Canada

David Sabel
Computer Science and Mathematics Department
Goethe-University Frankfurt am Main, Germany

Koichi Sato
RIEC
Tohoku University, Japan

Manfred Schmidt-Schauß
Computer Science and Mathematics Department
Goethe-University Frankfurt am Main, Germany

Sjaak Smetsers
Institute for Computing and Information Sciences
Radboud University Nijmegen, the Netherlands

Yoshihito Toyama
RIEC
Tohoku University, Japan

Marko van Eekelen
School of Computer Science
Open University of the Netherlands, the Netherlands

Germán Vidal
DSIC
Universitat Politècnica de València, Spain

Mechanizing Meta-Theory in Beluga*

Brigitte Pientka

School of Computer Science, McGill University
Montreal, Canada
`bpientka@cs.mcgill.ca`

Abstract

Mechanizing formal systems, given via axioms and inference rules, together with proofs about them plays an important role in establishing trust in formal developments. In this talk, I will survey the proof environment Beluga. To specify formal systems and represent derivations within them, Beluga provides a sophisticated infrastructure based on the logical framework LF; in particular, its infrastructure not only supports modelling binders via binders in LF, but extends and generalizes LF with first-class contexts to abstract over a set of assumptions, contextual objects to model derivations that depend on assumptions, and first-class simultaneous substitutions to relate contexts. These extensions allow us to directly support key and common concepts that frequently arise when describing formal systems and derivations within them.

To reason about formal systems, Beluga provides a dependently typed functional language for implementing inductive proofs about derivations as recursive functions on contextual objects following the Curry-Howard isomorphism. Recently, the Beluga system has also been extended with a totality checker which guarantees that recursive programs are well-founded and correspond to inductive proofs and an interactive program development environment to support incremental proof / program construction. Taken together these extensions enable direct and compact mechanizations. To demonstrate Beluga's strength, we develop a weak normalization proof using logical relations. The Beluga system together with examples is available from `http://complogic.cs.mcgill.ca/beluga/`.

1998 ACM Subject Classification D.3.1 Formal Definitions and Languages, F.3.2 Semantics of Programming Languages, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Type systems, Dependent Types, Logical Frameworks

Digital Object Identifier 10.4230/OASIcs.WPTE.2015.1

Category Invited Talk

Acknowledgements. Many of the ideas I describe in this talk are joint work with Andrew Cave. Over the past 6 years, several undergraduate students, graduate students, and postdocs have contributed to the implementation of BELUGA: M. Boespflug, A. Cave, S. Cooper,A. Marchildon, F. Ferreira, O. Savary Belanger, D. Thibodeau, T. Xue.

* This work was supported by Natural Sciences and Engineering Research Council of Canada (NSERC) and Fonds de Recherche Nature et technologies Quebec (FQRNT).

Head reduction and normalization in a call-by-value lambda-calculus

Giulio Guerrieri

Laboratoire PPS, UMR 7126, Université Paris Diderot, Sorbonne Paris Cité
F-75205 Paris, France
giulio.guerrieri@pps.univ-paris-diderot.fr

Abstract

Recently, a standardization theorem has been proven for a variant of Plotkin's call-by-value lambda-calculus extended by means of two commutation rules (sigma-reductions): this result was based on a partitioning between head and internal reductions. We study the head normalization for this call-by-value calculus with sigma-reductions and we relate it to the weak evaluation of original Plotkin's call-by-value lambda-calculus. We give also a (non-deterministic) normalization strategy for the call-by-value lambda-calculus with sigma-reductions.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory, F.3.2 Semantics of Programming Language, F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems.

Keywords and phrases sequentialization, lambda-calculus, sigma-reduction, call-by-value, head reduction, internal reduction, (strong) normalization, evaluation, confluence, reduction strategy.

Digital Object Identifier 10.4230/OASIcs.WPTE.2015.3

1 Introduction

The call-by-value λ-calculus (λ_v-calculus or λ_v for short) and the operational machine for its evaluation has been introduced by Plotkin [15] inspired by Landin's seminal work [9] on the programming language ISWIM and the SECD machine. The λ_v-calculus is a paradigmatic language able to capture two features of many functional programming languages: call-by-value parameter passing policy (parameters are evaluated before being passed) and weak evaluation (the body of a function is evaluated only when parameters are supplied).

The syntax of λ_v is the same as that of the ordinary (i.e. call-by-name) λ-calculus (λ for short), but the reduction rule for λ_v, called β_v, is a restriction of the β-rule for λ: β_v allows the contraction of a redex $(\lambda x.M)N$ only in case the argument N is a value, i.e. a variable or an abstraction. Unfortunately, the semantic analysis of the λ_v-calculus has turned out to be more elaborate than that of ordinary λ-calculus. This is due essentially to the "weakness" of (full) β_v-reduction, a fact widely recognized: indeed, there are many proposals of alternative call-by-value λ-calculi extending Plotkin's one [11, 10, 8, 2, 1]. To have an example of the "weakness" of the rewriting rules of λ_v, it is sufficient to consider that it is impossible to have an internal operational characterization (i.e. one that uses the β_v-reduction) of the semantically meaningful notions of call-by-value solvability and potential valuability, as shown in [13, 14, 2].

In this paper we will study the λ_v^σ-calculus (λ_v^σ for short), a call-by-value extension of λ_v recently proposed in [4]: it keeps the λ_v (and λ) syntax and it adds to the β_v-reduction two commutation rules, called σ_1 and σ_3, which unblock "hidden" β_v-redexes that are concealed by the "hyper-sequential structure" of terms. The λ_v^σ-calculus enjoy some basic properties we expect from a calculus, namely confluence (see [4]) and standardization (see [7]). Moreover, λ_v^σ provides elegant characterizations of many semantic properties, e.g. solvability and potential valuability (see [4]), and it is conservative with respect to Plotkin's λ_v: in particular, [7] shows that the notions of solvability and potential valuability for λ_v^σ coincide with those for λ_v.

The v-reduction (i.e. the reduction for λ_v^σ) can be partitioned into head v-reduction and internal v-reduction; the head v-reduction is in turn decomposed into head β_v- and head σ-reduction. The head β_v-reduction is just the deterministic weak evaluation strategy for Plotkin's λ_v-calculus. According to a sequentialization theorem proven in [7, Theorem 22], any v-reduction sequence can be sequentialized in an initial head β_v-reduction sequence followed by a head σ-reduction sequence followed by an internal v-reduction sequence. Similar well-known results hold for λ and λ_v, and starting from them one can define a normalization strategy for λ and λ_v, i.e. a deterministic reduction strategy that reaches a normal form if and only if one exists: for example the leftmost reduction, see [19, Theorem 2.8] and [3, Theorem 13.2.2].

Is there a normalization strategy for λ_v^σ? Theorem 24, one of the main results of this paper, proves that, starting from the sequentialization theorem mentioned above, a normalization strategy can be defined for λ_v^σ, based on the notions of head β_v- and head σ-reductions.

A first difference appears here between λ_v^σ and λ_v (or λ): the normalization strategy for λ_v^σ is not deterministic. Indeed, while the head β_v-reduction (or the call-by-name head reduction) is deterministic (i.e. a partial function), the head v-reduction is non-deterministic and, still worse, non-confluent and there are terms having several head v-normal forms: this might appear disappointing. So, three natural questions arise:

- With respect to head v-reduction, do normalization and strong normalization coincide?[1]
- Can we relate the termination of head β_v-reduction and head v-reduction?
- Can we characterize the terms having a unique head v-normal form?

Our Theorem 21 gives a positive answer to the first two questions. Observe that the lack of any form of confluence for head v-reduction requires a more complex reasoning, passing through a syntactic characterization of head β_v- and head v-normal forms. Theorem 21 not only shows that the head v-reduction and the head β_v-reduction are deeply related (and hence, again, λ_v^σ is conservative with respect to λ_v) but also that both enjoy good properties analogous to the ones of the (call-by-name) head reduction for ordinary λ-calculus.

Our Proposition 27 gives a partial answer to the third question above: it shows that in some cases (of interest) a head v-normalizable term has a unique head v-normal form; in particular, every closed head v-normalizable term has a unique head v-normal form.

So, λ_v^σ appears as an extension of Plotkin's λ_v-calculus that enjoys many meaningful conservation properties with respect to λ_v and therefore it is a useful tool for theoretical and semantic investigations about λ_v and call-by-value setting. See also conclusions in Section 6 for further and more precise motivations for this paper and future work.

Related work. The λ_v^σ-calculus has been recently introduced in [4] and further investigated in [7]. It is an extension of Plotkin's λ_v-calculus inspired by the call-by-value translation of λ-terms into linear logic proof-nets [6]. Other variants of λ_v have been introduced in the literature for modeling the call-by-value computation. We would like to cite here at least the contributions of Moggi [11], Felleisen and Sabry [18], Maraist et al. [10], Herbelin and Zimmerman [8], Accattoli and Paolini [2] (the latter is inspired by the call-by-value translation of λ-terms into linear logic proof-nets, see [1]). All these proposals are based on the introduction of new constructs to the syntax of λ_v, so the comparison between them is

[1] The answer is trivially positive in the case of call-by-name head normalization (for λ) and head β_v-normalization, since these reductions are deterministic.

not easy with respect to syntactical properties (some detailed comparison is given in [2]). We point out that the calculi introduced in [11, 18, 10, 8] present some variants of our σ_1 and/or σ_3 rules, often in a setting with explicit substitutions. Regnier [16, 17] used the rule σ_1 (but not σ_3) in ordinary (i.e. call-by-name) λ-calculus.

The head v-reduction investigated here has been introduced in [7]. Some results of this paper are inspired by the Takahashi's results [19] on the ordinary (i.e. call-by-name) λ-calculus, partially adapted by Crary [5] for λ_v.

Outline. In Section 2 we introduce the syntax and the reduction rules of the λ_v^σ-calculus. In Section 3 we define the head v-reduction and the internal v-reduction, and we recall some results already proven in [7] concerning them. Section 4 is devoted to proving the first main result of our paper: Theorem 21, which studies the normalization for the head v-reduction and relates it to the weak evaluation strategy for Plotkin's λ_v-calculus. In Section 5 we show that the head v-reduction can be used to define a normalization strategy for the λ_v^σ-calculus (Theorem 24), and moreover in some cases the head v-normal form (if any) of a term is unique (Proposition 27). In Section 6 we summarize the findings and suggest future work.

2 The call-by-value lambda calculus with sigma-rules

In this section we present λ_v^σ, a call-by-value λ-calculus introduced in [4] that adds two σ-reduction rules to pure (i.e. without constants) call-by-value λ-calculus defined by Plotkin in [15].

The syntax of terms of λ_v^σ [4] is the same as the one of ordinary λ-calculus and Plotkin's call-by-value λ-calculus λ_v [15] (without constants). Given a countable set $\mathcal{V}$ of *variables* (denoted by $x, y, z, \dots$), the sets Λ of *terms* and Λ_v of *values* are defined by mutual induction:

$$\begin{array}{lrcll} (\Lambda_v) & V, U & ::= & x \mid \lambda x.M & \textit{values} \\ (\Lambda) & M, N, L & ::= & V \mid MN & \textit{terms} \end{array}$$

Clearly, $\Lambda_v \subsetneq \Lambda$. All terms are considered up to α-conversion. The set of free variables of a term M is denoted by $\mathsf{fv}(M)$. Given $V_1, \dots, V_n \in \Lambda_v$ and pairwise distinct variables $x_1, \dots, x_n$, $M\{V_1/x_1, \dots, V_n/x_n\}$ denotes the term obtained by the *capture-avoiding simultaneous substitution* of V_i for each free occurrence of x_i in the term M (for all $1 \le i \le n$). Note that, for all $V, V_1, \dots, V_n \in \Lambda_v$ and pairwise distinct variables $x_1, \dots, x_n$, $V\{V_1/x_1, \dots, V_n/x_n\} \in \Lambda_v$.

Contexts (with exactly one hole $(\!|\cdot|\!)$), denoted by $\mathtt{C}$, are defined as usual via the grammar:

$$\mathtt{C} ::= (\!|\cdot|\!) \mid \lambda x.\mathtt{C} \mid \mathtt{C}M \mid M\mathtt{C}\,.$$

We use $\mathtt{C}(\!|M|\!)$ for the term obtained by the capture-allowing substitution of the term M for the hole $(\!|\cdot|\!)$ in the context $\mathtt{C}$.

▶ **Notation.** From now on, we set $I = \lambda x.x$ and $\Delta = \lambda x.xx$.

The reduction rules of λ_v^σ consist of Plotkin's β_v-reduction rule, introduced in [15], and two simple commutation rules called σ_1 and σ_3, studied in [4, 7].

▶ **Definition 1** (Reduction rules). *We define the following binary relations on Λ (for any $M, N, L \in \Lambda$ and any $V \in \Lambda_v$):*

$$\begin{array}{rll} (\lambda x.M)V \mapsto_{\beta_v} & M\{V/x\} & \\ (\lambda x.M)NL \mapsto_{\sigma_1} & (\lambda x.ML)N & \textit{with } x \notin \mathsf{fv}(L) \\ V((\lambda x.L)N) \mapsto_{\sigma_3} & (\lambda x.VL)N & \textit{with } x \notin \mathsf{fv}(V). \end{array}$$

We set $\mapsto_{\sigma} = \mapsto_{\sigma_1} \cup \mapsto_{\sigma_3}$ *and* $\mapsto_{\mathsf{v}} = \mapsto_{\beta_v} \cup \mapsto_{\sigma}$.

For any $\mathsf{r} \in \{\beta_v, \sigma_1, \sigma_3, \sigma, \mathsf{v}\}$*, if* $M \mapsto_{\mathsf{r}} M'$ *then* M *is a* r-redex *and* M' *is its* r-contractum*. In this sense, a term of the shape* $(\lambda x.M)N$ *(for any* $M, N \in \Lambda$*) is a* β-redex*.*

The side conditions on $\mapsto_{\sigma_1}$ and $\mapsto_{\sigma_3}$ in Definition 1 can be always fulfilled by α-renaming.

Obviously, any β_v-redex is a β-redex but the converse does not hold: $(\lambda x.z)(yI)$ is a β-redex but not a β_v-redex.

▶ **Example 2.** Redexes of different kind may overlap: for example, the term $\Delta I \Delta$ is a σ_1-redex and it contains the β_v-redex ΔI; the term $\Delta(I\Delta)(xI)$ is a σ_1-redex and it contains the σ_3-redex $\Delta(I\Delta)$, which contains in turn the β_v-redex $I\Delta$.

▶ **Notation.** Let R be a binary relation on Λ. We denote by R^* (resp. R^+; $\mathsf{R}^=$) the reflexive-transitive (resp. transitive; reflexive) closure of R.

▶ **Definition 3** (Reductions). *Let* $\mathsf{r} \in \{\beta_v, \sigma_1, \sigma_3, \sigma, \mathsf{v}\}$*.*

The r-reduction $\to_{\mathsf{r}}$ *is the contextual closure of* $\mapsto_{\mathsf{r}}$*, i.e.* $M \to_{\mathsf{r}} M'$ *iff there is a context* $\mathtt{C}$ *and* $N, N' \in \Lambda$ *such that* $M = \mathtt{C}(\!|N|\!)$*,* $M' = \mathtt{C}(\!|N'|\!)$ *and* $N \mapsto_{\mathsf{r}} N'$*.*

The r-equivalence $\simeq_{\mathsf{r}}$ *is the reflexive-transitive and symmetric closure of* $\to_{\mathsf{r}}$*.*

Let M *be a term:* M *is* r-normal *if there is no term* N *such that* $M \to_{\mathsf{r}} N$*;* M *is* r-normalizable *if there is a* r*-normal term* N *such that* $M \to_{\mathsf{r}}^* N$*;* M *is* strongly r-normalizable *if there is no sequence* $(N_i)_{i \in \mathbb{N}}$ *of terms such that* $M = N_0$ *and* $N_i \to_{\mathsf{r}} N_{i+1}$ *for any* $i \in \mathbb{N}$*.*

Obviously, $\to_{\sigma} = \to_{\sigma_1} \cup \to_{\sigma_3} \subsetneq \to_{\mathsf{v}}$ and $\to_{\beta_v} \subsetneq \to_{\mathsf{v}}$ and $\to_{\mathsf{v}} = \to_{\beta_v} \cup \to_{\sigma}$.

▶ **Remark 4.** For any $\mathsf{r} \in \{\beta_v, \sigma_1, \sigma_3, \sigma, \mathsf{v}\}$ (resp. $\mathsf{r} \in \{\sigma_1, \sigma_3, \sigma\}$), values are closed under r-reduction (resp. r-expansion): for any $V \in \Lambda_v$, if $V \to_{\mathsf{r}} M$ (resp. $M \to_{\mathsf{r}} V$) then $M \in \Lambda_v$; more precisely, $V = \lambda x.N$ and $M = \lambda x.N'$ for some $N, N' \in \Lambda$ with $N \to_{\mathsf{r}} N'$ (resp. $N' \to_{\mathsf{r}} N$).

For any $\mathsf{r} \in \{\beta_v, \mathsf{v}\}$, values are not closed under r-expansion: $I\Delta \to_{\beta_v} \Delta \in \Lambda_v$ but $I\Delta \notin \Lambda_v$.

▶ **Proposition 5** (See [4]). *The* σ*-reduction is confluent and strongly normalizing. The* v*-reduction is confluent.*

The λ_v^σ*-calculus*, λ_v^σ for short, is the set Λ of terms endowed with the v-reduction $\to_{\mathsf{v}}$. The set Λ endowed with the β_v-reduction $\to_{\beta_v}$ is the λ_v-calculus (λ_v for short), i.e. the Plotkin's call-by-value λ-calculus [15] (without constants), which is thus a sub-calculus of λ_v^σ.

▶ **Example 6.** $M = (\lambda y.\Delta)(xI)\Delta \to_{\sigma_1} (\lambda y.\Delta\Delta)(xI) \to_{\beta_v} (\lambda y.\Delta\Delta)(xI) \to_{\beta_v} \dots$ and $N = \Delta((\lambda y.\Delta)(xI)) \to_{\sigma_3} (\lambda y.\Delta\Delta)(xI) \to_{\beta_v} (\lambda y.\Delta\Delta)(xI) \to_{\beta_v} \dots$ are the only possible v-reduction paths from M and N respectively: M and N are not v-normalizable, and $M \simeq_{\mathsf{v}} N$. Meanwhile, M and N are β_v-normal and different, hence $M \not\simeq_{\beta_v} N$ (by confluence of $\to_{\beta_v}$, see [15]).

Informally, σ-rules unblock β_v-redexes which are hidden by the "hyper-sequential structure" of terms. This approach is alternative to the one in [2, 1] where hidden β_v-redexes are reduced using rules acting at a distance (through explicit substitutions). It can be shown that the call-by-value λ-calculus with explicit substitution introduced in [2] can be embedded in λ_v^σ.

It is well-known that the β_v-reduction can be simulated by linear logic cut-elimination via the call-by-value translation $(\cdot)^v$ of λ-terms into proof-nets, called by Girard [6, pp. 81-82] "boring" and defined by $(A \Rightarrow B)^v = {!A^v} \multimap {!B^v}$ (see also [1]). The images under $(\cdot)^v$ of a σ-redex and its σ-contractum are equal modulo some non-structural cut-elimination steps.

3 Head and internal reductions

In this section we introduce the definitions of head v-reduction (which is decomposed in head β_v- and head σ-reductions) and internal v-reduction, then we recall some results proven in [7].

▶ **Notation.** From now on, we always assume that $V, V' \in \Lambda_v$.

Note that the generic form of a term is $VM_1 \dots M_m$ for some $m \in \mathbb{N}$ (in particular, values are obtained when $m = 0$). The sequentialization result is based on a partitioning of v-reduction between head v-reduction and internal v-reduction.

▶ **Definition 7** (Head β_v-reduction). *The* head β_v-reduction $\overset{h}{\to}_{\beta_v}$ *is the binary relation on* Λ *defined inductively by the following rules ($m \in \mathbb{N}$ in both rules):*

$$\frac{}{(\lambda x.M)VM_1 \dots M_m \overset{h}{\to}_{\beta_v} M\{V/x\}M_1 \dots M_m}\,\beta_v \qquad \frac{N \overset{h}{\to}_{\beta_v} N'}{VNM_1 \dots M_m \overset{h}{\to}_{\beta_v} VN'M_1 \dots M_m}\,\mathit{right}$$

The head β_v-reduction $\overset{h}{\to}_{\beta_v}$ is exactly the (pure) "left reduction" defined in [15, p. 136] for λ_v and called "(weak) evaluation" in [18, 5]. If $N \overset{h}{\to}_{\beta_v} N'$ then N' is obtained from N by reducing the leftmost-outermost β_v-redex, not in the scope of a λ: thus, the head β_v-reduction is deterministic (i.e. it is a partial function from Λ to Λ) and does not reduce values.

▶ **Definition 8** (Head σ- and head v-reductions). *The* head σ-reduction $\overset{h}{\to}_\sigma$ *is the binary relation on* Λ *defined inductively by the following rules ($m \in \mathbb{N}$ in all the rules, $x \notin \mathsf{fv}(L)$ in the rule σ_1, $x \notin \mathsf{fv}(V)$ in the rule σ_3):*

$$\frac{}{(\lambda x.M)NLM_1 \dots M_m \overset{h}{\to}_\sigma (\lambda x.ML)NM_1 \dots M_m}\,\sigma_1 \qquad \frac{N \overset{h}{\to}_\sigma N'}{VNM_1 \dots M_m \overset{h}{\to}_\sigma VN'M_1 \dots M_m}\,\mathit{right}$$

$$\frac{}{V((\lambda x.L)N)M_1 \dots M_m \overset{h}{\to}_\sigma (\lambda x.VL)NM_1 \dots M_m}\,\sigma_3$$

The head v-reduction *is* $\overset{h}{\to}_{\mathsf{v}} = \overset{h}{\to}_{\beta_v} \cup \overset{h}{\to}_\sigma$.

Let $\mathsf{r} \in \{\beta_v, \sigma, \mathsf{v}\}$ *and* $N \in \Lambda$*:* N *is* head r-normal *if there is no* $N' \in \Lambda$ *such that* $N \overset{h}{\to}_{\mathsf{r}} N'$*;* N *is* head r-normalizable *if there is a r-normal term* N' *such that* $N \overset{h}{\to}{}^*_{\mathsf{r}} N'$*;* N *is* strongly head r-normalizable *if there is no* $(N_i)_{i \in \mathbb{N}}$ *such that* $N = N_0$ *and* $N_i \overset{h}{\to}_{\mathsf{r}} N_{i+1}$ *for any* $i \in \mathbb{N}$.

Notice that $\mapsto_{\beta_v} \subsetneq \overset{h}{\to}_{\beta_v} \subsetneq \to_{\beta_v}$ and $\mapsto_\sigma \subsetneq \overset{h}{\to}_\sigma \subsetneq \to_\sigma$ and $\mapsto_{\mathsf{v}} \subsetneq \overset{h}{\to}_{\mathsf{v}} \subsetneq \to_{\mathsf{v}}$.

Informally, if $N \overset{h}{\to}_\sigma N'$ then N' is obtained from N by reducing "one of the leftmost" σ_1- or σ_3-redexes, not in the scope of a λ: in general, a term may contain several head σ_1- and σ_3-redexes. Indeed, differently from $\overset{h}{\to}_{\beta_v}$, the head σ-reduction $\overset{h}{\to}_\sigma$ is not deterministic, for example the leftmost-outermost σ_1- and σ_3-redexes may overlap: if $M = (\lambda y.y')(\Delta(xI))I$ then $M \overset{h}{\to}_\sigma (\lambda y.y'I)(\Delta(xI)) = N_1$ by applying the rule σ_1 and $M \overset{h}{\to}_\sigma (\lambda z.(\lambda y.y')(zz))(xI)I = N_2$ by applying the rule σ_3. Note that N_1 contains only a head σ_3-redex and $N_1 \overset{h}{\to}_\sigma (\lambda z.(\lambda y.y'I)(zz))(xI) = N$ which is head v-normal; meanwhile N_2 contains only a head σ_1-redex and $N_2 \overset{h}{\to}_\sigma (\lambda z.(\lambda y.y')(zz)I)(xI) = N'$ which is head v-normal: $N \neq N'$, so the head σ- and head v-reductions are not (locally) confluent and a term may have several head v-normal forms (this example does not contradict the confluence of σ-reduction because $N' \to_\sigma N$ but by performing an internal v-reduction step, see next Definition 9).

The head v-reduction $\overset{h}{\to}_{\mathsf{v}}$ is non-deterministic not only because the head σ-reduction $\overset{h}{\to}_\sigma$ is non-deterministic, but also because the leftmost-outermost β_v-redex of a term may overlap with "one of its leftmost" σ_1- or σ_3-redexes, as seen in Example 2.

▶ **Definition 9** (Internal v-reduction). *The* internal v-reduction $\xrightarrow{int}_{\mathsf{v}}$ *is the binary relation on* Λ *defined inductively by the following rules:*

$$\frac{(m \in \mathbb{N}) \qquad N \to_{\mathsf{v}} N'}{(\lambda x.N)M_1 \dots M_m \xrightarrow{int}_{\mathsf{v}} (\lambda x.N')M_1 \dots M_m}\ \lambda \qquad \frac{(m \in \mathbb{N}) \qquad N \xrightarrow{int}_{\mathsf{v}} N'}{VNM_1 \dots M_m \xrightarrow{int}_{\mathsf{v}} VN'M_1 \dots M_m}\ \mathit{right}$$

$$\frac{(m \in \mathbb{N}^+) \qquad M_i \to_{\mathsf{v}} M_i' \quad \text{for some } 1 \leq i \leq m}{VNM_1 \dots M_i \dots M_m \xrightarrow{int}_{\mathsf{v}} VNM_1 \dots M_i' \dots M_m}\ @\ .$$

The following fact collects many minor properties which can be easily proved by inspection of the rules of Definitions 7-9.

▶ **Fact 10.**

1. *The head* β_v*-reduction* $\xrightarrow{h}_{\beta_v}$ *does not reduce a value (in particular, does not reduce under* λ*'s), i.e., for any* $M \in \Lambda$ *and any* $V \in \Lambda_v$*, one has* $V \not\xrightarrow{h}_{\beta_v} M$*.*
2. *The head* σ*-reduction* $\xrightarrow{h}_{\sigma}$ *does neither reduce a value nor reduce to a value, i.e., for any* $M \in \Lambda$ *and any* $V \in \Lambda_v$*, one has* $V \not\xrightarrow{h}_{\sigma} M$ *and* $M \not\xrightarrow{h}_{\sigma} V$*.*
3. *Values are closed under* $\xrightarrow{int}_{\mathsf{v}}$*-expansion, i.e., for all* $M \in \Lambda$ *and* $V \in \Lambda_v$*, if* $M \xrightarrow{int}_{\mathsf{v}} V$ *then* $M \in \Lambda_v$*; more precisely,* $M = \lambda x.N$ *and* $V = \lambda x.N'$ *for some* $N, N' \in \Lambda$ *where* $N \to_{\mathsf{v}} N'$*.*
4. *If* $\mathsf{R} \in \{\xrightarrow{h}_{\beta_v}, \xrightarrow{h}_{\sigma}, \xrightarrow{h}_{\mathsf{v}}, \xrightarrow{int}_{\mathsf{v}}\}$ *and* $M \mathrel{\mathsf{R}} M'$*, then* $MN \mathrel{\mathsf{R}} M'N$ *for any* $N \in \Lambda$*.*

Clearly, $\xrightarrow{int}_{\mathsf{v}} \subsetneq \to_{\mathsf{v}}$. Next Proposition 11 (whose proof uses Fact 10.4) relates $\xrightarrow{int}_{\mathsf{v}}$ and $\xrightarrow{h}_{\mathsf{v}}$.

▶ **Proposition 11.** *One has* $\xrightarrow{int}_{\mathsf{v}} = \to_{\mathsf{v}} \smallsetminus \xrightarrow{h}_{\mathsf{v}}$*.*

Proof.

$\subseteq$: The proof that $\xrightarrow{int}_{\mathsf{v}} \subseteq \to_{\mathsf{v}}$ is trivial. The proof that $M \xrightarrow{int}_{\mathsf{v}} M'$ implies $M \not\xrightarrow{h}_{\mathsf{v}} M'$ is by induction on the derivation of $M \xrightarrow{int}_{\mathsf{v}} M'$. Let us consider its last rule r. If $\mathsf{r} \in \{\lambda, @\}$, then it is evident that there is no last rule to derive $M \xrightarrow{h}_{\mathsf{v}} M'$. If $\mathsf{r} = \mathit{right}$ then $M = VNM_1 \dots M_m$ and $M' = VN'M_1 \dots M_m$ with $m \in \mathbb{N}$ and $N \xrightarrow{int}_{\mathsf{v}} N'$; by induction hypothesis, $N \not\xrightarrow{h}_{\mathsf{v}} N'$ and hence there is no last rule to derive $M \xrightarrow{h}_{\mathsf{v}} M'$.

$\supseteq$: We show that $M \to_{\mathsf{v}} M'$ and $M \not\xrightarrow{h}_{\mathsf{v}} M'$ implies $M \xrightarrow{int}_{\mathsf{v}} M'$, for all $M, M' \in \Lambda$. Since $M \to_{\mathsf{v}} M'$, there exist a context $\mathtt{C}$ and terms N and N' such that $M = \mathtt{C}(\!|N|\!)$, $M' = \mathtt{C}(\!|N'|\!)$ and $N \mapsto_{\beta_v} N'$. We proceed by induction on $\mathtt{C}$.

If $\mathtt{C} = (\!|\cdot|\!)$ then $M = N \mapsto_{\beta_v} N' = M'$ and thus $M \xrightarrow{h}_{\mathsf{v}} M'$ since $\mapsto_{\beta_v} \subseteq \xrightarrow{h}_{\mathsf{v}}$, which contradicts the hypothesis.

If $\mathtt{C} = \lambda x.\mathtt{C}'$ for some context $\mathtt{C}'$, then $M \xrightarrow{int}_{\mathsf{v}} M'$ by applying the rule λ for $\xrightarrow{int}_{\mathsf{v}}$, since $\mathtt{C}'(\!|N|\!) \to_{\mathsf{v}} \mathtt{C}'(\!|N'|\!)$.

If $\mathtt{C} = \mathtt{C}'L$ for some context $\mathtt{C}'$ and term L, then $\mathtt{C}'(\!|N|\!) \to_{\mathsf{v}} \mathtt{C}'(\!|N'|\!)$ and $\mathtt{C}'(\!|N'|\!) \not\xrightarrow{h}_{\mathsf{v}} \mathtt{C}'(\!|N'|\!)$ (by Fact 10.4, since $\mathtt{C}'(\!|N|\!)L \not\xrightarrow{h}_{\mathsf{v}} \mathtt{C}'(\!|N'|\!)L$). By induction hypothesis, $\mathtt{C}'(\!|N|\!) \xrightarrow{int}_{\mathsf{v}} \mathtt{C}'(\!|N'|\!)$, then $M = \mathtt{C}'(\!|N|\!)L \xrightarrow{int}_{\mathsf{v}} \mathtt{C}'(\!|N'|\!)L = M'$ by Fact 10.4.

If $\mathtt{C} = V\mathtt{C}'$ for some context $\mathtt{C}'$ and value V, then $\mathtt{C}'(\!|N|\!) \to_{\mathsf{v}} \mathtt{C}'(\!|N'|\!)$. There are two cases:

- either $\mathtt{C}'(\!|N'|\!) \xrightarrow{h}_{\mathsf{v}} \mathtt{C}'(\!|N'|\!)$, hence $M = V\mathtt{C}'(\!|N|\!) \xrightarrow{h}_{\mathsf{v}} V\mathtt{C}'(\!|N'|\!) = M'$ by the rule *right* for $\xrightarrow{h}_{\beta_v}$ or $\xrightarrow{h}_{\sigma}$, which contradicts the hypothesis;
- or $\mathtt{C}'(\!|N'|\!) \not\xrightarrow{h}_{\mathsf{v}} \mathtt{C}'(\!|N'|\!)$, hence $\mathtt{C}'(\!|N'|\!) \xrightarrow{int}_{\mathsf{v}} \mathtt{C}'(\!|N'|\!)$ by induction hypothesis, thus $M = V\mathtt{C}'(\!|N|\!) \xrightarrow{int}_{\mathsf{v}} V\mathtt{C}'(\!|N'|\!) = M'$ by applying the rule *right* for $\xrightarrow{int}_{\mathsf{v}}$.

Finally, if $\mathtt{C} = L\mathtt{C}'$ for some context $\mathtt{C}'$ and term $L \notin \Lambda_v$, then $L = VN_0 \dots N_n$ for some $n \in \mathbb{N}$, thus $M = VN_0 \dots N_n\mathtt{C}'(\!|N|\!) \xrightarrow{int}_{\mathsf{v}} VN_0 \dots N_n\mathtt{C}'(\!|N'|\!) = M'$ by the rule @ for $\xrightarrow{int}_{\mathsf{v}}$.

