12th International Workshop on Higher-Order Rewriting (HOR 2025)

Pablo Barenbaum (editor)

14^{th} July 2025, Birmingham, United Kingdom

Preface

This report contains the informal proceedings of the *12th International Workshop on Higher-Order Rewriting (HOR 2025)*, to be held on 14th July 2025, Birmingham, United Kingdom:

`https://hor2025.github.io/`

HOR 2025 is affiliated with the 10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025).

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
- **Computing paradigms**: lambda-calculi, higher-order logic programming, quantum programming languages, process calculi.

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

The 12th Workshop on Higher-Order Rewriting features six extended abstracts, contained in this volume, and three invited talks:

1. Théo Winterhalter (INRIA Saclay, France). *Controlling computation in type theory.*
2. Vincent van Oostrom (University of Sussex, United Kingdom). *Accounting for the cost of substitution in structured rewriting*
3. Damiano Mazza (CNRS, LIPN, Université Sorbonne Paris Nord, France). *Revisiting Honda and Laurent's Correspondence between the Pi-Calculus and Linear Logic*

HOR is possible thanks to the effort of the participants, the programme commitee, the steering committee, and the local organisers. I would like to thank all the people involved in preparing and running the workshop.

Birmingham, July 2025 Pablo Barenbaum

Programme Committee

Zena Ariola	University of Oregon, United States
Thibaut Balabonski	Université Paris-Saclay, France
Pablo Barenbaum (chair)	UNQ (CONICET) & UBA, Argentina
Małgorzata Biernacka	University of Wroclaw, Poland
Willem Heijltjes	University of Bath, United Kingdom
Johannes Waldmann	HTWK Leipzig, Germany

Steering Committee

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

Table of Contents

Higher-order inductive theorems via recursor templates

K. Hagens[1] and C. Kop[2]

[1] Radboud University, Nijmegen, Netherlands
kasper.hagens@ru.nl
[2] Radboud University, Nijmegen, Netherlands
c.kop@cs.ru.nl

Abstract

Rewriting Induction (RI) is a formal system in term rewriting for proving inductive theorems. Recently, RI has been extended to higher-order Logically Constrained Term Rewriting Systems (LCSTRSs), which makes it an interesting tool for program verification with inductive theorems as an interpretation for program equivalence. A major challenge when proving inductive theorems with RI is the generation of suitable induction hypothesis, preferably automatically. Two existing heuristics often fail. Here, we consider another approach: rather than inventing new heuristics for proving individual cases, we consider classes of equivalences. This is achieved by introducing templates, describing specific tail and non-tail recursive programs. Whenever each of the two programs fit into such a template we can generate an equation which is guaranteed to be an inductive theorem.

1 Introduction

Rewriting Induction (RI) is a method for inductive theorem proving. Recently, it was extended to higher-order Logically Constrained Term Rewriting Systems (LCSTRSs) [5], making it an interesting tool for program verification with inductive theorems as interpretation for program equivalence. The RI proof system is based on well-founded induction, and proving an equation often requires to introduce another equation, to be used as induction hypothesis. Finding such an induction hypothesis is known to be a non-trivial problem, and the two existing generalization methods for RI do not always succeed.

Inspired by [1], we consider another approach: rather than inventing new heuristics for proving the equivalence of individual program pairs, we consider classes of equivalences. We introduce tail and non-tail recursors, specifically aimed at describing simple bounded loop constructions, governed by some binary integer operator. We then introduce templates for describing specific tail and non-tail recursive programs. Whenever each of the two programs fit into a template we can generate an equation which is guaranteed to be an inductive theorem.

Induction proofs with RI Figure 1 shows four implementations of the factorial function: Tail recursive Upward (TU), Tail recursive Downward (TD), Recursive Upward (RU) and Recursive Downward (RD). Figure 2 shows their LCSTRS representation. Provided $x \geq 1$, they all compute $x \mapsto \prod_{i=1}^{x} i$. We aim to prove all $\binom{4}{2} = 6$ program-pairs being equivalent. In the setting of LCSTRSs the equivalence of, for example, $\mathsf{factTU}\ x$ and $\mathsf{factRU}\ x$ for $x \geq 1$ is expressed by the equation $\mathsf{factTU}\ x \approx \mathsf{factRU}\ x\ [x \geq 1]$.

With RI we then prove that this equation is an *inductive theorem*, meaning that for every ground substitution γ that satisfies $[\![(x \geq 1)\gamma]\!] = \top$ we have $(\mathsf{factTU}\ x)\gamma \leftrightarrow^*_{\mathcal{R}} (\mathsf{factRU}\ x)\gamma$. Here, $\leftrightarrow^*_{\mathcal{R}}$ is the transitive, reflexive closure of $\rightarrow_{\mathcal{R}} \cup \leftarrow_{\mathcal{R}}$, with $\rightarrow_{\mathcal{R}}$ the rewrite relation generated by $\mathcal{R}$ (and $\mathcal{R}$ the set of all rules involved in the definition of factTU and factRU).

A pleasant property of constrained rewriting is that it incorporates primitive data structures (such as the integers) non-inductively. This in turn is beneficial when it comes to inductive

```
int factTU(int x){
    int a = 1; int i = 1;
    while (i<=x){
        a = i*a; i = i+1;}
    return a; }
```

(a) Tail recursive Upward

```
int factTD(int x){
    int a = 1;
    while (x>0){
        a = a*x; x = x-1;}
    return a; }
```

(b) Tail recursive Downward

```
int factRD(int x){
    if (x > 1)
        return(x*factRD(x-1));
    else
        return 1; }
```

(c) Recursive Downward

```
int factRU(int x) = R(1, x);
int R(int i, int x){
    if (i<x)
        return(i*R(i+1, x));
    else
        return x; }
```

(d) Recursive Upward

Figure 1: Four equivalent implementations of $x \mapsto \prod_{i=1}^{x} i$.

$$\mathsf{factTU}\ x \to \mathsf{u}\ x\ 1\ 1$$
$$\mathsf{u}\ x\ i\ a \to a \quad [i > x]$$
$$\mathsf{u}\ x\ i\ a \to \mathsf{u}\ x\ (i+1)\ (i*a) \quad [i \leq x]$$

(a) Tail recursive Upward

$$\mathsf{factTD}\ x \to \mathsf{d}\ x\ 1$$
$$\mathsf{d}\ x\ a \to a \quad [x \leq 0]$$
$$\mathsf{d}\ x\ a \to \mathsf{d}\ (x-1)\ (a*x) \quad [x > 0]$$

(b) Tail recursive Downward

$$\mathsf{factRD}\ x \to 1 \quad [x \leq 1]$$
$$\mathsf{factRD}\ x \to x * (\mathsf{factRD}\ (x-1)) \quad [x > 1]$$

(c) Recursive Downward

$$\mathsf{factRU}\ x \to \mathsf{R}\ 1\ x$$
$$\mathsf{R}\ i\ x \to x \quad [i > x-1]$$
$$\mathsf{R}\ i\ x \to i * (\mathsf{R}\ (i+1)\ x) \quad [i \leq x-1]$$

(d) Recursive Upward

Figure 2: The LCTRS representations of the programs in Figure 1.

theorem proving, as it allows us to more directly deal with the program definition itself, instead of getting involved in complicated interactions with underlying recursively defined data structures. As we will see below: when working with integer programs we can use polynomials over $\mathbb{Z}$ to express invariants that we need to generate an induction hypothesis (and we can use computer algebra systems to find such polynomials [4]).

We briefly try to give some intuition for the role of generalization during the generation of induction hypotheses in RI proofs, and why this is easier for some equivalences than for others.

Typically, the two existing generalization methods (InGen [2] and matrix invariants [4]) perform well when comparing a tail recursive implementation with a recursive implementation. The corresponding RI proof is usually generated in two stages, which we will illustrate below for the equivalence $\mathsf{factTU}\ x \approx \mathsf{factRD}\ x$.

Stage 1: Eliminating recursive term. The proof of $\mathsf{factTU}\ x \approx \mathsf{factRD}\ x$ generates an induction hypothesis $\mathsf{u}\ x\ 1\ 1 \approx \mathsf{factRD}\ x$, which is applied later in the proof process to transform the equation $\mathsf{u}\ x\ 3\ 2 \approx x * (\mathsf{factRD}\ x_1)\ [x \geq 2 \wedge x_1 = x - 1]$ into $\mathsf{u}\ x\ 3\ 2 \approx x * (\mathsf{u}\ x\ 1\ 1)\ [x \geq 2 \wedge x_1 = x - 1]$. This stage did not require any generalization because $\mathsf{u}\ x\ 1\ 1 \approx \mathsf{factRD}\ x$ was automatically obtained as a proof goal during the RI process, and RI always allows us to save a proof goal as an induction hypothesis.

Stage 2: Divergence solving. After eliminating the recursive term, the proof starts to

diverge into an infinite repeating process, each time producing a new proof obligation that we are not able to remove. We only show the first three:

$$\begin{aligned}
\mathsf{u}\ x\ 3\ 2 &\approx x * \mathsf{u}\ x_1\ 2\ 1 && [x \geq 2 \wedge x_1 = x - 1]\\
\mathsf{u}\ x\ 4\ 6 &\approx x * \mathsf{u}\ x_1\ 3\ 2 && [x \geq 3 \wedge x_1 = x - 1]\\
\mathsf{u}\ x\ 5\ 24 &\approx x * \mathsf{u}\ x_1\ 4\ 6 && [x \geq 4 \wedge x_1 = x - 1]
\end{aligned}$$

None of these equations can be saved as induction hypothesis to remove the succeeding equation. Fortunately, both generalization methods are able to generate an induction hypothesis $\mathsf{u}\ x\ i\ a \approx x * \mathsf{u}\ x_1\ i_1\ a_1\ [i_1 = i - 1 \wedge x \geq i_1 \wedge x_1 = x - 1 \wedge a = a_1 * i_1]$.

There is, however, no guarantee that comparing a tail recursive with a recursive will always lead to such a procedure. For example, when trying to prove $\mathsf{factTU}\ x \approx \mathsf{factRU}\ x\ [x \geq 1]$ we are not able to eliminate the recursive term and immediately run into the divergence shown in Figure 3a. The repeated unfolding of the recursive call $\mathsf{R}\ i\ x \to i * (\mathsf{R}\ (i+1)\ x)$ makes

$$\begin{aligned}
\mathsf{u}\ x\ 2\ 1 &\approx \mathsf{R}\ 1\ x && [x \geq 1]\\
\mathsf{u}\ x\ 3\ 2 &\approx 1 * (\mathsf{R}\ 2\ x) && [x \geq 2]\\
\mathsf{u}\ x\ 4\ 6 &\approx 1 * (2 * (\mathsf{R}\ 3\ x)) && [x \geq 3]\\
\mathsf{u}\ x\ 5\ 24 &\approx 1 * (2 * (3 * (\mathsf{R}\ 4\ x))) && [x \geq 4]
\end{aligned}$$

(a) Original divergence

$$\begin{aligned}
\mathsf{u}\ x\ 2\ 1 &\approx \mathsf{R}\ 1\ x && [x \geq 1]\\
\mathsf{u}\ x\ 3\ 2 &\approx 1 * (\mathsf{R}\ 2\ x) && [x \geq 2]\\
\mathsf{u}\ x\ 4\ 6 &\approx 2 * (\mathsf{R}\ 3\ x) && [x \geq 3]\\
\mathsf{u}\ x\ 5\ 24 &\approx 6 * (\mathsf{R}\ 4\ x) && [x \geq 4]
\end{aligned}$$

(b) Processed divergence

Figure 3: Divergence of $\mathsf{factTU}\ x \approx \mathsf{factRU}\ x\ [x \geq 1]$

it difficult to handle by both generalization methods, as it leads to a divergence where each equation has a different term shape. Once we turn the divergence into the shape shown in Figure 3b we can apply the matrix invariants method to generate an induction hypothesis $\mathsf{u}\ x\ i\ z \approx a * (\mathsf{R}\ j\ x)\ [z = a * j \wedge i = j + 1 \wedge j \leq x]$. InGen is not capable of producing this induction hypothesis because it is not able to find the crucial invariant $z = a * j$.

