11th International Workshop on Higher-Order Rewriting (HOR 2023)

Carsten Fuhs (Editor)

HOR 2023, 4th July 2023, Rome, Italy

Editor's Preface

This volume contains the informal proceedings of the *11th International Workshop on Higher-Order Rewriting*, to be held on 4th July 2023 in Rome, Italy:

`https://hor2023.github.io/`

HOR is a forum to present work concerning all aspects of higher-order rewriting. The aim is to provide an informal and friendly setting to discuss recent work and work in progress. The following is a non-exhaustive list of topics for the workshop:

- **Applications**: proof checking, theorem proving, generic programming, declarative programming, program transformation, automated termination/confluence/equivalence analysis tools.
- **Foundations**: pattern matching, unification, strategies, narrowing, termination, syntactic properties, type theory, complexity of derivations.
- **Frameworks**: term rewriting, conditional rewriting, graph rewriting, net rewriting, comparisons of different frameworks.
- **Implementation**: explicit substitution, rewriting tools, compilation techniques.
- **Semantics**: semantics of higher-order rewriting, categorical rewriting, higher-order abstract syntax, games and rewriting.

The International Workshop on Higher-Order Rewriting has played an important role in the last two decades of developments in the field. Starting in 2002, the workshop took place loosely every other year:

HOR 2002 (Copenhagen, Denmark), affiliated with FLoC 2002
HOR 2004 (Aachen, Germany), affiliated with RDP 2004
HOR 2006 (Seattle, USA), affiliated with FLoC 2006
HOR 2007 (Paris, France), affiliated with RDP 2007
HOR 2010 (Edinburgh, UK), affiliated with FLoC 2010
HOR 2012 (Nagoya, Japan), affiliated with RTA 2012
HOR 2014 (Vienna, Austria), affiliated with FLoC 2014
HOR 2016 (Porto, Portugal), affiliated with FSCD 2016
HOR 2018 (Oxford, UK), affiliated with FLoC 2018
HOR 2019 (Dortmund, Germany), affiliated with FSCD 2019

Information about previous editions can be found at `https://hor.irif.fr/`.

HOR 2023 is affiliated with the *8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023)* and the *29th International Conference on Automated Deduction (CADE-29)*.

The 11th Workshop on Higher-Order Rewriting features six regular extended abstracts, contained in this volume, and an invited talk by Pablo Barenbaum (Universidad de Buenos Aires, Argentina) on *Quantitative Types for Useful Reduction*.

I would like to thank everyone who helped to prepare and run the workshop: the participants, the programme committee, the steering committee, and the local organisers.

Rome, July 2023 Carsten Fuhs

Programme Committee

Takahito Aoto	Niigata University, Japan
Maribel Fernández	King's College London, United Kingdom
Carsten Fuhs (Chair)	Birkbeck, University of London, United Kingdom
Delia Kesner	Université Paris Cité, France
Cynthia Kop	Radboud Universiteit Nijmegen, The Netherlands
Damiano Mazza	Université Paris 13, France

Steering Committee

Delia Kesner	Université Paris Cité, France
Femke van Raamsdonk	Vrije Universiteit Amsterdam, The Netherlands

Table of Contents

A Deeper Study of λ!-Calculus Simulations

Victor Arrial

Université Paris Cité, CNRS, IRIF, France

Abstract

In this paper we study properties of the encodings of CBN and CBV into a CBPV like language

1 Introduction

Bang Calculus. P.B. Levy introduced Call-by-Push-Value (CBPV) [13] as a *subsuming* language, so that different evaluation strategies of the λ-calculus can be captured in a uniform framework by the simple use of two primitives: *thunk* (to pause a computation) and *force* (to resume a computation). This mechanism is powerful enough to encode, in particular, Call-by-Name (CBN) and Call-by-Value [14] (CBV). The original CBPV has been introduced in a simply typed framework, but the underlying (untyped) syntax and operational semantics –the ones we are interested in here– already provide a powerful untyped subsuming mechanism. Despite that, CBN and CBV have always been studied notably by developing different techniques for one and the other. Some rare exceptions are [5, 8, 12, 3], where some particular property for CBN/CBV (e.g. quantitative typing, factorization, tight typing, inhabitation) is derived from the corresponding property for a language that is a restriction of CBPV, via a suitable CBN/CBV encoding. Such a language can be the *bang calculus* [7, 10, 11] (in turn inspired by Ehrhard [6], combining ideas from Levy's CBPV [13] and Girard's linear logic [9]), or its variant the λ!-*calculus* [5, 12] where reduction rules act *at a distance*.

Normal Form, Simulation and Reverse Simulation. In this work, we are interested in the preservation of the reduction relation by the encoding. On one hand and from a static point of view, one may consider the *preservation of normal forms*: t is a normal form if and only if t^e is a normal form, where t^e is the e-encoding of t. On the second hand and from a more dynamical point of view, two properties can naturally be considered. *Simulation*: reduction steps are transported into the subsuming paradigm (i.e. if $t \to u$ in the subsumed language then $t^e \to u^e$ in the subsuming language). *Reverse simulation*: reduction steps are recovered from the subsuming paradigm (i.e. if $t^e \to u^e$ in the subsuming language then $t \to u$ in the subsumed language).

Some of these properties have been studied for CBPV [13], bang calculus [10] and λ!-calculus [5] but a full analysis is still missing. In this work, we look into the normal form preservation, simulation and reverse simulation for the λ!-calculus. In particular, taking into account distance is not a trivial matter in that it requires modifications of the call-by-value encoding.

2 The λ!-Calculus

We now briefly introduce the λ!-calculus [5]. Given a countably infinite set $\mathcal{X}$ of variables $x, y, z, ...$, the set $\Lambda_!$ of terms is given by the following inductive definition:

$$\textbf{(Terms)} \qquad t, u \quad ::= \quad x \in \mathcal{X} \mid \lambda x.t \mid tu \mid t[x \backslash u] \mid !t \mid \mathtt{der}(t)$$

$$
\begin{array}{rcl@{\qquad}rcl}
x^{\mathtt{cbn}} & = & x & x^{\mathtt{cbv}} & = & !x \\
(\lambda x.t)^{\mathtt{cbn}} & = & \lambda x.t^{\mathtt{cbn}} & (\lambda x.t)^{\mathtt{cbv}} & = & !\lambda x.t^{\mathtt{cbv}} \\
(tu)^{\mathtt{cbn}} & = & t^{\mathtt{cbn}} !u^{\mathtt{cbn}} & (tu)^{\mathtt{cbv}} & = & \begin{cases} \mathtt{L}\,\langle s\rangle\, u^{\mathtt{cbv}} & \text{if } t^{\mathtt{cbv}} = \mathtt{L}\,\langle !s\rangle \\ \mathtt{der}(t^{\mathtt{cbv}})u^{\mathtt{cbv}} & \text{otherwise} \end{cases} \\
(t[x\backslash u])^{\mathtt{cbn}} & = & t^{\mathtt{cbn}}[x\backslash !u^{\mathtt{cbn}}] & (t[x\backslash u])^{\mathtt{cbv}} & = & t^{\mathtt{cbv}}[x\backslash u^{\mathtt{cbv}}]
\end{array}
$$

Figure 1: CbN and CbV Embeddings for $\lambda!$-Calculus [5]

The set $\Lambda_!$ includes λ-terms (variables x, abstractions $\lambda x.t$ and applications tu) as well as three new constructors: a closure $t[x\backslash u]$ representing a pending explicit substitution $[x\backslash u]$ on a term t, a bang $!t$ to freeze the execution of t, and a dereliction $\mathtt{der}(t)$ to fire again the frozen term t. The usual notion of α-conversion [4] is extended to the whole set $\Lambda_!$, and terms are identified up to α-conversion. We denote by $t\{x := v\}$ the usual (capture avoiding) meta-level substitution of the term u for all free occurrences of the variable x in the term t.

The sets of list contexts $\mathtt{L}$, surface contexts $\mathtt{S}$ and full contexts $\mathtt{F}$, which can be seen as terms containing exactly one hole $\diamond$, are inductively defined as follows:

$$
\begin{array}{rrcl}
\textbf{(List)} & \mathtt{L} & ::= & \diamond \mid \mathtt{L}[x\backslash t] \\
\textbf{(Surface)} & \mathtt{S} & ::= & \diamond \mid \lambda x.\mathtt{S} \mid \mathtt{S}\,t \mid t\,\mathtt{S} \mid \mathtt{S}[x\backslash t] \mid t[x\backslash \mathtt{S}] \mid \mathtt{der}(\mathtt{S}) \\
\textbf{(Full)} & \mathtt{F} & ::= & \diamond \mid \lambda x.\mathtt{F} \mid \mathtt{F}\,t \mid t\,\mathtt{F} \mid \mathtt{F}[x\backslash t] \mid t[x\backslash \mathtt{F}] \mid \mathtt{der}(\mathtt{F}) \mid !\mathtt{F}
\end{array}
$$

The hole can occur everywhere in $\mathtt{F}$, while in $\mathtt{S}$ it cannot occur under a !. We write $\mathtt{L}\,\langle t\rangle$ for the term obtained by replacing the hole in $\mathtt{L}$ with the term t and similarly for $\mathtt{S}$ and $\mathtt{F}$.

The following three rewriting rules are the base components of our reduction relations:

$$
\mathtt{L}\,\langle\lambda x.t\rangle\, u \;\mapsto_{\mathtt{dB}}\; \mathtt{L}\,\langle t[x\backslash u]\rangle \qquad t[x\backslash \mathtt{L}\,\langle !u\rangle] \;\mapsto_{\mathtt{s!}}\; \mathtt{L}\,\langle t\{x := u\}\rangle \qquad \mathtt{der}(\mathtt{L}\,\langle !t\rangle) \;\mapsto_{\mathtt{d!}}\; \mathtt{L}\,\langle t\rangle
$$

Rule $\mathtt{dB}$ (resp. $\mathtt{s!}$) is assumed to be capture free, so no free variable of u (resp. t) is captured by the context $\mathtt{L}$. The rule $\mathtt{dB}$ fires a standard β-redex and generates an explicit substitution. The rule $\mathtt{s!}$ fires an explicit substitution provided that its argument is a bang. The rule $\mathtt{d!}$ defrosts a frozen term. In all of these rewrite rules, the reduction acts at a distance [1]: the main constructors involved in the rule can be separated by a finite –possibly empty– list $\mathtt{L}$ of explicit substitutions. This mechanism unblocks redexes that otherwise would be stuck, e.g. $(\lambda x.x)[y\backslash w]!z \mapsto_{\mathtt{dB}} x[x\backslash !z][y\backslash w]$ fires a β-redex by taking $\mathtt{L} = \diamond[y\backslash w]$ as the list context in between the function $\lambda x.x$ and the argument $!z$.

The surface reduction relation $\to_{\mathtt{S}}$ is the surface closure of any of the three rewrite rules $\mathtt{dB}$, $\mathtt{s!}$ and $\mathtt{d!}$, i.e. $\to_{\mathtt{S}}$ only fires redexes in surface contexts, and not under bang. Similarly, the full reduction relation $\to_{\mathtt{F}}$ is the full closure of any of the rewrite rules, so that $\to_{\mathtt{F}}$ reduces under full contexts and thus the bang loses its freezing behavior. For example,

$$
(\lambda x.!\mathtt{der}(!x))!y \quad \to_{\mathtt{S}} \quad (!\mathtt{der}(!x))[x\backslash !y] \quad \to_{\mathtt{S}} \quad !\mathtt{der}(!y) \quad \to_{\mathtt{F}} \quad !y
$$

Note that the first two steps are also $\to_{\mathtt{F}}$-steps, while the last step is not an $\to_{\mathtt{S}}$-step. More generally, we have $\to_{\mathtt{S}} \subseteq \to_{\mathtt{F}}$. Moreover, we denote by $\twoheadrightarrow_{\mathtt{S}}$ (resp. $\twoheadrightarrow_{\mathtt{F}}$) the reflexive and transitive closure of $\to_{\mathtt{S}}$ (resp. $\to_{\mathtt{F}}$).

3 Call-by-Name – $\lambda_{\mathtt{n}}$

We now briefly introduce the (call-by-name) $\lambda_{\mathtt{n}}$-calculus. Given a countably infinite set $\mathcal{X}$ of variables $x, y, z, ...$, the set $\Lambda_{\mathtt{n}}$ of terms is given by the following inductive definition:

$$\textbf{(Terms)} \qquad t, u \quad ::= \quad x \in \mathcal{X} \mid \lambda x.t \mid tu \mid t[x\backslash u]$$

Terms are identified up the the usual notion of α-conversion [4] and we denote by $t\{x := v\}$ the expected (capture avoiding) meta-level substitution. The sets of list contexts $\mathtt{L}$, surface contexts $\mathtt{N}$ and full contexts $\mathtt{C}$, are inductively defined as follows:

$$\begin{array}{rrcl} \textbf{(List)} & \mathtt{L} & ::= & \diamond \mid \mathtt{L}[x\backslash t] \\ \textbf{(Surface)} & \mathtt{N} & ::= & \diamond \mid \lambda x.\mathtt{N} \mid \mathtt{N}\, t \mid \mathtt{N}[x\backslash t] \\ \textbf{(Full)} & \mathtt{C} & ::= & \diamond \mid \lambda x.\mathtt{C} \mid \mathtt{C}\, t \mid t\, \mathtt{C} \mid \mathtt{C}[x\backslash t] \mid t[x\backslash \mathtt{C}] \end{array}$$

The hole can occur everywhere in $\mathtt{C}$, while in $\mathtt{N}$ it cannot occur in the argument of an application nor an explicit substitution.

The following rewriting rules are the base components of our reduction relations:

$$\mathtt{L}\,\langle \lambda x.t\rangle\, u \quad \mapsto_{\mathtt{dB}} \quad \mathtt{L}\,\langle t[x\backslash u]\rangle \qquad\qquad t[x\backslash u] \quad \mapsto_{\mathtt{s}} \quad t\{x := u\}$$

The surface reduction relation $\to_{\mathtt{N}}$ (resp. full reduction relation $\to_{\mathtt{C}}$) is defined as the surface closure $\mathtt{N}$ (resp. full closure $\mathtt{C}$) of the rules $\mathtt{dB}$ and $\mathtt{s}$ presented above. Moreover, we denote by $\twoheadrightarrow_{\mathtt{N}}$ (resp. $\twoheadrightarrow_{\mathtt{C}}$) the reflexive transitive closure of the surface reduction $\to_{\mathtt{N}}$ (resp. full reduction $\to_{\mathtt{C}}$). Finally, a term $t \in \Lambda_{\mathtt{n}}$ is said in to be a surface (resp. full) normal form if there is no u such that $t \to_{\mathtt{N}} u$ (resp. $t \to_{\mathtt{C}} u$).

The embedding $\cdot^{\mathtt{cbn}}$ presented in (Fig. 1) has been shown [5] to preserve surface normal forms. However, this property can be strengthen to also include full normal forms:

Theorem 3.1 (Normal Forms Preservation)**.** *Let $t \in \Lambda_{\mathtt{n}}$, then t is a surface (resp. full) normal form if and only if $t^{\mathtt{cbn}}$ is a surface (resp. full) normal form.*

Rather than just looking at static properties, we are actually interested in dynamic ones. In particular, surface reduction is well transported into the bang calculus (simulation property in [5]). Interestingly, we show that it also holds for the full reduction. Moreover, we show that reverse simulation is also verified for both surface and full reductions.

Theorem 3.2 (Surface/Full Simulation and Reverse Simulation)**.** *Let $t, u \in \Lambda_{\mathtt{n}}$, then:*

- *$t \to_{\mathtt{N}} u$ (resp. $t \twoheadrightarrow_{\mathtt{N}} u$) if and only if $t^{\mathtt{cbn}} \to_{\mathtt{S}} u^{\mathtt{cbn}}$ (resp. $t^{\mathtt{cbn}} \twoheadrightarrow_{\mathtt{S}} u^{\mathtt{cbn}}$).*
- *$t \to_{\mathtt{C}} u$ (resp. $t \twoheadrightarrow_{\mathtt{C}} u$) if and only if $t^{\mathtt{cbn}} \to_{\mathtt{F}} u^{\mathtt{cbn}}$ (resp. $t^{\mathtt{cbn}} \twoheadrightarrow_{\mathtt{F}} u^{\mathtt{cbn}}$).*

Moreover, the number of $\mathtt{dB}$ (resp. $\mathtt{s}$) steps exactly matches the number of $\mathtt{dB}$ (resp. $\mathtt{s}!$) steps.