◀

We end this section by recalling three results proven in [7] concerning head v-reduction and internal v-reduction: they will be used to prove the main results in Sections 4-5.

The following lemma (proven in [7, Lemma 14]) shows that a head σ-reduction step can be postponed after a head β_v-reduction step, and hence every head v-reduction sequence can be rearranged into a head β_v-reduction sequence followed by a head σ-reduction sequence.

▶ **Lemma 12** (Commutation of head β_v- and head σ-reductions, see [7]).
1. *If $M \overset{h}{\to}_{\sigma} L \overset{h}{\to}_{\beta_v} N$ then there exists $L' \in \Lambda$ such that $M \overset{h}{\to}_{\beta_v} L' \overset{h}{\to}^{=}_{\sigma} N$.*
2. *If $M \overset{h}{\to}^{*}_{\mathsf{v}} M'$ then there exists $N \in \Lambda$ such that $M \overset{h}{\to}^{*}_{\beta_v} N \overset{h}{\to}^{*}_{\sigma} M'$.*

Next Lemma 13 (proven in [7, Corollary 21]) says that internal v-reduction can be shifted after head v-reductions.[2]

▶ **Lemma 13** (Postponement, see [7]). *If $M \overset{int}{\to}_{\mathsf{v}} L$ and $L \overset{h}{\to}_{\beta_v} N$ (resp. $L \overset{h}{\to}_{\sigma} N$), then there exist $L', L'' \in \Lambda$ such that $M \overset{h}{\to}^{+}_{\beta_v} L' \overset{h}{\to}^{*}_{\sigma} L'' \overset{int}{\to}^{*}_{\mathsf{v}} N$ (resp. $M \overset{h}{\to}^{*}_{\beta_v} L' \overset{h}{\to}^{*}_{\sigma} L'' \overset{int}{\to}^{*}_{\mathsf{v}} N$).*

Next Theorem 14 is one of the main result proven in [7, Theorem 22] by adapting Takahashi's method [19, 5]: any v-reduction sequence can be sequentialized into a head β_v-reduction sequence followed by a head σ-reduction sequence, followed by an internal v-reduction sequence. In ordinary λ-calculus, the well-known result corresponding to our Theorem 14 states that a β-reduction sequence can be factorized in a head reduction sequence followed by an internal reduction sequence (see for example [19, Corollary 2.6]).

▶ **Theorem 14** (Sequentialization, see [7]). *If $M \to^{*}_{\mathsf{v}} M'$ then there exist $L, N \in \Lambda$ such that $M \overset{h}{\to}^{*}_{\beta_v} L \overset{h}{\to}^{*}_{\sigma} N \overset{int}{\to}^{*}_{\mathsf{v}} M'$.*

The sequentialization of Theorem 14 imposes no order between head σ-reductions. Indeed, the example in [7, p. 10] shows that it is impossible to sequentialize them by giving way to head σ_1- or head σ_3-redexes: a head σ_1-reduction step can create a head σ_3-redex, and vice versa.

In [7, Definition 27 and Corollary 29] it has also been proven that the v-equivalence (and in particular the σ-equivalence) is contained in the call-by-value observational equivalence.

4 Head normalization

In this section we prove the first main result of our paper: Theorem 21, which studies the normalization for head v-reduction and relates it to the head β_v-reduction (i.e. the weak evaluation strategy for Plotkin's λ_v-calculus). Let us start with a preliminary remark.

▶ **Remark 15.** According to Facts 10.1-2, every $V \in \Lambda_v$ is head β_v- and head σ-normal, and hence is head v-normal. The converse does not hold: xI is head v-normal but $xI \notin \Lambda_v$.

First, we give a syntactic characterization of head v- and head β_v-normal forms.

▶ **Definition 16.** *We define the subsets Λ_a, Λ_b and Λ_c (whose elements are denoted by A, B and C respectively) of Λ as follows (for any variable x, any $V \in \Lambda_v$ and any $N \in \Lambda$):*

$$(\Lambda_a) \quad A ::= xV \mid xA \mid AN \qquad (\Lambda_b) \quad B ::= (\lambda x.N)A \qquad (\Lambda_c) \quad C ::= xV \mid VC \mid CN$$

[2] In [7, Corollary 21] there is a more informative statement of our Lemma 13, involving a notion of internal parallel reduction $\overset{int}{\Rightarrow}$. Our Lemma 13 follows immediately from [7, Corollary 21] since $\overset{int}{\to}_{\mathsf{v}} \subseteq \overset{int}{\Rightarrow} \subseteq \overset{int}{\to}^{*}_{\mathsf{v}}$.

Notice that $\Lambda_a \cup \Lambda_b \subsetneq \Lambda_c$ and $M, N \in \Lambda_c \smallsetminus (\Lambda_a \cup \Lambda_b)$ where $M = (\lambda y.\Delta)(xI)\Delta$ and $N = \Delta((\lambda y.\Delta)(xI))$ (as in Example 6). Moreover, $\Lambda_v \cap \Lambda_a = \Lambda_v \cap \Lambda_b = \Lambda_v \cap \Lambda_c = \Lambda_a \cap \Lambda_b = \emptyset$ and all terms in $\Lambda_a \cup \Lambda_b \cup \Lambda_c$ are not closed. All terms in Λ_b are β-redexes that are not β_v-redexes; all terms in Λ_a have a free "head variable" and are neither a value nor a β-redex.

▶ **Proposition 17** (Characterization of head β_v-normal forms)**.** *Let M be a term.*

1. *M is head β_v-normal and is not a λ-value if and only if $M \in \Lambda_c$.*
2. *M is head β_v-normal if and only if $M \in \Lambda_v \cup \Lambda_c$.*

Proof. Statement (2) is an immediate consequence of statement (1) and Remark 15.

$\Rightarrow$: We prove the left-to-right direction of statement (1), by induction on $M \in \Lambda$.
The case where $M \in \Lambda_v$ is impossible by hypothesis.
If $M = M_1 M_2$ (for some $M_1, M_2 \in \Lambda$) is head β_v-normal then M is not a λ-value and M_1 and M_2 are head β_v-normal, moreover either $M_1 \neq \lambda x.N$ (for any $N \in \Lambda$) or $M_2 \notin \Lambda_v$ (otherwise M would be a head β_v-redex). Therefore, there are only three cases:
- either $M_1 \notin \Lambda_v$, thus $M_1 \in \Lambda_c$ by induction hypothesis, and hence $M \in \Lambda_c$;
- or $M_1 \in \Lambda_v$ and $M_2 \notin \Lambda_v$, so $M_2 \in \Lambda_c$ by induction hypothesis, and thus $M \in \Lambda_c$;
- or M_1 is a variable and $M_2 \in \Lambda_v$, hence $M \in \Lambda_c$ (this is the base case).

$\Leftarrow$: The right-to-left direction of statement (1) can easily be proved by induction on $M \in \Lambda_c$. ◀

A consequence of Proposition 17 is that all closed head β_v-normal forms are abstractions.

▶ **Proposition 18** (Characterization of head v-normal forms)**.** *Let $M \in \Lambda$.*

1. *M is head v-normal and is neither a λ-value nor a β-redex if and only if $M \in \Lambda_a$.*
2. *M is head v-normal and is a β-redex if and only if $M \in \Lambda_b$.*
3. *M is head v-normal if and only if $M \in \Lambda_v \cup \Lambda_a \cup \Lambda_b$.*

Proof. Statement (3) is an immediate consequence of statements (1)-(2) and Remark 15.

$\Rightarrow$: We prove simultaneously the left-to-right direction of statements (1) and (2), by induction on $M \in \Lambda$. The case where $M \in \Lambda_v$ is impossible by hypothesis.
If $M = M_1 M_2$ (for some $M_1, M_2 \in \Lambda$) is head v-normal then M is not a λ-value and M_1 and M_2 are head v-normal, moreover M_1 is not a β-redex (otherwise M would be a head σ_1-redex), and either $M_1 \neq \lambda x.N$ (for any $N \in \Lambda$) or $M_2 \notin \Lambda_v$ (otherwise M would be a head β_v-redex), and either $M_1 \notin \Lambda_v$ or M_2 is not a β-redex (otherwise M would be a head σ_3-redex). There are only three cases:
- either M_1 is a variable and M_2 is not a β-redex, so M is not a β-redex; if $M_2 \in \Lambda_v$ then $M \in \Lambda_a$ (this is the base case); otherwise $M_2 \in \Lambda_a$ by induction hypothesis, so $M \in \Lambda_a$;
- or $M_1 \notin \Lambda_v$, thus M is not a β-redex and $M_1 \in \Lambda_a$ by induction hypothesis, so $M \in \Lambda_a$;
- or $M_1 = \lambda x.N$ for some $N \in \Lambda$ and M_2 is neither a λ-value nor a β-redex, so M is a β-redex, furthermore $M_2 \in \Lambda_a$ by induction hypothesis, and thus $M \in \Lambda_b$.

$\Leftarrow$: The right-to-left direction of statement (1) can easily be proved by induction on $M \in \Lambda_a$. Let us prove the right-to-left direction of statement (2): if $M \in \Lambda_b$ then $M = (\lambda x.N)A$ for some $N \in \Lambda$ and $A \in \Lambda_a$, thus M is a β-redex. For any $M' \in \Lambda$, the last rule of the derivation of $M \xrightarrow{h}_{\mathsf{v}} M'$ might be neither σ_1 nor σ_3 (because A is not a β-redex by statement 1) nor β_v (because $A \notin \Lambda_v$ by statement 1 again) nor *right* (because A is head v-normal, by statement 1 again). Therefore, M is head v-normal. ◀

As a consequence of Proposition 18, all closed head v-normal forms are abstractions.

The sets of terms Λ_a, Λ_b and Λ_c of Definition 16 enjoy the closure properties summarized in Lemma 19 below. Together with the syntactic characterizations of head β_v-normal forms

(Proposition 17) and head v-normal forms (Proposition 18), these closure properties allow one to reason about head v-reduction in spite of its non-confluence: they will be used to prove our main results, Theorems 21 and 24 and Proposition 27.

▶ **Lemma 19** (Closure properties).

1. *The set Λ_a is closed under v-internal reduction and expansion, i.e., for any $N' \in \Lambda$ and $N \in \Lambda_a$, if $N' \xrightarrow{int}_{\mathsf{v}} N$ or $N \xrightarrow{int}_{\mathsf{v}} N'$ then $N' \in \Lambda_a$.*
2. *The set Λ_b is closed under v-internal reduction and expansion, i.e., for any $N' \in \Lambda$ and $N \in \Lambda_b$, if $N' \xrightarrow{int}_{\mathsf{v}} N$ or $N \xrightarrow{int}_{\mathsf{v}} N'$ then $N' \in \Lambda_b$.*
3. *Head v-normal forms are closed under v-internal reduction and expansion, i.e., for any $N, N' \in \Lambda$ where N is head v-normal, if $N' \xrightarrow{int}_{\mathsf{v}} N$ or $N \xrightarrow{int}_{\mathsf{v}} N'$ then N' is head v-normal.*
4. *Head β_v-normal forms are closed under head σ-reduction and expansion, i.e., for any $N, N' \in \Lambda$ where N is head β_v-normal, if $N' \xrightarrow{h}_{\sigma} N$ or $N \xrightarrow{h}_{\sigma} N'$ then N' is head β_v-normal.*

Proof.

1. We show that if $N \in \Lambda_a$ and $N' \xrightarrow{int}_{\mathsf{v}} N$ (resp. $N \xrightarrow{int}_{\mathsf{v}} N'$) then $N' \in \Lambda_a$, by induction on the derivation of $N' \xrightarrow{int}_{\mathsf{v}} N$ (resp. $N \xrightarrow{int}_{\mathsf{v}} N'$). Let us consider its last rule r.
   Since $N \in \Lambda_a$ (see Definition 16), $N = xLN_1 \dots N_n$ for some $n \in \mathbb{N}$, some variable x, some $L \in \Lambda_v \cup \Lambda_a$ and some $N_1, \dots, N_n \in \Lambda$, thus $\mathsf{r} \neq \lambda$ and hence either $\mathsf{r} = \mathit{right}$ or $\mathsf{r} = @$.
   If $\mathsf{r} = \mathit{right}$ then $N' = xL'N_1 \dots N_n$ where $L' \xrightarrow{int}_{\mathsf{v}} L$ (resp. $L \xrightarrow{int}_{\mathsf{v}} L'$). Since $L \in \Lambda_v \cup \Lambda_a$, there are two cases:
   - either $L \in \Lambda_a$ and then $L' \in \Lambda_a$ by induction hypothesis, so $N' = xL'N_1 \dots N_n \in \Lambda_a$;
   - or $L \in \Lambda_v$ and then $L' \in \Lambda_v$ by Fact 10.3 (resp. Remark 4, since $\xrightarrow{int}_{\mathsf{v}} \subseteq \to_{\mathsf{v}}$), therefore $N' = xL'N_1' \dots N_n' \in \Lambda_a$.

   Finally, if $\mathsf{r} = @$ then $n \in \mathbb{N}^+$ and $N' = xLN_1 \dots N_i' \dots N_n$ for some $1 \leq i \leq n$ with $N_i' \to_{\mathsf{v}} N_i$ (resp. $N_i \to_{\mathsf{v}} N_i'$), hence $N' \in \Lambda_a$ because $xL \in \Lambda_a$.
2. We show that if $N \in \Lambda_b$ and $N' \xrightarrow{int}_{\mathsf{v}} N$ (resp. $N \xrightarrow{int}_{\mathsf{v}} N'$) then $N' \in \Lambda_b$, by induction on the derivation of $N' \xrightarrow{int}_{\mathsf{v}} N$ (resp. $N \xrightarrow{int}_{\mathsf{v}} N'$). Let us consider its last rule r. Since $N \in \Lambda_b$, then $N = (\lambda x.M)A$ for some $M \in \Lambda$ and $A \in \Lambda_a$, hence $\mathsf{r} \neq @$ because N has not the shape $VLM_1 \dots M_m$ for any $m \in \mathbb{N}^+$; therefore either $\mathsf{r} = \lambda$ or $\mathsf{r} = \mathit{right}$:
   - if $\mathsf{r} = \lambda$, then $N' = (\lambda x.M')A$ where $M' \to_{\mathsf{v}} M$ (resp. $M \to_{\mathsf{v}} M'$), hence $N' \in \Lambda_b$;
   - if $\mathsf{r} = \mathit{right}$, then $N' = (\lambda x.M)A'$ where $A' \xrightarrow{int}_{\mathsf{v}} A$ (resp. $A \xrightarrow{int}_{\mathsf{v}} A'$), thus $A' \in \Lambda_a$ by Lemma 19.1, hence $N' \in \Lambda_b$;
3. Thanks to Proposition 18.3, it is sufficient to show that if $N \in \Lambda_v \cup \Lambda_a \cup \Lambda_b$ and $N' \xrightarrow{int}_{\mathsf{v}} N$ (resp. $N \xrightarrow{int}_{\mathsf{v}} N'$) then $N' \in \Lambda_v \cup \Lambda_a \cup \Lambda_b$. If $N \in \Lambda_v$ then $N' \in \Lambda_v$ by Fact 10.3 (resp. Remark 4, since $\xrightarrow{int}_{\mathsf{v}} \subseteq \to_{\mathsf{v}}$). If $N \in \Lambda_a$ then $N' \in \Lambda_a$ by Lemma 19.1. Finally, if $N \in \Lambda_b$ then $N' \in \Lambda_b$ by Lemma 19.2.
4. By Proposition 17.2, $N \in \Lambda_v \cup \Lambda_c$. Since $M \xrightarrow{h}_{\sigma} N$ or $N \xrightarrow{h}_{\sigma} M$, $N \notin \Lambda_v$ by Fact 10.2. We prove by induction on $N \in \Lambda_c$ that $M \in \Lambda_c$. By Definition 16, there are only two cases:
   - either $N = xVN_1 \dots N_n$ for some $n \in \mathbb{N}$, variable x, $V \in \Lambda_v$ and $N_1, \dots, N_n \in \Lambda$, but this is impossible since the last rule of the derivation of $M \xrightarrow{h}_{\sigma} N$ or $N \xrightarrow{h}_{\sigma} M$ can be neither σ_1 nor σ_3 (because of the subterm xV) nor right (because of Fact 10.2);
   - or $N = VLN_1 \dots N_n$ for some $n \in \mathbb{N}$, $V \in \Lambda_v$, $L \in \Lambda_c$ and $N_1, \dots, N_n \in \Lambda$, and then there are three sub-cases, depending on the last rule r of the derivation of $M \xrightarrow{h}_{\sigma} N$ (resp. $N \xrightarrow{h}_{\sigma} M$):
     - if $\mathsf{r} = \sigma_1$ then $V = \lambda x.N'N_0$ (resp. $\lambda x.N'$) and $M = (\lambda x.N')LN_0 \dots N_n$ (resp. $M = (\lambda x.N'N_1)LN_2 \dots N_n$ with $n > 0$) for some $N', N_0 \in \Lambda$, hence $M \in \Lambda_c$;
     - if $\mathsf{r} = \sigma_3$ then $V = \lambda x.V'N'$ (resp. $L = (\lambda x.N')L'$) and $M = V'((\lambda x.N')L)N_1 \dots N_n$ (resp. $M = (\lambda x.VN')L'N_1 \dots N_n$) for some $V' \in \Lambda_v$ (resp. $L' \in \Lambda_c$) and $N' \in \Lambda$, thus $(\lambda x.N')L \in \Lambda_c$ (resp. $(\lambda x.VN')L' \in \Lambda_c$) and hence $M \in \Lambda_c$;

- if $\mathsf{r} = \mathit{right}$ then $M = VL'N_1 \dots N_n$ for some $L' \in \Lambda$ such that $L' \overset{h}{\to}_\sigma L$ (resp. $L \overset{h}{\to}_\sigma L'$), so $L' \in \Lambda_c$ by induction hypothesis, and hence $M \in \Lambda_c$. ◀

Lemma 19.4 is a formalization of the two following facts: (a) a head σ-reduction step may create a new β_v-redex but in this case it is not a head β_v-redex; (b) when $M \overset{h}{\to}_\sigma N$, the head β_v-redex of M (if any) has a residual in N which is the head β_v-redex of N.

▶ **Lemma 20.** *There exists no infinite head* v*-reduction sequence with finitely many head* β_v*-reduction steps.*

Proof. Suppose the opposite holds: then there would exist $m \in \mathbb{N}$ and an infinite sequence of terms $(M_i)_{i \in \mathbb{N}}$ such that $M_i \overset{h}{\to}_{\mathsf{v}} M_{i+1}$ for any $1 \le i \le m$, $M_m \overset{h}{\to}_{\beta_v} M_{m+1}$ and $M_i \overset{h}{\to}_\sigma M_{i+1}$ for any $i > m$ (since $\overset{h}{\to}_{\mathsf{v}} = \overset{h}{\to}_{\beta_v} \cup \overset{h}{\to}_\sigma$). But this is impossible because $\overset{h}{\to}_\sigma$ is strongly normalizing (by Proposition 5 and since $\overset{h}{\to}_\sigma \subseteq \to_\sigma$). Contradiction. ◀

Now we can state and prove our main result about head β_v- and head v-normalization.

▶ **Theorem 21** (Head normalization)**.** *Let $M \in \Lambda$. The following are equivalent:*
1. *there exists a head β_v-normal form N such that $M \simeq_{\beta_v} N$;*
2. *there exists a head* v*-normal form N such that $M \simeq_{\mathsf{v}} N$;*
3. *M is head* v*-normalizable;*
4. *M is head β_v-normalizable;*
5. *there is no* v*-reduction sequence from M with infinitely many head β_v-reduction steps;*
6. *M is strongly head* v*-normalizable.*

Proof.
(1)⇒(2) By hypothesis, there exists a head β_v-normal $N \in \Lambda$ such that $M \simeq_{\beta_v} N$, thus $M \simeq_{\mathsf{v}} N$. Since $\overset{h}{\to}_\sigma$ is strongly normalizing (by Proposition 5 and because $\overset{h}{\to}_\sigma \subseteq \to_\sigma$), there exists a head σ-normal $N' \in \Lambda$ such that $N \overset{h}{\to}{}^*_\sigma N'$, therefore $M \simeq_{\mathsf{v}} N'$ since $\overset{h}{\to}_\sigma \subseteq \to_{\mathsf{v}}$. By Lemma 19.4, N' is also head β_v-normal and hence head v-normal.
(2)⇒(3) Since $M \simeq_{\mathsf{v}} N$, there is $L \in \Lambda$ such that $M \to^*_{\mathsf{v}} L$ and $N \to^*_{\mathsf{v}} L$, by confluence of $\to_{\mathsf{v}}$ (Proposition 5). By Theorem 14, there are $M_1, M_2, N_1, N_2 \in \Lambda$ such that $M \overset{h}{\to}{}^*_{\beta_v} M_1 \overset{h}{\to}{}^*_\sigma M_2 \overset{int}{\to}{}^*_{\mathsf{v}} L$ and $N \overset{h}{\to}{}^*_{\beta_v} N_1 \overset{h}{\to}{}^*_\sigma N_2 \overset{int}{\to}{}^*_{\mathsf{v}} L$. As N is head v-normal, $N = N_1 = N_2 \overset{int}{\to}{}^*_{\mathsf{v}} L$. By Lemma 19.3, L and M_2 are v-head normal. So, $M \overset{h}{\to}{}^*_{\mathsf{v}} M_2$ with M_2 head v-normal.
(3)⇒(4) By hypothesis, there is $N \in \Lambda$ head v-normal such that $M \overset{h}{\to}{}^*_{\mathsf{v}} N$. By Lemma 12.2, there is $L \in \Lambda$ such that $M \overset{h}{\to}{}^*_{\beta_v} L \overset{h}{\to}{}^*_\sigma N$. Since N is head v-normal and in particular head β_v-normal, L is head β_v-normal according to Lemma 19.4. So M is head β_v-normalizable.
(4)⇒(5) Lemma 12.1 says that if $N \overset{h}{\to}_\sigma L \overset{h}{\to}_{\beta_v} N'$ then there exists $L' \in \Lambda$ such that $N \overset{h}{\to}_{\beta_v} L' \overset{h}{\to}{}^=_\sigma N'$; Lemma 13 and Fact 10.3 show that if $N \overset{int}{\to}_{\mathsf{v}} L \overset{h}{\to}_{\beta_v} N'$ then there exist $L', L'' \in \Lambda$ such that $N \overset{h}{\to}{}^+_{\beta_v} L' \overset{h}{\to}{}^*_\sigma L'' \overset{int}{\to}{}^*_{\mathsf{v}} N'$. Since $\to_{\mathsf{v}} = \overset{h}{\to}_{\beta_v} \cup \overset{h}{\to}_\sigma \cup \overset{int}{\to}_{\mathsf{v}}$, this means that if there is an infinite v-reduction sequence from M with infinitely many head β_v-reduction steps, then for any $n \in \mathbb{N}$ there is a head β_v-reduction sequence from M whose length is at least n. Therefore, M is not head β_v-normalizable, since the head β_v-reduction is deterministic.
(5)⇒(6) If M is not strongly head v-normalizable then there exists an infinite head v-reduction sequence. By Lemma 20, this head v-reduction (and hence v-reduction, since $\overset{h}{\to}_{\mathsf{v}} \subseteq \to_{\mathsf{v}}$) sequence has infinitely many head β_v-reduction steps.

(6)⇒(1) As M is strongly head v-normalizable, in particular is head v-normalizable, hence there exists $N \in \Lambda$ head v-normal and in particular head β_v-normal such that $M \overset{h}{\to}{}^*_{\mathsf{v}} N$. By Lemma 12.2, there exists $L \in \Lambda$ such that $M \overset{h}{\to}{}^*_{\beta_v} L \overset{h}{\to}{}^*_{\sigma} N$. Therefore $M \simeq_{\beta_v} L$ since $\overset{h}{\to}_{\beta_v} \subseteq \to_{\beta_v}$. According to Lemma 19.4, L is head β_v-normal. ◀

In Theorem 21, the equivalence (3)⇔(6) means that (weak) normalization and strong normalization are equivalent for head v-reduction (for head β_v-reduction they are trivially equivalent since the head β_v-reduction is deterministic), therefore if one is interested in studying the termination of head v-reduction, no difficulty arises from its non-determinism. The equivalence (4)⇔(3) or (4)⇔(6) says that the weak evaluation process defined for Plotkin's λ_v-calculus (the head β_v-reduction) terminates if and only if the weak evaluation process defined for λ_v^σ (the head v-reduction) terminates: σ-rules play no role in deciding the termination of a head v-reduction sequence. The equivalence (3)⇔(2) (resp. (4)⇔(1)) is the version for λ_v^σ (resp. λ_v) of a well-known theorem for ordinary λ-calculus (see for example [3, Theorem 8.3.11]): in some sense, it claims that the head v-reduction (resp. head β_v-reduction) is complete with respect to the v-equivalence (resp. β_v-equivalence). The equivalence (5)⇔(2) (resp. (5)⇔(1)) can be seen as the version for λ_v^σ (resp. λ_v) of the Quasi-Head Reduction Theorem [19, Theorem 2.10] stated by Takahashi for ordinary λ-calculus.

5 Normalization strategy and other results

Theorems 14 and 21 strengthen the idea that, in spite of non-determinism and non-confluence of head v-reduction and non-sequentiability of head σ-reduction steps, the head v-reduction can be used to define a normalization strategy for the λ_v^σ-calculus, as proven in next Theorem 24, the second main result of our paper: given a term M, one starts the (unique) head β_v-head reduction sequence from M as long as a head β_v-normal form N is reached (recall that, according to Theorem 21, a term is (strongly) head v-normalizable if and only if it is head β_v-normalizable); then, one starts a head σ-reduction sequence from N (where head σ_1- and head σ_3-reduction steps can be performed in whatever order) as long as a head σ-normal form N' is reached (such a N' always exists because $\overset{h}{\to}_\sigma$ is strongly normalizing, and it is head v-normal by Lemma 19.4); finally, one performs the internal v-reduction steps starting from N' by iterating the head β_v-reduction sequences and then the head σ-reduction sequences as above on the subterms of N', from the left to the right. More precisely:

▶ **Definition 22** (Successors path). *Let $M \in \Lambda$.*

A successor of M *is a $M' \in \Lambda$ defined by induction on $M \in \Lambda$ as follows:*

- *if M is not head β_v-normal, then M' is such that $M \overset{h}{\to}_{\beta_v} M'$;*
- *if M is head β_v-normal but not head σ-normal, then M' is such that $M \overset{h}{\to}_\sigma M'$;*
- *if M is head v-normal then:*
  - *if M is a variable then $M' = M$,*
  - *if $M = \lambda x.N$ for some $N \in \Lambda$, then $M' = \lambda x.N'$ for some successor N' of N,*
  - *if $M = NL$ for some $N, L \in \Lambda$, then either N is not v-normal and $M' = N'L$ where N' is a successor of N, or N is v-normal and $M' = NL'$ where L' is a successor of L.*

A successors path of M *is an infinite sequence $(M_i)_{i \in \mathbb{N}}$ of terms such that $M_0 = M$ and M_{i+1} is a successor of M_i, for any $i \in \mathbb{N}$.*

Clearly, for every term M there is at least one successor M' of M; moreover, this successor M' is unique when M is not head β_v-normal, since the head β_v-reduction is deterministic, and $M = M'$ when M is v-normal.

▶ **Remark 23.** Let $M \in \Lambda$ and let $(M_i)_{i\in\mathbb{N}}$ be a successors path of M.

1. For every $i \in \mathbb{N}$, there exist $0 \le j \le k \le i$ such that $M \xrightarrow{h}{}^{*}_{\beta_v} M_j \xrightarrow{h}{}^{*}_{\sigma} M_k \xrightarrow{int}{}^{*}_{\mathsf{v}} M_i$.
2. For every $i \in \mathbb{N}$, if M_i is v-normal then M_j is v-normal for any $j \ge i$.

A successors path of a term M is a call-by-value left-to-right v-evaluation strategy starting from M that can reduce under a λ only when a head v-normal from is reached. Due to the non-determinism of the head σ-reduction, a term M may have several successors paths. We cannot get rid of the non-determinism of the successors path of M because of the non-sequentiability of head σ-reductions, see p. 9 and [7, p. 10].

▶ **Theorem 24** (Normalization strategy). *Let $M \in \Lambda$. Every successors path $(M_i)_{i\in\mathbb{N}}$ of M is a normalization strategy for M, i.e. if M is v-normalizable then there exists $j,k,\ell \in \mathbb{N}$ such that $j \le k \le \ell$, M_j is head β_v-normal, M_k is head v-normal and M_ℓ is v-normal.*

Proof. Let $(M_i)_{i\in\mathbb{N}}$ be a successors path of M and $N \in \Lambda$ be such that N is v-normal and $M \to_{\mathsf{v}}^{*} N$: we prove by induction on $N \in \Lambda$ that there exist $j,k,\ell \in \mathbb{N}$ such that M_j is head β_v-normal, M_k is head v-normal and M_ℓ is v-normal.

Since M is v-normalizable, then it is head β_v-normalizable (because $\xrightarrow{h}_{\beta_v} \subseteq \to_{\mathsf{v}}$), thus there exists $j \in \mathbb{N}$ such that M_j is head β_v-normal because $\xrightarrow{h}_{\beta_v}$ is deterministic. As $\xrightarrow{h}_{\sigma}$ is strongly normalizing (by Proposition 5, since $\xrightarrow{h}_{\sigma} \subseteq \to_{\sigma}$), there exists $k \in \mathbb{N}$ with $j \le k$ such that M_k is head σ-normal. According to Lemma 19.4, M_k is also head β_v-normal, hence M_k is head v-normal. Certainly, $M_k = VN_1 \dots N_n$ for some $n \in \mathbb{N}$, $V \in \Lambda_v$ and $N_1, \dots, N_n \in \Lambda$. By confluence of $\to_{\mathsf{v}}$ (Proposition 5) and since N is v-normal and M_k is head v-normal, one has $M_k \xrightarrow{int}{}^{*}_{\mathsf{v}} N$ and hence $N = V'N'_1 \dots N'_n$ for some v-normal $V' \in \Lambda_v$ and some v-normal $N'_1, \dots, N'_n \in \Lambda$ such that $V \to_{\mathsf{v}}^{*} V'$ and $N_r \to_{\mathsf{v}}^{*} N'_r$ for any $1 \le r \le n$. By induction hypothesis, for every successors path $(V_i)_{i\in\mathbb{N}}$ of V and, for any $1 \le r \le n$, for every successors path $(L^r_i)_{i\in\mathbb{N}}$ of N^r there exist $p, p_1, \dots, p_n \in \mathbb{N}$ such that $V_p, L^1_{p_1}, \dots, L^n_{p_n}$ are v-normal: by confluence of $\to_{\mathsf{v}}$ (Proposition 5), $V_p = V'$ and $N'_r = L^r_{p_r}$ for any $1 \le r \le n$.

Let us consider the infinite sequence of terms $s = (M = M_0, \;\dots\dots\; , M_k = VN_1 \dots N_n = V_0 N_1 \dots N_n, \;\dots\dots\; , V_p N_1 \dots N_n = V'L^1_0 N_2 \dots N_n, \;\dots\dots\; , V'L^1_{p_1} N_2 \dots N_n = V'N'_1 L^2_0 \dots N_n, \;\dots\dots\; , V'N'_1 N'_2 \dots N'_n = N, N, \dots\dots)$: this is a successors path of M and, for an opportune choice of the successors paths $(V_i)_{i\in\mathbb{N}}$, $(L^1_i)_{i\in\mathbb{N}}$, $\dots$, $(L^n_i)_{i\in\mathbb{N}}$, one has that $s = (M_i)_{i\in\mathbb{N}}$, in particular there exists $\ell \in \mathbb{N}$ such that $j \le k \le \ell$ and $M_\ell = N$. ◀

In ordinary λ-calculus, the well-known theorem corresponding to our Theorem 24 is the Leftmost Reduction Theorem, see [19, Theorem 2.8] or [3, Theorem 13.2.2]. Differently from the leftmost reduction of ordinary λ-calculus, our normalization strategy is not deterministic, i.e., our Theorem 24 provides a family of normalization strategies.

Finally, we have shown at p. 7 that the head σ- and head v-reductions are not (locally) confluent and a term may have several head v-normal forms. Nevertheless, the characterization of head v-normal forms given by Proposition 18 allows us to claim that (see next Proposition 27) in some cases (of interest), more precisely when a term has a head v-normal form which is a value or an element of Λ_a, the head v-normal form is unique (Proposition 27.1): all terms having several head v-normal forms are such that all their head v-normal forms are in Λ_b. In particular, every head v-normalizable closed term has a unique head v-normal form, which is an abstraction and coincides with its head β_v-normal form (Proposition 27.2).

▶ **Remark 25.** By inspection on the rules of Definition 8, it easy to check that the head σ-reduction does not reduce to a term in Λ_a, i.e., for any $M \in \Lambda$ and $N \in \Lambda_a$, one has $M \not\xrightarrow{h}_{\sigma} N$.