For $\mathsf{factRU}\ x \approx \mathsf{factTD}\ x\ [x \geq 1]$ the situation is worse. We obtain a divergence

$$\begin{aligned}
\mathsf{d}\ i_1\ a_1 &\approx \mathsf{R}\ 1\ x && [i_1 = x - 1 \ \wedge \ a_1 = x && \wedge\ x \geq 1]\\
\mathsf{d}\ i_2\ a_2 &\approx 1 * (\mathsf{R}\ 2\ x) && [i_2 = x - 2 \ \wedge \ a_2 = x * (x-1) && \wedge\ x \geq 2]\\
\mathsf{d}\ i_3\ a_3 &\approx 1 * (2 * (\mathsf{R}\ 3\ x)) && [i_3 = x - 3 \ \wedge \ a_3 = x * (x-1) * (x-2) && \wedge\ x \geq 3]
\end{aligned}$$

This time, we cannot find an induction hypothesis of the shape $\mathsf{d}\ i\ a \approx a * (\mathsf{R}\ j\ x)\ [\varphi]$, where φ only contains polynomial arithmetical expressions (we can find an invariant $i + j = x$ but this is not sufficient to obtain an induction hypothesis). We are not able to prove this equation.

For $\mathsf{factRU}\ x \approx \mathsf{factRD}\ x\ [x \geq 1]$ and $\mathsf{factTU}\ x \approx \mathsf{factTD}\ x$ we encounter similar problems: the invariants that we can find are not sufficient to generate induction hypothesis.

Recursor templates In section 2 we will introduce recursor templates for LCSTRSs. This allows us to circumvent the need for executing explicit RI proofs, which as we just motivated can be quite cumbersome due to the need of finding induction hypotheses. The dirty work only needs to be done once, when proving the correctness of our templates and the corresponding inductive theorems. After this, we can check whether a specific example can be matched with a template and automatically generate a corresponding inductive theorem.

We will show that we can prove all 6 inductive theorems from Figure 2 with recursor templates. In addition, we will show we can handle higher-order examples as well.

1.1 Prerequisites

LCSTRSs [3] are a higher-order rewriting formalism with built-in support for integers and booleans (or in fact any arbitrary theory such as bitvectors, floating point numbers or integer arrays) as well as logical constraints to model control flow. This considers *applicative* higher-order term rewriting (without λ abstractions) and *first-order* constraints. We will introduce the minimal necessary prerequisites.

We assume a set of sorts (base types) $\mathcal{S}$; the set $\mathcal{T}$ of types is defined by $\mathcal{T} ::= \mathcal{S} \mid \mathcal{T} \to \mathcal{T}$. Here, $\to$ is right-associative. Assume a subset $\mathcal{S}_{theory} \subseteq \mathcal{S}$ of *theory sorts* (e.g., int and bool), and define the *theory types* by $\mathcal{T}_{theory} ::= \mathcal{S}_{theory} \mid \mathcal{S}_{theory} \to \mathcal{T}_{theory}$. Every $\iota \in \mathcal{S}_{theory}$ corresponds to a non-empty interpretation set $\mathcal{I}_\iota$. Here, we will use theory sorts bool and int, with $\mathcal{I}_{\mathsf{bool}} = \{\top, \bot\}$ and $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$ (the set of all integers).

We assume a signature Σ of *function symbols* and a disjoint set $\mathcal{V}$ of variables, and a function *typeof* from $\Sigma \cup \mathcal{V}$ to $\mathcal{T}$. The set of terms $T(\Sigma, \mathcal{V})$ over Σ and $\mathcal{V}$ are the well-typed expressions in $\mathbb{T}$, defined by $\mathbb{T} ::= \Sigma \mid \mathcal{V} \mid \mathbb{T}\ \mathbb{T}$. For a term t, let $Var(t)$ be the set of variables in t. A term t is *ground* if $Var(t) = \emptyset$. We assume that Σ is the disjoint union $\Sigma_{theory} \uplus \Sigma_{terms}$, where $\mathit{typeof}(\mathsf{f}) \in \mathcal{T}_{theory}$ for all $\mathsf{f} \in \Sigma_{theory}$.

Each $\mathsf{f} \in \Sigma_{theory}$ has an interpretation $[\![\mathsf{f}]\!] \in \mathcal{I}_{\mathit{typeof}(\mathsf{f})}$. Here, we will fix $\Sigma_{theory} = \{+, -, *\} \cup \{<, \leq, >, \geq, =, \wedge, \vee, \neg\} \cup \{\mathtt{true}, \mathtt{false}\} \cup \{\mathsf{n} \mid n \in \mathbb{Z}\}$, where each of these symbols is typed and interpreted as expected (e.g. $* :: \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ is interpreted as multiplication on $\mathbb{Z}$). We use $[\mathsf{f}]$ for prefix or partially applied notation (e.g., $[+]\ x\ y$ and $x + y$ are the same). Symbols in Σ_{terms} (such as $\mathsf{factRD} :: \mathsf{int} \to \mathsf{int}$) do not have an interpretation since their behavior will be defined through the rewriting system.

Values are theory symbols of base type, i.e. $\mathcal{V}al = \{v \in \Sigma_{theory} \mid \mathit{typeof}(v) \in \mathcal{S}_{theory}\}$, which in our setting are $\mathtt{true}, \mathtt{false}$ and all n. Elements of $T(\Sigma_{theory}, \mathcal{V})$ are *theory terms*. A *constraint* is a theory term $\varphi :: \mathsf{bool}$, such that $\mathit{typeof}(x) \in \mathcal{S}_{theory}$ for all $x \in Var(\varphi)$. For example, we have theory terms $x + \mathsf{3}$, $\mathtt{true}$ and $\mathsf{7} * \mathsf{0}$. The latter two are ground. We have $[\![\mathsf{7} * \mathsf{0}]\!] = 0$. An example of a constraint is $x * y > \mathsf{0}$.

A rewrite rule is an expression $\ell \to r\ [\varphi]$ with $\mathit{typeof}(\ell) = \mathit{typeof}(r)$, $\ell = \mathsf{f}\ \ell_1 \cdots \ell_k$ with $\mathsf{f} \in \Sigma$ and $k \geq 0$, φ a constraint and $Var(r) \subseteq Var(\ell) \cup Var(\varphi)$. If $\varphi = \mathtt{true}$, we write $\ell \to r$. We assume familiarity with contexts and substitutions. A substitution γ respects constraint φ if $\gamma(Var(\varphi)) \subseteq \mathcal{V}al$ and $[\![\varphi\gamma]\!] = \top$. We define $\mathcal{R}_{calc} = \{\mathsf{f}\ x_1 \cdots x_m \to y\ [y = \mathsf{f}\ x_1 \cdots x_m] \mid \mathsf{f} \in \Sigma_{theory} \setminus \mathcal{V}al,\ \mathit{typeof}(\mathsf{f}) = \iota_1 \to \ldots \to \iota_m \to \kappa\}$. The reduction relation $\to_{\mathcal{R}}$ is defined by:

$$C[l\gamma] \to_{\mathcal{R}} C[r\gamma] \text{ if } \ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{calc} \text{ and } \gamma \text{ respects } \varphi$$

For example, we have a reduction $\mathsf{factRD}\ \mathsf{2} \to_{\mathcal{R}} \mathsf{2} * (\mathsf{factRD}\ (\mathsf{2} - \mathsf{1})) \xrightarrow{\mathcal{R}_{calc}} \mathsf{2} * (\mathsf{factRD}\ \mathsf{1}) \to_{\mathcal{R}} \mathsf{2} * \mathsf{1} \xrightarrow{\mathcal{R}_{calc}} \mathsf{2}$. An *equation* is a triple $s \approx t\ [\varphi]$ with $\mathit{typeof}(s) = \mathit{typeof}(t)$ and φ a constraint. A substitution γ respects $s \approx t\ [\varphi]$ if γ respects φ and $Var(s) \cup Var(t) \subseteq dom(\gamma)$. An equation $s \approx t\ [\varphi]$ is an *inductive theorem* if $s\gamma \leftrightarrow^*_{\mathcal{R}} t\gamma$ for every ground substitution γ that respects it. Here $\leftrightarrow_{\mathcal{R}} = \to_{\mathcal{R}} \cup \leftarrow_{\mathcal{R}}$, and $\leftrightarrow^*_{\mathcal{R}}$ is its transitive, reflexive closure.

RI is a deduction system on proof states, which are pairs of the shape $(\mathcal{E}, \mathcal{H})$. Intuitively, $\mathcal{E}$ is a set of equations, describing all proof goals, and $\mathcal{H}$ is the set of induction hypotheses that have been assumed. At the start $\mathcal{E}$ consists of all equations that we want to prove to be inductive theorems, and $\mathcal{H} = \emptyset$. With a deduction rule we may transform a proof state $(\mathcal{E}, \mathcal{H})$ into another proof state $(\mathcal{E}', \mathcal{H}')$. This is denoted as $(\mathcal{E}, \mathcal{H}) \vdash (\mathcal{E}', \mathcal{H}')$. We write $\vdash^*$ for the reflexive, transitive closure of $\vdash$. If a RI deduction removes every proof goal in $\mathcal{E}$ then $\mathcal{E}$ only contains inductive theorems. This is expressed the following soudness principle: "If $(\mathcal{E}, \mathcal{H}) \vdash^* (\emptyset, \mathcal{H})$ for some set $\mathcal{H}$, then every equation in $\mathcal{E}$ is an inductive theorem".

Template		Inductive theorem
$C_{x,y}[i,a] \to a$	$[i > y]$	$C_{x,y}[i,a] \approx \mathsf{tailup}\ \mathsf{f}\ i\ x\ y\ a$
$C_{x,y}[i,a] \to C_{x,y}[i+1, \mathsf{f}\ i\ a]$	$[i \leq y]$	$[x \leq i \leq y]$
$C_{x,y}[i,a] \to a$	$[i < x]$	$C_{x,y}[i,a] \approx \mathsf{taildown}\ \mathsf{f}\ i\ x\ y\ a$
$C_{x,y}[i,a] \to C_{x,y}[i-1, \mathsf{f}\ a\ i]$	$[i \geq x]$	$[x \leq i \leq y]$
$C_{x,y}[i] \to D[i]$	$[i > y]$	$C_{x,y}[i] \approx \mathsf{recup}\ \mathsf{f}\ i\ x\ y\ D[z]$
$C_{x,y}[i] \to \mathsf{f}\ i\ C_{x,y}[i+1]$	$[i \leq y]$	$[x \leq i \leq y \wedge z = y+1]$
$C_{x,y}[i] \to D[i]$	$[i < x]$	$C_{x,y}[i] \approx \mathsf{recdown}\ \mathsf{f}\ i\ x\ y\ D[z]$
$C_{x,y}[i] \to \mathsf{f}\ i\ C_{x,y}[i-1]$	$[i \geq x]$	$[x \leq i \leq y \wedge z = x-1]$

Table 1: Recursor templates and corresponding inductive theorems

2 Recursor templates

We define recursors of type $(\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ as follows

$$\begin{aligned}
\mathsf{tailup}\ f\ i\ x\ y\ a &\to \mathsf{tailup}\ f\ (i+1)\ x\ y\ (f\ i\ a) && [x \leq i \leq y] \\
\mathsf{taildown}\ f\ i\ x\ y\ a &\to \mathsf{taildown}\ f\ (i-1)\ x\ y\ (f\ a\ i) && [x \leq i \leq y] \\
\mathsf{recup}\ f\ i\ x\ y\ a &\to f\ i\ (\mathsf{recup}\ f\ (i+1)\ x\ y\ a) && [x \leq i \leq y] \\
\mathsf{recdown}\ f\ i\ x\ y\ a &\to f\ i\ (\mathsf{recdown}\ f\ (i-1)\ x\ y\ a) && [x \leq i \leq y]
\end{aligned}$$

We furthermore define $\mathsf{F}\ f\ i\ x\ y\ a \to a\ [i < x \vee i > y]$ for all $\mathsf{F} \in \{\mathsf{tailup}, \mathsf{taildown}, \mathsf{recup}, \mathsf{recdown}\}$.