4 Call-by-Value – $\lambda_{\mathtt{vsub}}$

We now briefly introduce the (distant call-by-value) $\lambda_{\mathtt{vsub}}$-calculus [2]. The set $\Lambda_{\mathtt{v}}$ of terms is given by the following inductive definitions:

$$\begin{array}{rrcl} \textbf{(Terms)} & t, u & ::= & v \mid tu \mid t[x\backslash u] \\ \textbf{(Values)} & v & ::= & x \in \mathcal{X} \mid \lambda x.t \end{array}$$

$$\begin{array}{lccc} \text{In } \Lambda_{\mathtt{n}}: & (\lambda x.((\lambda y.y)z))z & \not\to_{\mathtt{v}} & (\lambda x.(y[y\backslash z]))z \\ & \downarrow_{\cdot^{\mathtt{cbv}}} & & \downarrow_{\cdot^{\mathtt{cbv}}} \\ \text{In } \Lambda_{!}: & (\lambda x.((\lambda y.!y)\,(!z)))\,(!z) & \to_{\mathtt{s}} & (\lambda x.((!y)[y\backslash !z]))\,(!z) \end{array}$$

Figure 2: A Counterexample of the CbV Reverse Simulation Property

$$\begin{array}{rcl} x^{\mathtt{CBV}} & = & !x \\ (\lambda x.t)^{\mathtt{CBV}} & = & !\lambda x.!t^{\mathtt{CBV}} \\ (tu)^{\mathtt{CBV}} & = & \begin{cases} \mathtt{der}(\mathtt{L}\,\langle s\rangle\, u^{\mathtt{CBV}}) & \text{if } t^{\mathtt{CBV}} = \mathtt{L}\,\langle !s\rangle \\ \mathtt{der}(\mathtt{der}(t^{\mathtt{CBV}})u^{\mathtt{CBV}}) & \text{otherwise} \end{cases} \\ (t[x\backslash u])^{\mathtt{CBV}} & = & t^{\mathtt{CBV}}[x\backslash u^{\mathtt{CBV}}] \end{array}$$

Figure 3: A New Call-by-Value Embedding for Bang-calculus

Again, we consider the set of terms up to the usual α-conversion [4] and we denote by $t\{x := v\}$ the expected (capture avoiding) meta-level substitution. The sets of list contexts $\mathtt{L}$, surface contexts $\mathtt{V}$ and full contexts $\mathtt{C}$, are inductively defined as follows:

$$\begin{array}{rrcl} \textbf{(List)} & \mathtt{L} & ::= & \diamond \mid \mathtt{L}[x\backslash t] \\ \textbf{(Surface)} & \mathtt{V} & ::= & \diamond \mid \mathtt{V}\,t \mid t\,\mathtt{V} \mid \mathtt{V}[x\backslash t] \mid t[x\backslash \mathtt{V}] \\ \textbf{(Full)} & \mathtt{C} & ::= & \diamond \mid \lambda x.\mathtt{C} \mid \mathtt{C}\,t \mid t\,\mathtt{C} \mid \mathtt{C}[x\backslash t] \mid t[x\backslash \mathtt{C}] \end{array}$$

In particular, the hole can occur everywhere in $\mathtt{C}$, while in $\mathtt{V}$ it cannot occur under an abstraction. The following rewriting rules are the base components of our reduction relations.

$$\mathtt{L}\,\langle \lambda x.t\rangle\, u \quad \mapsto_{\mathtt{dB}} \quad \mathtt{L}\,\langle t[x\backslash u]\rangle \qquad\qquad t[x\backslash \mathtt{L}\,\langle v\rangle] \quad \mapsto_{\mathtt{sV}} \quad \mathtt{L}\,\langle t\{x := v\}\rangle$$

In both these rules the reduction acts at a distance in order to deal with stuck redexes since this phenomenon also appear in call-by-value. The surface reduction relation $\to_{\mathtt{v}}$ (resp. full reduction relation $\to_{\mathtt{c}}$) is defined as the surface closure $\mathtt{V}$ (resp. full closure $\mathtt{C}$) of the rules $\mathtt{dB}$ and $\mathtt{sV}$ presented above. We denote by $\twoheadrightarrow_{\mathtt{v}}$ (resp. $\twoheadrightarrow_{\mathtt{c}}$) the reflexive transitive closure of the surface reduction $\to_{\mathtt{v}}$ (resp. full reduction $\to_{\mathtt{c}}$) and finally, a term $t \in \Lambda_{\mathtt{v}}$ is said in to be a surface (resp. full) normal form if there is no u such that $t \to_{\mathtt{v}} u$ (resp. $t \to_{\mathtt{c}} u$).

The embedding $\cdot^{\mathtt{cbv}}$ presented in Fig. 1 preserves surface normal forms and satisfies the simulation property for the surface reduction [5]. However, reverse simulation fails for the surface reduction. A counter example can be found in Fig 2.

We introduce in Fig. 3 a new call-by-value embedding $\cdot^{\mathtt{CBV}}$ solving the issue. The main difference can be found in the abstraction case where two ! are used instead of one. The application is then adapted by placing an additional dereliction on the outside. This dereliction gets rid of the bang on the body of the abstraction once the $\mathtt{dB}$ redex has been fired. This restores access to the body of the previously existing abstraction, matching the phenomenon at play in distance.

This new embedding has the same good static property of preserving the surface normal forms and is additionally shown to preserve full normal forms.

Theorem 4.1 (Normal Forms Preservation)**.** *Let $t \in \Lambda_{\mathtt{v}}$, then t is a surface (resp. full) normal form if and only if $t^{\mathtt{CBV}}$ is a surface (resp. full) normal form.*

Moreover and as intended, this new encoding is satisfying both simulation and reverse simulation properties for both surface and full reductions:

Theorem 4.2 (Surface/Full Simulation and Reverse Simulation)**.** *Let $t, u \in \Lambda_{\mathtt{vsub}}$, then:*

- $t \to_{\mathtt{v}} u$ *(resp.* $t \twoheadrightarrow_{\mathtt{v}} u$*) if and only if* $t^{\mathtt{CBV}} \twoheadrightarrow_{\mathtt{s}} u^{\mathtt{CBV}}$.
- $t \to_{\mathtt{c}} u$ *(resp.* $t \twoheadrightarrow_{\mathtt{c}} u$*) if and only if* $t^{\mathtt{CBV}} \twoheadrightarrow_{\mathtt{F}} u^{\mathtt{CBV}}$.

where the number of dB *(resp.* s*) steps exactly matches the number of* dB *(resp.* s!*) steps.*

References

[1] Beniamino Accattoli and Delia Kesner. The structural *lambda*-calculus. In Anuj Dawar and Helmut Veith, editors, *Proceedings of 24th EACSL Conference on Computer Science Logic*, volume 6247 of *LNCS*, pages 381–395. Springer, August 2010.

[2] Beniamino Accattoli and Luca Paolini. Call-by-value solvability, revisited. In Tom Schrijvers and Peter Thiemann, editors, *FLOPS*, volume 7294 of *LNCS*, pages 4–16. Springer, 2012.

[3] Victor Arrial, Giulio Guerrieri, and Delia Kesner. Quantitative inhabitation for different lambda calculi in a unifying framework. *Proc. ACM Program. Lang.*, 7(POPL):1483–1513, 2023.

[4] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundation of mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[5] Antonio Bucciarelli, Delia Kesner, Alejandro Ríos, and Andrés Viso. The bang calculus revisited. In Keisuke Nakano and Konstantinos Sagonas, editors, *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings*, volume 12073 of *Lecture Notes in Computer Science*, pages 13–32. Springer, 2020.

[6] Thomas Ehrhard. Call-by-push-value from a linear logic point of view. In Peter Thiemann, editor, *ESOP 2016*, volume 9632 of *Lecture Notes in Computer Science*, pages 202–228. Springer, 2016.

[7] Thomas Ehrhard and Giulio Guerrieri. The bang calculus: an untyped lambda-calculus generalizing call-by-name and call-by-value. In James Cheney and Germán Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 174–187. ACM, 2016.

[8] Claudia Faggian and Giulio Guerrieri. Factorization in call-by-name and call-by-value calculi via linear logic. In Stefan Kiefer and Christine Tasson, editors, *FOSSACS 2021*, volume 12650 of *Lecture Notes in Computer Science*, pages 205–225. Springer, 2021.

[9] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[10] Giulio Guerrieri and Giulio Manzonetto. The bang calculus and the two girard's translations. In *Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity-TLLA@FLoC 2018*, volume 292 of *EPTCS*, pages 15–30, 2018.

[11] Giulio Guerrieri and Federico Olimpieri. Categorifying non-idempotent intersection types. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference)*, volume 183 of *LIPIcs*, pages 25:1–25:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[12] Delia Kesner and Andrés Viso. Encoding tight typing in a unified framework. *CoRR*, abs/2105.00564, 2021.

[13] Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, pages 228–243, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[14] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

A Confluence Criterion for Non Left-Linearity in a $\beta\eta$-Free Reformulation of HRSs

Thiago Felicissimo

Université Paris-Saclay, INRIA project Deducteam, LMF, ENS Paris-Saclay
thiago.felicissimo@inria.fr

Abstract

We give a criterion for showing confluence of non-left linear (Pattern) Higher-order Rewrite Systems (HRSs). More precisely, our criterion concerns 2nd order signatures and allows one to show object confluence, that is, confluence when restricting to terms only with variables of order 0. As a second contribution, we give a reformulation of HRSs in which one never needs to speak of meta-level $\beta\eta$-equality.

1 Introduction

When proving confluence in Nipkow's (Pattern) Higher-order Rewrite Systems (HRSs) [8], one generally has to be in one of the two following cases. If the rewrite system being considered is (1) strongly normalizing (s.n.), then by the critical pair lemma it suffices to check that critical pairs are joinable [8]. In most situations this is too strong of a requirement, but fortunately if the system is (2) left-linear then many criteria exist, such as orthogonality [9] and development-closedness [10]. If however neither (1) nor (2) hold, then no known criteria allows for showing confluence. This is problematic even when non-linearity is only necessary for silly reasons, such as in the following example.

$$\Sigma_{\lambda\pi^\uparrow} = \quad \lambda : (\mathtt{t} : \mathsf{tm} \to \mathsf{tm}) \to \mathsf{tm}, \quad @ : (\mathtt{t} : \mathsf{tm}, \mathtt{u} : \mathsf{tm}) \to \mathsf{tm}, \quad \uparrow : (\mathtt{n} : \mathsf{lvl}, \mathtt{m} : \mathsf{lvl}, \mathtt{t} : \mathsf{tm}) \to \mathsf{tm},$$
$$\pi : (\mathtt{n} : \mathsf{lvl}, \mathtt{a} : \mathsf{tm}, \mathtt{b} : \mathsf{tm} \to \mathsf{tm}) \to \mathsf{tm}, \quad 0 : \mathsf{lvl}, \quad \mathsf{S} : (\mathtt{n} : \mathsf{lvl}) \to \mathsf{lvl}$$

$$\mathcal{R}_{\lambda\pi^\uparrow} = \quad \uparrow^{\mathtt{m}}_{\mathtt{n}} (\pi_{\mathtt{n}}(\mathtt{A}, x.\mathtt{B}(x))) \longmapsto \pi_{\mathtt{m}}(\uparrow^{\mathtt{m}}_{\mathtt{n}} (\mathtt{A}), x.\, \uparrow^{\mathtt{m}}_{\mathtt{n}} (\mathtt{B}(x))) \qquad \uparrow^{\mathtt{n}}_{\mathtt{n}} (\mathtt{t}) \longmapsto \mathtt{t}$$
$$\lambda(x.\mathtt{t}(x))_{@}\mathtt{u} \longmapsto \mathtt{t}(\mathtt{u}) \qquad \uparrow^{\mathtt{p}}_{\mathtt{m}} (\uparrow^{\mathtt{m}}_{\mathtt{n}} (\mathtt{t})) \longmapsto \uparrow^{\mathtt{p}}_{\mathtt{n}} (\mathtt{t})$$

Example 1. Consider the signature (where $\mathsf{tm}, \mathsf{lvl}$ are sorts) and the rewrite system given above, containing an excerpt of the rules used when defining a cumulative Tarski-style universe — similar ones can be found in [3]. Because of the beta rule, the rewrite system is not s.n. and because of the other rules the system is also non left-linear. Non-linearity is only used to obtain a finite signature, a pre-requisite for some applications.

Actually, there is no hope of showing confluence for $\mathcal{R}_{\lambda\pi^\uparrow}$, given that it is possible to simulate the rewrite system $\mathcal{R}_k = \{\lambda(x.\mathtt{t}(x))_{@}\mathtt{u} \longmapsto \mathtt{t}(\mathtt{u}),\ f(\mathtt{t},\mathtt{t}) \longmapsto a\}$ shown by Klop to be not confluent [7]. Indeed, by taking variables $x : \mathsf{tm} \to \mathsf{lvl}, y : \mathsf{tm}$ we can translate $[\![f(t,u)]\!] := \uparrow^{x([\![u]\!])}_{x([\![t]\!])} (y)$ and $[\![a]\!] := y$, and then show that $t \longrightarrow u$ implies $[\![t]\!] \longrightarrow [\![u]\!]$ and $[\![t]\!] \longrightarrow u$ implies $t \longrightarrow u'$ for some u' with $[\![u']\!] = u$. Using these two facts it is easy to see that the confluence of $\mathcal{R}_{\lambda\pi^\uparrow}$ implies that of $\mathcal{R}_k$.

This counterexample however makes essential use of the fact that we have access to a variable $x : \mathsf{tm} \to \mathsf{lvl}$, allowing us to perform a beta step inside a non-linear position of the lhs in $\uparrow^{\mathtt{n}}_{\mathtt{n}} (\mathtt{t}) \longmapsto \mathtt{t}$. However, in most applications one is only interested in terms containing 0-order variables, and higher-order variables are only used as metavariables to define the rewrite rules. If we instead restrict our attention to confluence over terms containing only variables of order 0 (a property we will call *object confluence*), can we prove that $\mathcal{R}_{\lambda\pi^\uparrow}$ satisfies this property?

In this article, we propose a criterion that allows us to do that. More precisely, given two rewrite systems $\mathcal{R}_l$ and $\mathcal{R}_{nl}$ over a signature of order at most 2 such that (1) both are

object confluent, (2) $\mathcal{R}_l$ is linear, (3) there are no critical pairs between them and (4) the sorts of non-linear lhs variables of $\mathcal{R}_{nl}$ are inaccessible from the sorts of the rules in $\mathcal{R}_l$, our criterion allows one to conclude object confluence of their union. The proof is a simple adaptation of the proof of confluence by orthogonality, by using condition (4) to show that a $\mathcal{R}_l$ step cannot destroy a $\mathcal{R}_{nl}$ redex. As shown in the end of the article, our criterion proves the object confluence of Example 1.

As a second contribution, we give a reformulation of Nipkow's HRSs in which one never needs to talk about $\beta\eta$-equivalence. This is achieved by adopting a canonical forms only presentation of the simply-typed λ-calculus, and replacing regular substitution by hereditary substitution [6]. This avoids the technicalities of switching $\beta\eta$ representatives and allows for a presentation of higher-order rewriting that we believe can be clearer.

Related work The problem of higher-order confluence with non-left linear rules has been studied by [2] and [3] in the setting of rewriting union β. The notion of *confinement* introduced in [2] was an essential inspiration for us. We omit a detailed discussion because of size constraints, but remark that our criterion's proof is much shorter and less technical.

2 Higher-order rewriting

We start by introducing our reformulation of Nipkow's (Pattern) Higher-order Rewriting Systems (HRSs). We suppose we are given three infinite and disjoint sets of *variables* $\mathcal{V}$, refered to by x, y, z or by letters in typewriter font such as $\mathtt{a}, \mathtt{b}, \mathtt{t}$, *(syntactic) constructors* $\mathcal{C}$, refered to by c, d, f, g, and *sorts* $\mathcal{S}$, refered to by s. A *head* h is either a constructor c or a variable x. We define *arities*, *scopes* and *signatures* by the grammars

$$\boxed{\mathsf{Arity}} \ni \sigma, \tau ::= \delta \to s \qquad \boxed{\mathsf{Scope}} \ni \gamma, \delta ::= \cdot \mid \gamma, x : \tau \qquad \boxed{\mathsf{Sig}} \ni \Sigma ::= \cdot \mid \Sigma, c : \tau$$

and abbreviate $\cdot \to s$ as simply s. We write $\vec{x}_\gamma$ for the sequence of variables in γ, and $\gamma.\delta$ for concatenation. A subscope γ' of γ is a subsequence of γ, written $\gamma' \sqsubseteq \gamma$.