Remark 25 does not hold if we replace $\xrightarrow{h}_{\sigma}$ with $\xrightarrow{h}_{\beta_v}$: for instance, $x(II) \xrightarrow{h}_{\beta_v} xI \in \Lambda_a$.

▶ **Fact 26.** *For every $N \in \Lambda_v \cup \Lambda_a$, one has $M \xrightarrow{h}{}^{*}_{\beta_v} N$ if and only if $M \xrightarrow{h}{}^{*}_{\mathsf{v}} N$.*

Proof. The left-to-right direction follows from $\xrightarrow{h}_{\beta_v} \subseteq \xrightarrow{h}_{\mathsf{v}}$. The right-to-left direction is a consequence of Lemma 12.2 and either Fact 10.2 (if $N \in \Lambda_v$) or Remark 25 (if $N \in \Lambda_a$). ◀

Fact 26 means that, given a head v-reduction sequence, the head σ-reduction plays no role not only in deciding its termination (as stated in Theorem 21), but also in reaching a particular value or term in Λ_a. Fact 26 will be used in the proof of Proposition 27.

▶ **Proposition 27** (Uniqueness of "some" head v-normal forms)**.** *Let $M \in \Lambda$ and $M \xrightarrow{h}{}^{*}_{\mathsf{v}} N$.*

1. *If $N \in \Lambda_v \cup \Lambda_a$ then, for every head v-normal $L \in \Lambda$, $M \xrightarrow{h}{}^{*}_{\mathsf{v}} L$ implies $N = L$.*
2. *If M is closed and N is head v-normal, then $M \xrightarrow{h}{}^{*}_{\beta_v} N$ and $N = \lambda x.N'$ for some $N' \in \Lambda$ such that $\mathsf{fv}(N') \subseteq \{x\}$; moreover, for any head v-normal $L \in \Lambda$, $M \xrightarrow{h}{}^{*}_{\mathsf{v}} L$ implies $N = L$.*

Proof.

1. Since $N \in \Lambda_v \cup \Lambda_a$, $M \xrightarrow{h}{}^{*}_{\mathsf{v}} N$ implies $M \xrightarrow{h}{}^{*}_{\beta_v} N$ by Fact 26. According to Proposition 18.3, N is head v-normal.
   Let $L \in \Lambda$ be head v-normal and such that $M \xrightarrow{h}{}^{*}_{\mathsf{v}} L$: by Proposition 18.3, $L \in \Lambda_v \cup \Lambda_a \cup \Lambda_b$. We claim that $L \notin \Lambda_b$. Otherwise, $L \in \Lambda_b$ and then, by confluence of $\to_{\mathsf{v}}$ there would exist $M' \in \Lambda$ such that $N \to_{\mathsf{v}}^{*} M'$ and $L \to_{\mathsf{v}}^{*} M'$. According to Proposition 11 and since N and L are head v-normal, $N \xrightarrow{int}{}^{*}_{\mathsf{v}} M'$ and $L \xrightarrow{int}{}^{*}_{\mathsf{v}} M'$. By Remark 4 (since $\xrightarrow{int}_{\mathsf{v}} \subseteq \to_{\mathsf{v}}$) and Lemma 19.1, $M' \in \Lambda_v \cup \Lambda_a$. By Lemma 19.2, $M' \in \Lambda_b$. But $\Lambda_v \cap \Lambda_b = \emptyset = \Lambda_a \cap \Lambda_b$: contradiction, therefore $L \notin \Lambda_b$.
   So, $L \in \Lambda_v \cup \Lambda_a$ and thus $M \xrightarrow{h}{}^{*}_{\beta_v} L$ by Fact 26, hence $N = L$ since $\xrightarrow{h}_{\beta_v}$ is deterministic.
2. Since M is closed, N is closed too. Hence, by Proposition 18.3, $N \in \Lambda_v$ (since the terms in $\Lambda_a \cup \Lambda_b$ are not closed) and N is not a variable, therefore $N = \lambda x.N'$ for some $N' \in \Lambda$ such that $\mathsf{fv}(N') \subseteq \{x\}$. By Fact 26, $M \xrightarrow{h}{}^{*}_{\beta_v} N$. According to Proposition 27.1, for every head v-normal $L \in \Lambda$, $M \xrightarrow{h}{}^{*}_{\mathsf{v}} L$ implies $N = L$. ◀

Recall that all head v-normal terms are head β_v-normal, since $\xrightarrow{h}_{\beta_v} \subseteq \xrightarrow{h}_{\mathsf{v}}$.

6 Conclusions and future work

In this paper, we have investigated the λ_v^σ-calculus introduced in [4], an extension of Plotkin's call-by-value λ-calculus λ_v [15] with the same syntax as λ_v (without constants) and ordinary (i.e. call-by-name) λ-calculus. The peculiarity of λ_v^σ is in its reduction rules: the v-reduction adds to Plotkin's β_v-reduction two commutation rules called σ_1 and σ_3 which unblock "hidden" β_v-redexes. We have studied the head v-reduction, a non-confluent sub-reduction of the v-reduction already introduced in [7]. We now summarize our main contributions:

1. Theorem 21 is about head v-normalization, it shows that:
   - for the head v-reduction, normalization coincides with strong normalization;
   - the head v-reduction is deeply related to Plotkin's deterministic weak evaluation strategy for λ_v (the former terminates if and only if the latter terminates);
   - both head v-reduction and weak evaluation strategy for λ_v enjoy good properties analogous to the ones of the (call-by-name) head reduction for ordinary λ-calculus.
2. Theorem 24 is about v-normalization: it proves that a top-down extension of the head v-normalization provides a family of normalization strategies for the (full) v-reduction.
3. Proposition 27 is about the uniqueness of the head v-normal form: it shows that, even if there are terms having several head v-normal forms, in some case of interest (for instance, closed terms) the head v-normal form, if any, is unique.

These results, together with the results proven in [4, 7], shows that λ_v^σ is a useful tool to study some theoretical and semantic properties of Plotkin's λ_v-calculus, for instance the notions of call-by-value solvability and potential valuability. This is hard (or impossible) to obtain directly in λ_v because of the "weakness" of Plotkin's β_v-reduction. In the case of ordinary (i.e. call-by-name) λ-calculus, head reduction and solvability are the starting point to investigate separability, semi-separability and Böhm's trees. Hence, it may reasonably be supposed that we have all the ingredients for tackling the question of separability, semi-separability and Böhm's trees in a call-by-value setting. In particular, one may reasonably hope to improve in λ_v^σ the separability theorem already proven by Paolini [12] for λ_v.

Acknowledgements. The author would like to express his gratitude to Luca Paolini and Simona Ronchi Della Rocca for discussions that inspired this work. Also the author wishes to thank Michele Pagani and the anonymous referees for many helpful comments.

References

1 Beniamino Accattoli. Proof nets and the call-by-value lambda-calculus. In Delia Kesner and Petrucio Viana, editors, *Proceedings Seventh Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2012*, volume 113 of *EPTCS*, pages 11–26, 2012.

2 Beniamino Accattoli and Luca Paolini. Call-by-Value Solvability, Revisited. In Tom Schrijvers and Peter Thiemann, editors, *Functional and Logic Programming*, volume 7294 of *Lecture Notes in Computer Science*, pages 4–16. Springer-Verlag, 2012.

3 Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundation of mathematics*. North Holland, 1984.

4 Alberto Carraro and Giulio Guerrieri. A Semantical and Operational Account of Call-by-Value Solvability. In Anca Muscholl, editor, *Foundations of Software Science and Computation Structures*, volume 8412 of *Lecture Notes in Computer Science*, pages 103–118. Springer-Verlag, 2014.

5 Karl Crary. A Simple Proof of Call-by-Value Standardization. Technical Report CMU-CS-09-137, Carnegie Mellon University, 2009.

6 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

7 Giulio Guerrieri, Luca Paolini, and Simona Ronchi Della Rocca. Standardization of a call-by-value lambda-calculus. In *To appear in the Proceedings of the 13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, 2015. Available at `http://www.pps.univ-paris-diderot.fr/~giuliog/standard.pdf`.

8 Hugo Herbelin and Stéphane Zimmermann. An Operational Account of Call-by-Value Minimal and Classical lambda-Calculus in "Natural Deduction" Form. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications*, volume 5608 of *Lecture Notes in Computer Science*, pages 142–156. Springer-Verlag, 2009.

9 Peter J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101; 158–165, 1965.

10 John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theoretical Computer Science*, 228(1–2):175–210, 1999.

11 Eugenio Moggi. Computational Lambda-Calculus and Monads. In *Logic in Computer Science*, pages 14–23. IEEE Computer Society, 1989.

12 Luca Paolini. Call-by-Value Separability and Computability. In Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi, editors, *Italian Conference in Theoretical Computer Science*, volume 2202 of *Lecture Notes in Computer Science*, pages 74–89. Springer-Verlag, 2002.

13 Luca Paolini and Simona Ronchi Della Rocca. Call-by-value Solvability. *Theoretical Informatics and Applications*, 33(6):507–534, 1999. RAIRO Series, EDP-Sciences.

14 Luca Paolini and Simona Ronchi Della Rocca. *The Parametric λ-Calculus: a Metamodel for Computation.* Texts in Theoretical Computer Science: An EATCS Series. Springer-Verlag, 2004.

15 Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

16 Laurent Regnier. *Lambda calcul et réseaux.* PhD thesis, Université Paris 7, 1992.

17 Laurent Regnier. Une équivalence sur les lambda-termes. *Theoretical Computer Science*, 126(2):281–292, April 1994.

18 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and symbolic computation*, 6(3-4):289–360, 1993.

19 Masako Takahashi. Parallel reductions in lambda-calculus. *Information and Computation*, 118(1):120–127, 1995.

Towards Modelling Actor-Based Concurrency in Term Rewriting*

Adrián Palacios and Germán Vidal

MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
`{apalacios,gvidal}@dsic.upv.es`

Abstract

In this work, we introduce a scheme for modelling actor systems within sequential term rewriting. In our proposal, a TRS consists of the union of three components: the functional part (which is specific of a system), a set of rules for reducing concurrent actions, and a set of rules for defining a particular scheduling policy. A key ingredient of our approach is that concurrent systems are modelled by terms in which concurrent actions can never occur inside user-defined function calls. This assumption greatly simplifies the definition of the semantics for concurrent actions, since no term traversal will be needed. We prove that these systems are well defined in the sense that concurrent actions can always be reduced.

Our approach can be used as a basis for modelling actor-based concurrent programs, which can then be analyzed using existing techniques for term rewrite systems.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases concurrency, actor model, rewriting

Digital Object Identifier 10.4230/OASIcs.WPTE.2015.19

1 Introduction

In this work, we consider the so called *actor model* [1] of concurrency, the model underlying programming languages like Erlang [3] or Scala [7]. In this model, a program consists of a pool of processes –the actors, each one identified by a unique process identifier– that interact by exchanging messages in an asynchronous manner. Each process has a (local) mailbox where incoming messages are stored, which is not shared with the other processes. When a process receives a message, it can update its local state, send messages, or create new actors. Here, we aim at modelling an Erlang-like language within sequential term rewriting. The basic objects of the language are variables (denoted by identifiers starting with a capital letter, e.g., $X, Y, \ldots$), atoms (denoted by a, b, ...), process identifiers –pids– (denoted by p, p′, ...), constructors (which are fixed in Erlang to lists, tuples and atoms), and defined functions (denoted by $f/n, g/m, \ldots$). In Erlang, programs are sequences of function definitions. Each function f/n is defined by a rule $f(X_1, \ldots, X_n) \to s$. where $X_1, \ldots, X_n$ are distinct variables and the body of the function, s, can be an expression, a sequence of expressions, a case distinction, message sending (e.g., $\mathrm{main}\,!\,\{\mathrm{hello}, \mathrm{world}\}$ sends a message $\{\mathrm{hello}, \mathrm{world}\}$ to the

* This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* under grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEOII2015/013. Adrián Palacios was partially supported by the the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores de la Secretaría de Estado de Investigación, Desarrollo e Innovación del Ministerio de Economía y Competitividad* under FPI grant BES-2014-069749.

process with pid main) and receiving (e.g., receive $\{A, B\} \to A$ end reads a message from the process mailbox that matches the pattern $\{A, B\}$ and returns A), pattern matching where the right-hand side can be an expression, the primitive self() (that returns the pid of the current process) or a process creation (e.g., spawn(*foo*, [1, 2]) creates a new process that executes *foo*(1, 2)).

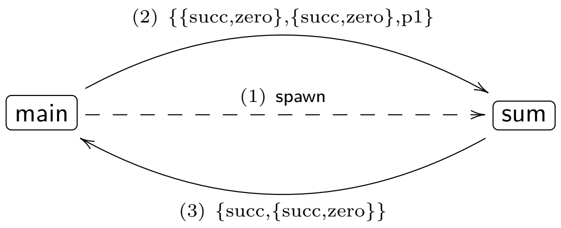
Consider, for instance, the following Erlang program:

```
main(X, Y)   →  P = spawn(sum, [ ]),
                P ! {X, Y, self()},
                receive
                    Z → Z
                end.

sum()        →  receive
                    {N, M, P} → P ! add(N, M)
                end,
                sum().

add(N, M)    →  case N of
                  zero → M
                  {succ, X} → {succ, add(X, M)}
                end.
```

Here, natural numbers are represented by zero, {succ, zero}, {succ, {succ, zero}}, etc. When we execute this program, e.g., with main({succ, zero}, {succ, zero}), the function main first spawns a new process, then sends both numbers and the own process identifier –obtained with the predefined function self()– to the new process, say {{succ, zero}, {succ, zero}, p1}, and receive suspends until some message arrives. The new process executes function sum which is initially suspended until a message arrives. When the message {{succ, zero}, {succ, zero}, p1} arrives, it calls to function add and returns the value {succ, {succ, zero}} to the process that requested the sum. Graphically,



We do not formally describe the semantics of Erlang here, but refer the reader to, e.g., [12]. Nevertheless, let us recall some relevant points about the semantics of Erlang:

- Erlang considers eager evaluation for function calls.
- The clauses of a case expression are tried in the textual order. Once the argument of a case expression matches a pattern, the remaning branches are discarded; note that this is the only sensible semantics when we have overlapping clauses (e.g., for defining default cases).
- Receive expressions proceed analogously to a case expression, but considers the *first* message in the process mailbox that matches some branch. If no one matches (or the mailbox is empty), the process' computation suspends until a new message arrives.

- As for message passing, Erlang's rule is that the direct messages between two processes should arrive in the same order they are sent. Nothing is said, though, when the messages follow different paths (i.e., when they traverse some other intermediate processes that resend the message). This is actually the source of many unexpected errors, and the reason to introduce a semantics like that of [11, 14].

In the following, we present an approach to model actor systems in the context of (sequential) rewriting, so that they can then be analyzed using existing techniques for term rewrite systems.

2 Term Rewriting

We assume familiarity with basic concepts of term rewriting. We refer the reader to, e.g., [4] for further details.

A *signature* $\mathcal{F}$ is a set of function symbols. Given a set of variables $\mathcal{V}$ with $\mathcal{F} \cap \mathcal{V} = \emptyset$, we denote the domain of *terms* by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We assume that $\mathcal{F}$ always contains at least one constant $\mathsf{f}/0$. We use $\mathsf{f}, \mathsf{g}, \ldots$ to denote functions and $x, y, \ldots$ to denote variables. A *position* p in a term t is represented by a finite sequence of natural numbers, where ϵ denotes the root position. The set of positions of a term t is denoted by $\mathcal{P}os(t)$. We let $t|_p$ denote the *subterm* of t at position p and $t[s]_p$ the result of *replacing the subterm* $t|_p$ by the term s. $\mathcal{V}ar(t)$ denotes the set of variables appearing in t. A term t is *ground* if $\mathcal{V}ar(t) = \emptyset$.

A *substitution* $\sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a mapping from variables to terms such that $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is its domain. Substitutions are extended to morphisms from $\mathcal{T}(\mathcal{F}, \mathcal{V})$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ in the natural way. We denote the application of a substitution σ to a term t by $t\sigma$ rather than $\sigma(t)$. The identity substitution is denoted by *id*.

A set of rewrite rules $l \to r$ such that l is a non-variable term and r is a term whose variables appear in l is called a *term rewriting system* (TRS for short); terms l and r are called the left-hand side (lhs) and the right-hand side (rhs) of the rule, respectively. We restrict ourselves to finite signatures and TRSs. Given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the *defined* symbols $\mathcal{D}_\mathcal{R}$ are the root symbols of the lhs's of the rules and the *constructors* are $\mathcal{C}_\mathcal{R} = \mathcal{F} \setminus \mathcal{D}_\mathcal{R}$. *Constructor terms* of $\mathcal{R}$ are terms over $\mathcal{C}_\mathcal{R}$ and $\mathcal{V}$, i.e., $\mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$. We omit $\mathcal{R}$ from $\mathcal{D}_\mathcal{R}$ and $\mathcal{C}_\mathcal{R}$ if it is clear from the context. A substitution σ is a *constructor substitution* (of $\mathcal{R}$) if $x\sigma \in \mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$ for all variables x. A TRS $\mathcal{R}$ is a *constructor system* if the lhs's of its rules have the form $\mathsf{f}(s_1, \ldots, s_n)$ where s_i are constructor terms, i.e., $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, for all $i = 1, \ldots, n$.

For a TRS $\mathcal{R}$, we define the associated rewrite relation $\to_\mathcal{R}$ as the smallest binary relation satisfying the following: given terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have $s \to_\mathcal{R} t$ iff there exist a position p in s, a rewrite rule $l \to r \in \mathcal{R}$ and a substitution σ with $s|_p = l\sigma$ and $t = s[r\sigma]_p$; the rewrite step is usually denoted by $s \to_{p,l\to r} t$ to make explicit the position and rule used in this step. A term t is called *irreducible* or in *normal form* w.r.t. a TRS $\mathcal{R}$ if there is no term s with $t \to_\mathcal{R} s$. A *derivation* is a (possibly empty) sequence of rewrite steps. Given a binary relation $\to$, we denote by $\to^*$ its reflexive and transitive closure. Thus $t \to^*_\mathcal{R} s$ means that t can be reduced to s in $\mathcal{R}$ in zero or more steps. We say that $s \overset{i}{\to}_{p,l\to r} t$ is an *innermost* rewrite step if $s \to_{p,l\to r} t$ and all proper subterms of $s|_p$ are irreducible.

3 Modelling Concurrency

In this section, we introduce our approach to model actor systems within the context of (sequential) rewriting. First, note that we do not intend to model the *semantics* of a concurrent language (as it is done, e.g., in [9], where basically an interpreter of Erlang is specified in Maude [5]), but to present a proposal for specifying actor systems in term rewriting. In particular, we aim at keeping the *functional* part of the model as independent as possible from the concurrent actions. Let us introduce first the structure of the specification a *system*:

▶ **Definition 1** (system specification structure)**.** An actor system is specified as a constructor TRS $\mathcal{R} = \mathcal{E} \cup \mathcal{A} \cup \mathcal{S}$, where $\mathcal{E}$ is the functional component, $\mathcal{A}$ specifies the evaluation of concurrent actions, and $\mathcal{S}$ defines a scheduling policy. Usually, only the functional component $\mathcal{E}$ changes from one system to another.

A given system will be modelled as a term, that is then reduced using the rules of the system specification. Let us first introduce the term representation for processes:

▶ **Definition 2** (process)**.** A process is denoted by a term $\mathsf{p}(p, t, m)$, where $\mathsf{p}/3$ is a constructor symbol, p is the process identifier (a constructor constant), t is the process' term, and m is the mailbox (a list of constructor terms).

Here, we use natural numbers (built from $\mathsf{0}$ and succ) as pids. Other data structures are possible, but we choose natural numbers for simplicity. Also, the systems have a *global mailbox* which is used to model actor systems following the semantics of Erlang. Whenever a process p sends a message t to another process p', we store a term $\mathsf{m}(p, p', t)$ in the global mailbox. It is then the scheduler who determines when and in which order these messages should be dispatched to the local mailboxes of the processes. Using this strategy, one can easily ensure that all possible interleavings are explored, which is essential for, e.g., model checking techniques. A similar strategy is presented in [11, 14].

A system is basically a *pool* of processes. In our approach, we model a system by means of a *defined* function –so systems are reducible– as follows:

▶ **Definition 3** (system)**.** A system is denoted by a term $\mathsf{s}(k, m, procs)$, where $\mathsf{s}/3$ is a defined symbol, k is a natural number (used to produce fresh pids), m is a global mailbox of the system, and $procs$ is a pool of processes.

In principle, there are several ways in which we can specify a *pool* of processes. For instance, we can introduce a constructor symbol $\odot/2$ and assume that it is associative and commutative (AC). Using an AC symbol greatly simplifies the presentation of the rules for concurrent actions. Note that, in this case, the system rules below are modelling all possible interleavings. In practice, though, we will be interested only in some of them. For instance, one can use a standard list to store the processes, and implement a simple Round Robin strategy that takes the process in the head of the list, performs a reduction step, then moves it to the end of the list, takes the next one, and so forth (this is the approach followed, e.g., in [13]). Here, for simplicity, we will consider the first approach.

For instance, an initial system could be represented by the following term:

$$\mathsf{s}(\mathsf{succ}(\mathsf{0}), [\,], \mathsf{p}(0, \mathsf{main}(t_1, t_2), [\,]))$$

where main is a call to a user-defined function with two arbitrary terms t_1 and t_2 as arguments. After a sequence of reduction steps, we may get a system like

$$\mathsf{s}(\mathsf{succ}(\mathsf{succ}(\mathsf{0})), [\,], \mathsf{p}(\mathsf{0}, \mathsf{self}(\ldots), [\,]) \odot \mathsf{p}(\mathsf{succ}(\mathsf{0}), \mathsf{sum}, [\,]))$$

with two processes with pids $\mathsf{0}$ and $\mathsf{succ}(\mathsf{0})$, such that the first one is *suspended* waiting for the concurrent action self to be processed by the system rules. The complete example will be shown in Section 3.3.

In the following, we consider each component of a system specification separately.

3.1 The Functional Component

The functional component of an actor system is defined by means of a constructor TRS $\mathcal{E}$. If the system performs no concurrent actions, then $\mathcal{E}$ is a standard TRS (and, moreover, there is no need for the other two components). In general, though, we consider four basic concurrent actions:

- self: this is the simplest action, and just returns the pid of the process.
- spawn: this action is used to create new processes. The argument of spawn is typically a function call that will start the execution of the new process. The pid of the new process is returned by the call to spawn.
- send: this action sends a message to a process identified by a pid. The call to send returns the value of the message.
- receive: this action consists in finding a message in the mailbox that matches some of the given patterns. If there is no such message (or the mailbox is empty), execution suspends.

In order to model these concurrent actions, we face a difficult problem. In a language like Erlang, expressions include *sequences* of actions and/or function calls. Moreover, as mentioned before, we do not want to specify an interpreter that traverses terms and executes both user-defined functions and concurrent actions.

Therefore, we took the following decision to avoid defining a complete interpreter for concurrent systems (i.e., in order to avoid the approach followed in [9], as mentioned before):

> *during the reduction of a system, concurrent actions must always occur in the topmost position of a process' term.*

Thanks to this requirement, one can model concurrent actions using a few simple rules (see Section 3.2 below). In order for this requirement to hold, we model concurrent actions using a sort of *continuations*. Basically, for each concurrent action, we have an associated auxiliary function that is used to bind some variables and to set the continuation after the concurrent action. This scheme will ensure that concurrent actions never occur below a user-defined constructor or defined function.

The constructor symbols used to represent concurrent actions are the following: self/2, spawn/3, send/3, and rec/2. For self/2, spawn/3 and and rec/2, we have auxiliary functions called fself/3, fspawn/3 and frec/3 (where the prefix f stands for defined "function", in contrast to constructor), which are indexed by a constructor constant. For send/3, an auxiliary function is not needed since no additional bindings are necessary, and the continuation is set in the third argument of the constructor function. Concurrent actions are reduced to a term, but they also produce some side effect (e.g., creating a new process or sending a message). For instance, in order to model a concurrent action, say $p = \mathsf{self}()$ followed by the evaluation of a term t, we should have a term of the form $\mathsf{self}(\mathsf{f1}, vs)$, together with a rule

$$\mathsf{fself}(\mathsf{f1}, vs, p) \to t$$

The rules for the evaluation of concurrent actions will be responsible of reducing $\mathsf{self}(\mathsf{f1}, vs)$ to $\mathsf{fself}(\mathsf{f1}, vs, k)$, where k is the pid of the process where $\mathsf{self}(\mathsf{f1}, vs)$ occurs. The situation is similar for spawn and rec. The case of send is slightly different since no variables need to be bound and, thus, the continuation term is just an argument of send. E.g., sending a term t to a process with pid p, and then continue with the evaluation of t' is written as $\mathsf{send}(p, t, t')$. The rules for concurrent actions will then rewrite $\mathsf{send}(p, t, t')$ to t' also storing the message in the global mailbox.

$$
\begin{array}{rcl}
\mathsf{main}(x,y) & \to & \mathsf{spawn}(\mathsf{main1},[x,y],\mathsf{sum}) \\
\mathsf{fspawn}(\mathsf{main1},[x,y],p) & \to & \mathsf{self}(\mathsf{main1},[x,y,p]) \\
\mathsf{fself}(\mathsf{main1},[x,y,p],s) & \to & \mathsf{send}(p,\mathsf{d}(x,y,s), \\
 & & \quad \mathsf{rec}(\mathsf{main1},[x,y]) \\
 & &) \\
\mathsf{frec}(\mathsf{main1},[x,y],z) & \to & z \\
 & & \\
\mathsf{sum} & \to & \mathsf{rec}(\mathsf{sum1},[\,]) \\
\mathsf{frec}(\mathsf{sum1},[\,],\mathsf{d}(n,m,p)) & \to & \mathsf{send}(p,\mathsf{add}(n,m), \\
 & & \quad \mathsf{sum} \\
 & &) \\
 & & \\
\mathsf{frec}(h,vs,t) & \to & \mathsf{no\_match}(h,vs) \\
 & & \\
\mathsf{add}(0,m) & \to & m \\
\mathsf{add}(\mathsf{succ}(n),m) & \to & \mathsf{succ}(\mathsf{add}(n,m))
\end{array}
$$

Figure 1 Functional component $\mathcal{E}$ of an actor system.

Let us briefly explain the constructor symbols introduced to model concurrent actions:[1]

- $\mathsf{self}(h, vars)$, where h is a constructor constant that is used to identify the associated continuation function. This expression will be rewritten –by the rules of $\mathcal{A}$– to $\mathsf{fself}(h, vars, k)$, where k is the pid of the current process. Here, $vars$ is a list of variables that must be passed to the auxiliary function.
- $\mathsf{spawn}(h, vars, t)$, where h is a constructor constant that is used to identify the associated continuation function, $vars$ are the variables that must be passed to this auxiliary function, and t is the term that must be evaluated by the new process. As we will see, this term will be replaced by $\mathsf{fspawn}(h, vars, k)$, where k is a fresh pid; also, a new process will be added to the pool of processes, initialized with t and with pid k.
- $\mathsf{send}(p, t_1, t_2)$, where p is a pid and t_1, t_2 are terms. This action will be replaced with t_2 –the continuation– and will add t_1 to the mailbox of the process with pid p.
- $\mathsf{rec}(h, vars)$, where h is a constructor constant that is used to identify the associated continuation function and $vars$ are the variables that must be passed to this auxiliary function. This is the most complex action. It is replaced with $\mathsf{frec}(h, vars, m)$ where m is the first message in the process' mailbox. If there is no matching rule, the next message will be tried, and so forth.

Note that we do not introduce a special symbol for case expressions. Here, we assume that case expressions can be modelled by means of ordinary rewrite rules (as in, e.g., [13]).

For instance, the functional component $\mathcal{E}$ of the Erlang program shown in Section 1 can be specified in our context with the rewrite rules of Figure 1. Here, the rules of function add are essentially the same as in the original Erlang program since they contained no concurrent actions. Functions main and sum now use a number of auxiliary functions to deal with concurrent actions so that we get four rules for specifying the original function main and three more rules for specifying the original function sum. Moreover, some Erlang data structures are now represented using constructor functions. For instance, the three element

[1] Note, however, that the rules that deal with concurrent actions will be introduced in Section 3.2.

$$\mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{self}(h, vs), ms') \odot ps) \to \mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{fself}(h, vs, p), ms') \odot ps)$$

$$\begin{aligned}\mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{spawn}(h, vs, t), ms') \odot ps) \to\ & \mathsf{s}(\mathsf{succ}(k), ms, \mathsf{p}(p, \mathsf{fspawn}(h, vs, k), ms') \\ & \odot\ \mathsf{p}(k, t, [\,]) \odot ps)\end{aligned}$$

$$\mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{send}(p', t, t'), ms') \odot ps) \to \mathsf{s}(k, ms{+\!+}[\mathsf{m}(p, p', t)], \mathsf{p}(p, t', ms') \odot ps)$$

$$\begin{aligned}\mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{rec}(h, vs), m : ms') \odot ps) &\to \mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{frec}(h, vs, m), ms') \odot ps)\\ \mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{no\_match}(h, vs), m : ms') \odot ps) &\to \mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{frec}(h, vs, m), ms') \odot ps)\\ \mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{no\_match}(h, vs), [\,]) \odot ps) &\to \mathsf{s}(k, ms, \mathsf{p}(p, \mathsf{rec}(h, vs), [\,]) \odot ps)\end{aligned}$$

Figure 2 Concurrent component $\mathcal{A}$: the system rules.

tuples of the original program are now represented using the constructor function $\mathsf{d}(\_, \_, \_)$. Specifying a system in this style might seem difficult, but one can generate it automatically from some higher-level language (as it is done in [13] for a similar formalism). The purpose of this paper, though, is to describe and analyze the properties of the modelling language, independently of the way the rules are produced.

The language could be more expressive by introducing additional constructs (e.g., a let expression to avoid duplicating the same term). Nevertheless, here we prefer to keep the previous minimal set of actions for simplicity.

3.2 Concurrent Actions

In this section, we present the rules of the second component $\mathcal{A}$ of an actor system. As mentioned before, we assume that concurrent actions always occur in the topmost position of a process' term, which greatly simplify the rules that deal with concurrent actions.

Let us now introduce the second component, $\mathcal{A}$, with the rules that define the evaluation of concurrent actions. Here, we assume that $\odot$ is an AC operator, so that $\mathsf{p}(p, t, m) \odot ps$ denotes a (non-empty) pool of processes, where $\mathsf{p}(p, t, m)$ is an arbitrary process.

▶ **Definition 4** (concurrent actions rules)**.** The component $\mathcal{A}$ for concurrent actions is given by the rewrite rules of Figure 2.[2]

Let us briefly explain the system rules. The first rule deals with the concurrent action self (a constructor), which is then replaced by a function call to fself also adding the pid p of the current process. The second rule first performs a *side effect* by creating a new process with pid k, and then replaces the constructor spawn with a call to the function fspawn where the new pid is also passed as argument. The third rule adds a message to the global mailbox as a side effect and, then, replaces the constructor send with the *continuation* t'. The fourth rule replaces the constructor rec by a call to function frec where the first message of the local mailbox is also passed as argument. The last two rules are used when the message does not match any pattern in the receive construct (which is denoted with the constructor

[2] We use Haskell's infix notation for lists, where $[\,]$ denotes the empty list and $x : xs$ a list with head element x and tail xs. When the number of elements is fixed, we also use the notation $[t_1, \ldots, t_n]$. Moreover, we use the operator $++$ for list concatenation.

$\mathsf{no\_match}$), and we can either perform a new call to function frec passing the next message in the mailbox, or suspend the evaluation when the mailbox is empty. Observe that in Figure 1 we have a rule

$$\mathsf{frec}(h, vs, t) \to \mathsf{no\_match}(h, vs)$$

This rule is used as a *default* case when a call to function frec with a message does not match any of the patterns in the previous rules. In order to correctly model this behavior, one should ensure that once a redex matches a rule, all other rules are discarded (this is further discussed below in Section 3.4).

3.3 The Scheduler

Finally, we present the specification of the third component, $\mathcal{S}$, the scheduler. These rules take care of choosing a message from the global mailbox and dispatch it to the corresponding process. Here, we show a trivial scheduler that just dispatches the messages in the same order they are sent:

$$\mathsf{s}(k, \mathsf{m}(p, p', t) : ms, \mathsf{p}(p', t', ms') \odot ps) \to \mathsf{s}(k, ms, \mathsf{p}(p', t', ms'{+\!+}[t]) \odot ps)$$

In [14], for instance, we propose to first normalize a system before applying the scheduling rules. Here, we achieve the same effect by only applying the rules of the scheduler when no previous rule from $\mathcal{A}$ is applicable. Of course, more complex schedulers are possible. In particular, we might be interested in nondeterministically exploring all possible interleavings, as it is common in the context of model checking.