What these recursors have in common is that in each call the iterator i is increased/decreased by 1 (executing its recursive call) until it surpasses the lower bound x or upperbound y (returning accumulator a). The same can be said about the examples in Figure 2. For example, Figure 2c is equivalent to $\mathcal{R} = \{\mathsf{factRD}\ i \to 1\ [i < 2],\ \mathsf{factRD}\ i \to i * (\mathsf{factRD}\ (i-1))\ [i \geq 2]\}$, satisfying this behavior (except that here, we have lower bound $x = 2$ but no upper bound y).

Let us consider downward recursion more abstractly, using the following template consisting of two rewrite rules (Figure 2c fits in by taking $C_{x,y}[i] = \mathsf{recdown}\ i$, $\mathsf{f} = *$, $D[i] = 1$, $x = 2$).

$$\begin{cases} C_{x,y}[i] \to D[i] & [i < x] \\ C_{x,y}[i] \to \mathsf{f}\ i\ C_{x,y}[i-1] & [i \geq x] \end{cases}$$

With RI we can prove that this template corresponds to the inductive theorem $C_{x,y}[i] \approx \mathsf{recdown}\ \mathsf{f}\ i\ x\ y\ D[z]\ [x \leq i \leq y \wedge z = x-1]$. For Figure 2c this yields $\mathsf{factRD}\ i \approx \mathsf{recdown}\ [*]\ i\ 2\ y\ 1\ [2 \leq i \leq y]$. The absence of an upper bound in the definition of factRD is reflected by the corresponding inductive theorem: variable y occurs only on the right-hand side and in the constraint. We can freely choose any y which satisfies $i \leq y$.

Table 1 summarizes the templates and corresponding inductive theorems for all the 4 types of recursion that we will consider.

Example 2.1. By renaming $x := y$ in Figure 2a we obtain the equivalent LCSTRS with rules $\mathsf{factTU}\ y \to \mathsf{u}\ y\ 1\ 1$, $\mathsf{u}\ y\ i\ a \to a\ [i > y]$, $\mathsf{u}\ y\ i\ a \to \mathsf{u}\ y\ (i+1)\ (i*a)\ [i \leq y]$. The u-rules fit into the tailup template of Table 1 (take $\mathsf{f} = *$ and $C_{x,y}[i,a] = \mathsf{u}\ y\ i\ a$). This yields an inductive theorem $\mathsf{u}\ y\ i\ a \approx \mathsf{tailup}\ [*]\ i\ x\ y\ a\ [x \leq i \leq y]$. Since $\mathsf{factTU}\ y \to_{\mathcal{R}} \mathsf{u}\ y\ 1\ 1$, we obtain the inductive theorem $\mathsf{factTU}\ y \approx \mathsf{tailup}\ [*]\ 1\ x\ y\ 1\ [x \leq 1 \leq y]$.

3 Proving inductive theorems with recursor templates

With Table 1 we can automatically generate inductive theorems, relating program definitions to one of the pre-defined recursors tailup, taildown, recup and recdown. To conclude program equivalence we in addition need to derive inductive theorems between the recursors themselves.

Lemma 3.1. $\mathsf{recdown}\ f\ y\ x\ n\ a \approx \mathsf{tailup}\ f\ x\ m\ y\ a\ [m \le x \le y \le n]$ *is an inductive theorem.*

This lemma is easily proven with RI.

Example 3.1. Variable-renaming the inductive theorem from Example 2.1 yields $\mathsf{factTU}\ i \approx \mathsf{tailup}\ [*]\ 1\ x\ i\ 1\ [x \le 1 \le i]$. We also deduced $\mathsf{factRD}\ i \approx \mathsf{recdown}\ [*]\ i\ 2\ y\ 1\ [2 \le i \le y]$. Moreover, we easily prove the $*$-specific equation $\mathsf{recdown}\ [*]\ i\ 2\ y\ a \approx \mathsf{recdown}\ [*]\ i\ 1\ y\ a$, which gives us

$$\begin{array}{ll} \mathsf{factTU}\ i \approx \mathsf{tailup}\ [*]\ 1\ x\ i\ 1 & [x \le 1 \le i] \\ \mathsf{factRD}\ i \approx \mathsf{recdown}\ [*]\ i\ 1\ y\ 1 & [2 \le i \le y] \end{array}$$

fitting into Lemma 3.1 by substituting $[x := 1,\ m := x,\ y := i,\ a := 1,\ n := y]$. We obtain $\mathsf{factTU}\ i \approx \mathsf{factRD}\ i\ [x \le 1 \le 2 \le i \le y]$, or equivalently $\mathsf{factTU}\ i \approx \mathsf{factRD}\ i\ [i \ge 2]$.

The remaining recursor equivalences we can only prove conditionally: under assumption $f = \mathsf{f} \in \Sigma$ satisfies extra properties (here, we need commutativity/associativity). We collect our assumptions in a set $\mathcal{A}$ of axioms, required to be proven by a RI deduction $(\mathcal{A}, \emptyset) \vdash^* (\emptyset, \mathcal{H})$.

Definition 3.1 (Conditional inductive theorems)**.** Let $\mathcal{A}$ be a set of equations (axioms) and $\mathcal{E}$ be a set of equations. We define the *conditional inductive theorem* $\mathcal{A} \models^{\mathsf{c}} \mathcal{E}$ as follows: "If there is a set $\mathcal{H}$ and a RI-deduction $(\mathcal{A}, \emptyset) \vdash^* (\emptyset, \mathcal{H})$ then every equation in $\mathcal{E}$ is an inductive theorem."

For $\mathsf{f} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \in \Sigma$, we define axiom sets $\mathsf{C}(\mathsf{f}) = \{\mathsf{f}\ x\ y \approx \mathsf{f}\ y\ x\}$ and $\mathsf{AC}(\mathsf{f}) = \{\mathsf{f}\ x\ (\mathsf{f}\ y\ z) \approx \mathsf{f}\ (\mathsf{f}\ x\ y)\ z,\ \mathsf{f}\ x\ y \approx \mathsf{f}\ y\ x\}$.

Lemma 3.2. *Let* $\mathsf{f} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \in \Sigma$*. The following are conditional inductive theorems*

$$\begin{array}{rl} \mathsf{AC(f)} \models^{\mathsf{c}} \mathsf{recdown\ f}\ y\ x\ n\ a \approx \mathsf{recup\ f}\ x\ m\ y\ a & [m \le x \wedge x \le y \wedge y \le n] \\ \mathsf{AC(f)} \models^{\mathsf{c}} \mathsf{taildown\ f}\ y\ x\ n\ a \approx \mathsf{tailup\ f}\ x\ m\ y\ a & [m \le x \wedge x \le y \wedge y \le n] \\ \mathsf{C(f)} \models^{\mathsf{c}} \mathsf{taildown\ f}\ y\ x\ n\ a \approx \mathsf{recup\ f}\ x\ m\ y\ a & [m \le x \wedge x \le y \wedge y \le n] \\ \mathsf{AC(f)} \models^{\mathsf{c}} \mathsf{taildown\ f}\ i\ x\ y\ a \approx \mathsf{recdown\ f}\ i\ x\ y\ a & \\ \mathsf{AC(f)} \models^{\mathsf{c}} \mathsf{tailup\ f}\ i\ x\ y\ a \approx \mathsf{recup\ f}\ i\ x\ y\ a & \end{array}$$

With RI we easily show $(\mathsf{AC}(*), \emptyset) \vdash^* (\emptyset, \mathcal{H})$ for $\mathcal{H} = \emptyset$. Using Lemma 3.2 we derive the remaining inductive theorems in Figure 2, such as $\mathsf{factTD}\ x \approx \mathsf{factTU}\ x\ [x \ge 1]$.

Higher-order equivalences With Lemma 3.1, 3.2 we prove all equivalences from Figure 1. For higher-order equivalences, however, they no longer suffice. Consider the following higher-order variants of the LCSTRSs in Figure 2a and Figure 2b, both computing $(f, x) \mapsto \prod_{i=1}^{x} f(i)$

$$\begin{array}{lll} \mathsf{funfactTU}\ f\ y \to \mathsf{u}\ f\ y\ 1\ 1 & & \mathsf{funfactTD}\ f\ i \to \mathsf{d}\ f\ i\ 1 \\ \mathsf{u}\ f\ y\ i\ a \to a & [i > y] & \mathsf{d}\ f\ i\ a \to a & [i < 1] \\ \mathsf{u}\ f\ y\ i\ a \to \mathsf{u}\ f\ y\ (i+1)\ ((f\ i) * a) & [i \le y] & \mathsf{d}\ f\ i\ a \to \mathsf{d}\ f\ (i-1)\ (a * (f\ i)) & [i \ge 1] \end{array}$$

By Table 1 we obtain $\mathsf{funfactTD}\ f\ i \approx \mathsf{taildown}\ (\lambda a, i.(f\ i) * a)\ i\ 1\ y\ 1\ [1 \le i \le y]$ and $\mathsf{funfactTU}\ f\ y \approx \mathsf{tailup}\ (\lambda i, a.(f\ i) * a)\ 1\ x\ y\ 1\ [x \le 1 \le y]$. Here, λ is used as meta-language

notation. For example, $\lambda a, i.(f\ i) * a$ denotes a function symbol $\mathsf{G}_f \in \Sigma$ defined by $\mathsf{G}_f\ a\ i \to (f\ i) * a$. Note that Lemma 3.2 does not apply: we do not even have $\lambda a, i.(f\ i) * a = \lambda i, a.(f\ i) * a$.