In other works, one usually calls γ a context and τ a simple type. We however prefer to insist here on a different point of view, in which τ is seen as a higher-order generalization of the regular notion of arity.

Given a fixed signature Σ, *terms* and *spines* are mutually defined by the following inference rules. From the perspective of the λ-calculus, our terms can be seen as the simply-typed λ-terms of some base type, and in β-normal η-long form (or canonical form). However, our definition allows us to capture directly the terms of interest, and unlike [8] we never need to speak about the non canonical forms, which play only a bureaucratic role. The definition also clarifies the fact that higher-order rewriting is not (or at least does not need to be seen as) a form of rewriting modulo, but instead rewriting in which one adopts a different notion of substitution (as we will see next).

In the following, when convenient we abbreviate $h(\varepsilon)$ as h. We write $e \in \mathsf{Expr}\ \gamma$ when either $e \in \mathsf{Tm}\ \gamma\ s$ or $e \in \mathsf{Sp}\ \gamma\ \delta$, and we call e an expression. Finally, given a spine $\mathbf{t} \in \mathsf{Sp}\ \gamma\ \delta$ and a variable $x : \gamma_x \to s_x \in \delta$, we write $\mathbf{t}_x \in \mathsf{Tm}\ \gamma.\gamma_x\ s_x$ for the term in $\mathbf{t}$ at variable x.

$$h : \delta \to s \in \Sigma \cup \gamma \frac{\mathbf{t} \in \mathsf{Sp}\ \gamma\ \delta}{h(\mathbf{t}) \in \mathsf{Tm}\ \gamma\ s} \qquad \frac{}{\varepsilon \in \mathsf{Sp}\ \gamma\ \cdot} \qquad \frac{\mathbf{t} \in \mathsf{Sp}\ \gamma\ \delta \qquad t \in \mathsf{Tm}\ \gamma.\gamma'\ s}{\mathbf{t}, \vec{x}_{\gamma'}.t \in \mathsf{Sp}\ \gamma\ (\delta, x : \gamma' \to s)}$$

Example 2. If we take $\Sigma = \lambda : (\mathtt{t} : \mathtt{tm} \to \mathtt{tm}) \to \mathtt{tm}, @ : (\mathtt{t} : \mathtt{tm}, \mathtt{u} : \mathtt{tm}) \to \mathtt{tm}$, then $\mathsf{Tm}\ (\vec{x} : \vec{\mathtt{tm}})\ \mathtt{tm}$ contains exactly the λ-term with free variables in $\vec{x}$. This justifies why, unlike in the original formulation of HRS, we do not consider $x.t$ to be a term, as $\lambda(x.t)$ corresponds to a term in the λ-calculus, but $x.t$ or $y.\lambda(x.t)$ do not. Moreover, this makes the restriction of rules to base types in [8] completely automatic in our formulation.

Remark 1. We present the syntax informally using names and α-equivalence as a convenience, but we expect that everything can be formally carried out using deBruijn indices.

Substitution Because of our definition of terms, naive substitution would not work: for instance, syntactically replacing x by $z.\mathsf{S}(z)$ and y by $\mathsf{0}$ in $x(y)$ would yield $(z.\mathsf{S}(z))(\mathsf{0})$, which is not a valid term. We instead use *hereditary substitution* [6], which in this case recursively replaces z by $\mathsf{0}$, giving $\mathsf{S}(\mathsf{0})$. This is defined by the following clauses, by lexographic induction on γ_2 and the expression being substituted. In the following, we write just $e[\mathbf{t}]$ instead of $e[\mathbf{t}/\delta]$ if no ambiguity arises.

$$\_[\_/\gamma_2] : \mathsf{Tm}\ \gamma_1.\gamma_2.\gamma_3\ s \to \mathsf{Sp}\ \gamma_1\ \gamma_2 \to \mathsf{Tm}\ \gamma_1.\gamma_3\ s$$
$$x(\mathbf{v})[\mathbf{u}/\gamma_2] := v[\mathbf{v}[\mathbf{u}/\gamma_2]/\delta] \qquad \text{if } x : \delta \to s \in \gamma_2 \text{ and } \mathbf{u}_x = v$$
$$h(\mathbf{v})[\mathbf{u}/\gamma_2] := h(\mathbf{v}[\mathbf{u}/\gamma_2]) \qquad \text{if } h \in \Sigma, \gamma_1, \gamma_3$$
$$\_[\_/\gamma_2] : \mathsf{Sp}\ \gamma_1.\gamma_2.\gamma_3\ \delta \to \mathsf{Sp}\ \gamma_1\ \gamma_2 \to \mathsf{Sp}\ \gamma_1.\gamma_3\ \delta$$
$$\varepsilon[\mathbf{u}/\gamma_2] := \varepsilon$$
$$(\mathbf{v}, \vec{y}.t)[\mathbf{u}/\gamma_2] := \mathbf{v}[\mathbf{u}/\gamma_2], \vec{y}.t[\mathbf{u}/\gamma_2]$$

Sometimes we need a spine $\mathbf{v} \in \mathsf{Sp}\ \gamma\ \gamma$ that satisfies $e[\mathbf{v}] = e$ for all e. Normally one takes $\mathbf{v} = \vec{x}_\gamma$, but in general this is not a valid spine. Instead, we need to define the *identity spine* $\mathsf{id}_\gamma \in \mathsf{Sp}\ \gamma\ \gamma$ by $\mathsf{id}_{(\cdot)} := \varepsilon$ and $\mathsf{id}_{\gamma, x:\delta\to s} := \mathsf{id}_\gamma, \vec{y}_\delta.x(\mathsf{id}_\delta)$. Intuitively, it η-expands each variable in $\vec{x}_\gamma$ so that the resulting sequence is indeed a valid spine. We can now verify that $e[\mathsf{id}_\gamma] = e$ for all $e \in \mathsf{Expr}\ \gamma$, and moreover $\mathsf{id}_\delta[\mathbf{t}] = \mathbf{t}$ for all $\mathbf{t} \in \mathsf{Sp}\ \gamma\ \delta$.

Rewriting Given an expression e, we write $\mathsf{Pos}\ e$ for its set of positions and $\mathsf{FPos}\ e$ for its set of functional positions. For each $p \in \mathsf{Pos}\ e$ let $\gamma_p \in \mathsf{Scope}$ be the scope introduced between the root and p, and $s_p \in \mathcal{S}$ the sort at p. Given $e \in \mathsf{Expr}\ \gamma$ and $p \in \mathsf{Pos}\ e$ we write $e|_p \in \mathsf{Tm}\ \gamma.\gamma_p\ s_p$ for the subterm at position p, and given a term $t \in \mathsf{Tm}\ \gamma.\gamma_p\ s_p$ we write $e\{t\}_p$ for the result of replacing $e|_p$ by t in e.

A pattern $e \in \mathsf{Patt}\ \gamma\ \gamma'$ is an expression $e \in \mathsf{Expr}\ \gamma.\gamma'$ in which each variable $x \in \gamma$ appearing at position p occurs applied to $\mathsf{id}_{\gamma''}$ where $\gamma'' \sqsubseteq \gamma'.\gamma_p$. We think of variables in γ as flexible and γ' as rigid. We write $t \in \mathsf{Tm}^\mathsf{P}\ \gamma\ \gamma'\ s$ for a term pattern and $\mathbf{t} \in \mathsf{Sp}^\mathsf{P}\ \gamma\ \gamma'\ \delta$ for a spine pattern. We have $\mathsf{Tm}^\mathsf{P}\ \gamma\ \gamma'\ s \subseteq \mathsf{Tm}\ \gamma.\gamma'\ s$ and $\mathsf{Sp}^\mathsf{P}\ \gamma\ \gamma'\ \delta \subseteq \mathsf{Sp}\ \gamma.\gamma'\ \delta$.

Given a pattern $e \in \mathsf{Patt}\ \gamma\ \gamma'$, we write $\mathsf{ffv}(e)$ for the subscope of γ containing exactly the free flexible variables of e. A pattern is linear if any $x \in \mathsf{ffv}(e)$ occurs only once.

A rewrite rule $\gamma \Vdash t \longmapsto u : s$ is given by $t \in \mathsf{Tm}^\mathsf{P}\ \gamma\ \cdot\ s$ and $u \in \mathsf{Tm}\ \gamma\ s$ st $\gamma = \mathsf{ffv}(t)$ and t is not a variable. It is linear if l is a linear pattern. We define the rewrite relation $e \longrightarrow e'$ for $e, e' \in \mathsf{Expr}\ \gamma$ if there is a rewrite rule $\delta \Vdash l \longmapsto r : s$ and position $p \in \mathsf{Pos}\ e$ and spine $\mathbf{v} \in \mathsf{Sp}\ \gamma.\gamma_p\ \delta$ such that $s = s_p$ and $e|_p = l[\mathbf{v}]$ and $e' = e\{r[\mathbf{v}]\}_p$.

Critical pairs Given $t, u \in \mathsf{Tm}\ \delta.\gamma\ s$ we call $\delta \mid \gamma \Vdash t =^? u : s$ a unification problem. A unifier is a spine $\mathbf{v} \in \mathsf{Sp}\ \delta'\ \delta$ st $t[\mathbf{v}] = u[\mathbf{v}]$. When $t, u \in \mathsf{Tm}^\mathsf{P}\ \delta\ \gamma\ s$, the problem has either a most general unifer (mgu) or no unifier.

It is known that one of the difficulties when going from first order to higher-order rewriting is adapting the definition of critical pairs. Given $\delta_i \Vdash l_i \longmapsto r_i : s_i$ and $p \in \mathsf{FPos}\ l_1$, if one tries naively to unify $\delta_1.\delta_2 \mid \gamma_p \Vdash l_1|_p =^? l_2 : s_2$, then because the variables in γ_p introduced between the root and p in l_1 do not appear in l_2, any unifier must throw dependencies on such variables away, which is not what is intended. Instead, we first need to add γ_p as dependencies to the variables appearing in l_2. This is achieved by a substitution Nipkow calls a $\vec{x}_{\gamma_p}$*-lifter*, but it can also be understood more algebraically.

Given $\gamma, \delta \in \mathsf{Scope}$, we define the exponential scope $\gamma \rightarrowtail \delta \in \mathsf{Scope}$ by replacing each entry $x : \gamma_x \to s \in \delta$ by $x^* : \gamma.\gamma_x \to s$. We have an evaluation spine pattern $\mathsf{eval}_{\gamma,\delta} \in \mathsf{Sp}^\mathsf{P}\ (\gamma \rightarrowtail \delta)\ \gamma\ \delta$, containing at entry $x : \gamma_x \to s \in \delta$ the argument $\vec{x}_{\gamma_x}.x^*(\mathsf{id}_\gamma, \mathsf{id}_{\gamma_x})$. For each $\mathbf{t} \in \mathsf{Sp}\ \gamma'.\gamma\ \delta$ we define its curryfication $\mathsf{cur}\ \mathbf{t} \in \mathsf{Sp}\ \gamma'\ (\gamma \rightarrowtail \delta)$ by replacing each entry $\vec{x}_{\gamma_x}.t$ in $\mathbf{t}$ by $\vec{x}_{\gamma.\gamma_x}.t$. The important property we have is that for all $\mathbf{t} \in \mathsf{Sp}\ \gamma'.\gamma\ \delta$, $\mathsf{cur}\ \mathbf{t}$ is the unique spine satisfying $\mathbf{t} = \mathsf{eval}_{\gamma,\delta}[\mathsf{cur}\ \mathbf{t}/\gamma \rightarrowtail \delta]$. Our notation evidences the fact that $\gamma \rightarrowtail \delta$ is the exponential object in the category of scope and spines, with $\mathsf{eval}_{\gamma,\delta}$ its corresponding evaluation morphism.

We can now define overlaps and critical pairs. A pattern $e \in \mathsf{Patt}\ \gamma\ \gamma'$ overlaps a rewrite rule $\delta \Vdash l \longmapsto r : s$ at functional position $p \in \mathsf{FPos}\ e$ if the unification problem

$$\gamma.(\gamma'.\gamma_p \rightarrowtail \delta) \mid \gamma'.\gamma_p \Vdash e|_p \stackrel{?}{=} l[\mathsf{eval}_{\gamma'.\gamma_p,\delta}] : s$$

has a unifier — in which case, it also has a most general one. A pattern overlap is proper if $e = l$ implies $p \neq \varepsilon$. A rule overlap is given by two rules $\delta_1 \Vdash l_1 \longmapsto r_1 : s_1$ and $\delta_2 \Vdash l_2 \longmapsto r_2 : s_2$ and a functional position $p \in \mathsf{FPos}\ l_1$ st l_1 properly overlaps $l_2 \longmapsto r_2$ at position p. Each rule overlap gives rise to a critical pair $\langle r_1[\mathbf{v}], l_1\{r_2[\mathsf{eval}_{\gamma_p,\delta_2}]\}_p[\mathbf{v}]\rangle$, where $\mathbf{v}$ is the mgu of the associated unification problem.

Superdevelopments Like with many proofs of confluence, ours will employ Aczel's superdevelopments [1], defined by the following rules.

$$\delta \Vdash l \longmapsto r : s \in \mathcal{R} \frac{f(\mathbf{u}) = l[\mathbf{v}] \qquad \mathbf{t} \Longrightarrow \mathbf{u} \in \mathsf{Sp}\ \gamma\ \delta_f}{f(\mathbf{t}) \Longrightarrow r[\mathbf{v}] \in \mathsf{Tm}\ \gamma\ s} \qquad h : \delta \to s \in \gamma \text{ or } \Sigma \frac{\mathbf{v} \Longrightarrow \mathbf{v}' \in \mathsf{Sp}\ \gamma\ \delta}{h(\mathbf{v}) \Longrightarrow h(\mathbf{v}') \in \mathsf{Tm}\ \gamma\ s}$$

$$\frac{}{\varepsilon \Longrightarrow \varepsilon \in \mathsf{Sp}\ \gamma\ \cdot} \qquad \frac{\mathbf{t} \Longrightarrow \mathbf{t}' \in \mathsf{Sp}\ \gamma\ \delta \qquad t \Longrightarrow t' \in \mathsf{Tm}\ \gamma.\gamma_x\ s}{\mathbf{t}, \vec{x}_{\gamma_x}.t \Longrightarrow \mathbf{t}', \vec{x}_{\gamma_x}.t' \in \mathsf{Sp}\ \gamma\ (\delta, x : \gamma_x \to s)}$$

Recall that we have $\longrightarrow \subseteq \Longrightarrow \subseteq \longrightarrow^*$ and thus $\longrightarrow^* = \Longrightarrow^*$. Moreover, superdevelopments are closed under substitution: if $e \Longrightarrow e' \in \mathsf{Expr}\ \gamma_1.\gamma_2.\gamma_3$ and $\mathbf{u} \Longrightarrow \mathbf{u}' \in \mathsf{Sp}\ \gamma_1\ \gamma_2$ then $e[\mathbf{u}/\gamma_2] \Longrightarrow e'[\mathbf{u}'/\gamma_2] \in \mathsf{Expr}\ \gamma_1.\gamma_3$. The following proposition is at the heart of most proofs of confluence by orthogonality [8]. We will also need it to show our criterion.