Consider now a system defined by the functional component shown in Figure 1 and the rules of Sections 3.2 and 3.3. Here, we consider the following initial system:

$$\mathsf{s}(1, [\,], \mathsf{p}(0, \mathsf{main}(\mathsf{succ}(0), \mathsf{succ}(0)), [\,]))$$

For clarity, we underline the selected redex at each reduction step; besides, we denote pids with 0,1,2,... instead of 0, $\mathsf{succ}(0)$, ... Moreover, we use different colors for each element of the system: green for the global mailbox, blue for the first process with pid 0 and red for the second process with pid 1. Then, reduction proceeds as shown in Figure 3. In this derivation, we have marked with $\overset{\alpha}{\to}$ the reduction steps where the scheduler dispatches a message to a local mailbox. Observe that this rule is only applied when no other rule is applicable.

Observe that the normal form contains defined functions –the case of s– and concurrent actions –the case of rec–. This is not unusual, since in general actor systems run forever (e.g., following a client-server architecture).

3.4 Intended Semantics

In this section, we briefly discuss the intended semantics for action systems specified with TRSs. First, we consider that reductions are performed using leftmost innermost rewriting, which essentially coincides with the reduction strategy of the functional and concurrent language Erlang. However, in order to precisely model the semantics of an Erlang-like language, one should further assume that

only the first rewrite rule that matches a redex is considered.

This is a strong requirement, but it is essential to be able to express the semantics of case and receive expressions. Overlapping cases –or default cases where the pattern is just a variable–

$$\begin{array}{ll}
 & \mathsf{s}(1,[\,],\mathsf{p}(0,\underline{\mathsf{main}(\mathsf{succ}(0),\mathsf{succ}(0))},[\,])) \\
\to & \underline{\mathsf{s}(1,[\,],\mathsf{p}(0,\mathsf{spawn}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)],\mathsf{sum}),[\,]))} \\
\to & \mathsf{s}(2,[\,],\mathsf{p}(0,\underline{\mathsf{fspawn}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)],1)},[\,]) \odot \mathsf{p}(1,\mathsf{sum},[\,])) \\
\to & \underline{\mathsf{s}(2,[\,],\mathsf{p}(0,\mathsf{self}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0),1]),[\,]) \odot \mathsf{p}(1,\mathsf{sum},[\,]))} \\
\to & \mathsf{s}(2,[\,],\mathsf{p}(0,\underline{\mathsf{fself}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0),1],0)},[\,]) \odot \mathsf{p}(1,\mathsf{sum},[\,])) \\
\to & \mathsf{s}(2,[\,],\mathsf{p}(0,\underline{\mathsf{send}(1,\mathsf{d}(\mathsf{succ}(0),\mathsf{succ}(0),0),\ldots)},[\,]) \odot \mathsf{p}(1,\mathsf{sum},[\,])) \\
\to & \mathsf{s}(2,[\mathsf{m}(0,1,\mathsf{d}(\mathsf{succ}(0),\mathsf{succ}(0),0))],\mathsf{p}(0,\mathsf{rec}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)]),[\,]) \odot \mathsf{p}(1,\underline{\mathsf{sum}},[\,])) \\
\to & \underline{\mathsf{s}(2,[\mathsf{m}(0,1,\mathsf{d}(\mathsf{succ}(0),\mathsf{succ}(0),0))],\mathsf{p}(0,\mathsf{rec}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)]),[\,])} \\
 & \underline{\odot \mathsf{p}(1,\mathsf{rec}(\mathsf{sum1},[\,]),[\,]))} \\
\overset{\alpha}{\to} & \underline{\mathsf{s}(2,[\,],\mathsf{p}(0,\mathsf{rec}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)]),[\,]) \odot \mathsf{p}(1,\mathsf{rec}(\mathsf{sum1},[\,]),[\mathsf{d}(\mathsf{succ}(0),\mathsf{succ}(0),0)]))} \\
\to & \mathsf{s}(2,[\,],\mathsf{p}(0,\mathsf{rec}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)]),[\,]) \odot \mathsf{p}(1,\underline{\mathsf{frec}(\mathsf{sum1},[\,],\mathsf{d}(\mathsf{succ}(0),\mathsf{succ}(0),0))},[\,])) \\
\to & \mathsf{s}(2,[\,],\mathsf{p}(0,\mathsf{rec}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)]),[\,]) \odot \mathsf{p}(1,\mathsf{send}(0,\underline{\mathsf{add}(\mathsf{succ}(0),\mathsf{succ}(0))},\mathsf{sum}),[\,])) \\
\to & \mathsf{s}(2,[\,],\mathsf{p}(0,\mathsf{rec}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)]),[\,]) \odot \mathsf{p}(1,\mathsf{send}(0,\mathsf{succ}(\underline{\mathsf{add}(0,\mathsf{succ}(0))}),\mathsf{sum}),[\,])) \\
\to & \underline{\mathsf{s}(2,[\,],\mathsf{p}(0,\mathsf{rec}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)]),[\,]) \odot \mathsf{p}(1,\mathsf{send}(0,\mathsf{succ}(\mathsf{succ}(0)),\mathsf{sum}),[\,]))} \\
\to & \mathsf{s}(2,[\mathsf{m}(1,0,\mathsf{succ}(\mathsf{succ}(0)))],\mathsf{p}(0,\mathsf{rec}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)]),[\,]) \odot \mathsf{p}(1,\underline{\mathsf{sum}},[\,])) \\
\to & \underline{\mathsf{s}(2,[\mathsf{m}(1,0,\mathsf{succ}(\mathsf{succ}(0)))],\mathsf{p}(0,\mathsf{rec}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)]),[\,]) \odot \mathsf{p}(1,\mathsf{rec}(\mathsf{sum1},[\,]),[\,]))} \\
\overset{\alpha}{\to} & \underline{\mathsf{s}(2,[\,],\mathsf{p}(0,\mathsf{rec}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)]),[\mathsf{succ}(\mathsf{succ}(0))]) \odot \mathsf{p}(1,\mathsf{rec}(\mathsf{sum1},[\,]),[\,]))} \\
\to & \mathsf{s}(2,[\,],\mathsf{p}(0,\underline{\mathsf{frec}(\mathsf{main1},[\mathsf{succ}(0),\mathsf{succ}(0)],\mathsf{succ}(\mathsf{succ}(0)))},[\,]) \odot \mathsf{p}(1,\mathsf{rec}(\mathsf{sum1},[\,]),[\,])) \\
\to & \mathsf{s}(2,[\,],\mathsf{p}(0,\mathsf{succ}(\mathsf{succ}(0)),[\,]) \odot \mathsf{p}(1,\mathsf{rec}(\mathsf{sum1},[\,]),[\,]))
\end{array}$$

Figure 3 Example of the reduction of a system.

are common and we do not see any other viable alternative (note that it is also essential for the scheduling strategy mentioned above, where the rules of the scheduler are only applied when no other rule is applicable). Here, one may replace the default case alternatives by adding a rule for every non-matching constructor. However, this would only work as long as the number of atoms is finite (e.g., it will not work for integers). Of course, by ignoring this requirement, we would compute an overapproximation of the computations of the original system, but in general this is not a satisfactory solution (even more since a state explosion is a common problem when analyzing concurrent systems).

Another relevant point is that, for a system to be reduced using only the concurrent action rules $\mathcal{A}$ and the scheduling rules $\mathcal{S}$, one should further require that

> *in the rules of $\mathcal{E}$, the concurrent actions* $\mathsf{self}/2$, $\mathsf{spawn}/3$, $\mathsf{send}/3$, *and* $\mathsf{rec}/2$ *do not occur below a user-defined constructor or function symbol.*

In practice, these constructor symbols either occur in the topmost position of the right-hand side or in the *continuation* argument of $\mathsf{send}/3$. Moreover, we say that a term is *safe* if it contains no occurrences of $\mathsf{self}/2$, $\mathsf{spawn}/3$, $\mathsf{send}/3$ or $\mathsf{rec}/2$ below a user-defined constructor or defined function. A TRS is safe if the right-hand sides of all its rules are safe. This static condition ensures that the rules of $\mathcal{A} \cup \mathcal{S}$ suffice to deal with concurrent actions since they can only occur in the topmost position of a process and, thus, the rules of Figure 2 suffice.

▶ **Theorem 5.** *Let $\mathcal{R} = \mathcal{E} \cup \mathcal{A} \cup \mathcal{S}$, where $\mathcal{E}$ is a safe constructor TRS, and $\mathcal{A}$ and $\mathcal{S}$ are defined as in Sections 3.2 and 3.3, respectively. Let t_0 be a term of the form* $\mathsf{s}(1,[],\mathsf{p}(0,\mathsf{f}(t_1,\ldots,t_n),[]),$

where f *is a defined function of* $\mathcal{E}$ *and* $t_1, \ldots, t_n$ *are purely functional terms.*[3] *Then, for all* $t_0 \overset{i}{\to}{}^* t$ *we have that* t *is a safe term.*

Proof. We prove the claim by induction on the length of the reduction $t_0 \overset{i}{\to}{}^* t$. Since the base case $t_0 = t$ is trivial, we now consider the inductive case.

Assume that $t_0 \overset{i}{\to}{}^* t$ and t is safe. Let us consider now the innermost reduction step $t \overset{i}{\to}_{\mathcal{R}} t'$. First, we consider the case $t \overset{i}{\to}_{p,l\to r} t'$ with $l \to r \in \mathcal{E}$. By the inductive hypothesis, we have that t is safe and, thus, both $t[\,]_p$ and $t|_p = l\sigma$ are safe. Since r is also safe, we have that $t[r]_p$ is safe. It only remains to show that σ cannot introduce unsafe terms. Here, we know that $t|_p$ is rooted by a defined function of $\mathcal{E}$. Therefore, since $t|_p$ is safe, there are no occurrences of $\mathsf{self}/2$, $\mathsf{spawn}/3$, $\mathsf{send}/3$ or $\mathsf{rec}/2$. Hence, σ cannot introduce any occurrence of $\mathsf{self}/2$, $\mathsf{spawn}/3$, $\mathsf{send}/3$ or $\mathsf{rec}/2$ too, and $t[r\sigma]_p = t'$ is thus safe.

The case when $t \overset{i}{\to}_{p,l\to r} t'$ with $l \to r \in \mathcal{A} \cup \mathcal{S}$ is immediate by definition of $\mathcal{A} \cup \mathcal{S}$. ◀

4 Related Work

In this section, we review some related work on modelling actor-based concurrent systems in a language based on term rewriting. First, Giesl and Arts [6] deal with the verification of Erlang processes using dependency pairs. They transform Erlang programs to conditional rewrite systems by hand, so that termination can be analyzed. Another related approach –though for different source and target languages– is that of Albert *et al* [2], where a transformation from a concurrent object-oriented programming language based on message passing to a rule-based logic-like programming language is introduced.

As mentioned in the introduction, there are some approaches where the goal is to define a sort of interpreter in a rewriting based language like Maude [5]. This is the case of [9], who introduces an implementation of Erlang in *rewriting logic* [8], a unified semantic framework for concurrency. In this approach, Erlang programs are seen as data objects manipulated by a sort of interpreter implemented in rewriting logic. In contrast, we aim at specifying plain rewrite systems that can be analyzed using existing technologies.

This work can be seen as a continuation of [13]. While [13] focused on the transformation from programs written in a subset of Erlang to rewrite systems, in this work we focused on the modelling language within the term rewriting setting. The long-term goal is the definition of a modelling language that is rich enough to specify the main features of Erlang or Scala programming languages, so that program analysis and transformations can be designed. For instance, [13] already includes a preliminary deadlock analysis based on *narrowing* [10], an extension of rewriting to deal with logic variables.

5 Discussion

In this work, we have introduced a scheme for modelling actor-based concurrent systems in term rewriting. Our approach can be used as a basis for modelling Erlang-like programs, which can then be analyzed using existing techniques for term rewrite systems. For instance, [13] presents a translation scheme for Erlang programs to a specification which is closer to the one introduced in this paper. The transformation can be tested using the web interface from `http://kaz.dsic.upv.es/erlang2trs/`. A similar transformation can be defined from a subset of Erlang to the actor systems as specified in this paper.

[3] A *purely functional* term has no occurrences of $\mathsf{self}/2$, $\mathsf{spawn}/3$, $\mathsf{send}/3$, $\mathsf{rec}/2$, $\mathsf{p}/3$, $\mathsf{m}/3$, $\odot/2$ nor $\mathsf{s}/3$.

As a future work, we consider two main directions. On the first hand, we plan to investigate rewriting strategies to model the fact that only the first matching rule should be considered. This is the main difference with standard rewriting. For instance, one could define a transformation to the actor system –a sort of completion procedure– so that it can be reduced by standard rewriting with the desired behavior. On the other hand, and in order to be able to model integers, arithmetic operations, etc., we plan to extend the modelling language within the framework of integer rewriting. This will allow us to produce more precise models for programs written in, e.g., Erlang.

Acknowledgements. We thank the anonymous reviewers for their useful comments to improve this paper.

References

1 G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

2 E. Albert, P. Arenas, and M. Gómez-Zamalloa. Symbolic Execution of Concurrent Objects in CLP. In *PADL'12*, pages 123–137. Springer LNCS 7149, 2012.

3 Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG.* Prentice Hall, 1993.

4 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

5 M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C.L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic.* Springer LNCS 4350, 2007.

6 Jürgen Giesl and Thomas Arts. Verification of Erlang Processes by Dependency Pairs. *Appl. Algebra Eng. Commun. Comput.*, 12(1/2):39–72, 2001.

7 Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

8 José Meseguer. Conditioned Rewriting Logic as a United Model of Concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

9 Thomas Noll. A Rewriting Logic Implementation of Erlang. *Electr. Notes Theor. Comput. Sci.*, 44(2):206–224, 2001.

10 James R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

11 H. Svensson, L.-A. Fredlund, and C. Benac Earle. A unified semantics for future Erlang. In *Proc. of the 9th ACM SIGPLAN workshop on Erlang*, pages 23–32. ACM, 2010.

12 Hans Svensson and Lars-Ake Fredlund. A more accurate semantics for distributed Erlang. In Simon J. Thompson and Lars-Ake Fredlund, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang*, pages 43–54. ACM, 2007.

13 G. Vidal. Towards Erlang Verification by Term Rewriting. In *LOPSTR'13*, pages 109–126. Springer LNCS 8901, 2014.

14 G. Vidal. Towards Symbolic Execution in Erlang (short paper). In *PSI'14*, pages 351–360. Springer LNCS 8974, 2014.

Observing Success in the Pi-Calculus

David Sabel and Manfred Schmidt-Schauß

Goethe-University, Frankfurt am Main
`{sabel,schauss}@ki.cs.uni-frankfurt.de`

Abstract

A contextual semantics – defined in terms of successful termination and may- and should-convergence – is analyzed in the synchronous pi-calculus with replication and a constant Stop to denote success. The contextual ordering is analyzed, some nontrivial process equivalences are proved, and proof tools for showing contextual equivalences are provided. Among them are a context lemma and new notions of sound applicative similarities for may- and should-convergence. A further result is that contextual equivalence in the pi-calculus with Stop conservatively extends barbed testing equivalence in the (Stop-free) pi-calculus and thus results on contextual equivalence can be transferred to the (Stop-free) pi-calculus with barbed testing equivalence.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages, D.3.1 Formal Definitions and Theory F.4.2 Grammars and Other Rewriting Systems,

Keywords and phrases concurrency, process calculi, pi-calculus, rewriting, semantics

Digital Object Identifier 10.4230/OASIcs.WPTE.2015.31

1 Introduction

The π-calculus [9, 8, 20] is a well-known model for concurrent processes with message passing. Its minimalistic syntax includes parallel process-composition, named channels, and input/output-capabilities on the channels. The data flow in the π-calculus is programmed by communication between processes. There is a lot of research on several bisimulations of π-processes (see e.g. [20, 10]). They equate processes if testing the processes (using reduction) exhibits that they have the same input and output capabilities and that they reach equivalent states. Bisimulations occur in *strong* variants, where bisimilar processes must have an identical reduction behavior for every single reduction step, and there are *weak* bisimulations, where the numbers of internal reduction need not coincide, but equivalent states w.r.t. the reflexive-transitive closure of reduction must be reached.

While proving processes bisimilar is often easy and elegant, the bisimilarities (in weak and strong variants) are very fine grained notions, and thus may not allow to equate processes even if they can be seen as semantically equal. We are interested in coarser notions of process equivalences as the *semantic base* and view bisimulations (provided that they are sound w.r.t. the semantics) as very helpful proof tools for investigating the (contextual) semantics.

For program calculi based on the lambda-calculus the usual approach to program equivalence is Morris style-contextual equivalence [11, 15] which can be used in a uniform way for a lot of those calculi. For deterministic languages, contextual equivalence is based on the notion of a terminated (or successful) program and it equates programs, if the ability to terminate (i.e. so-called *may-convergence*) is indistinguishable when exchanging one program by the other in any surrounding program context. For non-deterministic languages this equivalence is too coarse, but it can be strengthened by additionally observing whether the program successfully terminates on all execution paths, i.e. whether the program *must-converges*, or as a slightly different approach, whether the program *should-converges*, which holds, if the property of being *may-convergent* holds on all execution paths. In contrast to must-convergence, should-convergence (see e.g. [12, 2, 16, 21]) has some kind of fairness built-in: the predicate does not change even if instead of all reduction sequences only *fair* ones are taken into account, where fair reduction sequences ensure that every reducible expression is reduced after finitely many reduction steps.

In this paper we analyze contextual equivalence in the synchronous π-calculus with replication and – for simplicity – without sums. Since the π-calculus has no notion of successful termination, we use the same approach as e.g. [6, 13] and add a syntactic construct (the constant Stop) which indicates successful termination. We call the extended calculus Π_{Stop}. We develop two proof tools for showing program equivalences: We prove that a context lemma holds, which restricts the required class of contexts to show contextual equivalence. We introduce notions of applicative similarities for the may- and the should-convergence. There are soundness results on bisimilarities and barbed may- and should-testing for the asynchronous π-calculus in [5], but to the best of our knowledge our notion of an applicative similarity for should-convergence is new. We prove soundness of these similarities w.r.t. the contextual preorders and thus they can be used for co-inductive proofs to show contextual equivalence. Even though the test for may-convergence is subsumed by testing should-convergence, our reasoning tools require also reasoning about may-convergence, and thus we will consider both predicates. Equipped with these tools we show that process interaction is correct, if it is deterministic, prove some further process equations, and investigate the contextual ordering. We show that a contextually least element does not exist in Π_{Stop}, but a largest element exists – the constant Stop.

Our notion of contextual equivalence seems to be close to testing equivalences (see e.g. [3] for CCS, [1] for a restricted variant of the π-calculus, [5] for the asynchronous π-calculus, and [7] for the join-calculus), which are defined analogously to contextual equivalence, but instead of observing successful termination, other observations are relevant. For *barbed testing equivalences* [5] the capability of receiving (or emitting) a name on an (open) input (or output) channel is observed (i.e. the process has an input or output *barb*). There is also some work where testing is a combination of may- and must- or should-testing (which is sometimes called fair must-testing, e.g. [5]). Roughly speaking, these predicates require an input or output capability on every execution path. We prove a strong connection between contextual equivalence and barbed may- and should-testing: On Stop-free processes, barbed testing equivalence coincides with contextual equivalence. This connection enables us to transfer several of our results to the classic π-calculus without Stop.

Outline. In Sect. 2 we introduce the synchronous π-calculus with Stop. The context lemma and soundness of applicative similarity is proven in Sect. 3. We analyze the contextual ordering and prove correctness of deterministic process interaction in Sect. 4. In Sect. 5 we analyze the connection between contextual equivalence in Π_{Stop} and barbed testing equivalence in the π-calculus without Stop and transfer our results obtained in Π_{Stop} to the Stop-free calculus. Finally, we conclude in Section 6.

2 The π-Calculus Π_{Stop} with Stop

We consider the synchronous π-calculus Π_{Stop} with replication and a constant Stop. For simplicity, we neither include sums nor name matching. Let $\mathcal{N}$ be a countably infinite set of *names*. *Processes* $\mathit{Proc}_{\mathsf{Stop}}$ and *action prefixes* π are defined as follows, where $x, y \in \mathcal{N}$:

$$P,Q,R \in \mathit{Proc}_{\mathsf{Stop}} \quad ::= \quad \pi.P \mid P_1 \mathbin{\boldsymbol{|}} P_2 \mid \mathbf{!}\,P \mid \mathbf{0} \mid \nu x.P \mid \mathsf{Stop} \qquad\qquad \pi \quad ::= \quad x(y) \mid \overline{x}y$$

$P_1 \mid P_2$ is the *parallel composition* of processes P_1 and P_2. A process $x(y).P$ has the capability to *receive* some name z along the channel x and then behaves like $P[z/y]$ where $[z/y]$ is the capture free substitution of y by z. A process $\overline{x}y.P$ can *send* the name y along the channel x and thereafter it behaves like P. $\mathbf{0}$ is the *silent process* and Stop is the successful process. A *restriction* $\nu z.P$ restricts the scope of the name z to process P. The *replication* $!\,P$ represents arbitrary many parallel copies of process P. The constructs $\nu x.P$ and $y(x).P$ bind the name x with scope P which induces a notion of α-renaming and α-equivalence $=_\alpha$ as usual. We use $\mathsf{fn}(P)$ for the set of *free names* of P and adopt the distinct name convention, and assume that free names are distinct from bound names and bound names are pairwise distinct. We also use name substitutions $\sigma : \mathcal{N} \to \mathcal{N}$. With Σ we denote the set of all name substitutions.

In the remainder of the paper, we use several binary relations on processes. Given a relation $\mathcal{R} \subseteq (\mathit{Proc}_{\mathsf{Stop}} \times \mathit{Proc}_{\mathsf{Stop}})$, we write $\mathcal{R}^{-1}$ for the relation $\{(Q,P) \mid (P,Q) \in \mathcal{R}\}$ and $\mathcal{R}^\sigma$ is defined as: $(P,Q) \in \mathcal{R}^\sigma$ iff $(\sigma(P),\sigma(Q)) \in \mathcal{R}$ for all $\sigma \in \Sigma$.

A *context* $C \in \mathcal{C}_{\mathsf{Stop}}$ is a process with a *hole* $[\cdot]$. Replacing the hole of C by process P is written as $C[P]$. *Structural congruence* $\equiv$ is the smallest congruence satisfying the axioms:

$$\begin{array}{rcc}
P \equiv Q, \text{ if } P =_\alpha Q & P \mid \mathbf{0} \equiv P & \nu x.\mathsf{Stop} \equiv \mathsf{Stop} \\
P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3 & P \mid Q \equiv Q \mid P & \nu z.\nu w.P \equiv \nu w.\nu z.P \\
\nu z.(P_1 \mid P_2) \equiv P_1 \mid \nu z.P_2, \text{ if } z \notin \mathsf{fn}(P_1) & \nu z.\mathbf{0} \equiv \mathbf{0} & !\,P \equiv P \mid !\,P
\end{array}$$

Processes $x(y).\mathbf{0}$ and $\overline{x}y.\mathbf{0}$ are abbreviated as $x(y)$ and $\overline{x}y$. Instead of $\nu x_1.\nu x_2.\ldots.\nu x_n.P$ we write $\nu x_1,\ldots,x_n.P$, or also $\nu\mathcal{X}.P$ if the concrete names $x_1,\ldots,x_n$ and the number $n \geq 0$ are not of interest. We also use set-notation for $\mathcal{X}$ and e.g. write $x_i \in \mathcal{X}$ with its obvious meaning. With $\mathtt{choice}(P,Q)$ we abbreviate the internal choice of two processes $\mathtt{choice}(P,Q) := \nu x,y.(x(y_1).P \mid x(y_2).Q \mid \overline{x}y)$ (where $x,y,y_1,y_2 \notin \mathsf{fn}(P \mid Q)$).

The main reduction rule of Stop expresses a synchronous communication between two process, i.e. the reduction rule $x(y).P \mid \overline{x}v.Q \xrightarrow{ia} P[v/y] \mid Q$ performs *interaction* between two processes. The standard reduction of Stop is the closure of the rule $\xrightarrow{ia}$ w.r.t. structural congruence and reduction contexts. We prefer to use reduction contexts instead of closing reduction by congruence rules for the reduction as done e.g. in [20]. However, this leads to the same notion of a standard reduction. Note that there are also approaches [19] to make even the structural transformations of $\equiv$ more deterministic (by using reduction rules), however, for our goals in this paper this does not seem to be helpful.

▶ **Definition 2.1.** *Reduction contexts* $\mathcal{D}$ are $D \in \mathcal{D} ::= [\cdot] \mid D \mid P \mid P \mid D \mid \nu x.D$. A *standard reduction* $\xrightarrow{sr}$ applies an $\xrightarrow{ia}$-reduction in a reduction context (modulo structural congruence):

$$\frac{P \equiv D[P'], P' \xrightarrow{ia} Q', D[Q'] \equiv Q, \text{ and } D \in \mathcal{D}}{P \xrightarrow{sr} Q}$$

The *redex* of an $\xrightarrow{sr}$-reduction is the subprocess $x(y).P \mid \overline{x}v.Q$ which is replaced by $P[v/y] \mid Q$. We define $\xrightarrow{sr,*} := \bigcup_{i\geq 0} \xrightarrow{sr,i}$ and $\xrightarrow{sr,+} := \bigcup_{i>0} \xrightarrow{sr,i}$ where for $P,Q \in \mathit{Proc}_{\mathsf{Stop}}$: $P \xrightarrow{sr,0} P$ and $P \xrightarrow{sr,i} Q$ if there exists $P' \in \mathit{Proc}_{\mathsf{Stop}}$ s.t. $P \xrightarrow{sr} P'$ and $P' \xrightarrow{sr,i-1} Q$.

We define contextual equivalence by observing whether a process may- or should become successful, i.e. may-observation means that the process can be reduced to a successful process, and should-observation means that the process never looses the ability to become successful.

▶ **Definition 2.2.** A process P is *successful* (denoted by $\mathsf{stop}(P)$) if $P \equiv \mathsf{Stop} \mid P'$ for some process P'. *May-convergence* $\downarrow$ is defined as $P{\downarrow}$ iff $\exists P' : P \xrightarrow{sr,*} P' \wedge \mathsf{stop}(P')$ and *should-convergence* $\Downarrow$ is defined as: $P{\Downarrow}$ iff $\forall P' : P \xrightarrow{sr,*} P' \implies P'{\downarrow}$. We write $P{\uparrow}$ (P is *may-divergent*) iff $P{\Downarrow}$ does not hold, and $P{\Uparrow}$ (P is *must-divergent*) iff $P{\downarrow}$ does not hold.

For $\xi \in \{\downarrow, \Downarrow, \uparrow, \Uparrow\}$, the *preservation preorders* $\leq_\xi$ are defined as $P \leq_\xi Q$ iff $P\xi \implies Q\xi$. *Contextual preorder* $\leq_c$ is defined as $\leq_c := \leq_{c,\downarrow} \cap \leq_{c,\Downarrow}$ where for $\xi \in \{\downarrow, \Downarrow, \uparrow, \Uparrow\}$: $P \leq_{c,\xi} Q$ iff $\forall C \in \mathcal{C}_{\mathsf{Stop}} : C[P] \leq_\xi C[Q]$. *Contextual equivalence* $\sim_c$ is defined as $\sim_c := \leq_c \cap \geq_c$.

For $\xi \in \{\downarrow, \Downarrow, \uparrow, \Uparrow\}$ we write $\geq_{c,\xi}$ for $(\leq_{c,\xi})^{-1}$ and $\sim_{c,\xi}$ for the intersection $\leq_{c,\xi} \cap \geq_{c,\xi}$.

Note that $P{\uparrow}$ is equivalent to $\exists P' : P \xrightarrow{sr,*} P' \wedge P'{\Uparrow}$ and note also that for any successful process P, any contractum P' is also successful, i.e. $\mathsf{stop}(P) \wedge P \xrightarrow{sr,*} P' \implies \mathsf{stop}(P')$.

Since reduction includes transforming processes using structural congruence, structural congruent processes are contextually equivalent:

▶ **Proposition 2.3.** *If $P \equiv Q$, then for $\xi \in \{\downarrow, \Downarrow, \uparrow, \Uparrow\}$: $P \leq_\xi Q$, $P \leq_{c,\xi} Q$, $Q \leq_\xi P$, $Q \leq_{c,\xi} P$ and thus in particular $P \sim_c Q$.*

Some easier properties are:

▶ **Lemma 2.4.** ***1.*** *If $P \xrightarrow{sr} Q$ then $\nu x.P \xrightarrow{sr} \nu x.Q$.*
2. *If $\nu x.P \xrightarrow{sr} Q$ then $P \xrightarrow{sr} Q'$ such that either $Q \equiv \nu x.Q'$ or $Q \equiv Q'$.*
3. *For $\xi \in \{\downarrow, \Downarrow, \uparrow, \Uparrow\}$: $P \leq_\xi \nu x.P$ and $\nu x.P \leq_\xi P$.*

3 Proof Methods for Contextual Equivalence

For disproving an equation $P \sim_c Q$, it suffices to find a distinguishing context. Proving an equation $P \sim_c Q$ is in general harder, since all contexts must be considered. Hence, we develop proof tools supporting those proofs. In Sect. 3.1 we show that a context lemma holds, which restricts the set of contexts that need to be taken into account for proving $P \sim_c Q$. In contrast to the co-inductive proofs given in [14, 20] for a context lemma for barbed congruence in the π-calculus, our context lemma is proved inductively and also for should-convergence. In Sect. 3.2 we show soundness of applicative similarities which permit co-inductive proofs by analyzing the input or output possibilities on open channels of P and Q. The applicative similarities are related to the "weak early bisimilarities" in the π-calculus, but there are some differences which are discussed after the definitions.

3.1 A Context Lemma for May- and Should-Convergence

As a preparation we show two extension lemmas for $\leq_\downarrow$ and $\leq_\uparrow$ w.r.t. contexts of hole depth 1, which are the contexts $[\cdot] \mid R$, $R \mid [\cdot]$, $x(y).[\cdot]$, $\overline{x}y.[\cdot]$, $\nu x.[\cdot]$, and $!\,[\cdot]$. To ease reading the proofs are given in the appendix.

▶ **Lemma 3.1.** *Let $P, Q \in Proc_{\mathsf{Stop}}$. If $\sigma(P) \mid R \leq_\downarrow \sigma(Q) \mid R$ for all $\sigma \in \Sigma$ and $R \in Proc_{\mathsf{Stop}}$, then $\sigma(C[P]) \mid R \leq_\downarrow \sigma(C[Q]) \mid R$ for all $C \in \mathcal{C}_{\mathsf{Stop}}$ of hole depth 1, all $\sigma \in \Sigma$ and $R \in Proc_{\mathsf{Stop}}$.*

▶ **Lemma 3.2.** *Let $P, Q \in Proc_{\mathsf{Stop}}$. Assume that $\sigma(C[Q]) \mid R \leq_\downarrow \sigma(C[P]) \mid R$ for all $\sigma \in \Sigma$, all $C \in \mathcal{C}_{\mathsf{Stop}}$, and all $R \in Proc_{\mathsf{Stop}}$. If $\sigma(P) \mid R \leq_\uparrow \sigma(Q) \mid R$ for all $\sigma \in \Sigma$ and $R \in Proc_{\mathsf{Stop}}$, then $\sigma(C[P]) \mid R \leq_\uparrow \sigma(C[Q]) \mid R$ for all $C \in \mathcal{C}_{\mathsf{Stop}}$ of hole depth 1, all $\sigma \in \Sigma$ and $R \in Proc_{\mathsf{Stop}}$.*

▶ **Theorem 3.3** (Context Lemma). *For all processes P, Q:*
- *If for all σ, R: $\sigma(P) \mid R \leq_\downarrow \sigma(Q) \mid R$, then $P \leq_{c,\downarrow} Q$.*
- *If for all σ, R: $\sigma(P) \mid R \leq_\downarrow \sigma(Q) \mid R \wedge \sigma(P) \mid R \leq_\Downarrow \sigma(Q) \mid R$, then $P \leq_c Q$.*

Proof. For the first part it suffices to show that $\sigma(C[P]) \mid R \leq_{\downarrow} \sigma(C[Q]) \mid R$ for all $C \in \mathcal{C}_{\mathsf{Stop}}$, $\sigma \in \Sigma$, and $R \in \mathit{Proc}_{\mathsf{Stop}}$, which follows from Lemma 3.1 by induction on the depth of the hole of the context C. For the second part we use the fact that $\sigma(P) \mid R \leq_{\downarrow} \sigma(Q) \mid R$ for all σ, R implies $\sigma(C[P]) \mid R \leq_{\downarrow} \sigma(C[Q]) \mid R$ for all σ, R, C. By induction on the depth of the hole of the context C, the fact that $\sigma(P) \mid R \leq_{\Downarrow} \sigma(Q) \mid R$ is equivalent to $\sigma(Q) \mid R \leq_{\uparrow} \sigma(P) \mid R$, and by Lemma 3.2 it follows $\sigma(C[Q]) \mid R \leq_{\uparrow} \sigma(C[P]) \mid R$ for all C, σ and thus $P \leq_{c,\Downarrow} Q$. ◀

▶ Remark. The condition for all σ, R: $\sigma(P) \mid R \leq_{\Downarrow} \sigma(Q) \mid R$ is in general not sufficient for $P \leq_{c,\Downarrow} Q$. Let $P := \nu x.\overline{x}y \mid x(y).\mathsf{Stop} \mid x(y)$ and $Q := \mathbf{0}$. Then $\sigma(P) \mid R \leq_{\Downarrow} \sigma(Q) \mid R$ for all σ and R, but $P \not\leq_{c,\Downarrow} Q$, since the context $!\,[\cdot]$ distinguishes P and Q: $!\,P\Downarrow$ while $!\,Q\Uparrow$.