However, we can prove the equivalence once we have the following result

Lemma 3.3. *Let* $\mathsf{F} :: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \in \Sigma$. *The following are conditional inductive theorems*

$\mathsf{AC}(\mathsf{F}) \models^{c} \mathsf{recdown}\ (\lambda i, a.\ (\mathsf{F}\ (f\ i)\ a))\ y\ x\ n\ a \approx \mathsf{recup}\ (\lambda i, a.\ (\mathsf{F}\ (f\ i)\ a))\ x\ m\ y\ a\ [m \le x \le y \le n]$

$\mathsf{AC}(\mathsf{F}) \models^{c} \mathsf{taildown}\ (\lambda a, i.\ (\mathsf{F}\ (f\ i)\ a))\ y\ x\ n\ a \approx \mathsf{tailup}\ (\lambda i, a.\ (\mathsf{F}\ (f\ i)\ a))\ x\ m\ y\ a\ [m \le x \le y \le n]$

$\mathsf{AC}(\mathsf{F}) \models^{c} \mathsf{taildown}\ (\lambda a, i.\ (\mathsf{F}\ (f\ i)\ a))\ i\ x\ y\ a \approx \mathsf{recdown}\ (\lambda i, a.\ (\mathsf{F}\ (f\ i)\ a))\ i\ x\ y\ a$

$\mathsf{AC}(\mathsf{F}) \models^{c} \mathsf{tailup}\ (\lambda i, a.\ (\mathsf{F}\ (f\ i)\ a))\ i\ x\ y\ a \approx \mathsf{recup}\ (\lambda i, a.\ (\mathsf{F}\ (f\ i)\ a))\ i\ x\ y\ a$

$\emptyset \models^{c} \mathsf{taildown}\ (\lambda a, i.\ (F\ (f\ i)\ a))\ y\ x\ n\ a \approx \mathsf{recup}\ (\lambda i, a.\ (F\ (f\ i)\ a))\ x\ m\ y\ a\ [m \le x \le y \le n]$

4 Closing remarks

Constrained rewriting The facility for non-inductively defined primitive data structures is very specific to constrained rewriting. This made it possible to define recursors and templates being able to describe integer loops in a manner that is intuitively very close to real-life programming (where we can also treat the integers as being given for free). The templates defined here, we cannot define in ordinary higher-order rewriting. We specifically aimed at loop constructions having an integer counter i which is increased/decreased by 1 in each loop iteration. In future work we can further extend our existing templates or add new ones, e.g. generalizing our templates to increases/decreases by some arbitrary number k. We could also introduce recursors that iterate over lists, obtaining foldl and foldr. However, such recursors we can already define in ordinary higher-order rewriting.

Related & future work The idea to use templates for inductive theorem proving is not new. A comparable work for unconstrained first-order rewriting is [1], where the authors define templates to verify program transformations. In contrast to our approach, equivalence between templates is proven directly (i.e. no intermediate recursors like tailup are used) using the notion *equivalent term rewriting systems* (instead of using RI), which they can prove with specifically designed transformation rules. It seems that the underlying mechanism is fundamentally different, because their method assumes confluence (whereas RI relies on termination).

In future work we could investigate if we could benefit from this and other existing work on program transformations based on term rewriting, such as context moving transformations [6].

Implementation We recently implemented RI for LCSTRSs [5] in Cora (see `https://github.com/hezzel/cora`), and we are currently working on implementing the template method as well.

References

[1] Y. Chiba, T. Aoto, and Y. Toyama. Program transformation templates for tupling based on term rewriting. *IEICE TRANSACTIONS on Information and Systems*, E93-D(5):963–973, 2010.

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

[3] L. Guo and C. Kop. Higher-order LCTRSs and their termination. In *Proc. ESOP 24*, volume 14577 of *LNCS*, pages 331–357, 2024.

[4] K. Hagens and C Kop. Matrix invariants for program equivalence in lctrss. In *Proc. WPTE 23*, 2023.

[5] K. Hagens and C. Kop. Rewriting induction for higher-order constrained term rewriting systems. In *Proc. LOPSTR 24*, volume 14919, pages 202–219, 2024.

[6] K. Sato, K. Kikuchi, T. Aoto, and Y. Toyama. Correctness of context-moving transformations for term rewriting systems. In *Proc. LOPSTR 15*, volume 9527 of *LNCS*, pages 331–345, 2015.

Basis-Sensitive Quantum Typing via Realizability

Alejandro Díaz-Caro[1,2], Octavio Malherbe[3], and Rafael Romero[4,5,6]

[1] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
[2] Universidad Nacional de Quilmes, Bernal, Buenos Aires, Argentina
[3] Universidad de la República, FIng, IMERL, Montevideo, Uruguay
[4] Universidad de Buenos Aires, FCEN, DC, Buenos Aires, Argentina
[5] CONICET-Universidad de Buenos Aires, ICC, Buenos Aires, Argentina
[6] PEDECIBA, Universidad de la República-MEC, Montevideo, Uruguay

1 Introduction

In the context of quantum physics, there are impossibility theorems stating that arbitrary quantum states cannot be copied or deleted [10]. For this reason, when designing quantum programming languages, quantum bits (qubits) are often modelled as linear resources (as in linear logic [6]) within type systems. Qubits are vectors in a Hilbert space[1], so our starting point will be a quantum lambda calculus that manipulates vectors to perform computations. There is, however, a subtlety in these impossibility theorems. *Arbitrary qubits* cannot be copied, but it is indeed possible to do so with known qubits. This implies that qubits with known values behave as classical data and can be treated accordingly. Moreover, it suffices to know the basis to which a qubit belongs in order to copy and delete it. This is a well-known fact in quantum information theory, a fact that underlies many quantum algorithms.

In most quantum programming languages, qubits are interpreted in a canonical basis (often called the *computational basis*); see, for instance, [4, 7–9]. In this fashion, classical bits are represented by the basis vectors, and qubits as norm-1 linear combinations of bits. We are allowed to copy and delete classical bits freely, while such operations on arbitrary qubits remain restricted.

In this work, we start from the quantum-control lambda calculus defined in [4]. This calculus was introduced using a realizability technique, which allows one to extract a type system from its operational semantics. Our aim is to extract a type system, via a realizability technique, that is able to track bases, so that qubits in known bases can be treated classically, while unknown qubits are still handled linearly.

2 The λ_B calculus

Realizability. Realizability is a technique for extracting type systems from the operational semantics of a calculus, resulting in a system in which safety properties hold by construction.

The steps to define a programming language using this technique are as follows. First, define a calculus equipped with a deterministic evaluation strategy. Second, define types as sets of closed values in the language, optionally introducing operations to build more complex types. Third, define the typing judgement $\Gamma \vdash t : A$, where Γ is a context of typed variables, t a term in the calculus, and A a type, as the property that for every valid substitution θ of Γ, the term $\theta(t)$ reduces to a value in A, i.e., $\theta(t) \twoheadrightarrow v \in A$.

[1] A vector space equipped with an inner product.

Pure values	v	::=	$x \mid \lambda x_B\,.\,\vec{t} \mid \lvert 0\rangle \mid \lvert 1\rangle$	
Pure terms	t	::=	$v \mid t\,t \mid (t,t) \mid \mathtt{let}_{B\otimes B}\ (x,y) = \vec{t}\ \mathtt{in}\ \vec{t}$	
		$\mid$	$\mathsf{case}\ \vec{t}\ \mathsf{of}\{\vec{v} \mapsto \vec{t} \mid \cdots \mid \vec{v} \mapsto \vec{t}\}$	
Value distributions	$\vec{v}$	::=	$v \mid \vec{v} + \vec{v} \mid \alpha \cdot \vec{v}$	$(\alpha \in \mathbb{C})$
Term distributions	$\vec{t}$	::=	$t \mid \vec{t} + \vec{t} \mid \alpha \cdot \vec{t}$	$(\alpha \in \mathbb{C})$

Table 1: Syntax of the calculus

In this setting, each typing rule corresponds to a provable theorem. For instance, if $\Gamma \vdash t : A$ implies $\Delta \vdash r : B$, then the following rule is derivable:

$$\frac{\Gamma \vdash t : A}{\Delta \vdash r : B}$$

For the first step, we adopt the grammar described in Table 1. It includes pair constructors and destructors, two boolean values, and a quantum conditional case construct. These terms are closed under linear combination, forming a vector space over $\mathbb{C}$. We define a language in the "quantum data and control" paradigm, which allows the superposition of programs. Such superpositions are represented as norm-1 linear combinations of terms.

The novel feature of the calculus lies in its abstraction construct, which is decorated with a basis B. This basis is defined as a set of value distributions that are pairwise orthogonal. Beta reduction is performed in a call-by-value fashion, but with a crucial twist: if the argument of the abstraction belongs to the specified basis, then standard beta-reduction is applied. Otherwise, if the value distribution can be rewritten as a linear combination of elements in that basis, the reduction proceeds by distributing the application over the components of the decomposition.

This mechanism generalises the *call-by-basis* strategy introduced in [2]. We now define a substitution operation that incorporates this behaviour:

Definition 2.1. *For a term distribution $\vec{t}$, value distribution $\vec{v}$, variable x and orthogonal basis A, we define the substitution $\vec{t}\langle\vec{v}/x\rangle_A$ as:*

$$\vec{t}\langle\vec{v}/x\rangle_A = \begin{cases} \sum_{i\in I} \alpha_i \vec{t}\,[\vec{b}_i/x] & A = \{\vec{b}_i\}_{i\in I} \text{ and } \vec{v} \equiv \sum_{i\in I} \alpha_i \vec{b}_i \\ \sum_{i\in I} \alpha_i \vec{t}\,[v_i/x] & A = \mathbb{K} \text{ and } \vec{v} = \sum_{i\in I} \alpha_i v_i \\ \textit{Undefined} & \textit{Otherwise} \end{cases}$$

We extend the substitution for more than one pair of variables. This definition is extended to pair of values in the following way. Let $\vec{v} = \sum_{i\in I} \alpha_i(\vec{v_i}, \vec{w_i})$:

$$\vec{t}\langle\vec{v}/x \otimes y\rangle_{A\otimes B} = \sum_{i\in I} \alpha_i \vec{t}\langle\vec{v_i}/x\rangle_A\langle\vec{w_i}/y\rangle_B$$

The $\mathbb{K}$ basis represents the canonical basis and behaves analogously to the linear substitution defined in [4]. This special basis is necessary to enable higher-order substitution, since it is not possible to define a finite set of basis vectors that generates the vector space of functions $A \Rightarrow B$, for arbitrary types A and B.

With the substitution defined, we can design the reduction system in table 2. We take the liberty to omit the contextual rules, since they are fairly standard. The reduction is weak, meaning there is no action under lambda abstractions nor in the branches of the case construct.

$$\sum_{i=1}^{n} \alpha_i(\lambda x_A \,.\, \vec{t_i})\ \vec{v} \to \sum_{i=1}^{n} \alpha_i \vec{t_i}\langle \vec{v}/x\rangle_A \qquad \text{If } \vec{t_i}\langle \vec{v}/x\rangle_A \text{ is well-defined}$$

$$\mathtt{let}_{B\otimes B'}\ (x,y) = \vec{v}\ \mathtt{in}\ \vec{t} \to \vec{t}\langle \vec{v}/x\otimes y\rangle_{B\otimes B'}$$

$$\mathsf{case}\ \vec{v}\ \mathsf{of}\{\vec{v_1} \mapsto \vec{t_1} \mid \cdots \mid \vec{v_n} \mapsto \vec{t_2}\} \to \sum_{i=1}^{n} \alpha_i \vec{t_i} \qquad \text{If } \vec{v} \equiv \sum_{i=1}^{n} \alpha_i \vec{v_i}$$

Table 2: Reduction system

Example 2.2. *Using this system, we can consider the duplicator function in the Hadamard basis* $\mathbb{X} = \{|+\rangle, |-\rangle\}$ *applied to either* $|+\rangle$ *or* $|0\rangle$ *as such:*

$$(\lambda x^{\mathbb{X}}.(x,x))|+\rangle \twoheadrightarrow (|+\rangle, |+\rangle)$$

$$(\lambda x^{\mathbb{X}}.(x,x))|0\rangle = (\lambda x^{\mathbb{X}}.(x,x))(\frac{1}{\sqrt{2}}|+\rangle + \frac{1}{\sqrt{2}}|-\rangle) \twoheadrightarrow \frac{1}{\sqrt{2}}(|+\rangle, |+\rangle) + \frac{1}{\sqrt{2}}(|-\rangle, |-\rangle)$$

This is not a new operation, any language that implements arbitrary unitary gates is able to encode this program. For example, a language just manipulating unitary gates could implement this program. However, this term which duplicates the qubits in the basis $\mathbb{X}$, or generates Bell states in the computational basis, can be generalized to any basis.