Proposition 1. *Let $e \in \mathsf{Patt}\ \delta\ \gamma'$ be a linear pattern that does not overlap any lhs of $\mathcal{R}$ and suppose that for some $\mathbf{v} \in \mathsf{Sp}\ \gamma\ \delta$ and e' we have $e[\mathbf{v}/\delta] \Longrightarrow e'$. Then we have $\mathbf{v}' \Longrightarrow \mathbf{v}'' \in \mathsf{Sp}\ \gamma\ \mathsf{ffv}(e)$ with $\mathbf{v}'$ a subspine of $\mathbf{v}$ and $e[\mathbf{v}''/\mathsf{ffv}(e)] = e'$.*

Corollary 1. *Let $\delta \Vdash l \longmapsto r : s$ be a linear rule that does not overlap any rule in $\mathcal{R}$. If $l[\mathbf{v}] = f(\mathbf{t})$ and $\mathbf{t} \Longrightarrow \mathbf{t}' \in \mathsf{Sp}\ \gamma\ \delta_f$, then there is $\mathbf{v}'$ with $\mathbf{v} \Longrightarrow \mathbf{v}' \in \mathsf{Sp}\ \gamma\ \delta$ st $l[\mathbf{v}'] = f(\mathbf{t}')$.*

3 A confluence criterion for non-left linearity

Define the order of an arity, of a scope and of a signature by $\mathsf{ord}(\gamma \to s) = 1 + \mathsf{ord}(\gamma)$, $\mathsf{ord}(\cdot) = -1$, $\mathsf{ord}(\gamma, x : \tau) = \max\{\mathsf{ord}(\gamma), \mathsf{ord}(\tau)\}$, $\mathsf{ord}(\Sigma, c : \tau) = \max\{\mathsf{ord}(\Sigma), \mathsf{ord}(\tau)\}$. Note then that the variables of order zero are the ones whose arity is just a sort.

One can remark that for most rewrite systems of interest the underlying signature Σ is of order ≤ 2, and variables of order > 0 are only needed for defining the rewrite rules. For instance, this is the case of the λ-calculus, where one needs a 1st order variable $\mathtt{t}$ to play the role of a metavariable in the rule $\lambda(x.\mathtt{t}(x))_{@}\mathtt{u} \longmapsto \mathtt{t}(\mathtt{u})$, but when translating a λ-term into its HRS representation one only uses zero order variables. This is also the case of most logics and type theories.

The restriction to signatures of order ≤ 2 is even baked into the definition of second-order formalisms, such as in [5]. There, one distinguishes between variables (order 0) and metavariables (order 1), and confluence of terms without metavariables is called *object confluence*. In the following, we rephrase this notion in the setting of 2nd order HRSs.

Suppose now that the underlying signature Σ is of order ≤ 2. A term $t \in \mathsf{Tm}\ \gamma\ s$ is an object term t if $\mathsf{ord}(\gamma) \leq 0$. A spine $\mathbf{t} \in \mathsf{Sp}\ \gamma\ \delta$ is an object spine if $\mathsf{ord}(\gamma) \leq 0$ and $\mathsf{ord}(\delta) \leq 1$. We refer to them generically as object expressions. Note that if e is an object expression then all terms and spines appearing in e are object expressions. Indeed, because Σ is of order at most 2, each constructor f can only bind 0-order variables. A rewrite system $\mathcal{R}$ is object confluent if the rewriting relation restricted to object expressions is confluent.

Given two sorts s, s' we say that s is accessible from s' (written $s' \preceq s$) if there is some object term t of sort s and position p such that $t|_p$ is of sort s'. Given a rewrite system $\mathcal{R}$ we write $\mathcal{R} \preceq s$ if for some $\delta \Vdash l \longmapsto r : s'$ we have $s' \preceq s$. Note that this notion is only

interesting because we only consider object terms: if one has access to all variables, the condition is always verified by taking $t = x(y)$ with $x : s' \to s$ and $y : s'$. It is easy to see that accessibility is decidable when the signature is finite.

Lemma 1. *If $t \in \mathsf{Tm}\ \gamma\ s$ is an object term and $t \Longrightarrow_{\mathcal{R}} t'$ with $\mathcal{R} \not\trianglelefteq s$ then $t = t'$.*

The heart of our proof is the following proposition, which is very similar to Proposition 1 but replaces linearity by an inaccessibility condition.

Proposition 2. *Let $e \in \mathsf{Patt}\ \delta\ \gamma'$ be a pattern that does not overlap any lhs of $\mathcal{R}$, with $\mathsf{ord}(\delta) \leq 1$ and such that for any $x : \gamma_x \to s_x \in \delta$ occurring non-linearly in e we have $\mathcal{R} \not\trianglelefteq s_x$. If for some $\mathbf{v} \in \mathsf{Sp}\ \gamma\ \delta$ and e' we have $e[\mathbf{v}/\delta] \Longrightarrow e'$ with $e[\mathbf{v}/\delta]$ an object expression, then we have $\mathbf{v}' \Longrightarrow \mathbf{v}'' \in \mathsf{Sp}\ \gamma\ \mathsf{ffv}(e)$ with $\mathbf{v}'$ a subspine of $\mathbf{v}$ and $e[\mathbf{v}''/\mathsf{ffv}(e)] = e'$.*

Proof. We replay the proof of Proposition 1, but in the case $e = \mathbf{t}, \vec{x}_{\gamma_x}.t$ we use Lemma 1 to merge the substitution spines for $\mathbf{t}$ and t. □

Corollary 2. *Let $\delta \Vdash l \longmapsto r : s$ be a rule that does not overlap any rule in $\mathcal{R}$, and such that for any $x : \gamma_x \to s_x \in \delta$ occurring non-linearly in l we have $\mathcal{R} \not\trianglelefteq s_x$. If $l[\mathbf{v}] \in \mathsf{Tm}\ \gamma\ s$ is an object term with $l[\mathbf{v}] = f(\mathbf{t})$ and $\mathbf{t} \Longrightarrow_{\mathcal{R}} \mathbf{t}' \in \mathsf{Sp}\ \gamma\ \delta_f$ then we have $\mathbf{v} \Longrightarrow_{\mathcal{R}} \mathbf{v}' \in \mathsf{Sp}\ \gamma\ \delta$ and $l[\mathbf{v}'] = f(\mathbf{t}')$.*

We are now ready to give our criterion.

Theorem 1. *Let $\mathcal{R}_l$ and $\mathcal{R}_{nl}$ be two rewriting systems on Σ with $\mathsf{ord}(\Sigma) \leq 2$ such that*

(*A*) *$\mathcal{R}_{nl}$ and $\mathcal{R}_l$ are object confluent*

(*B*) *$\mathcal{R}_l$ is left-linear*

(*C*) *There are no critical pairs between $\mathcal{R}_l$ and $\mathcal{R}_{nl}$*

(*D*) *For each $\delta \Vdash t \longmapsto u : s \in \mathcal{R}_{nl}$ and $x : \gamma_x \to s_x \in \delta$ with x occurring non-linearly in t, we have $\mathcal{R}_l \not\trianglelefteq s_x$*

Then $\mathcal{R}_l \cup \mathcal{R}_{nl}$ is object confluent.

Proof. Because $\mathcal{R}_{nl}$ and $\mathcal{R}_l$ are both object confluent, it suffices to show that $\mathcal{R}_{nl}$ and $\mathcal{R}_l$ commute on object expressions. We show ${}_{\mathcal{R}_{nl}}\!\Longleftarrow\!\Longrightarrow_{\mathcal{R}_l} \subseteq \Longrightarrow_{\mathcal{R}_l}\!{}_{\mathcal{R}_{nl}}\!\Longleftarrow$ on object expressions, by induction on the superdevelopments. The proof is essentially the same as the one for orthogonal systems [8], but using Corollary 2 for the case $f(\mathbf{t}) \Longrightarrow_{\mathcal{R}_{nl}} r[\mathbf{v}]$. □

We can now prove that the system of Example 1 is object confluent. First note that Σ is second-order as required. Then, by taking $\mathcal{R}_l = \{\lambda(x.\mathtt{t}(x))_{@}\mathtt{u} \longmapsto \mathtt{t}(\mathtt{u})\}$ and $\mathcal{R}_{nl} = \mathcal{R}_{\lambda\pi^{\uparrow}} \setminus \mathcal{R}_l$, we have that $\mathcal{R}_l$ is confluent by orthogonality and $\mathcal{R}_{nl}$ is confluent by joining its critical pairs and seeing that it is also strongly normalizing. Therefore, both are also object confluent. Moreover, $\mathcal{R}_l$ is linear and there are no critical pairs between $\mathcal{R}_l$ and $\mathcal{R}_{nl}$. Finally, we can easily verify that there is no object term of sort lvl containing a subterm of sort tm, and thus we have $\mathcal{R}_l \not\trianglelefteq \mathsf{lvl}$. Hence, by our criterion $\mathcal{R}_{\lambda\pi^{\uparrow}} = \mathcal{R}_l \cup \mathcal{R}_{nl}$ is object confluent.

Final remarks Our theorem could instead be stated for second-order formalisms like [5, 4] — these correspond roughly to the second-order fragment of HRSs. There, the notion of object confluence is arguably more natural, as the restriction to 0 order variables is built in the formalism.

Finally, the criterion is designed for situations in which we can split the rewrite system into two parts: one that is s.n. but not left-linear (whose confluence can hopefully be shown with the critical pair lemma), and one that is left-linear but (possibly) not s.n. (whose confluence can hopefully be shown with criteria assuming left-linearity). Nevertheless, the condition (*B*) could also be replaced by a condition similar to (*D*), making symmetric the roles of the two systems in the theorem. However, we do not know of any interesting examples for which this generalization would apply.

Acknowledgments The author would like to thank Gilles Dowek for his careful reading of a preliminary version of the paper, Jean-Pierre Jouannaud, Gaspard Férey and Frédéric Blanqui for their thoughtful remarks on this work, and the anonymous reviewers for their very helpful comments and suggestions.

References

[1] Peter Aczel. A general Church–Rosser theorem. 1978.

[2] Ali Assaf, Gilles Dowek, Jean-Pierre Jouannaud, and Jiaxiang Liu. Untyped Confluence In Dependent Type Theories. working paper or preprint, April 2017.

[3] Gaspard Ferey. *Higher-Order Confluence and Universe Embedding in the Logical Framework*. These, Université Paris-Saclay, June 2021.

[4] Makoto Hamana. Universal algebra for termination of higher-order rewriting. In *Term Rewriting and Applications: 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005. Proceedings 16*, pages 135–149. Springer, 2005.

[5] Makoto Hamana, Tatsuya Abe, and Kentaro Kikuchi. Polymorphic computation systems: Theory and practice of confluence with call-by-value. *Science of Computer Programming*, 187:102322, 2020.

[6] Robert Harper and Daniel R Licata. Mechanizing metatheory in a logical framework. *Journal of functional programming*, 17(4-5):613–673, 2007.

[7] Jan Willem Klop. *Combinatory reduction systems*. PhD thesis, Rijksuniversiteit Utrecht, 1963.

[8] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical computer science*, 192(1):3–29, 1998.

[9] Vincent van Oostrom. Confluence for abstract and higher-order rewriting. *Ph. D. Thesis, Vrije Universiteit*, 1984.

[10] Vincent Van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997.

Higher-Order LCTRSs and Their Termination

Liye Guo and Cynthia Kop

Radboud University, Netherlands

1 Introduction

Logically constrained term rewriting systems (LCTRSs) [4, 1] are a formalism for analyzing programs. In real-world programming, data types such as integers, as opposed to natural numbers, and arrays are prevalent. Any practical program analyzing technique should be prepared to handle these. One of the defining features of the LCTRS formalism is its native support for such data types, which are not (co)inductively defined and need to be encoded if handled by more traditional TRSs. Another benefit of the formalism is its separation between logical constraints modeling the control flow and other terms representing the program states.

So far, program analysis on the basis of LCTRSs has concerned imperative programs since LCTRSs were introduced as a first-order formalism. We are naturally curious to see if functional programs can also be analyzed by constrained rewriting. What we present here is our ongoing exploration in this direction: First, we define a higher-order variant of the LCTRS formalism, which, despite the absence of lambda abstractions, is capable of representing some real-world functional programs straightforwardly. Then we take a brief look at the termination problem for this new formalism as termination analysis is by itself an important aspect of program analysis as well as a prerequisite for determining some other properties.

2 LCSTRS

We start defining *logically constrained simply-typed term rewriting systems* (LCSTRSs) with types and terms. We postulate a set $\mathcal{S}$, whose elements we call *sorts*, and a subset $\mathcal{S}_\vartheta$ of $\mathcal{S}$, whose elements we call *theory sorts*. The set $\mathcal{T}$ of *types* and its subset $\mathcal{T}_\vartheta$, called the set of *theory types*, are generated as follows: $\mathcal{T} ::= \mathcal{S} \mid (\mathcal{T} \to \mathcal{T})$ and $\mathcal{T}_\vartheta ::= \mathcal{S}_\vartheta \mid (\mathcal{S}_\vartheta \to \mathcal{T}_\vartheta)$. Right-associativity is assigned to $\to$ so we can omit some parentheses in types. We assume given disjoint sets $\mathcal{F}$ and $\mathcal{V}$, whose elements we call *function symbols* and *variables*, respectively. The grammar $\mathbb{T} ::= \mathcal{F} \mid \mathcal{V} \mid (\mathbb{T}\ \mathbb{T})$ generates the set $\mathbb{T}$ of *pre-terms*. Left-associativity is assigned to the juxtaposition operation in the above grammar so $t_0\ t_1\ t_2$ stands for $((t_0\ t_1)\ t_2)$, for example. We assume that every function symbol and variable is assigned a unique type. Typing works as expected: if pre-terms t_0 and t_1 have types $A \to B$ and A, respectively, $t_0\ t_1$ has type B. Pre-terms having a type are called *terms*. We write $t : A$ if a term t has type A. We postulate a subset $\mathcal{F}_\vartheta$ of $\mathcal{F}$, whose elements we call *theory (function) symbols*, and assume that theory symbols have theory types. Terms constructed with only theory symbols and variables are called *logical terms*. The set of variables in a term t, denoted by $\mathrm{Var}(t)$, is defined as follows: $\mathrm{Var}(f) = \emptyset$, $\mathrm{Var}(x) = \{\, x \,\}$ and $\mathrm{Var}(t_0\ t_1) = \mathrm{Var}(t_0) \cup \mathrm{Var}(t_1)$. A term t is called a *ground term* if $\mathrm{Var}(t) = \emptyset$. Note that ground logical terms always have theory types.

Logical terms are distinguished because they will be treated specially when we define the rewrite relation. First, let us define the interpretation of ground logical terms. We postulate an $\mathcal{S}_\vartheta$-indexed family of sets $(\mathfrak{X}_A)_{A \in \mathcal{S}_\vartheta}$, and extend it to a $\mathcal{T}_\vartheta$-indexed family of sets by letting $\mathfrak{X}_{A \to B}$ be the set of maps from $\mathfrak{X}_A$ to $\mathfrak{X}_B$. Now we assume given a $\mathcal{T}_\vartheta$-indexed family of maps $(\llbracket \cdot \rrbracket_A)_{A \in \mathcal{T}_\vartheta}$ where $\llbracket \cdot \rrbracket_A$ assigns to each theory symbol whose type is A an element of $\mathfrak{X}_A$ and is

bijective if $A \in \mathcal{S}_\vartheta$. Theory symbols whose type is a theory sort are called *values*. We extend each indexed map $[\![\cdot]\!]_B$ to a map that assigns to each **ground logical term** whose type is B an element of $\mathfrak{X}_B$ by letting $[\![t_0\ t_1]\!]_B$ be $[\![t_0]\!]_{A\to B}([\![t_1]\!]_A)$. We omit the type and write just $[\![\cdot]\!]$ whenever the type can be deduced from the context. $[\![t]\!]$ is called the *interpretation* of t.