3.2 Applicative Similarities

We first provide co-inductive definitions of $\leq_{\downarrow}$ and $\leq_{\uparrow}$, which will ease some of our proofs:

▶ **Definition 3.4.** We define the operators $F_{\downarrow}$ and $F_{\uparrow}$ on binary relation on processes:

- For $\eta \subseteq (\mathit{Proc}_{\mathsf{Stop}} \times \mathit{Proc}_{\mathsf{Stop}})$, $P\ F_{\downarrow}(\eta)\ Q$ holds iff
  1. If P is successful (i.e. $\mathsf{stop}(P)$), then $Q\downarrow$.
  2. If $P \xrightarrow{sr} P'$, then there exists Q' such that $Q \xrightarrow{sr,*} Q'$ and $P'\ \eta\ Q'$.
- For $\eta \subseteq (\mathit{Proc}_{\mathsf{Stop}} \times \mathit{Proc}_{\mathsf{Stop}})$, $P\ F_{\uparrow}(\eta)\ Q$ holds iff
  1. If $P\Uparrow$, then $Q\uparrow$.
  2. If $P \xrightarrow{sr} P'$, then there exists Q' such that $Q \xrightarrow{sr,*} Q'$ and $P'\ \eta\ Q'$.

The relation $\precsim_{\downarrow}$ ($\precsim_{\uparrow}$, resp.) is the greatest fixpoint of the operator $F_{\downarrow}$ ($F_{\uparrow}$, resp.).

For an operator F on binary relations, a relation η is *F-dense*, iff $\eta \subseteq F(\eta)$. The co-induction principle is that an F-dense relation η is contained in the greatest fixpoint of F.

Since$\leq_{\downarrow}$ is $F_{\downarrow}$-dense and $\leq_{\uparrow}$ is $F_{\uparrow}$-dense, the following lemma holds.

▶ **Lemma 3.5.** $\leq_{\downarrow} = \precsim_{\downarrow}$ *and* $\leq_{\uparrow} = \precsim_{\uparrow}$.

Before defining "applicative similarities" for may- and should-convergence, we define the property of a relation to preserve the input and output capabilities of one process w.r.t. another process. This definition is analogous to preserving actions in labeled bisimilarities. We prefer this definition here, since we can omit the definition of a labeled transition system.

▶ **Definition 3.6.** For processes $P, Q \in \mathit{Proc}_{\mathsf{Stop}}$ and a binary relation $\eta \subseteq (\mathit{Proc}_{\mathsf{Stop}} \times \mathit{Proc}_{\mathsf{Stop}})$, we say η *preserves the input/output capabilities of P w.r.t. Q* iff:

- *Open input:* If $P \equiv \nu\mathcal{X}.(x(y).P_1 \mid P_2)$ with $x \notin \mathcal{X}$, then for every name $z \in \mathcal{N}$ there exists a process $Q' \in \mathit{Proc}_{\mathsf{Stop}}$ s.t. $Q \xrightarrow{sr,*} Q'$, $Q' \equiv \nu\mathcal{Y}.(x(y).Q_1 \mid Q_2)$ with $x \notin \mathcal{Y}$, and $(\nu\mathcal{X}.(P_1[z/y] \mid P_2))\ \eta\ (\nu\mathcal{Y}.(Q_1[z/y] \mid Q_2))$.
- *Open output:* If $P \equiv \nu\mathcal{X}.(\overline{x}y.P_1 \mid P_2)$ with $x, y \notin \mathcal{X}$, then there exists a process Q' s.t. $Q \xrightarrow{sr,*} Q'$, $Q' \equiv \nu\mathcal{Y}.(\overline{x}y.Q_1 \mid Q_2)$ with $x, y \notin \mathcal{Y}$, and $(\nu\mathcal{X}.(P_1 \mid P_2))\ \eta\ (\nu\mathcal{Y}.(Q_1 \mid Q_2))$.
- *Bound output:* If $P \equiv \nu\mathcal{X}, \nu y.(\overline{x}y.P_1 \mid P_2)$ with $x \notin \mathcal{X}$, then there exists Q' s.t. $Q \xrightarrow{sr,*} Q'$, $Q' \equiv \nu\mathcal{Y}, \nu y.(\overline{x}y.Q_1 \mid Q_2)$ with $x \notin \mathcal{Y}$, and $(\nu\mathcal{X}.(P_1 \mid P_2))\ \eta\ (\nu\mathcal{Y}.(Q_1 \mid Q_2))$.

▶ **Definition 3.7** ((Full) Applicative Similarities)**.** We define the operators $F_{b,\downarrow}$ and $F_{b,\uparrow}$ on binary relations on processes, where *applicative $\downarrow$-similarity* $\precsim_{b,\downarrow}$ is the greatest fixpoint of $F_{b,\downarrow}$ and *applicative $\uparrow$-similarity* $\precsim_{b,\uparrow}$ is the greatest fixpoint of the operator $F_{b,\uparrow}$.

- For $\eta \subseteq (\mathit{Proc}_{\mathsf{Stop}} \times \mathit{Proc}_{\mathsf{Stop}})$, $P\ F_{b,\downarrow}(\eta)\ Q$ holds iff
  1. If P is successful (i.e. $\mathsf{stop}(P)$), then $Q\downarrow$.
  2. If $P \xrightarrow{sr} P'$, then $\exists Q'$ with $Q \xrightarrow{sr,*} Q'$ and $P'\ \eta\ Q'$.
  3. If P is not successful, then η preserves the input/output capabilities of P w.r.t. Q.

- For $\eta \subseteq (Proc_{\mathsf{Stop}} \times Proc_{\mathsf{Stop}})$, $P\ F_{b,\uparrow}(\eta)\ Q$ holds iff
  1. If $P\Uparrow$, then $Q\uparrow$.
  2. If $P \xrightarrow{sr} P'$, then $\exists Q'$ with $Q \xrightarrow{sr,*} Q'$ and $P'\ \eta\ Q'$.
  3. If $\neg P\Uparrow$, then η preserves the input/output capabilities of P w.r.t. Q.
  4. $Q \precsim_{b,\downarrow} P$.

Full applicative $\downarrow$-similarity $\precsim^{\sigma}_{b,\downarrow}$ and *full applicative $\uparrow$-similarity* $\precsim^{\sigma}_{b,\uparrow}$ are defined as $P \precsim^{\sigma}_{b,\downarrow} Q$ ($P \precsim^{\sigma}_{b,\uparrow} Q$, resp.) iff $\sigma(P) \precsim_{b,\downarrow} \sigma(Q)$ ($\sigma(P) \precsim_{b,\uparrow} \sigma(Q)$, resp.) for all $\sigma \in \Sigma$. *Full applicative similarity* $\precsim_b$ is defined as the intersection $\precsim_b := \precsim^{\sigma}_{b,\downarrow} \cap (\precsim^{\sigma}_{b,\uparrow})^{-1}$. *Mutual full $\Downarrow$-applicative similarity* $\simeq_{b,\Downarrow}$ is the intersection $\precsim^{\sigma}_{b,\uparrow} \cap (\precsim^{\sigma}_{b,\uparrow})^{-1}$ and *mutual full applicative similarity* $\simeq_b$ is the intersection $\simeq_b := \precsim_b \cap (\precsim_b)^{-1}$.

We discuss our definitions of applicative similarity. We first consider $\precsim_{b,\downarrow}$. Its definition is related to early labeled bisimilarity for the π-calculus [20], but adapted to the successfulness-test. However, there is a difference whether a similarity or a bisimilarity is used. Applicative $\downarrow$-bisimilarity would be defined as the largest relation $\mathcal{R}$ such that $\mathcal{R}$ and $\mathcal{R}^{-1}$ are $F_{b,\downarrow}$-dense. The relation $\precsim_{b,\downarrow} \cap (\precsim_{b,\downarrow})^{-1}$ is much coarser than applicative $\downarrow$-bisimilarity. For instance, the processes $P_{a,bc} := \mathtt{choice}(a(u_1), \mathtt{choice}(b(u_2), c(u_3)))$ and $P_{ab,c} := \mathtt{choice}(\mathtt{choice}(a(u_1), b(u_2)), c(u_3))$ are not applicative $\downarrow$-bisimilar, since after reducing $P_{a,bc} \xrightarrow{sr} P_0 \equiv \nu x, y.(x(y_1).a(u_1) \mid \mathtt{choice}(b(u_2), c(u_3)))$ there is no process P_1 with $P_{ab,c} \xrightarrow{sr,*} P_1$ s.t. P_0 and P' are applicative $\downarrow$-bisimilar. However, $P_{a,bc} \precsim_{b,\downarrow} P_{ab,c}$ and $P_{ab,c} \precsim_{b,\downarrow} P_{a,bc}$. The following example (adapted from an example in [20]) shows that even $\precsim_{b,\downarrow}$ is more discriminating than contextual may preorder:

▶ **Proposition 3.8.** *Let $S_{xy} := x(z).\overline{y}z$ and $S_{yx} := y(z).\overline{x}z$. For $P := \overline{a}x \mid !\,S_{xy} \mid !\,S_{yx}$ and $Q := \overline{a}y \mid !\,S_{xy} \mid !\,S_{yx}$, it holds: $\neg(P \precsim_{b,\downarrow} Q)$ (and thus also $\neg(P \precsim^{\sigma}_{b,\downarrow} Q)$), but $P \leq_{c,\downarrow} Q$.*

Proof. $P \precsim_{b,\downarrow} Q$ does not hold, since the output on channel a is different. $P \leq_{c,\downarrow} Q$ is proved in the appendix in Lemma A.1. ◀

The definition of applicative $\uparrow$-similarity includes the property $Q \precsim_{b,\downarrow} P$, i.e.:

▶ **Proposition 3.9.** $P \precsim_{b,\uparrow} Q \implies Q \precsim_{b,\downarrow} P$.

Thus – like the discussion before on bisimilarity – this requirement makes the relation $\precsim_{b,\uparrow}$ very fine-grained: the processes $P_{a,bc}$ and $P_{ab,c}$ are not applicative $\uparrow$-similar, although the processes are contextually equivalent. The reason for our choice of this definition is that we did not find a coarser $\uparrow$-similarity which is sound for contextual should-preorder. Properties that must hold for such a definition are that it preserves may-divergence w.r.t. Stop, i.e. $\uparrow$, but also (due to Theorem 5.5, see below) that it preserves the may-divergence w.r.t. barbs. The second condition holds for $\precsim_{b,\uparrow}$, since we added $Q \precsim_{b,\downarrow} P$ in Definition 3.7. Obviously, $\precsim_{b,\downarrow}$ preserves may-convergence and $\precsim_{b,\uparrow}$ preserves may-divergence:

▶ **Lemma 3.10.** $\precsim_{b,\downarrow} \subseteq \precsim_{\downarrow}$ *and* $\precsim_{b,\uparrow} \subseteq \precsim_{\uparrow}$.

We now show soundness of our applicative similarities.

▶ **Proposition 3.11.** *For all $P, Q, R \in Proc_{\mathsf{Stop}}$ and all $\mathcal{X}$, the following implications hold:*
1. $(P \precsim_{b,\downarrow} Q) \implies \nu\mathcal{X}.(P \mid R) \precsim_{\downarrow} \nu\mathcal{X}.(Q \mid R)$.
2. $(P \precsim_{b,\uparrow} Q) \implies \nu\mathcal{X}.(P \mid R) \precsim_{\uparrow} \nu\mathcal{X}.(Q \mid R)$.

Proof. The relation $\precsim_{\downarrow} \cup \{(\nu\mathcal{X}.(P \mid R), \nu\mathcal{X}.(Q \mid R)) \mid P \precsim_{b,\downarrow} Q$, for any $\mathcal{X}$, $R\}$ is $F_{\downarrow}$-dense (proved in the appendix, Lemma A.2) and thus the first part holds. The second part holds, since the relation $\precsim_{\uparrow} \cup \{(\nu\mathcal{X}.(P \mid R), \nu\mathcal{X}.(Q \mid R)) \mid P \precsim_{b,\uparrow} Q$, for any $\mathcal{X}$, $R\}$ is $F_{\uparrow}$-dense, which is proved in the appendix, Lemma A.3. ◀

▶ **Theorem 3.12** (Soundness of Full Applicative Similarities)**.** *The following inclusions hold:*
1. $\precsim^{\sigma}_{b,\downarrow} \;\subseteq\; \leq_{c,\downarrow}$,
2. $\precsim_{b} \;\subseteq\; \leq_{c}$, *and*
3. $\simeq_{b,\Downarrow} = \simeq_{b} \;\subseteq\; \sim_{c}$.

Proof. For the first part Proposition 3.11 part (1) shows that $\sigma(P) \precsim_{b,\downarrow} \sigma(Q)$ implies $\sigma(P) \mid R \precsim_{\downarrow} \sigma(Q) \mid R$ for all σ, R. Thus, $P \precsim^{\sigma}_{b,\downarrow} Q$ implies $\sigma'(P) \mid R \precsim_{\downarrow} \sigma'(Q) \mid R$ for all σ', R. Since $\precsim_{\downarrow} = \leq_{\downarrow}$ (Lemma 3.5) the context lemma (Theorem 3.3) shows $P \leq_{c,\downarrow} Q$.

For the second part we apply both parts of Proposition 3.11 which shows that $P \precsim^{\sigma}_{b,\downarrow} Q$ and $Q \precsim^{\sigma}_{b,\uparrow} P$ imply that $\sigma'(P) \mid R \precsim_{\downarrow} \sigma'(Q) \mid R$ and $\sigma'(Q) \mid R \precsim_{\uparrow} \sigma'(P) \mid R$ for all σ', R. Since $\precsim_{\downarrow} = \leq_{\downarrow}$ and $\precsim_{\uparrow} = \geq_{\Downarrow}$ (Lemma 3.5), Theorem 3.3 shows $P \leq_{c} Q$.

The equation of the last part follows from Proposition 3.9. The inclusion of the last part follows from the second part and the definitions of $\simeq_{b}$ and $\sim_{c}$. ◀

4 Equivalences and the Contextual Ordering

In this section we analyze the contextual ordering and also show some contextual equivalences.

4.1 Correctness of Deterministic Interaction

We demonstrate our developed techniques for an exemplary program optimization and apply Theorem 3.12 to show correctness of a restricted variant of the *ia*-reduction that ensures determinism. Moreover, the result can be used to show a completeness result w.r.t. the tests in the context lemma (Corollary 4.3).

▶ **Theorem 4.1** (Correctness of Deterministic Interaction)**.** *For all processes P, Q the equation $\nu x.(x(y).P \mid \overline{x}z.Q)) \sim_{c} \nu x.(P[z/y] \mid Q)$ holds.*

Proof. We use Theorem 3.12 and show that $\nu x.(x(y).P \mid \overline{x}z.Q)) \simeq_{b,\Downarrow} \nu x.(P[z/y] \mid Q)$.

Let $\mathcal{S} := \{(\sigma(\nu x.(x(y).P \mid \overline{x}z.Q)), \sigma(\nu x.(P[z/y] \mid Q))) \mid$ for all $x, y, z, P, Q, \sigma\} \;\cup \equiv$. We show that $\mathcal{S}$ and $\mathcal{S}^{-1}$ are $F_{b,\downarrow}$-dense and $F_{b,\uparrow}$-dense.

Let $(R_1, R_2) = (\sigma(\nu x.(x(y).P \mid \overline{x}z.Q)), \sigma(\nu x.(P[z/y] \mid Q)))$. Then $(R_1, R_2) \in F_{b,\downarrow}(\mathcal{S})$:
1. R_1 is not successful, so there is nothing to show.
2. If $R_1 \xrightarrow{sr} R_1'$, then $R_1' \equiv R_2$ and $(R_2, R_2) \in \mathcal{S}$.
3. R_1 does not have an open input or output, thus there is nothing to show.

Also $(R_2, R_1) \in F_{b,\downarrow}(\mathcal{S}^{-1})$:
1. If R_2 is successful, then $R_1{\downarrow}$, since $R_1 \xrightarrow{sr} R_2$.
2. If $R_2 \xrightarrow{sr} R_2'$, then $R_1 \xrightarrow{sr,2} R_2'$ and $(R_2', R_2') \in \mathcal{S}^{-1}$.
3. If R_2 has an open input or output, then $R_1 \xrightarrow{sr} R_2$ and the condition of $F_{b,\downarrow}$ can be fulfilled.

Thus $\mathcal{S}$ and $\mathcal{S}^{-1}$ are $F_{b,\downarrow}$-dense, and thus $R_1 \precsim_{b,\downarrow} R_2$ and $R_2 \precsim_{b,\downarrow} R_1$ for any $(R_1, R_2) \in \mathcal{S}$.

Now we show that $(R_1, R_2) \in F_{b,\uparrow}(\mathcal{S})$:
1. If $R_1{\Uparrow}$ then $R_2{\Uparrow}$, since $R_1 \xrightarrow{sr} R_2$.
2. If $R_1 \xrightarrow{sr} R_1'$, then $R_1' \equiv R_2$ (since there is only one reduction possibility for R_1) and $(R_1', R_1') \in \mathcal{S}$.

3. R_1 does not have an open input or output, thus there is nothing to show.
4. $R_2 \precsim_{b,\downarrow} R_1$ is already proved.

Finally, also $(R_2, R_1) \in F_{b,\uparrow}(\mathcal{S}^{-1})$:

1. If $R_2{\Uparrow}$ then clearly $R_1{\uparrow}$.
2. If $R_2 \xrightarrow{sr} R_2'$, then $R_1 \xrightarrow{sr,2} R_2'$ and $(R_2', R_2') \in \mathcal{S}^{-1}$.
3. If R_2 has an open input or output, then $R_1 \xrightarrow{sr} R_2$ and the condition of $F_{b,\uparrow}$ can be fulfilled.
4. $R_1 \precsim_{b,\downarrow} R_2$ is already proved.

Thus $\mathcal{S}$ and $\mathcal{S}^{-1}$ are $F_{b,\uparrow}$-dense and $R_1 \precsim_{b,\uparrow} R_2$ and $R_2 \precsim_{b,\uparrow} R_1$ for all $(R_1, R_2) \in \mathcal{S}$ and thus Theorem 3.12 shows the claim. ◀

Contextual preorder does not change, if we additionally consider all name substitutions:

▶ **Lemma 4.2.** *For $\xi \in \{\downarrow, \Downarrow\}$: $P \leq_{c,\xi} Q$ iff $\forall C \in \mathcal{C}_{\mathsf{Stop}}$, $\sigma \in \Sigma$: $C[\sigma(P)] \leq_\xi C[\sigma(Q)]$.*

Proof. "$\Leftarrow$" is trivial. For "$\Rightarrow$" we define for $\sigma = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ the context $C_\sigma := \nu\mathcal{W}.(w_1(x_1).w_2(x_2).\ldots.w_n(x_n).[\cdot] \mid \overline{w_1}y_1 \mid \ldots \mid \overline{w_n}y_n)$ where $\mathcal{W} = \{w_1, \ldots, w_n\}$ and $\mathcal{W} \cap (\mathsf{fn}(P) \cup \mathsf{fn}(Q)) = \emptyset$. The reductions $C_\sigma[P] \xrightarrow{ia,*} \sigma(P)$ and $C_\sigma[Q] \xrightarrow{ia,*} \sigma(Q)$ are valid, where all *ia*-steps are deterministic and thus by Theorem 4.1 $C_\sigma[P] \sim_c \sigma(P)$ and $C_\sigma[Q] \sim_c \sigma(Q)$. Now let $P \leq_{c,\xi} Q$ and let C, σ s.t. $C[\sigma(P)]\xi$. Since $\sigma(P) \sim_c C_\sigma[P]$ also $C[C_\sigma(P)]\xi$ which in turn implies $C[C_\sigma(Q)]\xi$. Since $C_\sigma[Q] \sim_c \sigma(Q)$, this shows $C[\sigma(Q)]\xi$. Since C, σ were chosen arbitrarily, $C[\sigma(P)] \leq_\xi C[\sigma(Q)]$ holds for all $C \in \mathcal{C}_{\mathsf{Stop}}$ and $\sigma \in \Sigma$. ◀

Thus, the tests of the context lemma (Theorem 3.3) are complete w.r.t. $\leq_c$:

▶ **Corollary 4.3.** *For all $P, Q \in Proc_{\mathsf{Stop}}$:*

- *$P \leq_{c,\downarrow} Q$ iff for all $\sigma \in \Sigma, R \in Proc_{\mathsf{Stop}}$: $\sigma(P) \mid R \leq_\downarrow \sigma(Q) \mid R$.*
- *$P \leq_c Q$ iff for all $\sigma \in \Sigma, R \in Proc_{\mathsf{Stop}}, \xi \in \{\downarrow, \Downarrow\}$: $\sigma(P) \mid R \leq_\xi \sigma(Q) \mid R$.*

4.2 Results on the Contextual Ordering

We show several properties on the contextual ordering and equivalence. All successful processes are in the same equivalence class. More surprisingly, all may-convergent processes are equal w.r.t. contextual may-convergence, which is a strong motivation to also consider should-convergence. Further results are that Stop is the largest element in the contextual ordering, and there is no least element:

▶ **Theorem 4.4.**

1. *If P, Q are two successful processes, then $P \sim_c Q$.*
2. *If P, Q are two processes with $P{\downarrow}, Q{\downarrow}$, then $P \sim_{c,\downarrow} Q$.*
3. *There are may-convergent processes P, Q with $P \not\sim_c Q$.*
4. *Stop is the greatest process w.r.t. $\leq_c$.*
5. *$\mathbf{0}$ is the smallest process w.r.t. $\leq_{c,\downarrow}$.*
6. *There is no smallest process w.r.t. $\leq_c$.*

Proof. For (1) let P and Q be successful. Then for any $\sigma \in \Sigma$ and any $R \in Proc_{\mathsf{Stop}}$ also $\sigma(P) \mid R$ and $\sigma(Q) \mid R$ are successful. This implies $\sigma(P) \mid R{\downarrow}$, $\sigma(Q) \mid R{\downarrow}$, $\sigma(P) \mid R{\Downarrow}$, and $\sigma(Q) \mid R{\Downarrow}$ for all $R \in Proc_{\mathsf{Stop}}$ and $\sigma \in \Sigma$ and thus Theorem 3.3 shows $P \sim_c Q$.

Since $P{\downarrow} \implies \sigma(P) \mid R{\downarrow}$ for any process P, R and $\sigma \in \Sigma$, Theorem 3.3 shows part (2).

For (3) the empty context distinguishes $\mathtt{choice}(\mathsf{Stop}, \mathbf{0})$ and Stop: $\mathtt{choice}(\mathsf{Stop}, \mathbf{0}){\downarrow}$, and $\mathtt{choice}(\mathsf{Stop}, \mathbf{0}){\uparrow}$, while $\mathsf{Stop}{\Downarrow}$, hence $\mathtt{choice}(\mathsf{Stop}, \mathbf{0}) \not\sim_c \mathsf{Stop}$.

For part (4) clearly $\mathsf{Stop} \mid R{\Downarrow}$ for all R. Since $\mathsf{Stop} \mid R{\Downarrow} \implies \mathsf{Stop} \mid R{\downarrow}$, we have $\sigma(P) \mid R \leq_{\downarrow} \mathsf{Stop} \mid R$ and $\sigma(P) \mid R \leq_{\Downarrow} \mathsf{Stop} \mid R$ for any P, σ, and R. Thus Theorem 3.3 shows $P \leq_c \mathsf{Stop}$ for any process P.

Part (5) follows from Theorem 3.12, since $\{(\mathbf{0}, P) \mid P \in \mathit{Proc}_{\mathsf{Stop}}\}$ is $F_{b,\downarrow}$-dense.

For (6) assume that there is a process P_0 that is the smallest one, i.e. $P_0 \leq_c P$ for all processes P. Then $P_0{\Uparrow}$, since $P_0 \leq_c \mathbf{0}$. Let $P_0 \xrightarrow{*} P_1$, such that $P_1 = D[x(y).P_3]$, and where x is free. With $D_1 = \overline{x}y.\mathsf{Stop}$ we obtain $D_1[P_1]{\downarrow}$, but $D_1[\mathbf{0}] \equiv \overline{x}y.\mathsf{Stop}{\Uparrow}$. We argue similarly for outputs. Thus the reducts of P_0 do not have open outputs. Now let $P = x(y).\mathbf{0}$, where by our assumption $P_0 \leq_{c,\Downarrow} P$ holds. Let $D = [\cdot] \mid \overline{x}y.\mathbf{0} \mid x(y).\mathsf{Stop}$. Then $D[P_0]{\Downarrow}$, since there is no communication between the reducts of P_0 and D, but D can always be reduced to a successful process. Now consider $D[P]$. It is $D[P] \to \mathbf{0} \mid x(y).\mathsf{Stop}$, which is must-divergent, hence we have reached the contradiction $P_0 \not\leq_{c,\Downarrow} P$. ◀

We show that it is suffices to test should-convergence in all contexts, since all tests for may-convergence can be encoded:

▶ **Theorem 4.5.**
1. $\leq_{c,\Downarrow} = \leq_c$,
2. $\leq_c \neq \leq_{c,\downarrow}$,
3. *and* $\leq_{c,\Downarrow} \not\subseteq \sim_{c,\downarrow}$.

Proof. For part (1), we show that $\leq_{c,\Downarrow} \subseteq \leq_{c,\downarrow}$: let $C_{x,y,\mathcal{X}} := !\,\nu x, y, \nu\mathcal{X}.[\cdot]$. For any process P with $x, y \notin \mathsf{fn}(P)$ and $\mathcal{X} \supseteq \mathsf{fn}(P)$ one can verify that $P{\downarrow}$ iff $C_{x,y,\mathcal{X}}[P]{\Downarrow}$: If $P{\downarrow}$, then $P' := \nu x, y.\nu\mathcal{X}.P{\downarrow}$ by Lemma 2.4 and for $!\,P'$ we can generate a parallel copy of P', and thus $C_{x,y,\mathcal{X}}[P]{\Downarrow}$. If $C_{x,y,\mathcal{X}}[P]{\Downarrow}$, then $\nu x, y, \mathcal{X}.P{\downarrow}$, since parallel copies of $\nu x, y, \mathcal{X}.P$ cannot communicate due to the name restriction. Lemma 2.4 shows $P{\downarrow}$. Now let $P \leq_{c,\Downarrow} Q$, $C[P]{\downarrow}$, but $C[Q]{\Uparrow}$. With fresh names x, y, $\mathcal{X} = \mathsf{fn}(P) \cup \mathsf{fn}(Q)$: $C_{x,y}[C[P]]{\Downarrow}$ but $C_{x,y}[C[Q]]{\uparrow}$ which contradicts $P \leq_{c,\Downarrow} Q$.

The inequality of part (2) follows from Theorem 4.4 items (5), (6).

For part (3), clearly, $\mathbf{0} \leq_c \mathsf{Stop}$, since Stop is a largest element of $\leq_c$, but $\mathbf{0}{\Uparrow}$ while $\mathsf{Stop}{\downarrow}$, and thus in Π_{Stop} contextual should-preorder does not imply contextual may-equivalence. ◀

We conclude this subsection, by analyzing several equations, including the ones from [4].

▶ **Theorem 4.6.** *For all processes P, Q, the following equivalences hold:*
1. $!\,P \sim_c !\,!\,P$.
2. $!\,P \mid !\,P \sim_c !\,P$.
3. $!\,(P \mid Q) \sim_c !\,P \mid !\,Q$.
4. $!\,\mathbf{0} \sim_c \mathbf{0}$.
5. $!\,\mathsf{Stop} \sim_c \mathsf{Stop}$.
6. $!\,(P \mid Q) \sim_c !\,(P \mid Q) \mid P$.
7. $x(y).\nu z.P \sim_c \nu z.x(y).P$ *if* $z \notin \{x, y\}$.
8. $\overline{x}y.\nu z.P \sim_c \nu z.\overline{x}y.P$ *if* $z \notin \{x, y\}$.

Proof. This holds, since $\mathcal{S}_i \cup \precsim_{b,\uparrow}$ and $\mathcal{S}_i^{-1} \cup \precsim_{b,\uparrow}$ are $F_{b,\uparrow}$-dense, where $\mathcal{S}_i := \{(R \mid l_i, R \mid r_i) \mid$ for all $R\}$, and l_i, r_i are the left and right hand side of the ith equation. ◀

5 Results for the Stop-free Calculus

In this section we consider the π-calculus Π without the constant Stop but with barbed may- and should-testing as notion of process equivalence. We will show a strong connection between Π_{Stop} and Π which makes a lot of results transferable.

▶ **Definition 5.1.** Let Π be the subcalculus of Π_{Stop} that does not have the constant Stop as a syntactic construct. Processes, contexts, reduction, structural congruences are accordingly adapted for Π. We write *Proc* for the set of processes of Π and $\mathcal{C}$ for the set of contexts of Π.

We define the notion of a barb, i.e. that a process can receive a name on an open channel[1]. Barbed may- and should-testing is defined analogously to contextual equivalence, where the observation of success is replaced by observing barbs:

▶ **Definition 5.2.** Let $P \in \mathit{Proc}$ and $x \in \mathcal{N}$. A process P *has a barb on input x* (written as $P \upharpoonright^{x}$) iff $P \equiv \nu\mathcal{X}.(x(y).P' \mid P'')$ where $x \notin \mathcal{X}$. We write $P\downharpoonleft_x$ iff there exists P' s.t. $P \xrightarrow{sr,*} P'$ and $P' \upharpoonright^{x}$. We write $P \Downarrow_x$ iff for all P' with $P \xrightarrow{sr,*} P'$ also $P'\downharpoonleft_x$ holds. We write $P \Uparrow_x$ iff $P\downharpoonleft_x$ does not hold, and we write $P\upharpoonleft_x$ iff $P \Downarrow_x$ does not hold.

For a name $x \in \mathcal{N}$, *barbed may- and should-testing preorder* $\leq_{c,barb}$ and *barbed may- and should-testing equivalence* $\sim_{c,barb}$ are defined as $\leq_{c,barb} := \leq_{c,\downharpoonleft_x} \cap \leq_{c,\Downarrow_x}$ and $\sim_{c,barb} := \leq_{c,barb} \cap (\leq_{c,barb})^{-1}$ where for $\xi \in \{\downharpoonleft_x, \Downarrow_x, \upharpoonleft_x, \Uparrow_x\}$ and $P, Q \in \mathit{Proc}$ the inequality $P \leq_{c,\xi} Q$ holds iff for all contexts $C \in \mathcal{C} : C[P]\xi \implies C[Q]\xi$.

In difference to observing success, the barb behavior is not stable under reduction, e.g. for the process $P = x(z) \mid \overline{x}y$, $P \upharpoonright^{x}$ holds, but $P \xrightarrow{sr} \mathbf{0}$ and $\mathbf{0} \Uparrow_x$. We show that in $\sim_{c,barb}$ the concrete name x is irrelevant:

▶ **Proposition 5.3.** *For all $x, y \in \mathcal{N}$: $\leq_{c,\downharpoonleft_x} = \leq_{c,\downharpoonleft_y}$ and $\leq_{c,\Downarrow_x} = \leq_{c,\Downarrow_y}$.*

Proof. First assume $P \leq_{c,\downharpoonleft_x} Q$, $C[P]\downharpoonleft_y$, but $C[Q] \Uparrow_y$. Let $C' = \overline{y}w.x(w').\mathbf{0} \mid \nu\mathcal{X}.([\cdot])$ where $\mathcal{X} = (\mathsf{fn}(C[P]) \cup \mathsf{fn}(C[Q])) \setminus \{y\}$. From $C[P]\downharpoonleft_y$ also $C'[C[P]]\downharpoonleft_x$ follows. Hence $C'[C[Q]]\downharpoonleft_x$ holds, too. But, the structure of C' shows that this is only possible if $C[Q] \xrightarrow{sr,*} Q'$ with $Q' \upharpoonright^{y}$ and thus $C[Q] \Uparrow_y$ cannot hold. Now assume $P \leq_{c,\Downarrow_x} Q$ and $C[P] \Downarrow_y$, but $C[Q]\upharpoonleft_y$. Clearly, $C'[C[P]] \Downarrow_x$ holds. The assumption $C[Q]\upharpoonleft_y$ implies that $C[Q] \xrightarrow{sr,*} Q'$ with $Q' \Uparrow_y$. Then also $C'[Q'] \Uparrow_x$, and since $C'[C[Q]] \xrightarrow{sr,*} C'[Q']$ we also have $C'[C[Q]]\upharpoonleft_x$. This is a contradiction, since $P \leq_{c,\Downarrow_x} Q$ implies $C'[C[P]] \Downarrow_x \implies C'[C[Q]] \Downarrow_x$. ◀

▶ **Corollary 5.4.** *$P \leq_{c,barb} Q$ iff $\forall x \in \mathcal{N} : P \leq_{c,\downharpoonleft_x} Q \wedge P \leq_{c,\Downarrow_x} Q$.*

We show that contextual equivalence of Π_{Stop} conservatively extends barbed testing equivalence of the π-calculus: $P \sim_{c,barb} Q \implies P \sim_c Q$ for all stop-free P, Q. Moreover, on Stop-free processes[2] P, Q, contextual equivalence is also complete for barbed testing, i.e. $P \sim_c Q \implies P \sim_{c,barb} Q$. Thus the identity translation (see e.g. [22] for properties of translations) from Π into Π_{Stop} is fully-abstract w.r.t. $\sim_{c,barb}$ in Π and $\sim_c$ in Π_{Stop}.