The main point here is that *the operation is native to the language*, without even introducing the concept of unitary gate. Adding more information into the term allow us more flexibility when writing functions. In doing so, we hope to add a layer of abstraction between language and quantum circuits, more in line with modern classical programming languages.

2.1 Typing system

In order to define a type system, first we need to select which sets will act as types. We will choose sets of orthogonal qubits to act as bases and a simple algebra to form more complex types. The final piece is an operation which allows us to span a base of values, we call this operation $\sharp$ (*sharp*). Members of this type represent linear combinations of values, and will be our *unknown qubits*. The $\sharp$ operator can be thought as the opposite of the exponential bang !. While the ! marks resources as non-linear, the $\sharp$ instead marks them as linear (i.e. non-duplicable). The type semantics is described in Table 3, where $\mathcal{S}_1$ is the set of norm-1 vectors. Ultimately, this allows us to prove which typing rules are valid. There are infinite valid rules, since typing rules are just theorems that we can prove, but we are interested in those which will help us to build a sufficiently expressive language.

Marking a lambda abstracion with a base B_X introduces the possibility of stopping progress. If the argument cannot be written as the linear combination of members of X, then the reduction is stuck. We do not interpret these stuck terms as meaningful values, so whenever we bind a variable we wish to discriminate these cases on the typing rules. In order to do this, we will require the type A of the arguments to be a subtype of the space generated by B_X (noted $A \leq \sharp B_X$). So every member of A can be written as a linear combination of values in B_X. Since we are dealing with sets of values as the semantic of our types, the subtyping relationship is merely the set inclusion.

$$[\![B_X]\!] = X \qquad [\![A \times B]\!] := \left\{(\vec{v},\vec{w}) : \vec{v} \Vdash [\![A]\!],\ \vec{w} \in [\![B]\!]\right\}$$
$$[\![\sharp A]\!] := ([\![A]\!]^\perp)^\perp \qquad [\![A \to B]\!] := \left\{\vec{w}\ :\ \forall \vec{v} \in [\![A]\!],\ \vec{w}\vec{v} \Vdash B\right\}$$

Where X is an orthogonal basis and $A^\perp = \{\vec{v} \in \mathcal{S}_1 \mid \forall \vec{w} \in A,\ \vec{v} \perp \vec{w}\}$

Table 3: Unitary semantics of types

$$\frac{}{x : A \vdash x : A}\ \text{(Axiom)} \qquad \frac{\vec{v} \in X \quad \flat X}{\vdash \vec{v} : B_X}\ \text{(Basis)} \qquad \frac{\Gamma \vdash \vec{t} : A \quad \Delta \vdash \vec{s} : B}{\Gamma, \Delta \vdash (\vec{t},\vec{s}) : A \times B}\ \text{(Pair)}$$

$$\frac{\Gamma, x : A \vdash \sum_{i=1}^n \alpha_i \vec{t_i} : B \quad A \leq \sharp B_X}{\Gamma \vdash \sum_{i=1}^n \alpha_i \lambda x_{B_X}.\, \vec{t_i} : A \Rightarrow B}\ \text{(UnitLam)} \qquad \frac{\Gamma \vdash \vec{s} : A \Rightarrow B \quad \Delta \vdash \vec{t} : A}{\Gamma, \Delta \vdash \vec{s}\,\vec{t} : B}\ \text{(App)}$$

$$\frac{\Gamma \vdash \vec{t} : A_1 \times A_2 \quad \Delta, x : A_1, y : A_2 \vdash \vec{s} : C \quad A_1 \leq \sharp B_X \quad A_2 \leq \sharp B_Y}{\Gamma, \Delta \vdash \mathtt{let}_{B_X \otimes B_Y}\ (x,y) = \vec{t}\ \mathtt{in}\ \vec{s} : C}\ \text{(LetPair)}$$

$$\frac{\Gamma \vdash \vec{t} : A_1 \otimes A_2 \quad \Delta, x : \sharp A_1, y : \sharp A_2 \vdash \vec{s} : C \quad A_1 \leq \sharp B_X \quad A_2 \leq \sharp B_Y}{\Gamma, \Delta \vdash \mathtt{let}_{B_X \otimes B_Y}\ (x,y) = \vec{t}\ \mathtt{in}\ \vec{s} : \sharp C}\ \text{(LetTens)}$$

$$\frac{\Gamma \vdash \vec{t} : B_{\{\vec{v_i}\}_{i=1}^n} \quad \forall i,\ \Delta \vdash \vec{s_i} : C}{\Gamma, \Delta \vdash \mathsf{case}\ \vec{t}\ \mathsf{of}\{\vec{v_1} \mapsto \vec{s_1} \mid \cdots \mid \vec{v_n} \mapsto \vec{s_n}\} : C}\ \text{(Case)}$$

$$\frac{\Gamma \vdash \vec{t} : \sharp B_{\{\vec{v_i}\}_{i=1}^n} \quad \forall i \neq j,\ \Delta \vdash \vec{s_i} \perp \vec{s_j} : C}{\Gamma, \Delta \vdash \mathsf{case}\ \vec{t}\ \mathsf{of}\{\vec{v_1} \mapsto \vec{s_1} \mid \cdots \mid \vec{v_n} \mapsto \vec{s_n}\} : \sharp C}\ \text{(UnitCase)}$$

$$\frac{\forall i \neq j, \Gamma \vdash \vec{t_i} \perp \vec{t_j} : A \quad \sum_{i=1}^n |\alpha_i|^2 = 1}{\Gamma \vdash \sum_{i=1}^n \vec{t_i} : \sharp A}\ \text{(Sum)}$$

$$\frac{\Gamma \vdash \vec{t} : B \quad \flat A}{\Gamma, x : A \vdash \vec{t} : B}\ \text{(Weak)} \qquad \frac{\Gamma, x : A, y : A \vdash \vec{t} : B \quad \flat A}{\Gamma, x : A \vdash \vec{t}[y := x] : B}\ \text{(Contr)}$$

Where the property $\flat$ is defined as: $\flat X \iff \forall \vec{v},\vec{w} \in [\![X]\!],\ \vec{v} \neq \vec{w} \Rightarrow \langle \vec{v} \mid \vec{w} \rangle = 0$

Table 4: Some valid typing rules

Once we have a type algebra defined, we can start proving the validity of some rules. This will ensure that the extracted system will be correct by construction. Intuitively, we will state that a judgement $\Gamma \vdash \vec{t} : A$ is valid when: "Every substitution σ such that $\sigma(x) \in [\![\Gamma(x)]\!]$ validates that $\sigma(\vec{t})$, reduces to $\vec{w} \in [\![A]\!]$". In the same way, a rule is valid if starting from valid hypotheses we arrive to a valid conclusion. In table 4 we state some of these rules.

Most of the rules are fairly standard, but we would like to highlight the UnitCase rule. There, we ask for an orthogonality judgement $\Delta \vdash \vec{s_i} \perp \vec{s_j} : C$. This means that "For every σ substitution such that $\sigma(x) \in [\![\Delta(x)]\!]$, $\sigma(\vec{s_i})$ and $\sigma(\vec{s_j})$ reduce to orthogonal values in $[\![C]\!]$". This is necessary to preserve the norm of the terms when the condition is a linear combination of the different possible vectors. The argument must be provided externally, so in a possible implementation, the programmer would be tasked with the responsibility of providing a proof

of orthogonality. This follows the line pioneered in [1] and the refinements from [4].

We will say a lambda term represents a function $F : \mathbb{C}^n \to \mathbb{C}^n$ if it encodes the action of F on vectors. With the typing judgements we have, we can characterize unitary operators as functions in the unitary semantic of types $\sharp B_X \Rightarrow \sharp B_Y$.

Theorem 2.3. *Let B_X, B_Y be orthonormal bases of size n. A closed λ-abstraction $\lambda x_X \,.\, \vec{t}$ is a value of type $\sharp B_X \to \sharp B_Y$ if and only if it represents a unitary operator $F : \mathbb{C}^n \to \mathbb{C}^n$.*

This result extends to unitary distributions of lambda abstractions since $\lambda x_X \,.\, \sum_{i=1}^n \alpha_i \vec{t_i}$ is syntactically different but computationally equivalent to $\sum_{i=1}^n \alpha_i (\lambda x_X \,.\, \vec{t_i})$.

2.2 Specification system

We found a somewhat interesting extreme when we restrict the sets forming the types to be singletons. The resulting system replicates the term reduction inside the strict typing rules and forms a sort of specification system. In short, sequents take the following form: $x_1 : \vec{v}_1, \cdots, x_n : \vec{v}_n \vdash \vec{t} : \vec{w}$ (We omit the brackets in singleton sets). And so, we can read the previous sequent as: "Every substitution σ such that $\sigma(x_i) = \vec{v_i}$ validates that $\sigma(\vec{t})$, reduces to $\vec{w}$". With this in mind, we can design inference rules that only use singleton sets as types. For example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : |+\rangle \vdash x : |+\rangle}\,(\text{Ax}) \quad \dfrac{}{y : |+\rangle \vdash y : |+\rangle}\,(\text{Ax})}{x : |+\rangle, y : |+\rangle \vdash (x, y) : |+\rangle \times |+\rangle}\,(\text{Pair})}{x : |+\rangle \vdash (x, x) : |+\rangle \times |+\rangle}\,(\text{Contr})}{\vdash \lambda x^{|+\rangle}.(x, x) : |+\rangle \to (|+\rangle \times |+\rangle)}\,(\text{Lam}) \quad \dfrac{}{\vdash |+\rangle : |+\rangle}\,(\text{Ax})}{\vdash (\lambda x^{|+\rangle}.(x, x))|+\rangle : |+\rangle \times |+\rangle}\,(\text{App})$$

This result is not surprising. When dealing with classic computation, regardless of the basis, if we restrict the possible inputs to only one option, we can statically determine the output without having to reduce the term.

However this changes when entangled qubits are involved. Entangled quantum states are states where a qubit cannot be described precisely and independently from another qubit. So for example the state $\frac{1}{\sqrt{2}}((|0\rangle, |0\rangle) + (|1\rangle, |1\rangle))$ cannot be described as a pair of two separate values $\vec{v}$ and $\vec{w}$. We can however use the let construct to destroy the pair, and in that case, we will have to type two separate variables. As expected, there is an inherent loss of precision due to the physical nature of the state, and this is reflected in the type we are able to assign. This indicates that the system is well constructed, as no singleton can correctly capture the value.

3 Conclusions and future work

With this work we aim to define a quantum lambda calculus that provides the implementer with more options when writing functions. The added information of the basis on the lambda abstractions helps to build a more informative and precise type system than previous iterations. An important feature of this system is the characterization of unitary operators via the semantic of the types $\sharp B_X \Rightarrow \sharp B_Y$.

A subproduct of the technique we used to prove the type judgements, is the specification system which in a way mirrors Hoare logic. This opens up a possible research line of defining a rigorous set of rules to reason about the correctness of quantum programs via realizability.

The next steps would be to give a categorical model to the calculus following the one in [5], to study its semantics and how it relates to other calculi. Another line of work would be to define a translation into an intermediate language such as ZX-calculus alongside the lines of [3]. Proving that, even though the programs are detached from the circuitry, they can still be implemented concretely.