A *substitution* is a type-preserving map from variables to terms. Every substitution σ extends to a type-preserving map $\bar{\sigma}$ from terms to terms. We write $t\sigma$ for $\bar{\sigma}(t)$ and define it as follows: $f\sigma = f$, $x\sigma = \sigma(x)$ and $(t_0\ t_1)\sigma = (t_0\sigma)\ (t_1\sigma)$. Now we postulate a theory sort $\mathbb{B}$ and theory symbols $\bot : \mathbb{B}$ and $\top : \mathbb{B}$. Let $\mathfrak{X}_\mathbb{B}$ be $\{\, \mathfrak{o}, \mathfrak{i} \,\}$ and assume $[\![\bot]\!] = \mathfrak{o}$ and $[\![\top]\!] = \mathfrak{i}$. A *rewrite rule* $\ell \to r\ [\varphi]$ is a triple where (i) ℓ and r are terms which have the same type, (ii) ℓ is not a logical term, (iii) φ is a *logical constraint*, i.e., φ is a logical term whose type is $\mathbb{B}$ and the type of each variable in $\mathrm{Var}(\varphi)$ is a theory sort, and (iv) the type of each variable in $\mathrm{Var}(r) \setminus \mathrm{Var}(\ell)$ is a theory sort. A substitution σ is said to *respect* a rewrite rule $\ell \to r\ [\varphi]$ if $\sigma(x)$ is a value for all $x \in \mathrm{Var}(\varphi) \cup (\mathrm{Var}(r) \setminus \mathrm{Var}(\ell))$ and $[\![\varphi\sigma]\!] = \mathfrak{i}$. A set $\mathcal{R}$ of rewrite rules induces a *rewrite relation* $\to_\mathcal{R}$ on terms such that $t \to_\mathcal{R} t'$ if and only if one of the following conditions is true:

- $t = \ell\sigma$ and $t' = r\sigma$ for some $\ell \to r\ [\varphi] \in \mathcal{R}$ and some substitution σ that respects $\ell \to r\ [\varphi]$.
- $t = f\ v_1 \cdots v_n$ where f is a theory symbol but not a value while v_i is a value for all i, the type of t is a theory sort, and t' is the unique value such that $[\![f\ v_1 \cdots v_n]\!] = [\![t']\!]$.
- $t = t_0\ t_1$, $t' = {t_0}'\ t_1$ and $t_0 \to_\mathcal{R} {t_0}'$.
- $t = t_0\ t_1$, $t' = t_0\ {t_1}'$ and $t_1 \to_\mathcal{R} {t_1}'$.

Logical constraints are essentially first-order—higher-order variables are excluded and theory symbols take only first-order arguments. We adopt this restriction because many conditions in functional programs are still first-order and solving higher-order constraints is hard. That is not to say that higher-order constraints are of no interest; we simply leave them out of the scope of LCSTRSs.

Below is an example LCSTRS:

$$\begin{array}{rlll} \mathsf{init} \to \mathsf{fact}\ n\ \mathsf{exit} & [\top] & \mathsf{fact}\ n\ k \to k\ 1 & [n \leq 0] \\ \mathsf{comp}\ g\ f\ x \to g\ (f\ x) & [\top] & \mathsf{fact}\ n\ k \to \mathsf{fact}\ (n-1)\ (\mathsf{comp}\ k\ (*\ n)) & [n > 0] \end{array}$$

Here init and exit denote the start and the end of the program, respectively. The core of the program is fact, which computes the factorial function in continuation-passing style, and comp is an auxiliary function for function composition. Integer literals and operators are theory symbols. Note that we use infix notation to improve readability. The occurrence of n in the rewrite rule defining init is an example of a variable that occurs on the right-hand side but not on the left-hand side of a rewrite rule. Such variables can be used to model user input.

Let $\mathcal{R}$ denote the set of rewrite rules in the example and consider the rewrite sequence

$$\mathsf{fact}\ 1\ \mathsf{exit} \to_\mathcal{R} \mathsf{fact}\ (1-1)\ (\mathsf{comp}\ \mathsf{exit}\ (*\ 1)) \to_\mathcal{R} \mathsf{fact}\ 0\ (\mathsf{comp}\ \mathsf{exit}\ (*\ 1)) \to_\mathcal{R} \mathsf{comp}\ \mathsf{exit}\ (*\ 1)\ 1.$$

In the second step, no rewrite rule is invoked. Such rewrite steps are called *calculation steps*. We can write $\to_\emptyset$ for a calculation step. Terms s and t are said to be *joinable* by $\to_\emptyset$, written as $s \downarrow_\emptyset t$, if there exists a term r such that $s \to_\emptyset^* r$ and $t \to_\emptyset^* r$.

3 Termination

In order to prove that a given (unconstrained) TRS $\mathcal{R}$ is terminating, we usually look for a stable, monotonic and well-founded relation $\succ$ which orients every rewrite rule in $\mathcal{R}$, i.e., $\ell \succ r$ for all $\ell \to r \in \mathcal{R}$. This standard technique, however, requires a few tweaks to be applied to LCSTRSs. First, stability should be tightly coupled with rule orientation because every rewrite rule in an LCSTRS is equipped with a logical constraint, which decides what substitutions are expected when the rewrite rule is invoked. Therefore, we say that a type-preserving relation $\succ$ on terms *orients* a rewrite rule $\ell \to r\ [\varphi]$ if $\ell\sigma \succ r\sigma$ for each substitution σ that **respects** the rewrite rule. Second, the monotonicity requirement can be weakened because ℓ is never a logical term in a rewrite rule $\ell \to r\ [\varphi]$. We say that a type-preserving relation $\succ$ on terms is *rule-monotonic* if $t_0 \succ t_0{}'$ implies $t_0\ t_1 \succ t_0{}'\ t_1$ when t_0 is not a logical term, and $t_1 \succ t_1{}'$ implies $t_0\ t_1 \succ t_0\ t_1{}'$ when t_1 is not a logical term.

We present a tentative definition of HORPO [2] on LCSTRSs. For each theory sort A, we postulate theory symbols $\sqsupset_A : A \to A \to \mathbb{B}$ and $\sqsupseteq_A : A \to A \to \mathbb{B}$ such that $[\![\sqsupset_A]\!]$ is a well-founded ordering on $\mathfrak{X}_A$ and $[\![\sqsupseteq_A]\!]$ is the reflexive closure of $[\![\sqsupset_A]\!]$. We omit the sort and write just $\sqsupset$ and $\sqsupseteq$ whenever the sort can be deduced from the context. Given the *precedence* $\blacktriangleright$, a well-founded ordering on function symbols such that $f \blacktriangleright g$ for all $f \in \mathcal{F} \setminus \mathcal{F}_\vartheta$ and $g \in \mathcal{F}_\vartheta$, and the *status* stat, a map from $\mathcal{F}$ to $\{\mathfrak{l}, \mathfrak{m}_2, \mathfrak{m}_3, \dots\}$, the *higher-order recursive path ordering* (HORPO) $(\succsim_\varphi, \succ_\varphi)$ is a family of type-preserving relation pairs on terms indexed by logical constraints and defined as follows:

- $s \succsim_\varphi t$ if and only if one of the following conditions is true:
  - s and t are logical terms whose type is a theory sort, $\mathrm{Var}(\varphi) \supseteq \mathrm{Var}(s) \cup \mathrm{Var}(t)$ and $\varphi \models \sqsupseteq\ s\ t$.
  - $s \succ_\varphi t$.
  - $s \downarrow_\emptyset t$.
  - s is not a logical term, $s = s_1\ s_2$, $t = t_1\ t_2$, $s_1 \succsim_\varphi t_1$ and $s_2 \succsim_\varphi t_2$.
- $s \succ_\varphi t$ if and only if one of the following conditions is true:
  - s and t are logical terms whose type is a theory sort, $\mathrm{Var}(\varphi) \supseteq \mathrm{Var}(s) \cup \mathrm{Var}(t)$ and $\varphi \models \sqsupset\ s\ t$.
  - s and t have the same type and $s \triangleright_\varphi t$.
  - s is not a logical term, $s = x\ s_1 \cdots s_n$ where x is a variable, $t = x\ t_1 \cdots t_n$, $s_i \succsim_\varphi t_i$ for all i and there exists i such that $s_i \succ_\varphi t_i$.
- $s \triangleright_\varphi t$ if and only if s is not a logical term, $s = f\ s_1 \cdots s_m$ where f is a function symbol, and one of the following conditions is true:
  - $s_i \succsim_\varphi t$ for some i.
  - $t = t_0\ t_1 \cdots t_n$ and $s \triangleright_\varphi t_i$ for all i.
  - $t = g\ t_1 \cdots t_n$, $f \blacktriangleright g$ and $s \triangleright_\varphi t_i$ for all i.
  - $t = f\ t_1 \cdots t_n$, $\mathsf{stat}(f) = \mathfrak{l}$, $s_1 \cdots s_m \succ_\varphi^{\mathfrak{l}} t_1 \cdots t_n$ and $s \triangleright_\varphi t_i$ for all i.
  - $t = f\ t_1 \cdots t_n$, $\mathsf{stat}(f) = \mathfrak{m}_k$, $k \leq n$, $s_1 \cdots s_{\min(m,k)} \succ_\varphi^{\mathfrak{m}} t_1 \cdots t_k$ and $s \triangleright_\varphi t_i$ for all i.

In the above, $s_1 \cdots s_m \succ_\varphi^{\mathsf{l}} t_1 \cdots t_n$ if and only if $\exists i \leq \min(m,n)$ $(s_i \succ_\varphi t_i \wedge \forall j < i\ s_j \succsim_\varphi t_j)$, $\succ_\varphi^{\mathsf{m}}$ is the generalized multiset extension of $(\succsim_\varphi, \succ_\varphi)$ (see [3]), and $\varphi \models \varphi'$ denotes, on the assumption that φ and φ' are logical constraints such that $\mathrm{Var}(\varphi) \supseteq \mathrm{Var}(\varphi')$, that for each substitution σ which maps variables in $\mathrm{Var}(\varphi)$ to values, $[\![\varphi\sigma]\!] = \mathtt{1}$ implies $[\![\varphi'\sigma]\!] = \mathtt{1}$.

The design is that $\succ_\top$ should orient a rewrite rule $\ell \to r\ [\varphi]$ if $\ell \succ_\varphi r$. Then once a combination of $\sqsupset$, $\blacktriangleright$ and stat that guarantees $\ell \succ_\varphi r$ for all $\ell \to r\ [\varphi] \in \mathcal{R}$ is present, we can conclude that the LCSTRS $\mathcal{R}$ is terminating. The soundness of this method relies on the following properties of $\succ_\varphi$, which we must prove:

- $\succ_\top$ orients $\ell \to r\ [\varphi]$ if $\ell \succ_\varphi r$.
- $\succ_\top$ is rule-monotonic.
- $\succ_\top$ is well-founded.
- $\to_\emptyset \,;\, \succ_\top \subseteq \succ_\top$.

Note that $\to_\emptyset$ is well-founded because the size strictly decreases through every calculation step.

Consider the example LCSTRS given in the previous section. Any combination of $\sqsupset$, $\blacktriangleright$ and stat that satisfies the following properties would witness the well-foundedness of $\to_{\mathcal{R}}$: $[\![\sqsupset]\!] = \lambda xy.\ x > 0 \wedge x > y$, $\mathsf{init} \blacktriangleright \mathsf{fact} \blacktriangleright \mathsf{comp}$, $\mathsf{init} \blacktriangleright \mathsf{exit}$ and $\mathsf{stat}(\mathsf{fact}) = \mathsf{l}$.

4 Future Work

LCSTRSs are still a work in progress. While the formalism itself is in a somewhat stable state, the above method for termination analysis is in active development. First and foremost, we need to prove that HORPO on LCSTRSs has the expected properties. When the theory is complete, we would like to make a tool to automate the finding of HORPO on LCSTRSs. It would also be interesting to explore other methods for termination analysis on the new formalism, including StarHorpo [3] (a transitive variant of HORPO), interpretation-based methods and dependency pairs. Another direction is to go beyond LCSTRSs by augmenting the formalism with lambda abstractions or higher-order constraints.

References

[1] C. Fuhs, C. Kop, and N. Nishida. Verifying procedural programs via constrained rewriting induction. *ACM Transactions on Computational Logic*, 18(2):14:1–14:50, 2017.

[2] J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proc. LICS*, pages 402–411, 1999.

[3] C. Kop. *Higher Order Termination.* PhD thesis, VU Amsterdam, 2012.

[4] C. Kop and N. Nishida. Term rewriting with logical constraints. In *Proc. FroCoS*, pages 343–358, 2013.

Modular Termination for Second-Order Rewriting Systems and Application to Effect Handlers

Makoto Hamana[1]

Faculty of Informatics, Gunma University, Japan
hamana@gunma-u.ac.jp

Abstract

We present a new modular proof method of termination for second-order computation. The proof method is useful for proving termination of higher-order foundational calculi. To establish the method, we use a variation of the semantic labelling translation and Blanqui's General Schema: a syntactic criterion of strong normalisation. As an application, we show termination of a variant of call-by-push-value calculus with algebraic effects, an effect handler and effect theory. This is based on the author's paper "Modular Termination for Second-Order Computation Rules and Application to Algebraic Effect Handler", published in *Logical Methods in Computer Science*, Vol. 18, Issue 2, No.8, June 14, 2022.

1 Introduction

Computation rules such as the β-reduction of the λ-calculus and arrangement of let-expressions are fundamental mechanisms of functional programming. Computation rules for modern functional programming are necessarily higher-order and are presented as a λ-calculus extended with extra rules such as rules of let-expressions or first-order algebraic rules like "$0 + x \to x$".

The termination property is one of the most important properties of such a calculus because it is a key to ensuring the decidability of its properties. A powerful termination checking method is important in theory and in practice. Termination is ideally checked *modularly*. However, in general, strong normalisation is not a modular property [Toy87].

In this work we establish a new modular termination proof method of the following form: if $\mathbf{A}$ is SN and $\mathbf{B}$ is SN with some suitable conditions, then $\mathcal{C} = \mathbf{A} \uplus \mathbf{B}$ is SN. As an application of the modular termination proof method, we give a termination proof of effectful calculus using a variant of Levy's call-by-push value (CBPV) calculus [Lev06] called multi-adjunctive metalanguage (MAM) [FKLP19] with effect handlers and effect theory [PP13].

2 Main Theorem

We use a formal framework of second-order computation based on second-order algebraic theories [FH10]. This framework has been used in [Ham19, HAK20]. A ***computation system*** (CS) is a pair $(\Sigma, \mathcal{C})$ of a signature and a set $\mathcal{C}$ of computation rules. We write $s \Rightarrow_{\mathcal{C}} t$ to be one-step computation using $\mathcal{C}$.

Assumption 2.1. Let $(\Sigma_{\mathbf{A}} \uplus \Theta, \mathbf{A})$ and $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta, \mathbf{B})$ be computation systems satisfying:

(i) $\Sigma_{\mathbf{A}}$ is the set of defined function symbols of $\mathbf{A}$.
(ii) $\Sigma_{\mathbf{B}}$ is the set of defined function symbols of $\mathbf{B}$.
(iii) Θ is the signature for constructors of $\mathcal{C}$, where $\mathcal{C} \triangleq \mathbf{A} \uplus \mathbf{B}$.
(iv) $\mathcal{C}$ is finitely branching.

(v) Both sides of each rule in $\mathcal{C}$ satisfy the $\Sigma_{\mathbf{A}}$-layer condition.

Definition 2.2. We say that ***a meta-term*** u ***satisfies the*** $\Sigma_{\mathbf{A}}$***-layer condition*** if

- for every $f(\overline{\overline{x}.t}) \trianglelefteq u$ with $f \in \Sigma_{\mathbf{A}}$, $\mathsf{Fun}(f(\overline{\overline{x}.t})) \subseteq \Sigma_{\mathbf{A}} \uplus \Theta$ and $\overline{t}$ are second-order patterns, and
- if u is a meta-application $M[s_1, \ldots, s_n]$, every s_i does not contain function symbols. This condition has been called that u is *solid* [Ham07].

We say that $\mathbf{A}$ *is accessible* if for each $f(\overline{\overline{x}.t}) \Rightarrow r \in \mathbf{A}$, every metavariable occurring in r is accessible in some of $\overline{t}$.

The General Schema [Bla00], [Bla16] is a termination criterion of higher-order rewrite rules.

The main theorem is proved by using a variant of higher-order semantic labelling [Ham07] and the General Schema.

Theorem 2.3. (Modular Termination) Let $(\Sigma_{\mathbf{A}} \uplus \Theta, \mathbf{A})$ and $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta, \mathbf{B})$ be computation systems satisfying Assumption 2.1 and the following.

(i) $\mathbf{A}$ is accessible.

(ii) $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta \uplus \Sigma_{\mathsf{Proj}}, \mathbf{A}_{\mathsf{Proj}})$ is SN (not necessarily by GS), where $\mathbf{A}_{\mathsf{Proj}}$ is an extension of $\mathbf{A}$ with projections of pairs $\langle M_1, M_2 \rangle \Rightarrow M_i$ $(i = 1, 2)$.

(iii) $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta, \mathbf{B})$ is SN by the General Schema.

Then $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta,\ \mathbf{A} \uplus \mathbf{B})$ is SN.