▶ **Theorem 5.5.** *For all processes $P, Q \in \mathit{Proc}$: $P \leq_{c,barb} Q \iff P \leq_c Q$, and hence also $P \sim_{c,barb} Q \iff P \sim_c Q$.*

[1] We only consider an input capability here, since the barbed may- and should-testing equivalence does not change if also output capabilities are observed.
[2] Stop-*free* means without occurrences of Stop.

Proof. Let P, Q be Stop-free processes. It suffices to show that $P \leq_{c,\downharpoonleft_x} Q$ iff $P \leq_{c,\downarrow} Q$ and $P \leq_{c,\downdownharpoons_x} Q$ iff $P \leq_{c,\Downarrow} Q$. In the remainder of the proof let $\psi_{u,v}(R)$ be the process (or context) R with every occurrence of Stop replaced by $u(v)$ and let $\psi^{-1}_{u,v}(R)$ be the process (or context) R with every occurrence of the subprocess $u(v)$ be replaced by Stop.

- $P \leq_{c,\downharpoonleft_x} Q \implies P \leq_{c,\downarrow} Q$: Let $P \leq_{c,\downharpoonleft_x} Q$ and $C \in \mathcal{C}$ with $C[P]\downarrow$, i.e. $C[P] = P_0 \xrightarrow{sr} P_1 \ldots \xrightarrow{sr} P_n$ s.t. $\mathsf{stop}(P_n)$. Let u, v be fresh names. Then $\psi_{u,v}(C)[P] = \psi_{u,v}(P_0) \xrightarrow{sr} \psi_{u,v}(P_1) \xrightarrow{sr} \ldots \xrightarrow{sr} \psi_{u,v}(P_n)$, since ia-reductions do not use Stop and the axiom $\nu z.\mathsf{Stop} \equiv \mathsf{Stop}$ can be replaced by $\nu z.u(v) \equiv u(v)$ (since u is fresh). Since $P_n \equiv \mathsf{Stop} \mid R$, we have $\psi_{u,v}(P_n) \equiv (u(v) \mid \psi_{u,v}(R))$ and thus $\psi_{u,v}(P_n)\Rsh^u$ and $\psi_{u,v}(C)[P]\downdownharpoons_u$. By Proposition 5.3 and $P \leq_{c,\downharpoonleft_x} Q$ we have $P \leq_{c,\downharpoonleft_u} Q$ and thus $\psi_{u,v}(C)[Q]\downdownharpoons_u$, i.e. $\psi_{u,v}(C)[Q] \xrightarrow{sr} Q_1 \ldots \xrightarrow{sr} Q_m$ where $Q_m\Rsh^u$, i.e. $Q_m \equiv \nu\mathcal{X}.(u(v).R_1 \mid R_2)$. The reduction cannot perform an ia-reduction using the prefix $u(v)$, since u is fresh, and thus $R_1 = \mathbf{0}$. Thus the reduction $C[Q] \xrightarrow{sr} \psi^{-1}_{u,v}(Q_1) \ldots \xrightarrow{sr} \psi^{-1}_{u,v}(Q_m)$ exists, and $\psi^{-1}_{u,v}(Q_m) \equiv \nu x_1, \ldots x_n.(\mathsf{Stop} \mid \psi^{-1}_{u,v}(R_2))$ and thus $C[Q]\downarrow$.
- $P \leq_{c,\downarrow} Q \implies P \leq_{c,\downharpoonleft_x} Q$: Let $P \leq_{c,\downarrow} Q$, C be a Stop-free context, and $C[P]\downdownharpoons_x$. Then $C[P] \xrightarrow{sr} P_1 \ldots \xrightarrow{sr} P_n \equiv \nu\mathcal{X}.(x(y).P' \mid P'')$, and for $C_1 := ([\cdot] \mid \overline{x}y.\mathsf{Stop})$ we have $C_1[C[P]]\downarrow$, since the reduction for $C[P]$ can be used and results in $C_1[P_n]$ which reduces to a successful process. $P \leq_{c,\downarrow} Q$ also implies $C_1[C[Q]]\downarrow$ and the corresponding reduction $C_1[C[Q]] \xrightarrow{sr,*} Q_m$ with $\mathsf{stop}(Q_m)$ must include an ia-reduction with a redex of the form $x(z).R \mid \overline{x}y.\mathsf{Stop}$. Let $Q_i \xrightarrow{sr,*} Q_{i+1}$ be this step in $C_1[C[Q]] \xrightarrow{sr,*} Q_m$. The prefix $C_1[C[Q]] \xrightarrow{sr,i} Q_i$ can be used to construct a reduction $C[Q] \xrightarrow{sr,i} Q_i'$ where $Q_i'\Rsh^x$ and thus $C[Q]\downdownharpoons_x$.
- $P \leq_{c,\Downarrow} Q \implies P \leq_{c,\downdownharpoons_x} Q$: Let $P \leq_{c,\Downarrow} Q$. For any Stop-free context $C \in \mathcal{C}$ we have to show: $C[Q]\upharpoonright_x \implies C[P]\upharpoonright_x$. Let C be a Stop-free context with $C[Q]\upharpoonright_x$, i.e. $C[Q] \xrightarrow{sr,*} Q'$ and $\neg(Q'\downharpoonleft_x)$. Then also $C_1[C[Q]]\uparrow$ with $C_1 = [\cdot] \mid \overline{x}y.\mathsf{Stop}$, since $C_1[C[Q]] \xrightarrow{sr,*} C_1[Q']$ and $C_1[Q']$ cannot become successful (otherwise $Q'\downharpoonleft_x$ would hold). $P \leq_{c,\Downarrow} Q$ implies $C_1[C[P]]\uparrow$, i.e. $C_1[C[P]] \xrightarrow{sr,*} P'$ and $P'\Uparrow$. The reduction $C_1[C[P]] \xrightarrow{sr,*} P'$ can never reduce $\overline{x}y.\mathsf{Stop}$, since otherwise $P'\Uparrow$ cannot hold, and thus we can assume that $P' \equiv C_1[P'']$ and $C[P] \xrightarrow{sr,*} P''$. Again $P''\downarrow_{\Rsh^x}$ cannot hold (otherwise $C_1[P'']\Uparrow$ would not hold) and thus $C[P]\upharpoonright_x$.
- $P \leq_{c,\downdownharpoons_x} Q \implies P \leq_{c,\Downarrow} Q$. Let $P \leq_{c,\downdownharpoons_x} Q$ and C be a context with $C[Q]\uparrow$, i.e. $C[Q] = Q_0 \xrightarrow{sr} \ldots \xrightarrow{sr} Q_n$ and $Q_n\Uparrow$. Let u, v be fresh names. Then $\neg(\psi_{u,v}(Q_n)\downharpoonleft_u)$ and also $\psi_{u,v}(Q_i) \xrightarrow{sr} \psi_{u,v}(Q_{i+1})$ and thus $\psi_{u,v}(C)[Q]\upharpoonright_u$. From $P \leq_{c,\downdownharpoons_x} Q$ (using Proposition 5.3) we have $P \leq_{c,\downdownharpoons_u} Q$ and thus $\psi_{u,v}(C)[P]\upharpoonright_u$, i.e. $C'[P] \xrightarrow{sr,*} P_m$ and $\neg(P_m\downharpoonleft_u)$. Then $\psi^{-1}_{u,v}(P_n)\Uparrow$ (otherwise $P_n\downharpoonleft_u$ would hold). Also $\psi^{-1}_{u,v}(P_i) \xrightarrow{sr} \psi^{-1}_{u,v}(P_{i+1})$, since $P_i \xrightarrow{sr} P_{i+1}$ cannot reduce any occurrence of $u(v)$. This shows $C[P]\uparrow$.

◀

Theorem 5.5 enables us to transfer some of the results for Π_{Stop} into Π.

▶ **Corollary 5.6.** *All equations in Theorem 4.6 (except for equation 5) also hold in Π for* Stop*-free processes, and for barbed testing equivalence $\sim_{c,barb}$. Deterministic interaction (see Theorem 4.1) is correct in Π for $\sim_{c,barb}$.*

Also Theorem 4.5 can be transferred to Π by applying Theorem 5.5, which shows that barbed should-testing preorder implies barbed may-testing equivalence (which does not hold for Π_{Stop} and contextual preorders, see Theorem 4.5 (3)):

▶ **Corollary 5.7.** *For all* Stop*-free processes $P, Q \in Proc$: $P \leq_{c,\downdownharpoons_x} Q$ implies $P \sim_{c,\downharpoonleft_x} Q$.*

Proof. The inclusion $\leq_{c,\downharpoonleft\!\downharpoonleft_x} \subseteq \leq_{c,\downharpoonleft_x}$ follows from Theorems 4.5 and 5.5. Before proving the remaining part, we show that the equivalence $P{\downharpoonleft_x} \iff C_1[P]{\upharpoonright\!\upharpoonright_y}$ holds, where $C_1 := R \mid [\cdot]$ with $R = \nu z.(\overline{z}y \mid z(w).w(w') \mid \overline{x}x'.z(z'))$ and P is any process with $\mathsf{fn}(P) \cap \{w, w', z, z', x'\} = \emptyset$ We have to show two implications:

1. $P\downharpoonleft_x \implies C_1[P]\upharpoonright_y$: If $P\downharpoonleft_x$, then $C_1[P]$ can be reduced to $P'' := \nu z.(z(w).w(w')) \mid P'$ where P' is the contractum of P after receiving x' along x. Clearly, P'' cannot barb on y (i.e. $P'' \upharpoonright\!\upharpoonright_y$) and thus $C_1[P]\upharpoonright_y$.
2. $C_1[P]\upharpoonright_y \implies P\downharpoonleft_x$: We show its contrapositive $P\upharpoonright\!\upharpoonright_x \implies C_1[P]\downharpoonleft\!\downharpoonleft_y$. If $P\upharpoonright\!\upharpoonright_x$, then in any reduction of $C_1[P]$ the process P cannot interact with the process R, and since $R\downharpoonleft\!\downharpoonleft_y$, also $C_1[P]\downharpoonleft\!\downharpoonleft_y$ holds.

We show $\leq_{c,\downharpoonleft\!\downharpoonleft_x} \subseteq (\leq_{c,\downharpoonleft_x})^{-1}$: Let $P \leq_{c,\downharpoonleft\!\downharpoonleft_x} Q$, $C[Q]\downharpoonleft_x$, but $C[P]\upharpoonright\!\upharpoonright_x$. Proposition 5.3 and $P \leq_{c,\downharpoonleft\!\downharpoonleft_x} Q$ imply $Q \leq_{c,\upharpoonright_y} P$. But $C_1[C[Q]]\upharpoonright_y$ while $C_1[C[P]]\downharpoonleft\!\downharpoonleft_y$ which is a contradiction. ◀

Finally, we show that there is no surjective encoding from Π_{Stop} into Π which preserves the ordering of processes w.r.t. contextual preorder in Π_{Stop} and barbed testing preorder in Π.

▶ **Theorem 5.8.** *There is no surjective translation $\psi : \Pi_{\mathsf{Stop}} \to \Pi$ s.t. for all $P, Q \in Proc_{\mathsf{Stop}}$: $P \leq_c Q \implies \psi(P) \leq_{c,barb} \psi(Q)$.*

Proof. This holds since Stop is a largest element of Π_{Stop} w.r.t. $\leq_c$, but in Π there is no largest element w.r.t. $\leq_{c,barb}$: Assume the claim is false, and P is a largest element w.r.t. $\leq_{c,barb}$. Let $\mathcal{X} = \mathsf{fn}(P)$ and $x \notin \mathcal{X}$. Then $x(z) \not\leq_{c,barb} P$, since $\nu\mathcal{X}.x(z)\downharpoonleft_x$ but $\nu\mathcal{X}.P\upharpoonright\!\upharpoonright_x$. ◀

6 Conclusion

We analyzed contextual equivalence w.r.t. may- and should-convergence in a π-calculus with Stop. We proved a context lemma and showed soundness of an applicative similarity. Since Π_{Stop} with contextual equivalence conservatively extends the π-calculus without Stop and barbed testing equivalence, this also provides a method to show barbed testing equivalences.

Future work may investigate extensions or variants of the calculus Π_{Stop}, e.g. with (guarded) sums, or with recursion. The results of this paper may also open easier possibilities to define and analyze embeddings of Π_{Stop} into other concurrent program calculi (e.g., the CHF-calculus [17, 18]) which also use a contextual semantics.

References

1 M. Boreale and R. De Nicola. Testing equivalence for mobile processes. *Inform. and Comput.*, 120(2):279–303, 1995.

2 A. Carayol, D. Hirschkoff, and D. Sangiorgi. On the representation of McCarthy's amb in the pi-calculus. *Theoret. Comput. Sci.*, 330(3):439–473, 2005.

3 R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoret. Comput. Sci.*, 34:83–133, 1984.

4 J. Engelfriet and T. Gelsema. A new natural structural congruence in the pi-calculus with replication. *Acta Inf.*, 40(6-7):385–430, 2004.

5 C. Fournet and G. Gonthier. A hierarchy of equivalences for asynchronous calculi. *J. Log. Algebr. Program.*, 63(1):131–173, 2005.

6 D. Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010.

7 C. Laneve. On testing equivalence: May and must testing in the join-calculus. Technical Report Technical Report UBLCS 96-04, University of Bologna, 1996.

8 R. Milner. *Communicating and Mobile Systems: the π-calculus.* CUP, 1999.
9 R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i & ii. *Inform. and Comput.*, 100(1):1–77, 1992.
10 R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. ICALP 1992*, *LNCS* 623, pp. 685–695. Springer, 1992.
11 J.H. Morris. *Lambda-Calculus Models of Programming Languages.* PhD thesis, MIT, 1968.
12 V. Natarajan and R. Cleaveland. Divergence and fair testing. In Proc. *ICALP 1995*, *LNCS* 944, pp. 648–659. Springer, 1995.
13 K. Peters, T. Yonova-Karbe, and U. Nestmann. Matching in the pi-calculus. In *Proc. EXPRESS/SOS 2014*, *EPTCS* 160, pp. 16–29. Open Publishing Association, 2014.
14 B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Math. Structures Comput. Sci.*, 6(5):409–453, 1996.
15 Gordon D. Plotkin. Call-by-name, call-by-value, and the lambda-calculus. *Theoret. Comput. Sci.*, 1:125–159, 1975.
16 A. Rensink and W. Vogler. Fair testing. *Inform. and Comput.*, 205(2):125–198, 2007.
17 D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *Proc. PPDP 2011*, pp. 101–112. ACM, 2011.
18 D. Sabel and M. Schmidt-Schauß. Conservative concurrency in Haskell. In *Proc. LICS 2012*, pp. 561–570. IEEE, 2012.
19 D. Sabel. Structural rewriting in the pi-calculus. In *Proc. WPTE 2014*, volume 40 of *OASIcs*, pages 51–62. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.
20 D. Sangiorgi and D. Walker. *The π-calculus: a theory of mobile processes.* CUP, 2001.
21 M. Schmidt-Schauß and D. Sabel. Closures of may-, should- and must-convergences for contextual equivalence. *Inform. Process. Lett.*, 110(6):232 – 235, 2010.
22 M. Schmidt-Schauß, D. Sabel, J. Niehren, and J. Schwinghammer. Observational program calculi and the correctness of translations. *Theoret. Comput. Sci.*, 577:98–124, 2015.

A Proofs for the Calculus Π_{Stop}

Proofs of Lemmas 3.1 and 3.2. For Lemma 3.1, we analyze all contexts of hole depth 1:

1. $C = [\cdot] \mid S$: Then $\sigma(C[P]) \mid R \equiv (\sigma(P) \mid R')$ and $\sigma(C[Q]) \mid R \equiv \sigma(Q) \mid R'$ with $R' = \sigma(S) \mid R$. The precondition of the claim implies that $\sigma(P) \mid R' \leq_{\downarrow} \sigma(Q) \mid R'$ and thus Proposition 2.3 shows $\sigma(C[P]) \mid R \leq_{\downarrow} \sigma(C[Q]) \mid R$.
2. $C = S \mid [\cdot]$: The claim follows from the previous item and Proposition 2.3.
3. $C = \nu x.[\cdot]$: Since $\sigma(P) \mid R \leq_{\downarrow} \sigma(Q) \mid R$ holds by the precondition of the claim, Lemma 2.4 shows the claim.
4. $C = x(y).[\cdot]$: Let $\sigma(C[P]) \mid R \xrightarrow{sr,n} P_n$ where P_n is successful. We use induction on n. The base case $n = 0$ holds, since in this case R must be successful, and thus $\sigma(C[Q]) \mid R$ is successful, too. For the induction step assume $\sigma(x) = x_1$ and w.l.o.g. $\sigma(y) = y$. Let $x_1(y).\sigma(P) \mid R \xrightarrow{sr} \nu\mathcal{X}.\sigma(P)[z/y] \mid R'$ be the first reduction step of the reduction sequence, where $\mathcal{X} \subseteq \{z\}$. The same reduction step for $\sigma(x(y).Q) \mid R$ results in $\nu\mathcal{X}.\sigma'(Q)[z/y] \mid R'$. By induction assumption, the lemma holds for the pair $\sigma(P)[z/y]$ and $\sigma(Q)[z/y]$, and by item (3) also for extending it with ν.
5. $C = \overline{x}y.[\cdot]$: This case is similar to the previous item.
6. $C = \,![\cdot]$. Let $\sigma(!P) \mid R \xrightarrow{sr,n} P_n$ where $\mathsf{stop}(P_n)$. We show $\sigma(!Q) \mid R{\downarrow}$ by induction on n. If $n = 0$, then R or P is successful. Thus $\mathsf{stop}(\sigma(P) \mid R)$ holds and the precondition of the lemma shows $(\sigma(Q) \mid R){\downarrow}$, which implies $\sigma(!Q) \mid R{\downarrow}$. For $n > 0$, let $\sigma(!P) \mid R \xrightarrow{sr} P_1$ be the first reduction of $\sigma(!P) \mid R \xrightarrow{sr,n} P_n$: If the redex is inside R, then the same reduction can be performed for $\sigma(!Q \mid R)$ and then

the induction hypothesis shows the claim. If the redex uses one instance of $\sigma(P)$ and parts of R, i.e. $P_1 \equiv \sigma(!\,P) \mid R_P$, s.t. $R \mid \sigma(P) \xrightarrow{sr} R_P$, then we apply the induction hypothesis to P_1 and have $(\sigma(!\,Q) \mid R_P)\downarrow$. This implies $\sigma(!\,Q) \mid \sigma(P) \mid R\downarrow$, since $\sigma(!\,Q) \mid \sigma(P) \mid R \xrightarrow{sr} \sigma(!\,Q) \mid R_P$. By the precondition of the lemma we have $\sigma(!\,Q) \mid \sigma(P) \mid R \equiv \sigma(P) \mid (\sigma(!\,Q) \mid R) \leq_\downarrow \sigma(Q) \mid (\sigma(!\,Q) \mid R) \equiv \sigma(!\,Q) \mid R$, and thus we have $\sigma(!\,Q) \mid R\downarrow$. If the redex uses two instances of $\sigma(P)$, i.e. $P_1 \equiv \sigma(!\,P) \mid R \mid P'$, s.t. $\sigma(P) \mid \sigma(P) \xrightarrow{sr} P'$, then the induction hypothesis for P_1 shows $\sigma(!\,Q) \mid R \mid P'\downarrow$. Since $\sigma(P) \mid \sigma(P) \xrightarrow{sr} P'$, we have $\sigma(!\,Q) \mid R \mid \sigma(P) \mid \sigma(P)\downarrow$. We apply the precondition twice: $\sigma(!\,Q) \mid R \mid \sigma(P) \mid \sigma(P) \equiv \sigma(P) \mid (\sigma(P) \mid (\sigma(!\,Q) \mid R)) \leq_\downarrow \sigma(Q) \mid (\sigma(P) \mid (\sigma(!\,Q) \mid R)) \equiv \sigma(P) \mid (\sigma(!\,Q) \mid R) \leq_\downarrow \sigma(Q) \mid (\sigma(!\,Q) \mid R) \equiv \sigma(!\,Q) \mid R$ and thus $\sigma(!\,Q) \mid R\downarrow$.

The proof of Lemma 3.2 is analogous to Lemma 3.1 by replacing $\leq_\downarrow$ with $\leq_\uparrow$, and replacing the base cases "if $\sigma(C[P]) \mid R$ is successful, then $\sigma(C[Q]) \mid R\downarrow$" with "if $(\sigma(C[P]) \mid R)\Uparrow$, then $(\sigma(C[Q]) \mid R)\uparrow$" which holds, since $\sigma(C[Q]) \mid R \leq_\downarrow \sigma(C[P]) \mid R$. ◀

▶ **Lemma A.1.** *For P, Q, S_{xy}, S_{yx} as defined in Proposition 3.8: $P \leq_{c,\downarrow} Q$.*

Proof. Let $\mathcal{S} := \mathcal{S}_1 \cup \mathcal{S}_2 \cup \precsim_\downarrow$ where $\mathcal{S}_2 := \{(\sigma(P) \mid R, \sigma(Q) \mid R) \mid$ for any R and $\sigma\}$ and

$\mathcal{S}_1 := \{((!\,S_{xy} \mid !\,S_{yx} \mid R[x/w] \mid \overline{y}u_1 \mid \ldots \mid \overline{y}u_n), (!\,S_{xy} \mid !\,S_{yx} \mid R[y/w] \mid \overline{x}u_1 \mid \ldots \mid \overline{x}u_n))$
$\mid$ for any R, any x, y, w, u_i, and any $n \geq 0\}$

For proving $P \leq_{c,\downarrow} Q$, it suffices to show that the relation $\mathcal{S}$ is $F_\downarrow$-dense: This implies $\sigma(P) \mid R \leq_\downarrow \sigma(Q) \mid R$ for all R, σ and thus the context lemma (Theorem 3.3) shows $P \leq_{c,\downarrow} Q$.

First let $(P_1, P_2) \in \mathcal{S}_1$. If P_1 is successful, then clearly also P_2 is successful and thus $P_2\downarrow$. If $P_1 \xrightarrow{sr} P_1'$, then there are following cases:

- If the redex is inside $R[x/w]$, then either the same reduction can also be performed for P_2, then $P_2 \xrightarrow{sr} P_2'$ and $(P_1', P_2') \in \mathcal{S}$, or the name x occurs in R. We consider two cases, where we use the abbreviations $L_x := \overline{x}u_1 \mid \ldots \mid \overline{x}u_n$ and $L_y := \overline{y}u_1 \mid \ldots \mid \overline{y}u_n$:
  1. If $R = \nu\mathcal{W}.(w(z').R_1 \mid \overline{x}v.R_2 \mid R_3)$ and $P_1' = !\,S_{xy} \mid !\,S_{yx} \mid \nu\mathcal{W}.(R_1[v/z'] \mid R_2 \mid R_3)[x/w] \mid L_y$, then $P_2 \xrightarrow{sr} P_2' \xrightarrow{sr} P_2''$ with $P_2 = !\,S_{xy} \mid !\,S_{yx} \mid L_x \mid \nu\mathcal{W}.(w(z').R_1 \mid \overline{x}v.R_2 \mid R_3)[y/w]$ and $P_2'' = !\,S_{xy} \mid !\,S_{yx} \mid L_x \mid \nu\mathcal{W}.(R_1[v/z'] \mid R_2 \mid R_3))[y/w]$, since $\overline{x}v.R_2 \mid S_{xy} \xrightarrow{sr} R_2 \mid \overline{y}v$. Since $(P_1', P_2'') \in \mathcal{S}$, we are finished.
  2. If $R = \nu\mathcal{W}.(x(z').R_1 \mid \overline{w}v.R_2 \mid R_3)$ and for P_1' we have $P_1' = !\,S_{xy} \mid !\,S_{yx} \mid \nu\mathcal{W}.(R_1[v/z'] \mid R_2 \mid R_3)[x/w] \mid L_y$, then there exists the reduction $P_2 \xrightarrow{sr} P_2' \xrightarrow{sr} P_2''$ with $P_2 = !\,S_{xy} \mid !\,S_{yx} \mid L_x \mid \nu\mathcal{W}.(x(z').R_1 \mid \overline{w}v.R_2 \mid R_3)[y/w]$ and $P_2'' = !\,S_{xy} \mid !\,S_{yx} \mid L_x \mid \nu\mathcal{W}.(R_1[v/z'] \mid R_2 \mid R_3))[y/w]$, since $\overline{y}v.R_2 \mid S_{yx} \xrightarrow{sr} R_2 \mid \overline{x}v$ and thus $(P_1', P_2'') \in \mathcal{S}$.
- The redex is $S_{yx} \mid \overline{y}u_i$, i.e. with the abbreviation $L_y = \overline{y}u_1 \mid \ldots \overline{y}u_{i-1} \mid \overline{y}u_{i+1} \mid \ldots \overline{y}u_n$, the reduction is $P_1 = !\,S_{xy} \mid !\,S_{yx} \mid R[x/w] \mid \overline{y}u_i \mid L_y \xrightarrow{sr} !\,S_{xy} \mid !\,S_{yx} \mid R[x/w] \mid \overline{x}u_i \mid L_y \equiv !\,S_{xy} \mid !\,S_{yx} \mid (R \mid \overline{w}u_i)[x/w] \mid L_y = P_1'$. Then for $L_x := \overline{x}u_1 \mid \ldots \overline{x}u_{i-1} \mid \overline{x}u_{i+1} \mid \ldots \overline{x}u_n$, there is the following reduction for process P_2: $P_2 = !\,S_{xy} \mid !\,S_{yx} \mid R[y/w] \mid \overline{x}u_i \mid L_x \xrightarrow{sr} !\,S_{xy} \mid !\,S_{yx} \mid R[y/w] \mid \overline{y}u_i \mid L_x \equiv !\,S_{xy} \mid !\,S_{yx} \mid (R \mid \overline{w}u_i)[y/w] \mid L_x = P_2'$ and $(P_1', P_2') \in \mathcal{S}$.
- The redex is $S_{xy} \mid R[x/w]$, i.e. $R = \overline{w}v.R'$ and for $L_y := \overline{y}u_1 \mid \ldots \mid \overline{y}u_n$ we have $P_1 = !\,S_{xy} \mid !\,S_{yx} \mid \overline{x}v.R'[x/w] \mid L_y \xrightarrow{sr} !\,S_{xy} \mid !\,S_{yx} \mid R'[x/w] \mid \overline{y}v \mid L_y = P_1'$. Then for $L_x := \overline{x}u_1 \mid \ldots \mid \overline{x}u_n$ the reduction $P_2 = !\,S_{xy} \mid !\,S_{yx} \mid \overline{y}v.R'[y/w] \mid L_x \xrightarrow{sr} P_2'$ exists, where $P_2' := !\,S_{xy} \mid !\,S_{yx} \mid R'[y/w] \mid \overline{x}v \mid L_x = P_2'$ and thus $(P_1', P_2') \in \mathcal{S}$.

Now let $(P_1, P_2) \in \mathcal{S}_2$ and let $a' = \sigma(a), x' = \sigma(x), y' = \sigma(y), z' = \sigma(z)$. If P_1 is successful, then P_2 is successful. If $P_1 \xrightarrow{sr} P_1'$, then there are the cases:

- If the redex is inside R, then $P_2 \xrightarrow{sr} P_2'$ and $(P_1', P_2') \in \mathcal{S}$.

- If $R = \nu\mathcal{W}.(a'(w).R' \mid R'')$ and $P_1 \xrightarrow{sr} !\,\sigma(S_{xy}) \mid !\,\sigma(S_{yx}) \mid \nu\mathcal{W}.(R'[x/w] \mid R'') := P_1'$. Then $P_1' \equiv !\,\sigma(S_{xy}) \mid !\,\sigma(S_{yx}) \mid \nu\mathcal{W}.(R' \mid R'')[x/w]$, since we may assume that w was renamed fresh for R''. Then $P_2 \xrightarrow{sr} P_2'$ with $P_2' := !\,\sigma(S_{xy}) \mid !\,\sigma(S_{yx}) \mid \nu\mathcal{W}.(R'[y/w] \mid R'')$. Since $P_2' \equiv !\,\sigma(S_{xy}) \mid !\,\sigma(S_{yx}) \mid \nu\mathcal{W}.(R' \mid R'')[y/w] = P_2'$, this shows $(P_1', P_2') \in \mathcal{S}$.
- $R = \nu\mathcal{W}.(\overline{x'}u.R' \mid R'')$ and $P_1 \xrightarrow{sr} \overline{a'}x' \mid !\,\sigma(S_{xy}) \mid !\,\sigma(S_{yx}) \mid (R' \mid R) \mid \overline{y'}u = P_1'$. Then $P_2 \xrightarrow{sr} \overline{a'}y' \mid !\,\sigma(S_{xy}) \mid !\,\sigma(S_{yx}) \mid (R' \mid R) \mid \overline{y'}u = P_2'$. and $(P_1', P_2') \in \mathcal{S}$.
- $R = \nu\mathcal{W}.(\overline{y'}u.R' \mid R'')$ and $P_1 \xrightarrow{sr} \overline{a'}x' \mid !\,\sigma(S_{xy}) \mid !\,\sigma(S_{yx}) \mid (R' \mid R) \mid \overline{x'}u = P_1'$. Then $P_2 \xrightarrow{sr} \overline{a'}y' \mid !\,\sigma(S_{xy}) \mid !\,\sigma(S_{yx}) \mid (R' \mid R) \mid \overline{x'}u = P_2'$ and $(P_1', P_2') \in \mathcal{S}$. ◀

▶ **Lemma A.2.** *The relation $\mathcal{S} := \{(\nu\mathcal{X}.(P \mid R), \nu\mathcal{X}.(Q \mid R)) \mid P \precsim_{b,\downarrow} Q,$ for any $\mathcal{X}$, $R\} \cup \precsim_{\downarrow}$ is $F_{\downarrow}$-dense.*

Proof. Let $(\nu\mathcal{X}.(P \mid R), \nu\mathcal{X}.(Q \mid R)) \in \mathcal{S}$. We have to show $(\nu\mathcal{X}.(P \mid R), \nu\mathcal{X}.(Q \mid R)) \in F_{\downarrow}(\mathcal{S})$. If $\nu\mathcal{X}.(P \mid R)$ is successful, then P or R is successful too, and thus either $Q\downarrow$ and so does $\nu\mathcal{X}.Q \mid R$ or $\nu\mathcal{X}.(Q \mid R)$ is already successful. For $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} P_1$ we show that $\nu\mathcal{X}.(Q \mid R) \xrightarrow{sr,*} Q_1$, s.t. $(P_1, Q_1) \in \mathcal{S}$: If the redex of $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} P_1$ is inside P, i.e. $P_1 = \nu\mathcal{X}.(P' \mid R)$, then by $P \precsim_{b,\downarrow} Q$ there exists Q' with $Q \xrightarrow{sr,*} Q'$, $P' \precsim_{b,\downarrow} Q'$. Since also $\nu\mathcal{X}.(Q \mid R) \xrightarrow{sr,*} \nu\mathcal{X}.(Q' \mid R)$ and thus $(\nu\mathcal{X}.(P' \mid R), \nu\mathcal{X}.(Q' \mid R)) \in \mathcal{S}$, this case is finished. If the redex of $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} P_1$ is inside R, i.e. $P_1 = \nu\mathcal{X}.(P \mid R')$ then also $\nu\mathcal{X}.(Q \mid R) \xrightarrow{sr} \nu\mathcal{X}.(Q \mid R')$ and $(\nu\mathcal{X}.(P \mid R'), \nu\mathcal{X}.(Q \mid R')) \in \mathcal{S}$.

The remaining cases are that the redex uses parts of P and parts of R.