Introducing the basis information can lead to programs that are more readable and intuitive. The obvious tradeoff is that when writing these programs one must have in mind the basis for each function and act accordingly. We tried to strike a balance where the additional information leads to a more precise typing without being too cumbersome to read.

References

[1] T. Altenkirch and J. Grattage. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, page 249–258. IEEE, 2005.

[2] Pablo Arrighi and Gilles Dowek. Lineal: A linear-algebraic Lambda-calculus. *Logical Methods in Computer Science*, Vol. 13, Issue 1, 03 2017.

[3] Agustin Borgna and Rafael Romero. Encoding high-level quantum programs as szx-diagrams. *Electronic Proceedings in Theoretical Computer Science*, 394:141–169, November 2023.

[4] Alejandro Díaz-Caro, Mauricio Guillermo, Alexandre Miquel, and Benoit Valiron. Realizability in the unitary sphere. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, June 2019.

[5] Alejandro Díaz-Caro and Octavio Malherbe. Quantum control in the unitary sphere: Lambda-s1 and its categorical model. *Logical Methods in Computer Science*, Volume 18, Issue 3, September 2022.

[6] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université de Paris 7, 1972.

[7] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. *ACM SIGPLAN Notices*, 48(6):333–342, June 2013.

[8] Jennifer Paykin, Robert Rand, and Steve Zdancewic. Qwire: a core language for quantum circuits. *SIGPLAN Not.*, 52(1):846–858, jan 2017.

[9] Peter Selinger and Benoît Valiron. *A Lambda Calculus for Quantum Computation with Classical Control*, page 354–368. Springer Berlin Heidelberg, 2005.

[10] William K. Wootters, William K. Wootters, and Wojciech H. Zurek. A single quantum cannot be cloned. *Nature*, 299:802–803, 1982.

Compression for Coinductive Infinitary Rewriting (A Preliminary Account)

Rémy Cerda[1] and Alexis Saurin[1,2]

[1] Université Paris Cité, CNRS, IRIF, F-75013 Paris, France
[2] INRIA π^3, Paris, France
Remy.Cerda@math.cnrs.fr, Alexis.Saurin@irif.fr

In "traditional" infinitary rewriting based on ordinal-indexed rewriting sequences and strong Cauchy convergence, a key property of rewriting systems is *compression*, that is, the fact that rewriting sequences of arbitrary ordinal length can be compressed to sequences of length ω. Famous examples of compressible systems are left-linear first-order systems and infinitary λ-calculi.

In this work, we investigate compression in the equivalent setting of coinductive infinitary rewriting, introduced by Endrullis et al. [End+18], which we recall in Section 1 in a slightly augmented form: the original work only covered first-order rewriting, we extend it to rewriting of (possibly non-wellfounded) derivations in an arbitrary sytem of derivation rules.

Then in Section 2 we define the coinductive counterpart of compressed rewriting sequences, and we present a general coinductive procedure turning arbitrary infinitary rewriting derivations into compressed ones, without relying on a topological formalism. The coinductive presentation of the two aforementioned examples, that is left-linear first-order systems and the full infinitary λ-calculus, are endowed with compression lemmas as instances of our general method.

This is a preliminary work, as our main motivation is to tackle the rewriting induced on non-wellfounded proofs by eliminating cuts. For future work, we will focus on the system $\mu\mathsf{MALL}^\infty$ for multiplicative-additive linear logic with fixed points, the cut-elimination theorem of which crucially relies on a compression lemma [Sau23]. In particular, we hope to be able to use a coinductive compression step as a component of a fully coinductive cut-elimination proof.

1 Coinductive infinitary rewriting

In this first section, we recall the coinductive presentation of infinitary first-order rewriting from [End+18]. Then we provide an extension of this presentation to infinitary λ-calculus, and to a generic notion of rewriting system for non-wellfounded derivations.

1.1 First order infinitary rewriting

Fix a countable set $\mathcal{V}$ of variables. A first-order signature is a countable set Σ equipped with an arity function $\mathrm{ar} : \Sigma \to \mathbf{N}$; we fix such a signature. The set T_Σ of first-order terms on this signature can be defined inductively by the derivation rules:

$$\frac{x \in \mathcal{V}}{x \in \mathrm{T}_\Sigma} \qquad \frac{t_1 \in \mathrm{T}_\Sigma \quad \dots \quad t_{\mathrm{ar}(\mathsf{c})} \in \mathrm{T}_\Sigma}{\mathsf{c}(t_1, \dots, t_{\mathrm{ar}(\mathsf{c})}) \in \mathrm{T}_\Sigma} \quad (\text{for each } \mathsf{c} \in \Sigma).$$

The set T^∞_Σ of infinitary first-order terms on the signature Σ can then be defined by completing T_Σ with respect to the metric defined by $d(s,t) := 2^{-(\text{the smallest depth at which } s \text{ and } t \text{ differ})}$, or equivalently by treating the above rules coinductively [Bar93] (which we will denote by using double inference bars).

In this setting, an (infinitary) rewrite rule is a couple (p, q) where $p \in \mathrm{T}_\Sigma$ and $q \in \mathrm{T}^\infty_\Sigma$. An infinitary term rewriting system (ITRS) is a countable set of rewrite rules; we fix an ITRS $\mathcal{R}$. Two terms $s, t \in \mathrm{T}^\infty_\Sigma$ are related by a rewrite step, denoted $s \longrightarrow t$, whenever there are a rule $(p, q) \in \mathcal{R}$, a (single-hole) context $u[*]$ and a substitution $\sigma : \mathcal{V} \to \mathrm{T}^\infty_\Sigma$ such that $s = u[\sigma \cdot p]$ and $t = u[\sigma \cdot q]$ (where $\sigma \cdot p$ denotes the substitution of each $x \in \mathcal{V}$ by $\sigma(x)$ in p).

A rewriting sequence of ordinal length γ is given by terms $(s_\delta)_{\delta \leq \gamma}$ together with rewrite steps $(s_\delta \longrightarrow s_{\delta+1})_{\delta<\gamma}$. Such a rewriting sequence is said to be strongly convergent if for all limit ordinal $\delta \leq \gamma$, $\lim_{\epsilon \to \delta} d(s_\epsilon, s_{\epsilon+1}) = 0$ and in addition, for all limit ordinal $\delta < \gamma$, the steps $s_\epsilon \longrightarrow s_{\epsilon+1}$ occur at depths tending to infinity when $\epsilon \to \delta$ [Ken+95].

Definition 1. We say that $s \longrightarrow^\infty t$ when there is an ordinal γ such that $s \underset{\gamma}{\longrightarrow}^\infty t$ is derivable in the following system of rules (wheres simple bars denote inductive inferences and double bars denote coinductive inferences, *i.e.* non-wellfounded derivations are allowed provided each infinite branch crosses infinitely often a double bar):

$$\frac{s \underset{\gamma,n}{\rightsquigarrow} s' \qquad s' \underset{\gamma}{\longrightarrow}^\infty t}{s \underset{\gamma}{\longrightarrow}^\infty t}\ (\mathsf{split}) \qquad \frac{x \in \mathcal{V}}{x \underset{\gamma}{\longrightarrow}^\infty x}\ (\mathsf{var}) \qquad \frac{\forall 1 \leq i \leq n,\ s_i \underset{\gamma}{\longrightarrow}^\infty t_i}{\mathsf{c}(s_1, \ldots, s_n) \underset{\gamma}{\longrightarrow}^\infty \mathsf{c}(t_1, \ldots, t_n)}\ (\mathsf{lift_c})$$

where $s \underset{\gamma,n}{\rightsquigarrow} s'$ denotes any sequence $s \longrightarrow^* s'_1 \underset{\delta_1}{\longrightarrow}^\infty t_1 \longrightarrow^* s'_2 \underset{\delta_2}{\longrightarrow}^\infty \ldots \underset{\delta_n}{\longrightarrow}^\infty t_n \longrightarrow^* s'$ such that $\forall 1 \leq i \leq n,\ \delta_i < \gamma$. (Notice that we are in fact defining *two* relations, namely $\underset{\gamma}{\longrightarrow}^\infty$ and $\underset{\gamma}{\longrightarrow}^\infty$, the latter one indicating that the former one occurs under a constructor.)

This coinductive presentation was introduced (with slightly different notations) by Endrullis et al. [End+18], who prove that it is equivalent to the "traditional", topology-based one. Indeed:

Theorem 2 ([End+18])**.** For $s, t \in \mathrm{T}^\infty_\Sigma$, there is a strongly converging rewriting sequence from s to t iff. $s \longrightarrow^\infty t$.

1.2 Infinitary λ-calculus

An identical path can be followed to present an infinitary λ-calculus. (Finite) λ-terms are the elements of the set Λ defined by:

$$\frac{x \in \mathcal{V}}{x \in \Lambda} \qquad \frac{x \in \mathcal{V} \qquad M \in \Lambda}{\lambda x.M \in \Lambda} \qquad \frac{M \in \Lambda \qquad N \in \Lambda}{MN \in \Lambda}$$

and the set Λ^∞ of infinitary λ-terms can be defined either by metric completion [Ken+97] or by treating these rules coinductively [EP13]. In both cases we work modulo α-equivalence (which raises some subtleties, see [Kur+13] for a formal treatment.)

As usual, the reduction $\longrightarrow$ of β-reduction is defined on Λ^∞ by $(\lambda x.M)N \longrightarrow M[N/x]$ for all $M, N \in \Lambda^\infty$ (where $M[N/x]$ denotes the term obtained by substituting N to each free x in M), as well as lifting to contexts. As above, we define:

Definition 3. For $M, N \in \Lambda^\infty$, we say that $M \longrightarrow^\infty N$ when there is an ordinal γ such that $M \underset{\gamma}{\longrightarrow}^\infty N$ is derivable with the rules (split), (var),

$$\frac{M \underset{\gamma}{\longrightarrow}^\infty M'}{\lambda x.M \underset{\gamma}{\longrightarrow}^\infty \lambda x.M'}\ (\mathsf{lift}_\lambda) \qquad \text{and} \qquad \frac{M \underset{\gamma}{\longrightarrow}^\infty M' \qquad N \underset{\gamma}{\longrightarrow}^\infty N'}{MN \underset{\gamma}{\longrightarrow}^\infty M'N'}\ (\mathsf{lift}_@).$$

Theorem 4. For $M, N \in \Lambda^\infty$, there is a strongly converging β-reduction sequence from M to N iff. $M \longrightarrow^\infty N$.

This can in fact be extended to all *abc*-infinitary λ-calculi from [Ken+97]. The corresponding work (but the missing piece we intend to add) is presented in [Cer24; Cer25].

1.3 Rewriting non-wellfounded derivations

Consider the set $\mathcal{D}$ of all derivations obtained from a set of rules of one of the following shapes:

$$\frac{p_1 \quad \dots \quad p_k}{c}\ (\mathsf{r}) \qquad \text{or} \qquad \frac{p_1 \quad \dots \quad p_k}{\overline{c}}\ (\mathsf{r})$$

and some rewriting relation $\longrightarrow$ on $\mathcal{D}$. Just as we did in Definitions 1 and 3, we can define a relation $\underset{\gamma}{\longrightarrow}^\infty$ on $\mathcal{D}$ by the rules (split) as well as:

$$\frac{}{d \underset{\gamma}{\longrightarrow}^\infty d}\ (\mathsf{refl}) \qquad \text{and} \qquad \frac{d_1 \underset{\gamma}{\longrightarrow}^\infty d'_1 \quad \dots \quad d_k \underset{\gamma}{\longrightarrow}^\infty d'_k}{d \underset{\gamma}{\longrightarrow}^\infty d'}\ (\mathsf{lift_r})$$

where d (resp. d') is a derivation concluded by a rule (r) as above, $d_1, \dots, d_k$ (resp. $d'_1, \dots, d'_k$) are the sub-derivations rooted at the premises of this rule, and $(\mathsf{lift_r})$ is coinductive whenever (r) is. We say that $d \longrightarrow^\infty d'$ whenever there is γ such that $d \underset{\gamma}{\longrightarrow}^\infty d'$.