3 Application: Algebraic Effect Handler with Theory

As an application, we demonstrate that our theorem is useful to prove the termination of a calculus with algebraic effects. The background of this section is as follows. Plotkin and Power introduced the *algebraic theory of effects* to axiomatise various computational effects [PP02] and they correspond to computational monads of Moggi [Mog88]. Plotkin and Pretnar formulated *algebraic effects and handlers* [PP13], which provided an alternative to monads as a basis for effectful programming across a variety of functional programming languages.

First, we formulate the *multi-adjunctive metalanguage* (MAM) [FKLP19] for an effectful λ-calculus, which is an extension of Levy's call-by-push-value (CBPV) calculus [Lev06]. Secondly, we provide an effect theory. Thirdly, we give an effect handler. We formulate MAM as a second-order computation system $(\Sigma_{\mathrm{MAM}} \uplus \Theta, \mathtt{MAM})$. The signature Σ_{MAM} consists of the following defined function symbols

```
bang  : U(c) → c
caseP : Pair(a1,a2),(a1,a2 → c) → c
case  : Sum(a1,a2),(a1 → c),(a2 → c) → c
let : F(a),(a → c) → c
app : Arr(a,c),a → c
prj1 : CPair(c1,c2) → c1     ; prj2 : CPair(c1,c2) → c2
```

and the set of constructors Θ consists of

```
unit  : Unit             ; pair : a1,a2 → Pair(a1,a2)
inj1  : a1 → Sum(a1,a2) ; inj2 : a2 → Sum(a1,a2)
cpair : c1,c2 → CPair(c1,c2)
thunk : c → U(c)
```

```
return : a → F(a)
lam : (a → b) → Arr(a,b)
```

The set MAM of MAM's computation rules is given by[1]

```
(beta)  lam(x.M[x])@V                    ⇒ M[V]
(u)     bang(thunk(M))                   ⇒ M
(prod1) prj1(cpair(M1,M2))               ⇒ M1
(prod2) prj2(cpair(M1,M2))               ⇒ M2
(caseP) caseP(pair(V1,V2),x1.x2.M[x1,x2]) ⇒ M[V1,V2]
(case1) case(inj1(V),x.M1[x],y.M2[y])    ⇒ M1[V]
(case2) case(inj2(V),x.M1[x],y.M2[y])    ⇒ M2[V]
(f)     let(return(V),x.M[x])            ⇒ M[V]
```

Using this formulation, SN of MAM is immediate because it satisfies GS.

We define the signature Σ_{Gl} by

```
get : (N → F(N)) → F(N)
put : N,F(N) → F(N)
```

It consists of the operations get(v.t) (looking-up the state, binding the value to v, and continuing t) and put(v,t) (updating the state to v and continuing t). The theory of global state [PP02, FS14] can be stated as a computation system $(\Sigma_{\mathrm{Gl}} \uplus \{\mathtt{return}\}, \mathtt{gstate})$ defined by

```
(lu)   get(v.put(v,X))        ⇒ X
(ll)   get(w.get(v.X[v,w]))   ⇒ get(v.X[v,v])
(uu)   put(V,put(W,X))        ⇒ put(W,X)
(ul)   put(V,get(w.X[w]))     ⇒ put(V,X[V])
```

These axioms have intuitive reading. For example, the axiom (lu) says that looking-up the state, binding the value to v, then updating the state to v, is equivalent to doing nothing. The axiom (ul) says that updating the state to V, then looking-up and continuing X with the looked-up value, is equivalent to updating the state to V and continuing X with V.

Plotkin and Power showed that the monad corresponding to the theory of global state (of finitely many locations) is the state monad [PP02]. Although $(\Sigma_{\mathrm{Gl}} \uplus \{\mathtt{return}\}, \mathtt{gstate})$ does not satisfy GS, it can be shown to be SN by interpretation [HAK20].

An effect handler provides an implementation of effects by interpreting algebraic effects as actual effects. The handler [KLO13, FKLP19] for effect terms for global states can be formulated as a computation system $(\Sigma_{\mathrm{Gl}} \uplus \Sigma_{\mathrm{MAM}} \uplus \Sigma_{\mathrm{Handle}}, \mathtt{Handle})$ as follows.

```
handler : (N → F(N)), (Arr(N,F(N)) → F(N)), (N,F(N) → F(N)), F(N) → F(N)

(h_r) handler(RET,GET,PUT,return(X))  ⇒ RET[X]
(h_g) handler(RET,GET,PUT,get(x.M[x]))⇒ GET[  lam(x.handler(RET,GET,PUT,M[x]))]
(h_p) handler(RET,GET,PUT,put(P,M))   ⇒ PUT[P,lam(x.handler(RET,GET,PUT,M))]
```

Now, we consider the main problem of this section: SN of the whole computation system

$$(\Sigma_{\mathrm{Gl}} \uplus \Sigma_{\mathrm{MAM}} \uplus \Sigma_{\mathrm{Handle}} \uplus \Theta,\ \mathtt{gstate} \uplus \mathtt{MAM} \uplus \mathtt{Handle}) \tag{1}$$

[1] t@s is the abbreviation of app(t,s).

The General Schema does not work to show SN of it because `gstate` does not satisfy GS. Therefore, we divide it into

$$(\Sigma_{\mathbf{A}} \uplus \Theta, \mathbf{A}) = (\Sigma_{\mathrm{Gl}} \uplus \Theta,\ \mathtt{gstate}),$$
$$(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta, \mathbf{B}) = (\Sigma_{\mathrm{Gl}} \uplus (\Sigma_{\mathrm{MAM}} \uplus \Sigma_{\mathrm{Handle}}) \uplus \Theta,\ \mathtt{MAM} \uplus \mathtt{Handle}).$$

The computation system (1) is not a disjoint union of $\mathbf{A}$ and $\mathbf{B}$, and is actually a *hierarchical combination* that *shares constructors* Θ. The lhss of `Handle` ($\subseteq \mathbf{B}$) involve defined function symbols `get`,`put` in $\Sigma_{\mathbf{A}}$.

We apply the modularity Thm. 2.3. Assumption 2.1 is satisfied because the computation system (1) is finitely branching and satisfies the the $\Sigma_{\mathbf{A}}$-layer condition. We check the assumptions. We define the well-founded order on types by

$$T(a_1, \ldots, a, \ldots, a_n) >_{\mathcal{T}} a$$

for every n-ary type constructor T, and every type a, where the rhs's a is placed at the i-th argument for every $i = 1, \ldots, n$.

(i) $\mathbf{A} = \mathtt{gstate}$ is accessible. This is immediate because of the type comparison for the arguments of `get`,`put`,`return`, which holds by $\mathtt{N} <_{\mathcal{T}} \mathtt{F(N)}$.

(ii) $(\Sigma_{\mathbf{A}} \uplus \Theta \uplus \Sigma_{\mathsf{Proj}}, \mathtt{gstate} \uplus \mathsf{Proj})$ is SN. This is proved by well-founded interpretation.

(iii) $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta,\ \mathtt{MAM} \uplus \mathtt{Handle})$ is SN by GS. This is immediate by applying GS with the precedence

$$\mathtt{handler} >_{\Sigma} \mathtt{lam}$$

to the computation system. The rhss of `MAM` involve no function symbols and the rhss of `Handle` involve `handler`,`lam`. To check that each recursive call of `handler` happens with a smaller argument, here we use the structural subterm ordering [Bla16, Def.13] to establish that `M[x]` is smaller than `get(x.M[x])`. Every metavariable is accessible.

Hence we conclude that the computation system (1) is SN.

References

[Bla00] F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Rewriting Techniques and Application (RTA 2000)*, LNCS 1833, pages 47–61. Springer, 2000.

[Bla16] F. Blanqui. Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. *Theor. Comput. Sci.*, 611:50–86, 2016.

[FH10] M. Fiore and C.-K. Hur. Second-order equational logic. In *Proc. of CSL'10*, LNCS 6247, pages 320–335, 2010.

[FKLP19] Y. Forster, O. Kammar, S. Lindley, and M. Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.*, 29:e15, 2019.

[FS14] M. Fiore and S. Staton. Substitution, jumps, and algebraic effects. In *Proc of CSL-LICS'14*, pages 41:1–41:10, 2014.

[HAK20] M. Hamana, T. Abe, and K. Kikuchi. Polymorphic computation systems: Theory and practice of confluence with call-by-value. *Science of Computer Programming*, 187(102322), 2020.

[Ham07] M. Hamana. Higher-order semantic labelling for inductive datatype systems. In *Proc. of PPDP'07*, pages 97–108. ACM Press, 2007.

[Ham19] M. Hamana. How to prove decidability of equational theories with second-order computation analyser SOL. *Journal of Functional Programming*, 29(e20), 2019.

[KLO13] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158. ACM, 2013.

[Lev06] P. B. Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *High. Order Symb. Comput.*, 19(4):377–414, 2006.

[Mog88] E. Moggi. Computational lambda-calculus and monads. LFCS ECS-LFCS-88-66, University of Edinburgh, 1988.

[PP02] G. Plotkin and J. Power. Notions of computation determine monads. In *FoSSaCS'02*, pages 342–356, 2002.

[PP13] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[Toy87] Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Inf. Process. Lett.*, 25(3):141–143, 1987.

The algebraic λ-calculus is a conservative extension of the ordinary λ-calculus

Axel Kerinec[1] and Lionel Vaux Auclair[2*]

[1] Université Sorbonne Paris Nord, LIPN, CNRS UMR 7030, France
[2] Aix-Marseille Université, CNRS, I2M, France

Abstract

The algebraic λ-calculus is an extension of the ordinary λ-calculus with linear combinations of terms. We establish that two ordinary λ-terms are equivalent in the algebraic λ-calculus iff they are β-equal. Although this result was originally stated in the early 2000's (in the setting of Ehrhard and Regnier's differential λ-calculus), the previously proposed proofs were wrong: we explain why previous approaches failed and develop a new proof technique to establish conservativity.

1 Introduction

The algebraic λ-calculus was introduced by the second author [Vau07; Vau09] as a generic framework to study the rewriting theory of the λ-calculus in presence of weighted superpositions of terms. The latter feature is pervasive in the quantitative semantics of λ-calculus and linear logic, that have flourished in the past twenty years [Ehr05; Lai+13; DE11; Cas+18, *etc.*] and the algebraic λ-calculus is meant as a unifying syntactic counterpart of that body of works.

The algebraic λ-calculus was actually obtained by removing the differentiation primitives from Ehrhard and Regnier's differential λ-calculus [ER03], keeping only the dynamics associated with linear combinations of terms. This dynamics is surprisingly subtle in itself: for instance, if 1 has an opposite in the semiring R of coefficients, then the rewriting theory becomes trivial. We refer the reader to the original paper [Vau09] for a thorough discussion, and focus on the question of conservativity only, assuming R is **positive** — *i.e.* if $a + b = 0$ then $a = b = 0$. We briefly outline the main definitions, keeping the same notations as in the former paper, so that the reader can consistently refer to it for a more detailed account if need be.

Overview of the algebraic λ-calculus. The syntax of algebraic λ-terms is constructed in two stages. We first consider **raw terms**, which are terms inductively generated as follows:

$$\mathsf{L_R} \ni M, N, \ldots ::= x \mid \lambda x.M \mid (M)\, N \mid \mathbf{0} \mid M + N \mid a.M$$

where a ranges over the semiring R (beware that we use Krivine's convention for application). We consider raw terms up to α-equivalence: $\mathsf{L_R}$ contains the set Λ of pure λ-terms as a strict subset. We then consider **algebraic equality** $\triangleq$ on raw terms, which is the congruence generated by the equations of R-module, plus the following linearity axioms:

$$\begin{array}{lll} \lambda x.\mathbf{0} \triangleq \mathbf{0} & \lambda x.(M+N) \triangleq \lambda x.M + \lambda x.N & \lambda x.(a.M) \triangleq a.\lambda x.M \\ (\mathbf{0})\, P \triangleq \mathbf{0} & (M+N)\, P \triangleq (M)\, P + (N)\, P & (a.M)\, P \triangleq a.(M)\, P \end{array}$$

*This work was partially supported by the French ANR project PPS (ANR-19-CE48-0014).

which reflects the point-wise definition of the sum of functions. Note that, without these equations, a term such as $(\lambda x.M + \lambda x.N)\,P$ has no redex.

The terms of the algebraic λ-calculus, called **algebraic terms** below, are then the $\triangleq$-classes $\sigma = \underline{M}$ of raw terms $M \in \mathsf{L}_\mathsf{R}$. We extend syntactic constructs to algebraic terms (e.g., $\lambda x.\underline{M} = \underline{\lambda x.M}$, which is well defined because $\triangleq$ is a congruence). Among algebraic terms, we distinguish the **simple terms**, which are intuitively those without sums at top level: a term σ is simple if $\sigma = \underline{x}$ for some variable x, or $\sigma = \lambda x.\tau$ or $\sigma = (\tau)\,\rho$ where τ is itself simple (inductively). In particular, $\underline{M}$ is simple as soon as $M \in \Lambda$. By definition, algebraic terms form an R-module, and it is easy to check that it is freely generated by the set Δ_R of simple terms: we write $\mathsf{R}\langle\Delta_\mathsf{R}\rangle$ for the module of algebraic terms.

A seemingly natural way to extend the β-reduction $\to_\Lambda$ of λ-terms to algebraic terms is to define it contextually on raw terms, and then apply it *modulo* $\triangleq$: among other issues with this naïve definition, note that $M \triangleq M + 0.N$ would reduce to $M + 0.N' \triangleq M$ for any $N \to_\Lambda N'$, so that the obtained reduction would be reflexive and there would be no clear notion of normal form. Ehrhard and Regnier's solution is to rather consider two relations: $\to \subset \Delta_\mathsf{R} \times \mathsf{R}\langle\Delta_\mathsf{R}\rangle$ defined contextually on simple terms with β-reduction as a base case; and $\widetilde{\to} \subset \mathsf{R}\langle\Delta_\mathsf{R}\rangle \times \mathsf{R}\langle\Delta_\mathsf{R}\rangle$ on algebraic terms, obtained by setting $\sigma \mathrel{\widetilde{\to}} \sigma'$ iff $\sigma = a.\tau + \rho$ and $\sigma' = a.\tau' + \rho$ with $\tau \to \tau'$ and $a \neq 0$. Then $\widetilde{\to}$ is confluent [ER03; Vau09] and, *provided* R *is positive*, an algebraic term is in normal form iff it is the class of a raw term without β-redex.

Note that, for any fixed point combinator Y, by setting $\infty_\sigma = (\underline{Y})\,\lambda x.(\sigma + \underline{x})$, we obtain $\infty_\sigma \leftrightarrow \sigma + \infty_\sigma$ where $\leftrightarrow$ is the equivalence on $\mathsf{R}\langle\Delta_\mathsf{R}\rangle$ generated by the reduction relation $\widetilde{\to}$. In case $1 \in \mathsf{R}$ has an opposite -1, we can now exhibit the above-mentioned inconsistency of the theory: $\underline{\mathbf{0}} = \infty_\sigma + (-1).\infty_\sigma \leftrightarrow \sigma$ for any σ. From now on, we thus assume that R is positive.

Contributions. Our goal is to establish that, for any two λ-terms M and $N \in \Lambda$, we have $\underline{M} \leftrightarrow \underline{N}$ iff $M \leftrightarrow_\Lambda N$, where $\leftrightarrow_\Lambda$ is the usual β-equivalence on λ-terms. For that purpose, it is sufficient to establish a conservativity result on reduction relations rather than on the induced equivalences: if $\underline{M} \mathrel{\widetilde{\to}}^* \underline{N}$ then $M \to_\Lambda^* N$. This is our main result, theorem 4.3 below.

In the next section, we explain what was wrong with the previous two attempts, first by Ehrhard and Regnier, then by the second author, to establish conservativity, and we outline the new proof strategy we propose. The rest of the paper is dedicated to the proof of theorem 4.3.[1]

2 Two non-proofs and a new approach

Recall that an **ARS** (abstract rewriting system) is a pair $(A, \rightsquigarrow)$ of a set A and binary relation $\rightsquigarrow \subseteq A \times A$. An **extension** of $(A, \rightsquigarrow)$ is another ARS $(A', \rightsquigarrow')$ such that $A \subseteq A'$ and $\rightsquigarrow \subseteq \rightsquigarrow'$. This extension is **conservative** if, for every $a_1, a_2 \in A$, $a_1 \rightsquigarrow a_2$ iff $a_1 \rightsquigarrow' a_2$. An **equational system** is an ARS $(A, \sim)$ such that $\sim$ is an equivalence relation. Our goal is thus to establish that the equational system $(\mathsf{R}\langle\Delta_\mathsf{R}\rangle, \leftrightarrow)$ is a conservative extension of $(\Lambda, \leftrightarrow_\Lambda)$ — here we consider the injection $M \in \Lambda \mapsto \underline{M} \in \mathsf{R}\langle\Delta_\mathsf{R}\rangle$ as an inclusion.