- If $P \equiv \nu\mathcal{X}_1.(x(y).P' \mid P'')$, $R \equiv \nu\mathcal{X}_2.(\overline{x}z.R' \mid R'')$ with $z \notin \mathcal{X}_2$ and $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} \nu\mathcal{X}.(\nu\mathcal{X}_1.(P'[z/y] \mid P'') \mid \nu\mathcal{X}_2.R' \mid R'') = P_1$, then by $P \precsim_{b,\downarrow} Q$ there exists Q_0 s.t. $Q \xrightarrow{sr,*} Q_0 = \nu\mathcal{Y}_1.(x(y).Q' \mid Q'')$ and $\mathcal{X}_1.(P'[z/y] \mid P'') \precsim_{b,\downarrow} \nu\mathcal{Y}_1.(Q'[z/y] \mid Q'')$. Since $\nu\mathcal{X}.(Q \mid R) \xrightarrow{sr,*} \nu\mathcal{X}.(Q_0 \mid R) \xrightarrow{sr} \nu\mathcal{Y}_1.(Q'[z/y] \mid Q'') \mid \nu\mathcal{X}_2.R' \mid R'') = Q_1$, $(P_1, Q_1) \in \mathcal{S}$.
- If $P \equiv \nu\mathcal{X}_1.(x(y).P' \mid P'')$, $R \equiv \nu z, \mathcal{X}_2.(\overline{x}z.R' \mid R'')$ and $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} P_1$ with $P_1 := \nu\mathcal{X}.(\nu z.(\nu\mathcal{X}_1.P'[z/y] \mid P'' \mid \nu\mathcal{X}_2.(R' \mid R'')))$, then by $P \precsim_{b,\downarrow} Q$ there exists a process Q_0 s.t. $Q \xrightarrow{sr,*} Q_0$, $Q_0 = \nu\mathcal{Y}_1.(x(y).Q' \mid Q'')$ and $\nu\mathcal{X}_1.(P'[z/y] \mid P'') \precsim_{b,\downarrow} \nu\mathcal{Y}_1.(Q'[z/y] \mid Q'')$. Since $\nu\mathcal{X}.(Q \mid R) \xrightarrow{sr,*} \nu\mathcal{X}.(Q_0 \mid R) \xrightarrow{sr} \nu\mathcal{X}.(\nu z.(\nu\mathcal{Y}_1.(Q'[z/y] \mid Q'') \mid \nu\mathcal{X}_2.(R' \mid R''))) = Q_1$ we have $(P_1, Q_1) \in \mathcal{S}$.
- If $P \equiv \nu\mathcal{X}_1.(\overline{x}y.P' \mid P'')$ and $R \equiv \nu\mathcal{X}_2.(x(z).R' \mid R'')$ with $y \notin \mathcal{X}_1$ and $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} \nu\mathcal{X}.(\nu\mathcal{X}_1.P' \mid P'') \mid \nu\mathcal{X}_2.(R'[y/z] \mid R'') = P_1$, then by $P \precsim_{b,\downarrow} Q$ there exists Q_0 with $Q \xrightarrow{sr,*} Q_0$, $Q_0 = \nu\mathcal{Y}_1.(\overline{x}y.Q' \mid Q'')$ where $y \notin \mathcal{Y}_1$ s.t. $\nu\mathcal{X}_1.(P' \mid P'') \precsim_{b,\downarrow} \mathcal{Y}_1.(Q' \mid Q'')$. Since also $\nu\mathcal{X}.(Q \mid R) \xrightarrow{sr,*} \nu\mathcal{X}.(Q_0 \mid R) \xrightarrow{sr} \nu\mathcal{X}.(\nu\mathcal{Y}_1.(Q' \mid Q'') \mid \nu\mathcal{X}_2.(R'[y/z] \mid R'')) = Q_1$, we have $(P_1, Q_1) \in \mathcal{S}$.
- If $P \equiv \nu y.\nu\mathcal{X}_1.(\overline{x}y.P' \mid P'')$, $R \equiv \nu\mathcal{X}_2.(x(z).R' \mid R'')$, and $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} P_1$ where $P_1 := \nu\mathcal{X}.\nu y.(\nu\mathcal{X}_1.(P' \mid P'') \mid \nu\mathcal{X}_2.(R'[y/z] \mid R''))$, then by $P \precsim_{b,\downarrow} Q$ there exists Q_0 with $Q \xrightarrow{sr,*} Q_0$, $Q_0 = \nu y.\nu\mathcal{Y}_1.(\overline{x}y.Q' \mid Q'')$ s.t. $\nu\mathcal{X}_1.(P' \mid P'') \precsim_{b,\downarrow} \nu\mathcal{Y}_1.(Q' \mid Q'')$. Since also $\nu\mathcal{X}.\nu y.(Q \mid R) \xrightarrow{sr,*} \nu\mathcal{X}.\nu y.(Q_0 \mid R) \xrightarrow{sr} \nu\mathcal{X}.\nu y.(\nu\mathcal{Y}_1.(Q' \mid Q'') \mid \nu\mathcal{X}_2.(R'[y/z] \mid R'')) = Q_1$, we have $(P_1, Q_1) \in \mathcal{S}$. ◀

▶ **Lemma A.3.** *The relation $\mathcal{S} := \{(\nu\mathcal{X}.(P \mid R), \nu\mathcal{X}.(Q \mid R)) \mid P \precsim_{b,\uparrow} Q,$ for any $\mathcal{X}$, $R\} \cup \precsim_{\uparrow}$ is $F_{\uparrow}$-dense.*

Proof. Note that if $P \precsim_{b,\uparrow} Q$, then $Q \precsim_{b,\downarrow} P$. Let $(\nu\mathcal{X}.(P \mid R), \nu\mathcal{X}.(Q \mid R)) \in \mathcal{S}$. We have to show that $(\nu\mathcal{X}.(P \mid R), \nu\mathcal{X}.(Q \mid R)) \in F_{\uparrow}(\mathcal{S})$. If $\nu\mathcal{X}.(P \mid R) \Uparrow$, then $Q \precsim_{b,\downarrow} P$ and Proposition 3.11 show that $\nu\mathcal{X}.(Q \mid R) \precsim_{\downarrow} \nu\mathcal{X}.(P \mid R)$ which implies that $\nu\mathcal{X}.(P \mid R) \leq_{\Uparrow} \nu\mathcal{X}.(Q \mid R)$ and thus $\nu\mathcal{X}.(Q \mid R) \downarrow$. If $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} P_1$, then we have to show that $\nu\mathcal{X}.(Q \mid R) \xrightarrow{sr,*} Q_1$, s.t. $(P_1, Q_1) \in \mathcal{S}$. If the redex of $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} P_1$ is inside P, i.e. $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} \nu\mathcal{X}.(P' \mid R)$ then $P \precsim_{b,\uparrow} Q$ shows that $Q \xrightarrow{sr,*} Q'$ s.t. $P' \precsim_{b,\uparrow} Q'$. Since $\nu\mathcal{X}.(Q \mid R) \xrightarrow{sr,*} \nu\mathcal{X}.(Q' \mid R)$ and thus $(\nu\mathcal{X}.(P' \mid R), \nu\mathcal{X}.(Q' \mid R)) \in \mathcal{S}$ in this case. If the redex of $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} P_1$ is inside R, i.e. $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} \nu\mathcal{X}.(P \mid R')$ then also $\nu\mathcal{X}.(Q \mid R) \xrightarrow{sr} \nu\mathcal{X}.(Q \mid R')$ and thus $(\nu\mathcal{X}.(P \mid R'), \nu\mathcal{X}.(Q \mid R')) \in \mathcal{S}$ in this case.

It remains to consider the cases where the redex uses parts of P and parts of R.

- If $P \equiv \nu\mathcal{X}_1.(x(y).P' \mid P'')$, $R \equiv \nu\mathcal{X}_2.(\overline{x}z.R' \mid R'')$ with $z \notin \mathcal{X}_2$ and $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} \nu\mathcal{X}.(\nu\mathcal{X}_1.(P'[z/y] \mid P'') \mid \nu\mathcal{X}_2.R' \mid R'') = P_1$, then by $P \precsim_{b,\uparrow} Q$ there exists Q_0 with $Q \xrightarrow{sr,*} Q_0 = \nu\mathcal{Y}_1.(x(y).Q' \mid Q'')$ s.t. $\mathcal{X}_1.(P'[z/y] \mid P'') \precsim_{b,\uparrow} \nu\mathcal{Y}_1.(Q'[z/y] \mid Q'')$. Since $\nu\mathcal{X}.(Q \mid R) \xrightarrow{sr,*} \nu\mathcal{X}.(Q_0 \mid R) \xrightarrow{sr} \nu\mathcal{Y}_1.(Q'[z/y] \mid Q'') \mid \nu\mathcal{X}_2.R' \mid R'') = Q_1$ this shows $(P_1, Q_1) \in \mathcal{S}$.
- If $P \equiv \nu\mathcal{X}_1.(x(y).P' \mid P'')$, $R \equiv \nu z, \mathcal{X}_2.(\overline{x}z.R' \mid R'')$, and $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} P_1$ where $P_1 := \nu\mathcal{X}.(\nu z.(\nu\mathcal{X}_1.P'[z/y] \mid P'' \mid \nu\mathcal{X}_2.(R' \mid R'')))$, then $P \precsim_{b,\uparrow} Q$ shows that there exists Q_0 with $Q \xrightarrow{sr,*} Q_0 = \nu\mathcal{Y}_1.(x(y).Q' \mid Q'')$ s.t. $\nu\mathcal{X}_1.(P'[z/y] \mid P'') \precsim_{b,\uparrow} \nu\mathcal{Y}_1.(Q'[y/z] \mid Q'')$. Since $\nu\mathcal{X}.(Q \mid R) \xrightarrow{sr,*} \nu\mathcal{X}.(Q_0 \mid R) \xrightarrow{sr} \nu\mathcal{X}.(\nu z.(\nu\mathcal{Y}_1.(Q'[z/y] \mid Q'') \mid \nu\mathcal{X}_2.(R' \mid R''))) = Q_1$ we have $(P_1, Q_1) \in \mathcal{S}$.
- If $P \equiv \nu\mathcal{X}_1.(\overline{x}y.P' \mid P'')$, $R \equiv \nu\mathcal{X}_2.(x(z).R' \mid R'')$ with $y \notin \mathcal{X}_1$, and $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} P_1$ with $P_1 := \nu\mathcal{X}.(\nu\mathcal{X}_1.P' \mid P'') \mid \nu\mathcal{X}_2.(R'[y/z] \mid R'')$, then $P \precsim_{b,\uparrow} Q$ shows that $Q \xrightarrow{sr,*} Q_0$ with $Q_0 := \nu\mathcal{Y}_1.(y.Q' \mid Q'')$ where $y \notin \mathcal{Y}_1$ s.t. $\nu\mathcal{X}_1.(P' \mid P'') \precsim_{b,\uparrow} \mathcal{Y}_1.(Q' \mid Q'')$. Since $\nu\mathcal{X}.(Q \mid R) \xrightarrow{sr,*} \nu\mathcal{X}.(Q_0 \mid R) \xrightarrow{sr} \nu\mathcal{X}.(\nu\mathcal{Y}_1.(Q' \mid Q'') \mid \nu\mathcal{X}_2.(R'[y/z] \mid R'')) = Q_1$, we have $(P_1, Q_1) \in \mathcal{S}$.
- If $P \equiv \nu y, \mathcal{X}_1.(\overline{x}y.P' \mid P'')$, $R \equiv \nu\mathcal{X}_2.(x(z).R' \mid R'')$, and $\nu\mathcal{X}.(P \mid R) \xrightarrow{sr} P_1$ with $P_1 := \nu\mathcal{X}.\nu y.(\nu\mathcal{X}_1.(P' \mid P'') \mid \nu\mathcal{X}_2.(R'[y/z] \mid R''))$, then $P \precsim_{b,\uparrow} Q$ shows that $Q \xrightarrow{sr,*} Q_0$ with $Q_0 = \nu y.\nu\mathcal{Y}_1.(y.Q' \mid Q'')$ s.t. $\nu\mathcal{X}_1.(P' \mid P'') \precsim_{b,\uparrow} \nu\mathcal{Y}_1.(Q' \mid Q'')$. Since also the reduction $\nu\mathcal{X}.\nu y.(Q \mid R) \xrightarrow{sr,*} \nu\mathcal{X}.\nu y.(Q_0 \mid R) \xrightarrow{sr} Q_1$ exists, where the process Q_1 is $Q_1 := \nu\mathcal{X}.\nu y.(\nu\mathcal{Y}_1.(Q' \mid Q'') \mid \nu\mathcal{X}_2.(R'[y/z] \mid R''))$, this shows$(P_1, Q_1) \in \mathcal{S}$. ◀

Formalizing Bialgebraic Semantics in PVS 6.0

Sjaak Smetsers[1], Ken Madlener[1], and Marko van Eekelen[1,2]

1 **Institute for Computing and Information Sciences, Radboud University**
Postbus 9010, 6500 GL Nijmegen, The Netherlands
`{S.Smetsers,K.Madlener}@cs.ru.nl`

2 **School of Computer Science, Open University of the Netherlands**
Postbus 2960, 6401 DL Heerlen, The Netherlands
`{M.vanEekelen}@cs.ru.nl`

Abstract

Both operational and denotational semantics are prominent approaches for reasoning about properties of programs and programming languages. In the categorical framework developed by Turi and Plotkin both styles of semantics are unified using a single, syntax independent format, known as GSOS, in which the operational rules of a language are specified. From this format, the operational and denotational semantics are derived. The approach of Turi and Plotkin is based on the categorical notion of bialgebras. In this paper we specify this work in the theorem prover *PVS*, and prove the adequacy theorem of this formalization. One of our goals is to investigate whether *PVS* is adequately suited for formalizing metatheory. Indeed, our experiments show that the original categorical framework can be formalized conveniently. Additionally, we present a GSOS specification for the simple imperative programming language *While*, and execute the derived semantics for a small example program.

1998 ACM Subject Classification F.4.1 Mathematical Logic, F.3.2 Semantics of Programming Languages

Keywords and phrases operational semantics, denotational semantics, bialgebras, distributive laws, adequacy, theorem proving, *PVS*, *While*.

Digital Object Identifier 10.4230/OASIcs.WPTE.2015.47

1 Introduction

The formal definition of a real-world programming language is often a monumental undertaking. The process of verifying metatheory often exceeds human capabilities; due to its inherent complexity, mechanization time, even for the interesting core facets of the semantics of real-world programming languages, is prohibitive. The best alternative for complete verification is to employ well-established methods, such as type systems or the use of mechanized verification tools. These verification tools are usually based on typed higher-order logic. The specification languages of these tools often provide automatic code generation, which enables the execution of specifications. This feature can be used as an additional check of the developed concepts, before one starts with formally proving properties of these concepts.

In this paper, we present a formalization of both popular styles of semantic specifications: (structural) operational semantics and denotational semantics. Our main goal is to experiment with *PVS*'s latest feature, so-called *declaration parameters*, which enable the specification of polymorphic functions and data structures. The experiments are carried out with *PVS* version 6.0, released in February 2013. Previous versions of *PVS* already offered a limited form of polymorphism by means of theory level parameters. However, there are situations where these theory parameters are inconvenient, particularly if notions are involved with a generic nature, such as the categorical concepts used in our formalization.

Our approach is based on a framework developed by Turi and Plotkin [18] unifying both styles of semantics. By exploiting the language of category theory, they managed to disassociate from language-specific details such as concrete syntax and behavior. Given a set of operational rules, they derived both operational and denotational semantics using the concept of *bialgebras* equipped with a *distributive law*. The format in which these operational rules are specified is known as the *GSOS (Grand Structural Operational Semantics) format*, introduced by [3]. This format originates from the theory of SOS, and has been given a categorical interpretation by means of bialgebras. Turi and Plotkin prove that the operational and denotational semantics generated from any set of GSOS rules are consistent for every input program. Our work is also inspired by previous comparable experiments with the *Coq* proof assistant; see [14].

The contribution of our work is threefold. First, we investigate whether *PVS* 6.0 is sufficiently expressive for formalizing metatheory. Until now, such *PVS* developments have been constrained to only a few metatheoretical experiments. This seems to be a missed opportunity, because *PVS* has shown to be very successful in proving properties of computer programs. In our attempt, we extend to a very high abstraction level, namely we use Turi and Plotkin's categorical description [18] as the point of inception. Our goal is to formally prove that their construction is sound, which is expressed in the *adequacy theorem*. The second, but not less important contribution is a *PVS* formalization providing a framework for facilitating both formal reasoning about, and experimentation with semantics of programming languages[1]. The setup is such that the user is free to choose between either denotational or operational semantics at any point. And third, we illustrate the expressiveness of our framework with an elaborate example of a standard imperative language specified in the GSOS format. Until now, concrete applications of GSOS are mostly restricted to process algebras. The reader is expected to have a modest knowledge of Category Theory. If not, we recommend [1] as an easy-going starting point.

2 Background

A brief introduction of the basic prerequisite concepts for this paper follows. Let us start with an example following the approach as, for instance, taken by [9] and [11]. We explain and illustrate the technicalities using a very simple language of streams (see also [12]). The operational rules for this stream language are given in Figure 1. These rules inductively define a transition relation that is a subset of $\mathbb{T} \times L \times \mathbb{T}$, where $\mathbb{T}$, L denote the sets of closed terms and output labels, respectively. The two basic operations AS and BS generate constant (infinite) streams of As and Bs whereas the operation Alt yields an alternation of two streams. The latter operates by repeatedly taking the head of its first argument, and calling itself recursively on the swapped tails, discarding the head of the second argument. The first step towards the formalization of such a language is to express both the *signature* and *behaviour* (i.e., the result/effect of an operation) of the operations as *functors*. In languages with support for higher-order polymorphism, like the functional language *Haskell*, one would express a functor as a type constructor class that is parameterized with the type (the object map) of the functor. The type class itself contains the corresponding (morphism)

[1] All definitions and theorems given in this paper have been formalized and proven. The files of the development can be obtained via `http://www.cs.ru.nl/~sjakie/papers/adequacy/`.

$$\frac{}{AS \xrightarrow{A} AS} \qquad \frac{}{BS \xrightarrow{B} BS} \qquad \frac{x \xrightarrow{l} x' \qquad y \xrightarrow{m} y'}{Alt\ x\ y \xrightarrow{l} Alt\ y'\ x'}$$

Figure 1 A simple language for streams.

map. However, *PVS* only offers first-order type variables which forces us to specify functors in a slightly more ad hoc way.

Syntax and Σ-algebras

We start with representing the syntax of a language as (open) terms, by introducing the following datatype:

```
TER [V : type, F : type, ar : [F → nat]] : datatype begin
  tvar (var_id : V)                              : tvar ?
  tapp (op : F, args : [below (ar (op)) → TER]) : tapp ?
end TER
```

The datatype is parametric in the type V of variables and the signature which is represented by the set F of operator symbols and the arity map ar. It has two *constructors* ($tvar$ and $tapp$), and two *recognizers* ($tvar?$ and $tapp?$)[2]. These recognizers are used as predicates to test whether or not a term starts with the respective constructor. The field names var_id, op and $args$ are used as *accessors* to extract these components from a term. It is more convenient, however, to define operations on inductive datatypes by pattern matching. This datatype definition also illustrates *PVS*'s special typing facility: *dependent types*, i.e., types depending on values. For example, $below\ (n)$ denotes the set of natural numbers between 0 and n.

The signature (F and ar) is encoded as a functor which we will call the *signature functor*. The following *PVS* theory[3] defines the notion of Σ-functor (consisting of a type Σ, and a map function map_Σ), and Σ-algebra (categorically, an algebra is defined as a pair consisting of an *object* X, called the *carrier* of the algebra, and a structure map $\Sigma[X] \to X$)[4]. In *PVS* syntax, a function type $T_1 \to T_2$ has the form $[T_1 \to T_2]$, and tuple types have the form $[T_1, ..., T_n]$. Associated with every n-tuple type is a set of projection functions: ‘1, ‘2, ..., i.e., by t‘i one selects the i^{th} component from t.

```
ΣAlgebra [F : type, ar : [F → nat]] : theory begin
  Σ [a : type] : type     = [f : F, [below (ar (f)) → a]]
  map_Σ [a, b : type] (f : [a → b]) (sf : Σ [a]) : Σ [b] =
    (sf‘1, λ (i : below (ar (sf‘1))) : f (sf‘2 (i)))
  Alg_Σ [a : type] : type = [Σ [a] → a]
end ΣAlgebra
```

[2] *PVS* allows question marks as constituents of identifiers.
[3] *PVS* uses parameterized theories to organize specifications, i.e, datatypes, function definitions, and properties.
[4] Here, we tacitly omit the fact that the structure map is usually equipped with two so-called *functor laws*.

In this theory we have used the new feature of *PVS* 6.0: *declaration (level) parameters*, i.e. (type) variables ranging over first-order types. These are comparable to type variables in, for instance, *Haskell* and *Coq*. Combined with the dependent typing facility, these declaration parameters enable us, for example, to specify the type part of the functor $\Sigma\ [a]$ as a dependent pair being parametric in the object type a.

To illustrate how to use these notions in combination with the simple stream language, we introduce the following auxiliary theory:

$StreamL$: **theory begin**
  $SigS$: **type** $= \{AS, BS, Alt\}$
  $arS\ (s : sigS) : nat =$ **cases** (s) **of** $AS : 0, BS : 0, Alt : 2$ **endcases**
end $StreamL$

The enumeration type *SigS* represents the set of operator symbols, whereas the function *arS* assigns arities to each of these symbols. We can now use *SigS* and *arS* as actual parameters for $\Sigma Algebra$ theory in order to obtain the concrete signature functor.

The datatype *TER* introduced earlier is also a functor. To make this more explicit, we introduce $\mathbb{T}\ [v]$ as an abbreviation for $TER\ [v, F, ar]$. When *PVS* typechecks *TER* then it automatically generates for *TER* the corresponding map operation and a folding operation called *reduce*. To adhere to standard terminology and to the notation used in the present paper, we rename these generated operations to $map_{\mathbb{T}}$ and $fold_{\mathbb{T}}$, respectively. Incidentally, the latter is a standard operation for processing terms that avoids explicit recursion, see also [16] where this operation is called a *catamorphism*.

$Terms\ [F :$ **type**$, ar : [F \to nat]]$: **theory begin importing** $TER, \Sigma Algebra\ [F, ar]$
  $\mathbb{T}\ [v :$ **type**$]$: **type** $= TER\ [v, F, ar]$
  $fold_{\mathbb{T}}\ [v, x :$ **type**$]\ (e : [v \to x], a : Alg_{\Sigma}\ [x])\ (t : \mathbb{T}\ [v]) : x = reduce\ (e, a)\ (t)$
  $map_{\mathbb{T}}\ [a, b :$ **type**$]\ (f : [a \to b])\ (t : \mathbb{T}\ [a]) : \mathbb{T}\ [b] = map\ (f)\ (t)$
end $Terms$

The following uniqueness property is based on the categorical fact that *tapp* (which is an algebra for the functor Σ) is *initial*.

▶ **Proposition 1.** *Let $e : V \to X$ and $a : Alg_{\Sigma}\ [X]$. Then $fold_{\mathbb{T}}\ (e, a)$ is unique in making the following diagram commute*[5] *:*

$$\begin{tikzcd}
V \arrow[r, "tvar"] \arrow[dr, "\forall e"'] & \mathbb{T}[V] \arrow[d, dashed, "fold_{\mathbb{T}}\ (e{,}a)"] & \Sigma[\mathbb{T}[V]] \arrow[l, "tapp"'] \arrow[d, "\Sigma\ (fold_{\mathbb{T}}\ (e{,}a))"] \\
 & X & \Sigma[X] \arrow[l, "\forall a"]
\end{tikzcd}$$

This propery appears to be very useful as an alternative for structural induction in proofs of properties on terms. In fact, it allows for a direct translation of diagrammatic proofs into a *PVS* formalization. In our experience, these (hand-drawn) diagrammatic proofs are indispensable as the initial and most important step towards a fully formalized proof. The 'textual' *PVS* version of this proposition is:

[5] For the diagrams in this paper we adopted the categorical notation for functors by writing $\mathcal{F}$ instead of $map_{\mathcal{F}}$, for some functor $\mathcal{F}$.

$$\begin{array}{l}
\mathit{fold\_unique}\;[v, x : \textsf{type}] : \textsf{proposition} \\
\quad \forall\; (f : [\mathbb{T}\;[v] \to x], e : [v \to x], a : \mathit{Alg}_{\Sigma}\;[x]) : \\
\qquad f \circ \mathit{tvar} = e \;\wedge\; f \circ \mathit{tapp} = a \circ \mathit{map}_{\Sigma}\;(f) \;\Leftrightarrow\; f = \mathit{fold}_{\mathbb{T}}\;(e, a)
\end{array}$$

Next, we show that $\mathbb{T}$ is a monad in the categorical sense, by defining the corresponding operations (so-called *natural transformations*) *unit* (embedding) and *join* (composition):

$$\begin{array}{l}
\mathbb{T}\mathit{Monad}\;[F : \textsf{type}, \mathit{ar} : [F \to \mathit{nat}]] : \textsf{theory begin importing}\;\mathit{Terms}\;[F, \mathit{ar}] \\
\quad \mathit{unit}_{\mathbb{T}}\;[v : \textsf{type}]\;(\mathit{vid} : v) : \mathbb{T}\;[v] = \mathit{tvar}\;(\mathit{vid}) \\
\quad \mathit{join}_{\mathbb{T}}\;[v : \textsf{type}]\;(\mathit{tt} : \mathbb{T}\;[\mathbb{T}\;[v]]) : \mathbb{T}\;[v] = \mathit{fold}_{\mathbb{T}}\;(\mathit{id}, \mathit{tapp})\;(\mathit{tt}) \\
\textsf{end}\;\mathbb{T}\mathit{Monad}
\end{array}$$

Likewise, from category theory we borrow the notion of $\mathbb{T}$-algebra: A $\mathbb{T}$-algebra (or, more verbosely, an algebra for the $\mathbb{T}$ monad) is a 'plain' algebra a with two additional properties:

$$a \circ \mathit{unit}_{\mathbb{T}} = \mathit{id} \quad (1) \qquad a \circ \mathit{map}_{\mathbb{T}}\;(a) = a \circ \mathit{join}_{\mathbb{T}} \quad (2)$$

Below, these properties are encoded by the predicate $\mathbb{T}\mathit{Alg}?$. Additionally, we introduce a slightly modified version of $\mathit{fold}_{\mathbb{T}}$, named $\mathit{free}_{\mathbb{T}}$, taking a $\mathbb{T}$-algebra as argument instead of a Σ-algebra. In *PVS*:

$$\begin{array}{l}
\mathbb{T}\mathit{Algebra}\;[F : \textsf{type}, \mathit{ar} : [F \to \mathit{nat}]] : \textsf{theory begin importing}\;\mathit{TMonad}\;[F, \mathit{ar}] \\
\quad \mathit{Alg}_{\mathbb{T}}\;[a : \textsf{type}] : \textsf{type} = [\mathbb{T}\;[a] \to a] \\
\quad \mathit{free}_{\mathbb{T}}\;[v, w : \textsf{type}]\;(e : [v \to w], a : \mathit{Alg}_{\mathbb{T}}\;[w]) : [\mathbb{T}\;[v] \to w] = \mathit{fold}_{\mathbb{T}}\;(e, a \circ \mathit{tapp} \circ \mathit{map}_{\Sigma}\;(\mathit{tvar})) \\
\quad \mathbb{T}\mathit{Alg}?\;[w : \textsf{type}]\;(a : \mathit{Alg}_{\mathbb{T}}\;[w]) : \mathit{bool} = a \circ \mathit{unit}_{\mathbb{T}} = \mathit{id} \;\wedge\; a \circ \mathit{map}_{\mathbb{T}}\;(a) = a \circ \mathit{join}_{\mathbb{T}} \\
\textsf{end}\;\mathbb{T}\mathit{Algebra}
\end{array}$$

We end this section with a proposition supplying $\mathit{free}_{\mathbb{T}}$ with a proof principle.

▶ **Proposition 2.** *Let $e : V \to W$ and $a : \mathit{Alg}_{\mathbb{T}}\;[W]$ such that $\mathbb{T}\mathit{Alg}?\;(a)$ holds. Then, $\mathit{free}_{\mathbb{T}}\;(e, a)$ is unique in making the following diagram commute:*

$$\begin{tikzcd}
V \arrow[r, "\mathit{unit}_{\mathbb{T}}"] \arrow[dr, "\forall\; e"'] & \mathbb{T}[V] \arrow[d, dashed, "\mathit{free}_{\mathbb{T}}\;(e,a)"] & \mathbb{T}[\mathbb{T}[V]] \arrow[l, "\mathit{join}_{\mathbb{T}}"'] \arrow[d, "\mathbb{T}\;(\mathit{free}_{\mathbb{T}}\;(e,a))"] \\
& W & \mathbb{T}[W] \arrow[l, "\forall\; a"]
\end{tikzcd}$$

Behaviour and $\mathcal{B}$-coalgebras

The operational semantics of a language is given by a transition relation representing the execution steps of an abstract machine. These transition relations can be modelled in a categorial manner using coalgebras (e.g., see [10]). A coalgebra is the dual of an algebra: for a functor $\mathcal{B}$, the coalgebra consists of a carrier C, and a structure map $C \to \mathcal{B}[C]$. We express a transition relation as a $\mathcal{B}$-coalgebra, with carrier $\mathbb{T}\;[V]$, more specifically, as a function with type $\mathbb{T}\;[V] \to \mathcal{B}[\mathbb{T}[V]]$. We call this coalgebraic formalization the *operational model.*

For the stream language, the functor $\mathcal{B}$ is defined by $\mathcal{B}\;X = (L, X)$, i.e., $\mathcal{B}$ just pairs a label from L with X. Specifying the operational rules as a coalgebra is straightforward; e.g., see [14]. In Section 4 we will give a more elaborate example.

By making the behaviour functor $\mathcal{B}$ parametric in X we anticipate the fact that the terms can be executed according to the operational rules of the language yielding an infinite stream of labels. Categorically, this stream is constructed by taking the greatest fixpoint of $\mathcal{B}$. In order to express this in *PVS*, we use *PVS*'s capability to introduce co-inductive datatypes. However, we cannot do this as a general fixpoint construction that is parametric in the behaviour functor. Incidentally, also *Coq* forbids such a construction for exactly the same technical restrictions imposed by its underlying logic, namely, such a construction would admit instantiations that have no set-theoretic semantics. Instead, we define the output stream directly as a codatatype, named $\mathbf{N}_{\mathcal{B}}$, and extract the functor $\mathcal{B}$ from this definition. In fact, no definitions are required to obtain $\mathcal{B}$: it is automatically generated from the definition of the codatatype, together with the *unfold* operation (named *coreduce*). This unfolding operation is also know as an *anamorphism*, see [16].

```
$\mathbf{N}_{\mathcal{B}}$ $[L : \textbf{type}]$ : **codatatype begin**
  $nb\_in\ (el : L, next : \mathbf{N}_{\mathcal{B}}) : nb\_in\,?$
**end** $\mathbf{N}_{\mathcal{B}}$
```

The *PVS* specification of a coalgebra is as follows:

```
$\mathcal{B}Coalgebra\ [L : \textbf{type}]$ : **theory begin importing** $\mathbf{N}_{\mathcal{B}}\ [L]$
  $\mathcal{B}\ [x : \textbf{type}] : \textbf{type}$          $= NB\_struct\ [L, x]$
  $Coalg_{\mathcal{B}}\ [x : \textbf{type}] : \textbf{type} = [x \to \mathcal{B}\ [x]]$
  $out_{\mathcal{B}} : Coalg_{\mathcal{B}}\ [\mathbf{N}_{\mathcal{B}}] = \lambda\ (nb : \mathbf{N}_{\mathcal{B}}) : inj\_nb\_in\ (el\ (nb), next\ (nb))$
  $unfold_{\mathcal{B}}\ [x : \textbf{type}]\ (c : Coalg_{\mathcal{B}}\ [x])\ (z : x) : \mathbf{N}_{\mathcal{B}} = coreduce\ (c)\ (z)$
**end** $\mathcal{B}Coalgebra$
```

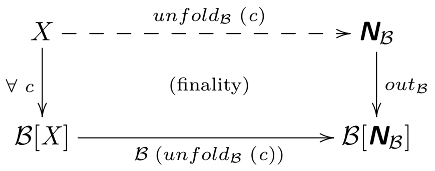
The functor $\mathcal{B}$ and operation $unfold_{\mathcal{B}}$ coincide with the generated datatype *NB_struct* and the (coinductive) function *coreduce*. As the components of *NB_struct* (e.g., see the definition of $out_{\mathcal{B}}$), are also used, we give the definition as is generated by *PVS*.

```
$NB\_struct\ [L, x : \textbf{type}]$ : **datatype begin**
  $inj\_nb\_in\ (inj\_el : L, inj\_next : x) : inj\_nb\_in\,?$
**end** $NB\_struct$
```

As said before, the operational model, which represents the transition relation of the language, is specified as a coalgebra with type $Coalg_{\mathcal{B}}\ [\mathbb{T}(V)]$. The evaluation of a term (with type $\mathbb{T}\ [V]$)[6] is obtained by unfolding the operational model, using the term as 'initial seed'.