Our presentations of first-order rewriting and β-reduction are instances of this construction. Furthermore, as we did in Theorems 2 and 4, one can show that $d \longrightarrow^\infty d'$ iff. there is a strongly convergent rewriting sequence from d to d'.

What remains to be investigated is whether this construction is compatible with validity criteria, that is, global criteria used on top of non-wellfounded derivation systems to sort out "incorrect" derivations, typically to avoid inconsistencies in non-wellfounded proof systems. Our hope is that reasonable validity criteria on the rewritten derivations can be transported to restrict the derivations defining $\longrightarrow^\infty$ in such a way that $\longrightarrow^\infty$ rewrites valid derivations into valid derivations.

2 Compression lemmas

Two standard instances of the Compression lemma, which we would like to transport to the coinductive setting we just presented, are the following:

Theorem 5 ([Ken+95])**.** Let $\mathcal{R}$ be a left-linear ITRS, that is, no variable occurs twice in the left component of a rule of $\mathcal{R}$. Then for all $s, t \in \mathrm{T}^\infty_\Sigma$, there is a strongly convergent rewriting sequence from s to t iff. there is such a sequence of length at most ω.

Theorem 6 ([Ken+97])**.** For all $M, N \in \Lambda^\infty$, there is a strongly convergent β-reduction sequence from M to N iff. there is such a sequence of length at most ω.

2.1 Compressed rewriting sequences, coinductively

Since the "length" of a derivation $d \longrightarrow^\infty d'$ is not defined, we first need to introduce a coinductive counterpart of strongly converging rewriting sequences of length ω, extending again a definition from [End+18].

Definition 7. With the notation of Section 1.3, a relation $\longrightarrow^{\omega}$ is defined on $\mathcal{D}$ by:

$$\frac{d \longrightarrow^{*} e \qquad e \longrightarrow\!\!\!\!\!\!\rightarrow^{\omega} d'}{d \longrightarrow^{\omega} d'}\ (\mathsf{split}^{\omega}) \qquad \frac{}{d \longrightarrow\!\!\!\!\!\!\rightarrow^{\omega} d}\ (\mathsf{refl}^{\omega}) \qquad \frac{d_1 \longrightarrow^{\omega} d'_1 \quad \dots \quad d_k \longrightarrow^{\omega} d'_k}{d \longrightarrow^{\omega} d'}\ (\mathsf{lift}^{\omega}_{\mathsf{r}})$$

Lemma 8. For $d, d' \in \mathcal{D}$, there is a strongly convergent rewriting sequence of length at most ω from d to d' iff. $d \longrightarrow^{\omega} d'$.

In particular, this construction defines relations $\longrightarrow^{\omega}$ on $\mathrm{T}^{\infty}_{\Sigma}$ and Λ^{∞}, capturing exactly strongly convergent sequences of rewriting steps through $\mathcal{R}$ and β-reductions, respectively.

A relation $\longrightarrow^{\infty}$ defined as in Section 1.3 has the (coinductive) compression property if any derivation of $d \longrightarrow^{\infty} d'$ can be turned into a derivation of $d \longrightarrow^{\omega} d'$, that is, if $\longrightarrow^{\infty} = \longrightarrow^{\omega}$. In particular for the relation $\longrightarrow^{\infty}$ on $\mathrm{T}^{\infty}_{\Sigma}$, the compression property can be obtained by translating $s \longrightarrow^{\infty} s'$ as a strongly convergent rewriting sequence (Theorem 2), compressing it (Theorem 5), and translating the compressed sequence again (Lemma 8). Similarly, the compression property in Λ^{∞} is a consequence of Theorems 4 and 6 and Lemma 8. However, we would like to build a direct, explicit proof, without resorting to ordinal-based infinitary rewriting.

2.2 A general proof structure

With the general notations from Section 1.3, consider the following properties:

$\mathfrak{P}_{\gamma,n}$: For all $d, d' \in \mathcal{D}$, if $d \underset{\gamma,n}{\rightsquigarrow} d'$ then there are $d'' \in \mathcal{D}$ and an ordinal $\delta < \gamma$ such that $d \longrightarrow^{*} d'' \underset{\delta}{\longrightarrow\!\!\!\!\!\!\rightarrow}^{\infty *} d'$.

$\mathfrak{Q}$: For any ordinal δ, if $\forall m \in \mathbf{N},\ \mathfrak{P}_{\delta,m}$ holds, then for any reduction $d'_n \underset{\delta}{\longrightarrow\!\!\!\!\!\!\rightarrow}^{\infty} e'_n \longrightarrow d'$ there is a $d''_n \in \mathcal{D}$ such that $d'_n \longrightarrow^{*} d''_n \underset{\delta}{\longrightarrow\!\!\!\!\!\!\rightarrow}^{\infty} d'$.

Theorem 9. If $\mathfrak{Q}$ holds then $\longrightarrow^{\infty}$ has the compression property.

Lemma 10. If $d \underset{\gamma}{\longrightarrow}^{\infty} e$, then for any ordinal $\epsilon \geq \gamma$ there is also a derivation of $d \underset{\epsilon}{\longrightarrow}^{\infty} e$.

Lemma 11. If $d \underset{\gamma}{\longrightarrow}^{\infty} e$ and $e \underset{\delta}{\longrightarrow}^{\infty} f$, then $d \underset{\epsilon}{\longrightarrow}^{\infty} f$ for $\epsilon := \max(\gamma + 1, \delta)$.

Lemma 12. If $\mathfrak{Q}$ holds then $\forall \gamma,\ \forall n \in \mathbf{N},\ \mathfrak{P}_{\gamma,n}$.

Proof. We proceed by well-founded induction over γ, and we suppose that $\forall \delta < \gamma,\ \forall m \in \mathbf{N},\ \mathfrak{P}_{\delta,m}$. Then we proceed by induction on $n \in \mathbf{N}$.

If $n = 0$ the result is immediate since $d \underset{\gamma,0}{\rightsquigarrow} d'$ means that $d \longrightarrow^{*} d'$. Otherwise, suppose that $d \underset{\gamma,n}{\rightsquigarrow} d'$. This can be decomposed as: $d \underset{\gamma,n-1}{\rightsquigarrow} d'_n \underset{\delta_n}{\longrightarrow\!\!\!\!\!\!\rightarrow}^{\infty} e'_n \longrightarrow^{*} d'$, with $\delta_n < \gamma$. Using $\mathfrak{Q}$ and the induction hypothesis (*i.e.* the fact that $\forall m \in \mathbf{N},\ \mathfrak{P}_{\delta_n,m}$ holds), there is a $d''_n \in \mathcal{D}$ such that $d \underset{\gamma,n-1}{\rightsquigarrow} d'_n \longrightarrow^{*} d''_n \underset{\delta_n}{\longrightarrow\!\!\!\!\!\!\rightarrow}^{\infty} d'$. Notice that $d \underset{\gamma,n-1}{\rightsquigarrow} d'_n \longrightarrow^{*} d''_n$ can be reformulated as $d \underset{\gamma,n-1}{\rightsquigarrow} d''_n$, whence we can apply the induction hypothesis on $n-1$ and obtain a term d'' and an ordinal $\delta' < \gamma$ such that $d \longrightarrow^{*} d'' \underset{\delta'}{\longrightarrow\!\!\!\!\!\!\rightarrow}^{\infty *} d''_n \underset{\delta_n}{\longrightarrow\!\!\!\!\!\!\rightarrow}^{\infty} d'$, which can be simplified as $d \longrightarrow^{*} d'' \underset{\delta}{\longrightarrow\!\!\!\!\!\!\rightarrow}^{\infty *} d'$ with $\delta := \max(\delta', \delta_n) < \gamma$ thanks to Lemma 10. □

Proof of Theorem 9. Suppose $\mathfrak{Q}$. We start with a derivation of $d \xrightarrow[\gamma]{}{}^{\infty} e$. It can only be obtained through the rule (split), hence $d \overset{\gamma,n}{\rightsquigarrow} d' \xrightarrow[\gamma]{}{}^{\infty} e$. Then:

1. We apply Lemma 12 to $d \overset{\gamma,n}{\rightsquigarrow} d'$, and obtain $d \longrightarrow^{*} d'' \xrightarrow[\delta]{}{}^{\infty *} d' \xrightarrow[\gamma]{}{}^{\infty} e$ for some $d'' \in \mathcal{D}$ and some ordinal $\delta < \gamma$.
2. We apply the transitivity Lemma 11, and obtain $d \longrightarrow^{*} d'' \xrightarrow[\gamma]{}{}^{\infty} e$.
3. We proceed coinductively in $d'' \xrightarrow[\gamma]{}{}^{\infty} e$, building $d \longrightarrow^{*} d'' \longrightarrow^{\omega} e$. We conclude with the rule (split$^{\omega}$). □

Our proof departs from the one presented in the Coq formalisation of [End+18] in three directions: first, as already stressed, it is parametric in the kind of rewriting we consider (whereas their proof only covers first-order rewriting); second, our definition of $\longrightarrow^{\infty}$ features ordinal annotations to constrain the use of coinduction, whereas their definition relies on mixing least and a greatest fixed points, which results in different treatments of the inductive part of the proof; third, our proof provides an explicit coinductive procedure for compressing derivations of infinitary rewritings. This suggests that compression may be computable, in a sense and under conditions that are yet to be made precise (which we leave for further work).

2.3 Back to our main examples

In the property $\mathfrak{Q}$, we isolated the precise step where the specific properties of the considered rewriting system come into play. Let us come back to the two previously described examples, that is left-linear ITRS and infinitary λ-calculus, and instantiate Theorem 9.

Lemma 13. If $\mathcal{R}$ is a left-linear ITRS, then the relation $\longrightarrow$ it defines on $\mathrm{T}^{\infty}_{\Sigma}$ satisfies the property $\mathfrak{Q}$.

Proof sketch. Let δ be an ordinal such that $\forall m \in \mathbf{N}$, $\mathfrak{P}_{\delta,m}$ holds, and consider a derivation of $s'_n \xrightarrow[\delta]{}{}^{\infty} t'_n \longrightarrow s'$. The last step can be described as $p[\sigma] \longrightarrow q[\sigma]$ for a substitution σ. The key observation is that since p is finite, we can analyse $s'_n \xrightarrow[\delta]{}{}^{\infty} t'_n$ inductively (using the hypothesis on δ when we meet $\overset{\delta,m}{\rightsquigarrow}$) and produce, on one hand a finite reduction $s'_n \longrightarrow^{*} p[\tau]$ for some substitution τ, on the other hand derivations $\tau(x) \xrightarrow[\delta]{}{}^{\infty} \sigma(x)$ for each $x \in \mathcal{V}$. This allows to conclude: $s'_n \longrightarrow^{*} p[\tau] \longrightarrow q[\tau] \xrightarrow[\delta]{}{}^{\infty} q[\sigma]$. The left-linearity assumption is used when we define τ: if a variable x appeared several times in p, then we might define $\tau(x)$ in several conflicting ways. □

Corollary 14. If $\mathcal{R}$ is a left-linear ITRS, then the relation $\longrightarrow^{\infty}$ it defines on $\mathrm{T}^{\infty}_{\Sigma}$ has the compression property.