In their paper on the differential λ-calculus [ER03], Ehrhard and Regnier claim that this follows directly from the confluence of $\widetilde{\to}$, but this argument is not valid: $\widetilde{\to}$ does contain $\to_\Lambda$, and it is indeed confluent, without any positivity assumption; but we have already stated that $\leftrightarrow$ is inconsistent in presence of negative coefficients, so this observation cannot be sufficient.

[1] These results were obtained during a research internship of the first author, in the first half of 2019; they were presented by the second author at the annual meeting of the working group Scalp (*Structures formelles pour le Calcul et les Preuves*) in Lyon in October 2019. This collaboration was unfortunately disrupted by the COVID-19 pandemic in the following year, which delayed dissemination to a wider audience.

$$\frac{M \to_\Lambda^* x}{M \vdash \underline{x}}\ (\mathrm{v}) \qquad \frac{M \to_\Lambda^* \lambda x.N \qquad N \vdash \tau}{M \vdash \lambda x.\tau}\ (\lambda) \qquad \frac{M \to_\Lambda^* (N)\,P \qquad N \vdash \tau \qquad P \Vdash \rho}{M \vdash (\tau)\,\rho}\ (\mathrm{a})$$

$$\frac{}{M \Vdash \mathbf{0}}\ (\mathbf{0}) \qquad \frac{M \vdash \sigma \qquad M \Vdash \tau}{M \Vdash a.\sigma + \tau}\ (+)$$

Figure 1: Inference rules for the mashup relations

Ehrhard and Regnier's mistake is certainly an erroneous application of a general conservativity result in Terese's textbook [Ter03], missing the fact that Terese's notion of extension is more demanding: for Terese, $(A, \rightsquigarrow)$ is a **sub-ARS** of $(A', \rightsquigarrow')$ if $A \subseteq A'$ and, for every $a \in A$ and $a' \in A'$, $a \rightsquigarrow' a'$ iff $a' \in A$ and $a \rightsquigarrow a'$. The latter is strictly stronger than the mere inclusion $\rightsquigarrow \subseteq \rightsquigarrow'$, and is indeed sufficient to deduce conservativity for the induced equational systems from the confluence of the super-ARS [Ter03, Exercice 1.3.21 (iii)]. But $(\Lambda, \to_\Lambda)$ is not a sub-ARS of $(\mathsf{R}\langle\Delta_\mathsf{R}\rangle, \widetilde{\to})$, even when R is positive: for instance, if $\mathsf{R} = \mathbf{Q}^+$ and $M \to_\Lambda M' \neq M$, we have $\underline{M} = \frac{1}{2}\underline{M} + \frac{1}{2}\underline{M} \mathrel{\widetilde{\to}} \frac{1}{2}\underline{M} + \frac{1}{2}\underline{M'} \notin \underline{\Lambda}$. So one must design another approach.

Given $\sigma \in \mathsf{R}\langle\Delta_\mathsf{R}\rangle$, one can consider the finite set of λ-terms $\Lambda(\sigma) \subset \Lambda$ obtained by keeping exactly one element in the support of each sum occurring in σ [Vau09, Definition 3.18]. The second author tried to establish the conservativity of $\widetilde{\to}^*$ over $\to_\Lambda^*$ by iterating the following:

Claim 2.1 ([Vau09, Lemma 3.20])**.** *If $\sigma \mathrel{\widetilde{\to}} \sigma'$ and $M' \in \Lambda(\sigma')$ then there exists $M \in \Lambda(\sigma)$ such that $M \to_\Lambda^* M'$.*

But the latter claim is wrong! Consider, for instance, $\sigma = (\lambda x.(\underline{x})\,\underline{x})\,(\underline{y} + \underline{z}) \mathrel{\widetilde{\to}} \sigma' = (\underline{y} + \underline{z})\,(\underline{y} + \underline{z})$. We have $M' = (y)\,z \in \Lambda(\sigma')$ but no term in $\Lambda(\sigma') = \{(\lambda x.(x)\,x)\,y, (\lambda x.(x)\,x)\,z\}$ β-reduces to M'. Note that, in this counter-example, there is no $M \in \Lambda$ such that $\underline{M} \mathrel{\widetilde{\to}}^* \sigma$: somehow, we must exploit this additional hypothesis to establish a correct version of claim 2.1.

Reasoning on $\widetilde{\to}^*$ directly is difficult, due to its definition as a reflexive and transitive closure. The technique we propose involves the definition of a mixed-type relation $M \Vdash \sigma$ between a λ-term M and a term $\sigma \in \mathsf{R}\langle\Delta_\mathsf{R}\rangle$: intuitively, $M \Vdash \sigma$ when σ is obtained by pasting together terms issued from various reductions of M, and we say $M \Vdash \sigma$ is a **mashup** of such reductions. In particular: $M \Vdash \underline{M'}$ as soon as $M \to_\Lambda^* M'$; and $M \Vdash \sigma + \tau$ as soon as $M \Vdash \sigma$ and $M \Vdash \tau$. We then show that $\Vdash$ is conservative over $\to_\Lambda^*$ (lemma 3.4) and that $M \Vdash \sigma$ as soon as $\underline{M} \mathrel{\widetilde{\to}}^* \sigma$ (lemmas 3.3 and 4.2): this ensures the conservativity of $\widetilde{\to}^*$ over $\to_\Lambda^*$ (theorem 4.3). Our whole approach thus rests on the careful definition of the mashup relation. Among other requirements, it must behave well w.r.t. the structure of terms: e.g., if $M \Vdash \sigma$ then $\lambda x.M \Vdash \lambda x.\sigma$.

3 Mashup of β-reductions

We define two relations $\vdash \subseteq \Lambda \times \Delta_\mathsf{R}$ and $\Vdash \subseteq \Lambda \times \mathsf{R}\langle\Delta_\mathsf{R}\rangle$ by mutual induction, with the rules of fig. 1. If $\sigma \in \mathsf{R}\langle\Delta_\mathsf{R}\rangle$, we write $\mathsf{Supp}(\sigma) \subset \Delta_\mathsf{R}$ for its support set.

Lemma 3.1. *We have $M \Vdash \sigma$ iff, for every $\sigma' \in \mathsf{Supp}(\sigma)$, $M \vdash \sigma'$.*

Proof. The forward implication is done by induction on the derivation of $M \Vdash \sigma$, noting that if $\sigma' \in \mathsf{Supp}(a\tau + \rho)$ with $M \vdash \tau$ and $M \Vdash \rho$ then $\sigma' = \tau$ or $\sigma' \in \mathsf{Supp}(\rho)$. For the reverse implication, we can write $\sigma = \sum_{i=1}^n a_i.\sigma_i$ with $\sigma_i \in \mathsf{Supp}(\sigma)$ for $1 \leq i \leq n$, and obtain a derivation of $M \Vdash \sigma$ by induction on n. □

$$\frac{M \vdash \sigma}{M \Vdash \sigma}\ (\mathrm{s})$$

$$\frac{M \to_\Lambda^* \lambda x.N \qquad N \Vdash \tau}{M \Vdash \lambda x.\tau}\ (\lambda\text{'}) \qquad \frac{M \to_\Lambda^* (N)\,P \qquad N \Vdash \tau \qquad P \Vdash \rho}{M \Vdash (\tau)\,\rho}\ (\mathrm{a}\text{'}) \qquad \frac{M \Vdash \sigma \qquad M \Vdash \tau}{M \Vdash a.\sigma + \tau}\ (+\text{'})$$

Figure 2: Admissible rules for the mashup relations

Lemma 3.2. *The rules of fig. 2 are admissible.*

Proof. For (s), it is sufficient to observe that $\sigma = 1\sigma + \mathbf{0}$. For the other three rules, we reason on the support sets, using lemma 3.1. □

Lemma 3.3 (Reflexivity of $\vdash$)**.** *For every $M \in \Lambda$, $M \vdash \underline{M}$.*

Proof. By a straightforward induction on M, using the reflexivity of $\to_\Lambda^*$ and rule (s). □

Lemma 3.4 (Conservativity of $\Vdash$)**.** *If $M \Vdash \underline{M'}$ then $M \to_\Lambda^* M'$.*

Proof. Note that $\underline{M'} \in \Delta_{\mathsf{R}}$, hence $M \vdash \underline{M'}$ by lemma 3.1. The proof is then by induction on M', inspecting the last rule of the derivation of $M \vdash \underline{M'}$:

(v) If $M \to_\Lambda^* x$ and $\underline{M'} = \underline{x}$ then we conclude directly since $M' = x$.

(λ) If $M \to_\Lambda^* \lambda x.N$ and $N \vdash \tau$ with $\underline{M'} = \lambda x.\tau$, then $M' = \lambda x.N'$ with $\tau = \underline{N'}$. By induction hypothesis, $N \to_\Lambda^* N'$, hence $M \to_\Lambda^* M'$.

(a) If $M \to_\Lambda^* (N)\,P$, $N \vdash \tau$ and $P \Vdash \rho$ with $\underline{M'} = (\tau)\,\rho$, then $M' = (N')\,P'$ with $\tau = \underline{N'}$ and $\rho = \underline{P'}$. In particular $\rho \in \Delta_{\mathsf{R}}$, hence $P \vdash \rho$ by lemma 3.1. By induction hypothesis, $N \to_\Lambda^* N'$ and $P \to_\Lambda^* P'$, hence $M \to_\Lambda^* M'$. □

Lemma 3.5 (Compatibility with $\to_\Lambda$)**.** *If $M \to_\Lambda^* M' \vdash \sigma$ then $M \vdash \sigma$. Similarly, if $M \to_\Lambda^* M' \Vdash \sigma$ then $M \Vdash \sigma$.*

Proof. For the first implication, it is sufficient to inspect the last rule of the derivation $M' \vdash \sigma$, and use the transitivity of $\to_\Lambda^*$. The second implication follows directly by induction on the derivation of $M' \Vdash \sigma$. □

4 Conservativity of algebraic reduction

Lemma 4.1 (Substitution lemma)**.** *If $M \Vdash \sigma$ and $P \Vdash \rho$ then $M[P/x] \Vdash \sigma[\rho/x]$.*

Proof. We prove the result, together with the variant assuming $M \vdash \sigma$ instead of $M \Vdash \sigma$, by induction on the derivations of those judgements.

(v) If $M \to_\Lambda^* y$ and $\sigma = \underline{y}$ then:

- if $y = x$, then $M[P/x] \to_\Lambda^* x[P/x] = P \Vdash \rho$ and we obtain $M[P/x] \Vdash \rho = \sigma[\rho/x]$ by lemma 3.5;
- otherwise, $M[P/x] \to_\Lambda^* y[P/x] = y$, hence $M[P/x] \Vdash \underline{y} = \sigma[\rho/x]$ by (v).

(λ) If $M \to_\Lambda^* \lambda y.N$ and $N \vdash \tau$ with $\sigma = \lambda y.\tau$ (choosing $y \neq x$ and y not free in P nor in ρ), then $M[P/x] \to_\Lambda^* \lambda y.N[P/x]$ and, by induction hypothesis $N[P/x] \Vdash \tau[\rho/x]$: we obtain $M[P/x] \Vdash \lambda y.\tau[\rho/x] = \sigma[\rho/x]$ by (λ').

(a) If $M \to_\Lambda^* (N_1)\, N_2$, $N_1 \vdash \tau_1$ and $N_2 \Vdash \tau_2$, with $\sigma = (\tau_1)\, \tau_2$, then we have $M[P/x] \to_\Lambda^* (N_1[P/x])\, N_2[P/x]$ and, by induction hypothesis, $N_1[P/x] \Vdash \tau_1[\rho/x]$ and $N_2[P/x] \Vdash \tau_2[\rho/x]$: we obtain $M[P/x] \Vdash (\tau_1[\rho/x])\, \tau_2[\rho/x] = \sigma[P/x]$ by (a').

(**0**) If $\sigma = \mathbf{0}$ then $\sigma[\rho/x] = \mathbf{0}$ and we conclude directly, by (**0**).

(+) If $\sigma = a.\tau_1 + \tau_2$ with $M \vdash \tau_1$ and $M \Vdash \tau_2$, then, by induction hypothesis, $M[P/x] \Vdash \tau_1[\rho/x]$ and $M[P/x] \Vdash \tau_2[\rho/x]$, hence $M[P/x] \Vdash a.\tau_1[\rho/x] + \tau_2[\rho/x] = \sigma[\rho/x]$ by (a'). □

Note that, by positivity, if $\sigma = a.\tau + \rho$ with $\tau \in \Delta_{\mathsf{R}}$ and $a \neq 0$, then $\tau \in \mathsf{Supp}(\sigma) \supseteq \mathsf{Supp}(\rho)$.

Lemma 4.2 (Compatibility with $\widetilde{\to}$)**.** *Let $M \in \Lambda$ and $\sigma' \in \mathsf{R}\langle\Delta_{\mathsf{R}}\rangle$. For every $\sigma \in \Delta_{\mathsf{R}}$ such that $M \vdash \sigma \to \sigma'$ (resp. every $\sigma \in \mathsf{R}\langle\Delta_{\mathsf{R}}\rangle$ such that $M \Vdash \sigma \mathrel{\widetilde{\to}} \sigma'$), we have $M \Vdash \sigma'$.*

Proof. The proof is by induction on the definition of the reduction $\sigma \to \sigma'$ or $\sigma \mathrel{\widetilde{\to}} \sigma'$.

- If $\sigma = (\lambda x.\tau)\, \rho \in \Delta_{\mathsf{R}}$ and $\sigma' = \tau[\rho/x]$, then the derivation of $M \vdash \sigma$ must be of the form
$$\dfrac{M \to_\Lambda^* (N)\, P \qquad \dfrac{N \to_\Lambda^* \lambda x.N' \qquad N' \vdash \tau}{N \vdash \lambda x.\tau}\ (\lambda) \qquad P \Vdash \rho}{M \vdash (\lambda x.\tau)\, \rho}\ (\mathrm{a})\ .$$
By lemma 4.1, we have $N'[P/x] \Vdash \sigma'$. Moreover, $M \to_\Lambda^* (N)\, P \to_\Lambda^* (\lambda x.N')\, P \to_\Lambda N'[P/x]$ and we obtain $M \Vdash \sigma'$ by lemma 3.5.

- If $\sigma = \lambda x.\tau$ and $\sigma' = \lambda x.\tau'$ with $\tau \to \tau'$, then the derivation of $M \vdash \sigma$ must be of the form
$$\dfrac{M \to_\Lambda^* \lambda x.N \qquad N \vdash \tau}{M \vdash \lambda x.\tau}\ (\lambda)\ .$$
We obtain $N \Vdash \tau'$ by induction hypothesis, and we conclude by (λ').

- If $\sigma = (\tau)\, \rho$ and $\sigma' = (\tau')\, \rho'$ with either $\tau \to \tau'$ and $\rho = \rho'$, or $\tau = \tau'$ and $\rho \mathrel{\widetilde{\to}} \rho'$, then the derivation of $M \vdash \sigma$ must be of the form
$$\dfrac{M \to_\Lambda^* (N)\, P \qquad N \vdash \tau \qquad P \Vdash \rho}{M \vdash (\tau)\, \rho}\ (\mathrm{a})\ .$$
We obtain $N \Vdash \tau'$ and $P \Vdash \rho'$ by induction hypothesis, and we conclude by (a').