The function $out_{\mathcal{B}}$ is the (unique) final $\mathcal{B}$-colagebra on $\mathbf{N}_{\mathcal{B}}$. The following property is the dual of Propostion 1.

▶ **Proposition 3.** *Let $c : Coalg_{\mathcal{B}}\ [X]$. Then, $unfold_{\mathcal{B}}\ (c)$ is unique in making the following diagram commute:*



$$\begin{tikzcd}
X \arrow[r, dashed, "unfold_{\mathcal{B}}\ (c)"] \arrow[d, "\forall\ c"'] \arrow[dr, phantom, "\text{(finality)}"] & \mathbf{N}_{\mathcal{B}} \arrow[d, "out_{\mathcal{B}}"] \\
\mathcal{B}[X] \arrow[r, "\mathcal{B}\ (unfold_{\mathcal{B}}\ (c))"'] & \mathcal{B}[\mathbf{N}_{\mathcal{B}}]
\end{tikzcd}$$

[6] Observe that the term might be open, i.e., it is possible to evaluate term containing variables.

The proof of this property is done by coinduction on $\mathbf{N}_{\mathcal{B}}$. The coinduction principle in *PVS* requires the construction of a proper *bisimulation relation*; e.g., see [8]. This principle is based on the fact that if two streams are bisimilar, then they are equal.

The "fusion law for anamorphisms" was introduced by [16]. We apply this law in a proof in Section 3.

$$\begin{array}{l} \mathit{unfold\_fusion}\ [a, b : \textbf{type}] : \textbf{lemma} \\ \quad \forall\ (f : [a \to b], \mathit{c1} : \mathit{Coalg}_{\mathcal{B}}\ [a], \mathit{c2} : \mathit{Coalg}_{\mathcal{B}}\ [b]) : \\ \qquad \mathit{map}_{\mathcal{B}}\ (f) \circ \mathit{c1} = \mathit{c2} \circ f \Rightarrow \mathit{unfold}_{\mathcal{B}}\ (\mathit{c1}) = \mathit{unfold}_{\mathcal{B}}\ (\mathit{c2}) \circ f \end{array}$$

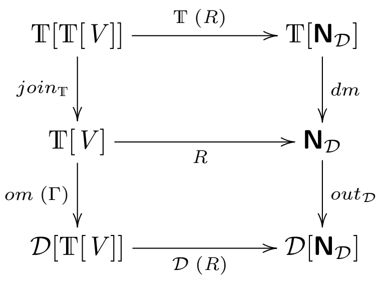
This lemma is proven by applying Proposition 3 twice: once to coalgebra *c1*, and the other to *c2*.

3 Bialgebraic semantics

In the previous section we already mentioned that the operational model (abbreviated as *om*) is specified as a coalgebra. The dual notion, the *denotational model* (*dm*), will be specified as an algebra. Before actually defining both *om* and *dm*, we introduce a general categorical concept, known as a *bialgebra*, which is used to link *om* and *dm* together.

Formally, let N, M be functors. Then, a bialgebra (for N, M) is a triple $\langle V, a, c\rangle$ such that a is a N-algebra and c is a M-coalgebra, sharing V as the common carrier. For two bialgebras, a *bialgebra homomorphism* is a mapping which is both a N-algebra homomorphism and a M-coalgebra homomorphism.

The concrete functors of our framework are $\mathbb{T}$ (for the syntax), and the so-called *free pointed functor* $\mathcal{D}$ *of* $\mathcal{B}$, i.e., $\mathcal{D}[X] = X \times \mathcal{B}[X]$ (for the behaviour). Furthermore, we consider two bialgebras with carriers $\mathbb{T}[V]$ and $\mathbf{N}_{\mathcal{D}}$ (i.e., greatest fixpoint of $\mathcal{D}$), namely, $\langle \mathbb{T}[V], \mathit{join}_{\mathbb{T}}, \mathit{om}\ (\Gamma)\rangle$ and $\langle \mathbf{N}_{\mathcal{D}}, \mathit{dm}, \mathit{out}_{\mathcal{D}}\rangle$. Here, $\Gamma : X \to \mathcal{D}[X]$ represents a *behaviour environment*. We explain later how *om* depends on Γ. The following diagram shows these two bialgebras together with a connecting homomorphism R.



$$\begin{tikzcd} \mathbb{T}[\mathbb{T}[V]] \arrow[r, "\mathbb{T}\ (R)"] \arrow[d, "\mathit{join}_{\mathbb{T}}"'] & \mathbb{T}[\mathbf{N}_{\mathcal{D}}] \arrow[d, "\mathit{dm}"] \\ \mathbb{T}[V] \arrow[r, "R"'] \arrow[d, "\mathit{om}\ (\Gamma)"'] & \mathbf{N}_{\mathcal{D}} \arrow[d, "\mathit{out}_{\mathcal{D}}"] \\ \mathcal{D}[\mathbb{T}[V]] \arrow[r, "\mathcal{D}\ (R)"'] & \mathcal{D}[\mathbf{N}_{\mathcal{D}}] \end{tikzcd}$$

Here, R is an (evaluation) function that maps each term to its execution result which is a (possibly infinite) value of type $\mathbf{N}_{\mathcal{D}}$. From this diagram we infer two different ways of defining R: (1) by considering the top square and using Proposition 2, or (2) by using the bottom square in combination with Proposition 3. This leads to the following two cases:

$$R_D\ (\Gamma) = \mathit{free}_{\mathbb{T}}\ (\mathit{unfold}_{\mathcal{D}}\ (\Gamma), \mathit{dm}) \qquad R_O\ (\Gamma) = \mathit{unfold}_{\mathcal{D}}\ (\mathit{om}\ (\Gamma))$$

From their construction it follows that both R_D (i.e., execution according to the denotational model) and R_O (i.e., execution according to the operational model) are unique. It remains to be shown that these functions are equal, which is called the *adequacy theorem*. Obviously, this depends on the way *om* and *dm* are defined. Now, the essence of the framework of

[18] is the following: instead of defining *om* and *dm* separately, the operational rules of the language are described by using a specific syntactic format from which both *om* and *dm* are obtained generically (i.e., syntax-independently). The interrelation between *om* and *dm* is given by a so-called *distributed law* $\Lambda\ [v : \textbf{type}] : \mathbb{T}[\mathcal{D}[v]] \to \mathcal{D}[\mathbb{T}[v]]$ from which both models can be derived.[7] Formally, a distributive law (between a monad, here $\mathbb{T}$, and an endofunctor, here $\mathcal{D}$) is a *natural transformation* for which the following two identities hold (e.g., see [4]):

$$\begin{aligned}&\mathit{law\_distributive} : \textbf{lemma}\\ &\quad \Lambda \circ \mathit{unit}_{\mathbb{T}} = \mathit{map}_{\mathcal{D}}\ (\mathit{unit}_{\mathbb{T}})\ \wedge\ \Lambda \circ \mathit{join}_{\mathbb{T}} = \mathit{map}_{\mathcal{D}}\ (\mathit{join}_{\mathbb{T}}) \circ \Lambda \circ \mathit{map}_{\mathbb{T}}\ (\Lambda)\end{aligned}$$

To be a natural transformation, Λ should satisfy:

$$\mathit{law\_natural} : \textbf{lemma}\ \Lambda\ \circ\ \mathit{map}_{\mathbb{T}}\ (\mathit{map}_{\mathcal{D}}\ (f)) = \mathit{map}_{\mathcal{D}}\ (\mathit{map}_{\mathbb{T}}\ (f))\ \circ\ \Lambda$$

In polymorphic functional languages, the latter property is an example of a so-called *theorem for free*; see [19]. In essence, this free theorem formalizes the fact that Λ is genuinely polymorphic.

As for *dm*, we observe that the codomain of this operation is $\mathbf{N}_{\mathcal{D}}$, calling for a definition based on $\mathit{unfold}_{\mathcal{D}}$. This leads to:

$$\mathit{dm} : \mathit{Alg}_{\mathbb{T}}\ [\mathbf{N}_{\mathcal{D}}] = \mathit{unfold}_{\mathcal{D}}\ (\Lambda\ \circ\ \mathit{map}_{\mathbb{T}}\ (\mathit{out}_{\mathcal{D}}))$$

Dually, we define *om* as a $\mathit{fold}_{\mathbb{T}}$ (actually, we use the special variant $\mathit{free}_{\mathbb{T}}$ of $\mathit{fold}_{\mathbb{T}}$):

$$\mathit{om}\ (\Gamma : [\,V \to \mathcal{D}[V]]) : \mathit{Coalg}_{\mathcal{D}}\ [\mathbb{T}(V)] = \mathit{free}_{\mathbb{T}}\ (\mathit{map}_{\mathcal{D}}\ (\mathit{unit}_{\mathbb{T}})\ \circ\ \Gamma, \mathit{map}_{\mathcal{D}}\ (\mathit{join}_{\mathbb{T}})\ \circ\ \Lambda)$$

For a detailed explanation, see [14].

The proof of the adequacy theorem (stating that $R_D\ (\Gamma) = R_O\ (\Gamma)$) is done by using Proposition 3 with $\mathit{om}\ (\Gamma)$ substituted for c. This will immediately lead to the following commutation property:

$$\begin{tikzcd}
\mathbb{T}[V] \arrow[r, "\mathit{om}\ (\Gamma)"] \arrow[d, "R_D\ (\Gamma)"'] & \mathcal{D}[\mathbb{T}[V]] \arrow[d, "\mathcal{D}\ (R_D\ (\Gamma))"] \\
\mathbf{N}_{\mathcal{D}} \arrow[r, "\mathit{out}_{\mathcal{D}}"'] & \mathcal{D}[\mathbf{N}_{\mathcal{D}}]
\end{tikzcd}$$

For the proof of this property we apply the alternative proof principle for terms (Proposition 2) using *dm* as $\mathbb{T}\mathit{Algebra}$, and thus we need to verify the fact that $\mathbb{T}\mathit{Alg}?\ (\mathit{dm})$. The key to this proof is the *unfold_fusion* lemma. Finally, nowhere in the above proofs it was required to use any (language-)specific properties of Λ, making the our approach fully syntax-independent.

4 Semantics of *While*

We have seen that our treatment of semantics is parametric in the concrete set of operational rules: the construction of the operational and denotational models did not depend

[7] The term distributed law can be explained by considering the type of Λ as a proposition, specifying that $\mathbb{T}$ distributes over $\mathcal{D}$.

$$\begin{array}{lcl}
\mathcal{A}[\![n]\!]s & = & n \\
\mathcal{A}[\![x]\!]s & = & s(x) \\
\mathcal{A}[\![a_1 + a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 - a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 \times a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s \times \mathcal{A}[\![a_2]\!]s
\end{array}$$

Figure 2 Semantics of arithmetic expressions.

$$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle} \qquad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$
$$\langle x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s] \qquad \langle \mathbf{skip}, s\rangle \Rightarrow s$$
$$\langle \mathbf{if}\ c\ \mathbf{then}\ S_t\ \mathbf{else}\ S_e, s\rangle \Rightarrow \langle S_t, s\rangle \ \text{if}\ s(c) \neq 0$$
$$\langle \mathbf{if}\ c\ \mathbf{then}\ S_t\ \mathbf{else}\ S_e, s\rangle \Rightarrow \langle S_e, s\rangle \ \text{if}\ s(c) = 0$$
$$\langle \mathbf{while}\ c\ \mathbf{do}\ S, s\rangle \Rightarrow \langle \mathbf{if}\ c\ \mathbf{then}\ (S; \mathbf{while}\ c\ \mathbf{do}\ S)\ \mathbf{else\ skip}, s\rangle$$

Figure 3 SOS for *While*.

on language specific properties. As such, our treatment could be classified as 'meta-meta-theoretical'.

In this section we demonstrate a paradigmatic example for many concrete (imperative) programming languages. The language we have chosen is *While*, appearing in many text-books on semantics; e.g., see [17]. The standard structural operational semantics is given as a set of transitions, each of the form $\langle S, s\rangle \Rightarrow \gamma$, where γ is either of the form $\langle S', s'\rangle$ or simply s'.

The complete set of operational rules is given in Figure 3, in *small-step* style.

This system is based on the evaluation of arithmetic expressions which is defined separately in Figure 2, in (a sort of) *big-step* style. Although it is very well possible to specify both styles of semantics in the bialgebraic framework (for example, see [11]) we will treat arithmetic expressions differently, by expressing the semantics directly in *PVS* as a recursive function; see the *PVS* proof files. We have simplified the language by omitting boolean expressions, and restricting conditions to solely (integer) variables. The value 0 will be interpreted as *false*; any other value as *true*.

We specify the operational rules, not directly as a distributive law, but in the so-called *GSOS-format* which is more restrictive: the rules given in this format are functions ρ of type

$$\rho\ [v : \mathbf{type}] : \Sigma[\mathcal{D}[v]] \to \mathcal{B}[\mathbb{T}[v]].$$

As a first step, we introduce appropriate functors for representing the syntax and behaviour.

Syntax

In our *PVS* formalization, the syntax functor is derived from an enumeration of operator symbols, and their corresponding arity function. These can subsequently be used as actual parameters of the *Terms* theory (see Section 2) to obtain Σ. Concretely,

```
SigW [ V : type] : datatype begin importing Expr [ V ]
  ass (dst : V, src : Expr) : ass ?
  skip :                      skip ?
  seq :                       seq ?
  $ifs\ (con : V) :$              $ifs\,?$
  $while\ (con : V) :$            $while\,?$
end $SigW$
```

The imported type *Expr* $[V]$ represents the arithmetic expressions. Aside from the function arW, the following theory contains some auxiliary definitions which facilitate the specification of the operational rules and concrete *While* programs. The representation of the syntax by $\mathbb{T}$ terms requires that the arity of each operation, as returned by arW, should correspond to the number of $\mathbb{T}$ arguments instead of the real number of arguments. This, for instance, explains why *ifs* and *while* have arities 2 and 1, respectively.

```
$WhileLang\ [V : \textbf{type}] :$ theory begin importing $SigW\ [V], Expr\ [V]$
  $arW\ (s : SigW\ [V]) : nat =$ cases $(s)$ of
    $ass\ (v, a) : 0, skip : 0, seq : 2, ifs\ (c) : 2, while\ (c) : 1$
  endcases
  importing $Terms\ [SigW\ [V], arW], EmptyFunP$
  $args0\ [x : \textbf{type}] : [below\ (0) \to \mathbb{T}\ [x]] = emptyFun\ [below\ (0), \mathbb{T}\ [x]]$
  $args1\ [x : \textbf{type}]\ (a : \mathbb{T}\ [x])\ (i : below\ (1)) : \mathbb{T}\ [x] = a$
  $args2\ [x : \textbf{type}]\ (a1, a2 : \mathbb{T}\ [x])\ (i : below\ (2)) : \mathbb{T}\ [x] =$ if $i = 0$ then $a1$ else $a2$ endif
  $assT\ [x : \textbf{type}]\ (v : V, src : Expr) :$  $\mathbb{T}\ [x] = tapp\ (ass\ (v, src), args0)$
  $skipT\ [x : \textbf{type}] :$                   $\mathbb{T}\ [x] = tapp\ (skip, args0)$
  $seqT\ [x : \textbf{type}]\ (s1, s2 : \mathbb{T}\ [x]) :$   $\mathbb{T}\ [x] = tapp\ (seq, args2\ (s1, s2))$
  $ifT\ [x : \textbf{type}]\ (co : V, t, e : \mathbb{T}\ [x]) :$  $\mathbb{T}\ [x] = tapp\ (ifs\ (co), args2\ (t, e))$
  $whileT\ [x : \textbf{type}]\ (co : V, b : \mathbb{T}\ [x]) :$ $\mathbb{T}\ [x] = tapp\ (while\ (co), args1\ (b))$
  $varT\ [x : \textbf{type}]\ (v : x) :$             $\mathbb{T}\ [x] = tvar\ (v)$
end $WhileLang$
```

Behaviour

As to the behaviour functor, we must remember that the semantic domain (being the greatest fixpoint $\mathbf{N}_{\mathcal{D}}$ of functor $\mathcal{D}$)[8], cannot be expressed in terms of $\mathcal{D}$ $(= X \times \mathcal{B}[X])$. Again, we will define $\mathbf{N}_{\mathcal{D}}$ as a coinductive data type, and let *PVS* generate the corresponding functor $\mathcal{D}$. The behaviour functor for *While* resembles the behaviour functor for the stream language. However, instead of using a label set, we now need a state transition function for passing the (possibly) modified store s. The store itself maps variables to integer values. Furthermore, to represent the two possibilities for γ in the state transitions, we use the *Maybe* functor which is given below. The standard map_{Mb} and $fold_{Mb}$ operations are defined in terms of the corresponding *map* and *reduce* functions generated by *PVS* itself.

```
$MBF :$ theory begin
  $Maybe\ [x : \textbf{type}] :$ datatype begin
    $nothing :$             $nothing\,?$
    $just\ (fromJust : x) : just\,?$
  end $Maybe$
  $map_{Mb}\ [a, b : \textbf{type}]\ (f : [a \to b]) : [Maybe\ [a] \to Maybe\ [b]] = map\ (f)$
```

[8] Some other approaches use $\mathbf{N}_{\mathcal{B}}$ as domain. However, this is not an essential difference since one can easily show that $\mathbf{N}_{\mathcal{B}}$ and $\mathbf{N}_{\mathcal{D}}$ are isomorphic.

```
  $fold_{Mb}\ [a, b : \textbf{type}]\ (nf : b, jf : [a \to b]) : [Maybe\ [a] \to b] \quad = reduce\ (nf, jf)$
$\textbf{end}\ MBF$
```

The data types, and basic operations for representing $\mathcal{D}$ in PVS are:

```
$BF\ [ST : \textbf{type}] : \textbf{theory begin importing}\ MBF$
  $\mathcal{B}\ [x : \textbf{type}] : \textbf{type} = [ST \to [ST, Maybe\ [x]]]$
  $map_{\mathcal{B}}\ [a, b : \textbf{type}]\ (f : [a \to b])\ (bf : \mathcal{B}\ [a]) : \mathcal{B}\ [b] =$
    $\lambda\ (s : ST) : \textbf{let}\ ber = bf\ (s)\ \textbf{in}\ (ber`1, map_{Mb}\ (f)\ (ber`2))$
$\textbf{end}\ BF$
$\mathbf{N}_{\mathcal{D}}\ [ST : \textbf{type}] : \textbf{codatatype begin}$
    $\textbf{importing}\ BF\ [ST]$
    $dz\_in\ (left : \mathbf{N}_{\mathcal{D}}, right : \mathcal{B}\ [\mathbf{N}_{\mathcal{D}}]) : dz\_in\,?$
$\textbf{end}\ \mathbf{N}_{\mathcal{D}}$
$\mathcal{D}Coalgebra\ [ST : \textbf{type}] : \textbf{theory begin importing}\ \mathbf{N}_{\mathcal{D}}\ [ST]$
  $\mathcal{D}\ [x : \textbf{type}] \qquad : \textbf{type} = DZ\_struct\ [ST, x]$
  $Coalg_{\mathcal{D}}\ [x : \textbf{type}] : \textbf{type} = [x \to \mathcal{D}\ [x]]$
  $injD\ [a, b : \textbf{type}]\ (f : [a \to b], g : [a \to \mathcal{B}\ [b]])\ (x : a) : \mathcal{D}\ [b] = inj\_dz\_in\ (f\ (x), g\ (x))$
  $outD : Coalg_{\mathcal{D}}\ [\mathbf{N}_{\mathcal{D}}] = injD\ (left, right)$
  $map_{\mathcal{D}}\ [a, b : \textbf{type}]\ (f : [a \to b]) : [\mathcal{D}\ [a] \to \mathcal{D}\ [b]] = injD\ (f\ \circ\ inj\_left, map_{\mathcal{B}}\ (f)\ \circ\ inj\_right)$
$\textbf{end}\ \mathcal{D}Coalgebra$
```

For the sake of completeness, we also give the definition of DZ_struct as is generated from $\mathbf{N}_{\mathcal{D}}$ by PVS.

```
  $DZ\_struct\ [ST, x : \textbf{type}] : \textbf{datatype begin importing}\ BF\ [ST]$
    $inj\_dz\_in\ (inj\_left : x, inj\_right : \mathcal{B}\ [x]) : inj\_dz\_in\,?$
  $\textbf{end}\ DZ\_struct$
```

Before specifying the operational rules, we have a closer look at the GSOS-format itself. The domain of ρ (i.e., $\Sigma[\mathcal{D}[v]]$) allows us to pattern-match on the outermost symbol. The symbol is parameterized (depending on the arity) with a $\mathcal{D}$ expression which will provide access to the premises of the rule. Since only the sequence operator has rules with premises, we will elaborate on the alternative for ρ that corresponds to this operation. The first component of $\mathcal{D}$ represents the meta-variable on the left-hand side of the premise, whereas the second component represents the right-hand side. Since we do not pass the state explicitly, we have to apply the second component to the state argument of the state transition function that is yielded as a result. By inspecting the outcome of that application we can decide which of the two rules for seq applies, and construct the corresponding right-hand side of the conclusion. For the latter, we use the fold operation for $Maybe$. In PVS:

```
  $WhileGSOS\ [V : \textbf{type}] : \textbf{theory begin}$
    $STORE : \textbf{type} = [V \to int]$
    $\rho\ [v : \textbf{type}]\ (sf : \Sigma[\mathcal{D}[v]]) : \mathcal{B}[\mathbb{T}[v]] = \textbf{cases}\ sf`1\ \textbf{of}$
      $\ldots$
      $seq : \textbf{let}\ a0 = sf`2\ (0), a1 = sf`2\ (1)\ \textbf{in}$
        $\lambda\ (st : STORE) : \textbf{let}\ rst = inj\_right\ (a0)\ (st)\ \textbf{in}$
          $fold_{Mb}\ ((rst`1, just\ (varT\ (inj\_left\ (a1)))),$
            $\lambda\ (s1 : x) : (rst`1, just\ (seqT\ (varT\ (s1), varT\ (inj\_left\ (a1))))))\ (rst`2),$
    ...
    **endcases**
**end** *WhileGSOS*
```

In order to obtain a distributive law of $\mathbb{T}$ over $\mathcal{D}$, ρ needs to undergo a two-step transformation. The first step is to expand ρ's codomain using the auxiliary function *injD* defined in the theory $\mathcal{D}Coalgebra$:

$$\tau\ [v : \textsf{type}] : [\Sigma[\mathcal{D}[v]] \to \mathcal{D}[\mathbb{T}[v]]] = \mathit{injD}\ (\mathit{tapp} \circ \mathit{map}_\Sigma\ (\mathit{tvar} \circ \mathit{inj\_left}), \rho)$$

Adjusting the domain is slightly more involved, and requires an appropriate use of $fold_{\mathbb{T}}$:

$$\Lambda\ [v : \textsf{type}] : [\mathbb{T}[\mathcal{D}[v]] \to \mathcal{D}[\mathbb{T}[v]]] = fold_{\mathbb{T}}\ (\mathit{map}_{\mathcal{D}}\ (\mathit{unit}_{\mathbb{T}}), \mathit{map}_{\mathcal{D}}\ (\mathit{join}_{\mathbb{T}})\ \circ\ \tau)$$

This construction does not affect the naturality property (Lemma *law_natural*). Moreover, it guarantees that Λ is indeed a distributive law (Lemma *law_distributive*).

Running a program

There is one discrepancy between our formalization and a standard denotational semantics for *While* as, for instance given by [17]. In our case, the mathematical object describing the effect of executing each construct is the greatest fixed point $\mathbf{N}_{\mathcal{D}}$ of $\mathcal{D}$. In the standard case, this is a (possibly partial) state transition function, which is obtained by composing the functions that correspond to the components of this construct. Moreover, for while statements, a standard denotational semantics also requires fixed points. The result of executing a program in our framework, $\mathbf{N}_{\mathcal{D}}$, is not a single state transition function, but a (possibly infinite) stream of functions that still need to be interconnected. In PVS, however, all functions have to be total. We solve this issue by defining the following total variant[9] of a compose function which returns the constructed transition function after N execution steps.

```
composeN $(N : nat, dz : \mathbf{N}_{\mathcal{D}})\ (ist : STORE)$ : **recursive** $ST$ =
  **if** $N = 0$ **then** $ist$
  **else let** $res = right\ (dz)\ (ist)$ **in**
    **cases** $res`2$ **of** $nothing : res`1$,
                        $just\ (x) : composeN\ (N-1, x)\ (res`1)$ **endcases endif measure** $N$
```

To illustrate program execution, we use the following program that computes the 12th Fibonacci number. It uses 3 variables each identified by a number. Variable 2 will hold the final result.

```
PFib10 : $\mathbb{T}$ = seqT (assT (0, enum (10)), seqT (assT (1, enum (1)), seqT (assT (2, enum (1)),
  (whileT (0, seqT (assT (3, eplus (evar (1), evar (2))), seqT (assT (1, evar (2)),
    seqT (assT (2, evar (3)), assT (0, emin (evar (0), enum (1))))))))))
```

We execute this program and apply the final store, obtained after 62 steps (just enough for the while loop to terminate), to variable 2, producing[10] the value 144.

[9] In Pvs, totality is enforced by a so-called *measure* specification which is mandatory when defining a recursive function. This measure is used to guarantee (by generating special proof obligations) that the function indeed terminates.

[10] The answer was obtained by executing our specification in the functional language *Clean*. The current, so-called, ground evaluator of PVS seems to have difficulties with evaluating expressions containing infinite codata structures.

$$EXEC : nat = composeN\ (62, R_O\ (emptyEnv)\ (PFib10))\ (\lambda\ (i : nat) : 0)\ (2)$$

Observe that we used R_O (based on the operational model) to execute the program. Equivalently, we could have used the denotational version R_D, obviously leading to the same result, since we proven that R_O and R_D equal.

5 PVS formalization

Our main motivation for developing this formalization was to investigate whether or not implementing abstract categorical concepts in *PVS* 6.0 is feasible. The case study we performed was based on previous, similar experiments with *Coq*.

As far as this case study is concerned, the main difference between *Coq* and *PVS* is that *Coq* is equipped with a rich type class system offering type classes as first class citizens. Therefore, in *Coq*, functors, monads, and (co)algebras can be naturally represented. *PVS* offers parameterized theories, but using these as a substitute for *Coq*'s type classes in general is definitely a setback. Moreover, like *PVS*, *Coq* suffers from the same restriction that polymorphism is only first-order. This definitely reduces the generality of the formalisation, however, we managed to separate the language-specific components from the more abstract categorial concepts such that changes in the syntax and/or behaviour of the programming language hardly affects the description in its entirety.

Additionally, for the most part these aspects of our formalization do not obstruct the proving process. There were no fundamental problems which could not be resolved due to restrictions of *PVS*'s specification language. The rich support for abstract (co)data types (including the facility for automatic generation of common theories) has shown to be adequate.

There was, however, a minor issue obstructing the proving process to some extent. When importing a parameterized theory, the user must explicitly specify which actual arguments are required. In a truly polymorphic case this matter would have been solved by the type checker (as is done in *Coq* or in *Haskell*). Unfortunately, *PVS* lacks the ability to resolve theory instantiations automatically. To some extent, this is also the case for instantiation of declaration parameters. We encountered situations in which the type checker was not capable of determining the correct instance types. However, from the discussions with the developers of *PVS* we concluded that this was not a fundamental issue but a temporary defect of the typing algorithm, that is expected to be repaired in the near future.

Finally, the goal of our experiment was not to compare *PVS* with *Coq*. In terms of development times, for our specific example these were about the same. Of course, we have to take into account that we first did our experiment with *Coq* and all theoretical difficulties were already solved when we started the exercise with *PVS*. We are convinced that the time it takes to formalize a relatively complex system, such as the bialgebraic framework, is mainly determined by the experience of the user. This development process is barely retarded by (the peculiarities of) the specific proof assistant itself.

6 Related work

This work was inspired by our earlier work on modularity, the formalization of Modular Structural Operational Semantics (MSOS) [15]. The present paper can be seen as a contribution to field of bialgebraic semantics, starting with Turi and Plotkin's research [18], and resulting in a uniform categorical treatment of semantics. They abstracted from concrete syntactical and semantical details by characterizing these language-dependent issues by a distributive law between syntax and behaviour. By means of a categorical construct, both an operational and a denotational model were obtained, and moreover the adequacy of these models could be proven. Klin [12] gives an introduction to the basics of bialgebras for operational semantics that was used in the present formalization. He also sketches the state-of-the-art in this field of research.

The distributive law actually describes a syntactic format for specifying operational rules. This abstract so-called GSOS format has been applied to several areas of computer science. For example, in his thesis [2] Bartels gives concrete syntactic rule formats for abstract GSOS rules in several concrete cases. Variable binding, which is a fundamental issue in, for example, λ-calculus or name passing π-calculi, is addressed in [6]. The authors show that name binding fits in the abstract GSOS format. This was refined further in [7].

In [5] a framework is introduced, called MTC, for defining and reasoning about extensible inductive datatypes which is implemented as a Coq library. It enables modular mechanized metatheory by allowing language features to be defined as reusable components. Similar to our work, MTC's modular reasoning is based on universal property of folds [16], offering an alternative to structural induction.

A significant contribution to the work on interpreters for programming languages, is that of the application of monads in order to structure semantics. Liang et al. [13] introduced monad transformers to compose multiple monads and build modular interpreters. Jaskelioff et al. use [11] as a starting point, and provide monad-based modular implementation of mathematical operational semantics in *Haskell*. The authors also give some concrete examples of small programming languages specified in GSOS-format. Our example in Section 2 is inspired by this work. Although, [11] strictly follows the approach of Turi and Plotkin, there is no formal evidence that their construction is correct, i.e., there are no 'pen and paper' or machine-checked proofs given. The latter issue is addressed by recent work of [9] who introduce modular proof techniques for equational reasoning about monads.

7 Conclusions

We presented a formalization in *PVS* version 6.0 of Turi and Plotkin's work based on category theory. Our main goal was to investigate whether this could be done in a convenient way. Except for some minor flaws in *PVS*'s type checker discussed in Section 5, we did not encountered any fundamental issues that seriously hindered the proving process. Our experiment also resulted in a *PVS* framework which can be used for formal reasoning about programming languages in general, in addition to reasoning about specific programs. Moreover, it offers the user the possibility to choose between either denotational or operational semantics at any point in his application.

Our future plans comprise of experimenting with our framework in formal reasoning with case studies in specific examples of denotational and operational semantics, and to extend the framework with an axiomatic semantics.

References

1 Michael Barr and Charles Wells. *Category theory for computing science*, volume 49. Prentice Hall New York, 1990.

2 Falk Bartels. *On generalised coinduction and probabilistic specification formats.* PhD thesis, CWI, Amsterdam, April 2004.

3 Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, January 1995.

4 Marcello M Bonsangue, Helle Hvid Hansen, Alexander Kurz, and Jurriaan Rot. Presenting distributive laws. In *Algebra and Coalgebra in Computer Science*, LNCS, pages 95–109. Springer, 2013.

5 Ben Delaware, Bruno C.d.S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, New York, NY, USA, 2013. ACM.

6 Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, pages 193–202, Washington, DC, USA, 1999. IEEE Computer Society.

7 Marcelo Fiore and Sam Staton. A congruence rule format for name-passing process calculi from mathematical structural operational semantics. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, LICS '06, pages 49–58, Washington, DC, USA, 2006. IEEE Computer Society.

8 Ulrich Hensel and Bart Jacobs. Coalgebraic theories of sequences in pvs. *J. Log. Comput.*, 9(4):463–500, 1999.

9 Ralf Hinze and Daniel W.H. James. Proving the unique fixed-point principle correct: an adventure with category theory. *SIGPLAN Not.*, 46(9):359–371, September 2011.

10 Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

11 Mauro Jaskelioff, Neil Ghani, and Graham Hutton. Modularity and implementation of mathematical operational semantics. *Electron. Notes Theor. Comput. Sci.*, 229(5):75–95, March 2011.

12 Bartek Klin. Bialgebras for structural operational semantics: An introduction. *Theoretical Computer Science*, 412(38):5043–5069, 2011. CMCS Tenth Anniversary Meeting.

13 Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM.

14 Ken Madlener and Sjaak Smetsers. GSOS formalized in Coq. In *The 7th International Symposium on Theoretical Aspects of Software Engineering (TASE2013)*, pages 199–206, 2013. Birmingham, UK, 2013. IEEE.

15 Ken Madlener, Sjaak Smetsers, and Marko C. J. D. van Eekelen. Formal component-based semantics. In Michel A. Reniers and Pawel Sobocinski, editors, *SOS*, volume 62 of *EPTCS*, pages 17–29, 2011.

16 Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, London, UK, UK, 1991. Springer-Verlag.

17 Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.

18 Daniele Turi and Gordon Plotkin. Towards a mathematical operational semantics. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, LICS '97, pages 280–291, Washington, DC, USA, 1997. IEEE Computer Society.

19 Philip Wadler. Theorems for free! In *Functional Programming Languages And Computer Architecture*, pages 347–359. ACM Press, 1989.