- If $\sigma = a.\tau + \rho$ and $\sigma' = a.\tau' + \rho$ with $\tau \to \tau'$ and $a \neq 0$, then we have already observed that $\tau \in \mathsf{Supp}(\sigma)$ and $\mathsf{Supp}(\rho) \subseteq \mathsf{Supp}(\sigma)$. Since $M \Vdash \sigma$, we obtain $M \vdash \tau$ and $M \Vdash \rho$ by lemma 3.1. The induction hypothesis gives $M \Vdash \tau'$, hence $M \Vdash a.\tau' + \rho = \sigma'$ by (+'). □

Theorem 4.3 (Conservativity of $\widetilde{\to}^*$)**.** *If $\underline{M} \mathrel{\widetilde{\to}}^* \underline{N}$ then $M \to_\Lambda^* N$.*

Proof. Assume $\underline{M} \mathrel{\widetilde{\to}}^* \underline{N}$. By lemma 3.3 and (s), we have $M \Vdash \underline{M}$. By iterating lemma 4.2, we deduce $M \Vdash \underline{N}$. We conclude by lemma 3.4. □

Corollary 4.4 (Conservativity of $\leftrightarrow$)**.** *If R is positive then $\underline{M} \leftrightarrow \underline{N}$ iff $M \leftrightarrow_\Lambda N$.*

Proof. Assuming $\underline{M} \leftrightarrow \underline{N}$, the confluence of $\widetilde{\to}$ ensures that there exist $\sigma \in \mathsf{R}\langle\Delta_{\mathsf{R}}\rangle$ and $k \in \mathbf{N}$, such that $\underline{M} \mathrel{\widetilde{\to}}^k \sigma$ and $\underline{N} \mathrel{\widetilde{\to}}^* \sigma$. It follows [Vau09, Lemma 3.23] that $\sigma \mathrel{\widetilde{\to}}^* \underline{M\downarrow^k}$ where $\tau\downarrow$ is the term obtained by reducing all the redexes of τ simultaneously. Observing that $\underline{M\downarrow} = \underline{M\downarrow}$, we obtain $\underline{N} \mathrel{\widetilde{\to}}^* \underline{M\downarrow^k}$ hence $N \to_\Lambda^* M\downarrow^k$ by theorem 4.3, which concludes the proof. □

References

[Cas+18] Simon Castellan, Pierre Clairambault, Hugo Paquet, and Glynn Winskel. "The concurrent game semantics of Probabilistic PCF". In: *LICS 2018*. ACM Press, July 2018. DOI: 10.1145/3209108.3209187.

[DE11] Vincent Danos and Thomas Ehrhard. "Probabilistic coherence spaces as a model of higher-order probabilistic computation". In: *Information and Computation* 209.6 (2011), pp. 966–991. DOI: 10.1016/j.ic.2011.02.001.

[Ehr05] Thomas Ehrhard. "Finiteness spaces". In: *Mathematical Structures in Computer Science* 15.4 (2005), pp. 615–646. DOI: 10.1017/S0960129504004645.

[ER03] Thomas Ehrhard and Laurent Regnier. "The differential lambda-calculus". In: *Theoretical Computer Science* 309.1-3 (2003). DOI: 10.1016/S0304-3975(03)00392-X.

[Lai+13] Jim Laird, Giulio Manzonetto, Guy McCusker, and Michele Pagani. "Weighted relational models of typed lambda-calculi". In: *LICS 2013*. IEEE Computer Society, 2013, pp. 301–310. DOI: 10.1109/LICS.2013.36.

[Ter03] Terese. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science 55. Cambridge University Press, 2003.

[Vau07] Lionel Vaux. "On linear combinations of λ-terms". In: *RTA 2007*. Vol. 4533. Lecture Notes in Computer Science. Springer, 2007. ISBN: 978-3-540-73447-5. DOI: 10.1007/978-3-540-73449-9_28.

[Vau09] Lionel Vaux. "The algebraic lambda calculus". In: *Mathematical Structures in Computer Science* 19.5 (2009), pp. 1029–1059. DOI: 10.1017/S0960129509990089.

Nijn/ONijn: A New Certification Engine for Higher-Order Termination*

Cynthia Kop, Deivid Vale, and Niels van der Weide

Institute for Computing and Information Sciences
Radboud University, Nijmegen, The Netherlands
{c.kop,deividvale,nweide}@cs.ru.nl

1 Introduction

In this short paper, we limn a new combination Nijn/ONijn for the certification of higher-order rewriting termination proofs. A complete version of this work has been accepted for publication at ITP 2023 [8]. We follow the following system design in Nijn/ONijn: Nijn [7] is the certifier, a Coq library providing a formalization of the underlying higher-order rewriting theory and ONijn [6] is a proof script generator, an application that when given a minimal description of a termination proof, i.e., *proof trace*, outputs a Coq *proof script*. The proof script is a fully formal description of the syntax signature used by the TRS and the specification of each rule in the system together with the formal steps needed to express its termination. The proof script then utilizes results from Nijn for checking the correctness of the traced proof. Examples of this system design are the combinations Cochinelle/CiME3 [2] and CoLoR/Rainbow [1].

The schematic below depicts the basic steps for producing proof certificates using Nijn/ONijn.

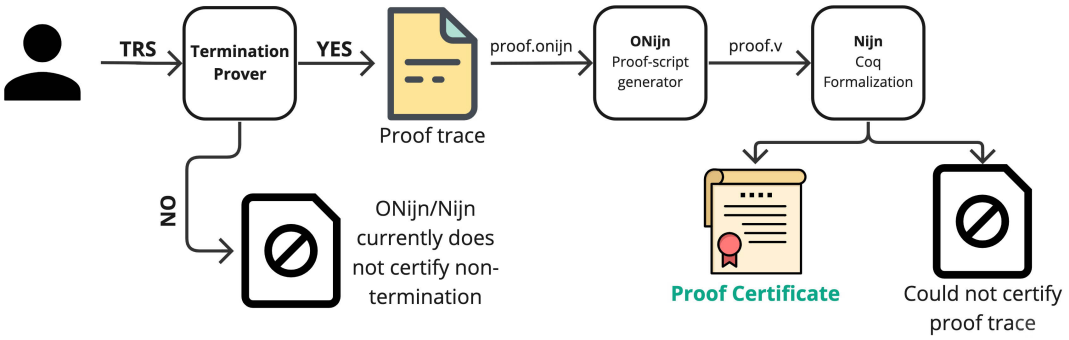

Figure 1: Nijn/ONijn schematics

A *termination prover* in this schematic is an abstract entity responsible for producing proof traces. It can be either a human, proving termination manually, or a termination tool like Wanda [4], which uses programmed techniques and automated reasoning tools such as SAT/SMT solvers. Whenever a prover outputs a proof trace, we can use ONijn to process it into a formal proof script in Coq. At this moment, we have formalized the polynomial interpretation method.

*This work is supported by the following NWO projects: "Implicit Complexity through Higher-Order Rewriting", NWO 612.001.803/7571; NWO VIDI project "Constrained Higher-Order Rewriting and Program Equivalence", NWO VI.Vidi.193.075; and "The Power of Equality" NWO OCENW.M20.380.

Notice that producing the certificates for only this proof method is an inherently incomplete task, since it would require a method to solve inequalities over arbitrary polynomials, which is undecidable in general.

While Nijn is the certified core part of our tool since it is checked by Coq, the proof script generation implemented in OCaml (ONijn) is not currently certified and must be trusted. For this reason, we deliberately keep ONijn as simple (small) as possible. The main task delegated to ONijn is that of parsing the proof trace given by the termination prover to a Coq proof script and perform sanitazation on the prover's input, so that syntax errors are avoided in the proof script. This approach does not pose significant drawbacks in our experience.

2 Encoding TRSs in Nijn

Let us encode $\mathcal{R}_{\mathsf{map}}$ in Coq using Nijn. This will be useful to demonstrate our choices in the formalization and show how to express rewriting systems directly in Coq. The file containing the full enconding can be found at `Map.v`. A simple example of a higher-order system is that of $\mathcal{R}_{\mathsf{map}}$. It represents the higher-order function that applies a function to each element of a list. Recall that $\mathcal{R}_{\mathsf{map}}$ is composed of two rules: $\mathsf{map}\,F\,\mathsf{nil} \to \mathsf{nil}$ and $\mathsf{map}\,F\,(\mathsf{cons}\,x\,xs) \to \mathsf{cons}\,(F\,x)\,(\mathsf{map}\,F\,xs)$. These rules are under a typing context where $F : \mathsf{nat} \Rightarrow \mathsf{nat}, x : \mathsf{nat}$, and $xs : \mathsf{list}$. We start by encoding base types.

```
Inductive base_types := TBtype | TList.
Definition Btype : ty base_types := Base TBtype.
Definition List : ty base_types := Base TList.
```

The abbreviations `Btype` and `List` is to smoothen the usage of the base types. There are three function symbols in this system:

```
Inductive fun_symbols := TNil | TCons | TMap.
```

The arity function `map_ar` maps each function symbol in `fun_symbols` to its type.

```
Definition map_ar f : ty base_types
  := match f with
     | TNil ⇒ List
     | TCons ⇒ Btype ⟶ List ⟶ List
     | TMap ⇒ (Btype ⟶ Btype) ⟶ List ⟶ List
     end.
```

So, `TNil` is a list and given an inhabitant of `Btype` and `List`, the function symbol `TCons` gives a `List`. Again we introduce some abbreviations to simplify the usage of the function symbols.

```
Definition Nil {C} : tm map_ar C _ := BaseTm TNil.
Definition Cons {C} x xs : tm map_ar C _ := BaseTm TCons · x · xs.
Definition Map {C} f xs : tm map_ar C _ := BaseTm TMap · f · xs.
```

The first rule, $\mathsf{map}\,F\,\mathsf{nil} \to \mathsf{nil}$, is encoded as the following Coq construct:

```
Program Definition map_nil :=
  make_rewrite
    (_ ,, •) _
    (let f := TmVar Vz in Map · f · Nil)
    Nil.
```

Notice that we only defined the *pattern* of the first two arguments of `make_rewrite`, leaving the types in the context (`_ ,, •`) and the type of the rule unspecified. Coq can fill in these holes

automatically, as long as we provide a context pattern of the correct length. In this particular rewrite rule, there is only one free variable. As such, the variable `TmVar Vz` refers to the only variable in the context. In addition, we use iterated `let`-statements to imitate variable names. For every position in the context, we introduce a variable in Coq, which we use in the left- and right-hand sides of the rule. This makes the rules more human-readable. Indeed, the lhs map F nil of this rule is represented as `Map · f · Nil` in code. The second rule for `map` is encoded following the same ideas.

```
Program Definition map_cons :=
  make_rewrite
    (_ ,, _ ,, _ ,, •) _
    (let f := TmVar Vz in let x := TmVar (Vs Vz) in let xs := TmVar (Vs (Vs Vz)) in
    Map · f · (Cons · x · xs))
    (let f := TmVar Vz in let x := TmVar (Vs Vz) in let xs := TmVar (Vs (Vs Vz)) in
    Cons · (f · x) · (Map · f · xs)).
```

3 Practical Aspects of Nijn/ONijn Certification

In this section, we discuss the practical aspects of our verification framework. In principle one can manually encode rewrite systems as Coq files and use the formalization we provide to verify their own termination proofs. However, this is cumbersome to do so. Indeed, in the last section we used abbreviations to make the formal description of $\mathcal{R}_{\mathsf{map}}$ more readable. A rewrite system with many more rules would be difficult to encode manually. Additionally, to formally establish termination we also need to encode proofs. The full formal encoding of $\mathcal{R}_{\mathsf{map}}$ and its termination proof is found in the file `Map.v`.

3.1 Proof traces for polynomial interpretation

This difficulty of manual encoding motivates the usage of proof traces. A proof trace is a human-friendly encoding of a TRS and the essential information needed to reconstruct the termination proof as a Coq script. Let us again consider $\mathcal{R}_{\mathsf{map}}$ as an example. The proof trace for this system starts with `YES` to signal that we have a termination proof for it. Then we have a list encoding the signature and the rules of the system.

```
YES
Signature: [
  cons : a -> list -> list ;
  map : list -> (a -> a) -> list ;
  nil : list
]
Rules: [
  map nil F => nil ;
  map (cons X Y) G => cons (G X) (map Y G)
]
```

Notice that the free variables in the rules do not need to be declared nor their typing information provided. Coq can infer this information automatically. The last section of the proof trace describes the interpretation of each function symbol in the signature.

```
Interpretation: [
  J(cons) = Lam[y0;y1].3 + 2*y1;
  J(map)  = Lam[y0;G1].3*y0 + 3*y0 * G1(y0);
  J(nil)  = 3
]
```

We can fully reconstruct a formal proof of termination for $\mathcal{R}_{\mathsf{map}}$, which uses the theory formalized in Nijn, with the information provided in the proof trace above. The full description of proof traces can be found in [6], the API for ONijn. Proof traces are not Coq files. So we need to further compile them into a proper Coq script. The schematics in fig. 1 describe the steps necessary for it. We use ONijn to compile proof traces to Coq script. It is invoked as follows:

```
onijn path/to/proof/trace.onijn -o path/to/proof/script.v
```

Here, the first argument is the file path to a proof trace file and the `-o` option requires the file path to the resulting Coq script. The resulting Coq script can be verified by Nijn as follows:

```
coqc path/to/proof/script.v
```

Instructions on how to locally install ONijn/Nijn can be found at [6].

3.2 Verifying Wanda's Polynomial Interpretations

It is worth noticing that the termination prover is abstract in our certification framework. This means that we are not bound to a specific termination tool. So we can verify any termination tool that implements the interpretation method described here and can output proof traces in ONijn format.

Since Wanda [4] is a termination tool that implements the interpretation method in [3], it is our first candidate for verification. We added to Wanda the runtime argument `--formal` so it can output proof traces in ONijn format. In [4] one can find details on how to invoke Wanda. For instance, we illustrate below how to run Wanda on the map AFS.

```
./wanda.exe -d rem --formal Mixed_HO_10_map.afs
```

The setting `-d rem` sets Wanda to disable rule removal. The option `--formal` sets Wanda to only use polynomial interpretations and output proofs to ONijn proof traces. Running Wanda with these options gives us the proof trace we used for $\mathcal{R}_{\mathsf{map}}$ above. The latest version of Wanda, which includes this parameter, is found at [5].

The table below describes our experimental evaluation on verifying Wanda's output with the settings above. The benchmark set consists of those 46 TRSs that Wanda outputs `YES` while using only polynomial interpretations and no rule removal. The time limit for certification of each system is set to 60 seconds.

The experiment was run in a machine with M1 Pro 2021 processor with 16GB of RAM. Memory usage of Nijn during certification ranges from 400MB to 750MB. We provide the experimental benchmarks at https://github.com/deividrvale/nijn-coq-script-generation.

	Wanda			Nijn/ONijn		
Technique	# YES	Pct.	Avg. Time	# Certified	Perc.	Avg. Time
Poly, no rule removal	46	23%	0.07s	46	100%	4.06s

Table 1: Experimental Results

Hence, we can certify all TRSs proven SN by Wanda using only polynomial interpretations.

4 Conclusion and Future Plans

In this formalization effort, we were successful in certifying higher-order polynomial interpretations. This line of work is far from finished, however. The initial setup of Nijn/ONijn presented here bootstraps the foundation of a full-fledged certification engine for more complex higher-order termination proof techniques. For instance, incorporating the so-called higher-order dependency pair framework is our next immediate future work plan. This will allow us to significantly improve the number of systems we can certify.

References

[1] Frédéric Blanqui and Adam Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Struct. Comput. Sci.*, 21(4):827–859, 2011. doi:10.1017/S0960129511000120.

[2] Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Automated certified proofs with cime3. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 of *LIPIcs*, pages 21–30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPIcs.RTA.2011.21.

[3] Carsten Fuhs and Cynthia Kop. Polynomial Interpretations for Higher-Order Rewriting. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, volume 15 of *LIPIcs*, pages 176–192. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.RTA.2012.176.

[4] Cynthia Kop. WANDA - a higher order termination tool (system description). In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 36:1–36:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSCD.2020.36.

[5] Cynthia Kop. Wanda's source code repository, 2023. URL: https://github.com/hezzel/wanda.

[6] Deivid Vale and Niels van der Weide. Onijn documentation, 2022. URL: https://deividrvale.github.io/nijn-coq-script-generation/onijn/index.html.

[7] Niels van der Weide and Deivid Vale. nmvdw/nijn: 1.0.0, May 2023. doi:10.5281/zenodo.7913023.

[8] Niels van der Weide, Deivid Vale, and Cynthia Kop. Certifying higher-order polynomial interpretations. In *Proc. ITP 2023 (to appear)*, 2023. URL: https://doi.org/10.48550/arXiv.2302.11892.