The same holds in the infinitary λ-calculus:

Lemma 15. The relation $\longrightarrow$ defined on Λ^{∞} satisfies the property $\mathfrak{Q}$.

Proof sketch. Let δ be an ordinal such that $\forall m \in \mathbf{N}$, $\mathfrak{P}_{\delta,m}$ holds, and consider a derivation of $M'_n \xrightarrow[\delta]{}{}^{\infty} N'_n \longrightarrow M'$. If the last β-reduction step occurs at top-level, *i.e.* $N'_n = (\lambda x.P')Q'$ and $M' = P'[Q'/x]$, then by analysing $M'_n \xrightarrow[\delta]{}{}^{\infty} N'_n$ and using the hypothesis $\mathfrak{P}_{\delta,m}$ we are able to identify terms P, Q such that $M'_n \longrightarrow^{*} (\lambda x.P)Q \longrightarrow P[Q/x]$, as well as $P \xrightarrow[\delta]{}{}^{\infty} P'$ and

$Q \underset{\delta}{\longrightarrow}^{\infty} Q'$. For the last two hypotheses we are able to deduce that $P[Q/X] \underset{\delta}{\longrightarrow}^{\infty} P'[Q'/x] = N'_n$. In general the last redex occurs in context, *i.e.* $N'_n = C[(\lambda x.P')Q']$, and we have to scan this context inductively, collecting finite reductions $P_0 \longrightarrow^* P_1 \longrightarrow^* \ldots \longrightarrow^* P_K$ as in Lemma 13. □

Corollary 16. The relation $\longrightarrow$ defined on Λ^{∞} has the compression property.

In particular, notice that this justifies the coinductive definition of infinitary β-reduction as it is usually written, that is, using $\longrightarrow^{\omega}$ instead of the rather impractical $\longrightarrow^{\infty}$ [EP13; Cza20; Cer24].

References

[Bar93] Michael Barr. "Terminal coalgebras in well-founded set theory". In: *Theoretical Computer Science* 114.2 (1993), pp. 299–315. DOI: 10.1016/0304-3975(93)90076-6.

[Cer24] Rémy Cerda. "Taylor Approximation and Infinitary λ-Calculi". Theses. Aix-Marseille Université, 2024. URL: https://hal.science/tel-04664728.

[Cer25] Rémy Cerda. *Nominal Algebraic-Coalgebraic Data Types, with Applications to Infinitary λ-Calculi*. To appear in the proceedings of FICS 2024. 2025. URL: https://www.i2m.univ-amu.fr/perso/remy.cerda/fichiers/papiers/nominal-nu-mu-fics.pdf.

[Cza20] Łukasz Czajka. "A new coinductive confluence proof for infinitary lambda calculus". In: *Logical Methods in Computer Science* 16.1 (2020). DOI: 10.23638/LMCS-16(1:31)2020.

[End+18] Jörg Endrullis et al. "Coinductive Foundations of Infinitary Rewriting and Infinitary Equational Logic". In: *Logical Methods in Computer Science* 14.1 (2018), pp. 1860–5974. DOI: 10.23638/LMCS-14(1:3)2018.

[EP13] Jörg Endrullis and Andrew Polonsky. "Infinitary Rewriting Coinductively". In: *18th International Workshop on Types for Proofs and Programs (TYPES 2011)*. 2013, pp. 16–27. DOI: 10.4230/LIPIcs.TYPES.2011.16.

[Ken+95] Richard Kennaway et al. "Transfinite Reductions in Orthogonal Term Rewriting Systems". In: *Information and Computation* 119.1 (1995), pp. 18–38. DOI: 10.1006/inco.1995.1075.

[Ken+97] Richard Kennaway et al. "Infinitary lambda calculus". In: *Theoretical Computer Science* 175.1 (1997), pp. 93–125. DOI: 10.1016/S0304-3975(96)00171-5.

[Kur+13] Alexander Kurz et al. "Nominal Coalgebraic Data Types with Applications to Lambda Calculus". In: *Logical Methods in Computer Science* 9.4 (2013). DOI: 10.2168/lmcs-9(4:20)2013.

[Sau23] Alexis Saurin. "A Linear Perspective on Cut-Elimination for Non-wellfounded Sequent Calculi with Least and Greatest Fixed-Points". In: *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2023)*. 2023, pp. 203–222. DOI: 10.1007/978-3-031-43513-3_12.

Constructing a Curry Algebra where Indeterminates are not Left Cancellative

Enno Folkerts

SAP SE, Germany
enno.folkerts@sap.com

1 Introduction

The origin of this extended abstract is the question which property of universal algebra is best suited to model the role of free variables in term models of the lambda calculus. In some mathematical contexts variables and indeterminates are just the same thing. In this paper however, a variable is a term that can be turned into a semantic object when considering term models. Indeterminates on the other hand are defined by a universal property. They are semantic objects. The main result of this paper is the identification of a central property of universal algebra that holds for variables in the typical term models while we construct a combinatory algebra where this property does not hold for indeterminates. The property we are after is the left cancellation property as defined below.

Definition 1. *Let $\mathcal{A}$ be a combinatory algebra. Some $u \in \mathcal{A}$ is left cancellative if for all $a, b \in \mathcal{A}$*

$$\mathcal{A} \models u \cdot a = u \cdot b \text{ implies } a = b.$$

Left cancellative elements are oftentimes also called injective as the map FUN_u defined by $a \mapsto u \cdot a$ for $a \in \mathcal{A}$ is injective on $\mathcal{A}$.

The goal of the paper is to construct a combinatory algebra where all indeterminates are not left cancellative.

Left Cancellation and Term Models Before we move on to the construction of a combinatory algebra where all indeterminates are not left cancellative let us first look at the *natural* case of the term models $\mathcal{M}(\lambda)$ or $\mathcal{M}(\lambda\eta)$ where all (elements representing) variables are indeterminates and left cancellative. The basic notions of lambda theories and term model of a lambda theory are defined in [2], 4.1.1 and 4.1.17. In what follows we allow lambda theories $\mathcal{T}$ to also contain constants. The language $\mathcal{L}(\mathcal{T})$ of the theory $\mathcal{T}$ is uniquely determined by the set of constants used, and the definition below is meant to extend [2], 4.1.17 for cases with constants in the play.

Definition 2. *For a lambda theory $\mathcal{T}$ and a term M in the language of $\mathcal{T}$ let $[M]_{\mathcal{T}}$ denote the equivalence class defined by $\mathcal{T}$ equality. The term model $\mathcal{M}(\mathcal{T})$ has as elements the set $\{[M]_{\mathcal{T}} | M \in \mathcal{L}(\mathcal{T})\}$ and the operations are defined as in [2], 4.1.17.*

Lemma 1. *For each variable x the element $[x]_{\lambda\eta} \in \mathcal{M}(\lambda\eta)$ is left cancellative.*

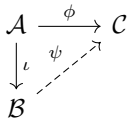
Proof. Assume $\mathcal{M}(\lambda\eta) \models [x]_{\lambda\eta} \cdot [M]_{\lambda\eta} = [x]_{\lambda\eta} \cdot [N]_{\lambda\eta}$. Then $\lambda\eta \vdash xM = xN$. Then $\lambda\eta \vdash M = N$ by CR and $\mathcal{M}(\lambda\eta) \models [M]_{\lambda\eta} = [N]_{\lambda\eta}$. □

Next we need to elaborate about indeterminates of a combinatory algebra. We start with the most general setting.

Indeterminates and Their Construction

Definition 3. *Let $\mathcal{V}$ be a variety of algebras, $\mathcal{A}, \mathcal{B} \in \mathcal{V}$ and $X \subset \mathcal{B}$. A homomorphism $\iota : \mathcal{A} \to \mathcal{B}$ is an extension by indeterminates X if for each homomorphism $\phi : \mathcal{A} \to \mathcal{C}$ and a map $h : X \to \mathcal{C}$ there is a unique homomorphism ψ such that*

1. $\psi \circ \iota = \phi$ *and*
2. $\psi\big|_X = h$.



Elements $x \in X$ are also called indeterminates of $\mathcal{B}$.

In the paper *The Lambda Calculus is Algebraic* [8] Selinger presents two different ways to construct extensions of a combinatory algebra $\mathcal{A}$ by a single indeterminate. The first construction ([8], page 4) is via polynomials in a variable z and denoted by $\mathcal{A}[z]$. In the second construction the combinatory algebra constructed is named $\mathcal{B}$ in proposition 8 of [8]. In case $\mathcal{A}$ is a Curry algebra (see [8], page 13) the second construction can be simplified and presented in the form below.

Definition 4. *(Explicit Extension) Let $\mathcal{A}$ be a Curry algebra.*
Then ∂A is the Curry algebra given by

1. $|\partial\mathcal{A}| := |\mathcal{A}|$ *and* $a \cdot_\partial b := \mathbf{S} \cdot a \cdot b$
2. $\mathbf{K}_\partial := \mathbf{K} \cdot \mathbf{K}$ *and* $\mathbf{S}_\partial := \mathbf{K} \cdot \mathbf{S}$

Then the map $\iota : \mathcal{A} \to \partial\mathcal{A}$ given by $\iota : a \mapsto \mathbf{K} \cdot a$ is a homomorphism and ι gives rise to an extension by indeterminates $\{\mathbf{I}\}$.

The polynomial and the explicit extension by a single indeterminate are isomorphic and can be expanded to extensions by n indeterminates. In case $\mathcal{T}$ is a lambda theory and the base algebra is the term model $\mathcal{M}(\mathcal{T})$ there is a third way to construct an extension by indeterminates. We can simply construct an extension by indeterminates via adding constants to the base theory.

Definition 5. *(Extension by Constants)*
Let $\mathcal{T}$ be a lambda theory and E be a set of new constants.

1. *Let $\mathcal{T}(E)$ denote the theory of generated by $\mathcal{T}$ and E.*
2. *Let $\iota_{\mathcal{T},E} : \mathcal{M}(\mathcal{T}) \to \mathcal{M}(\mathcal{T}(E))$ be the canonical mapping defined by $[M]_{\mathcal{T}} \mapsto [M]_{\mathcal{T}(E)}$.*

Lemma 2. *Let E be a finite set of new constants.*
Then $\iota_{\mathcal{T},E}$ defines an extension of indeterminates from $\mathcal{M}(\mathcal{T})$ to $\mathcal{M}(\mathcal{T}(E))$ by $\{[e]_{\mathcal{T}(E)} | e \in E\}$.

Note that this also holds for an infinite set of constants and also note that one can also use free variables for the construction.

The Proof Strategy We are now in the position to explain the strategy how to construct the counter model. We choose two distinct constants e_1, e_2 and having $E := \{e_1, e_2\}$ construct a sequence of Curry algebras and respective homomorphisms:

$$\mathcal{A}_0 \to_{\iota_1} \mathcal{A}_1 \to_{\phi} \mathcal{A}_2 \to_{\iota_3} \mathcal{A}_3 \quad \text{with}$$

1. $\mathcal{A}_0 := \mathcal{M}(\lambda\eta)$ - the term model of $\lambda\eta$.
2. $\mathcal{A}_1 := \mathcal{M}(\lambda\eta(E))$ - the term model of $\lambda\eta(E)$ and let $\iota_1 := \iota_{\lambda\eta,E}$.
3. Let $\mathcal{A}_2$ be the quotient model of $\mathcal{A}_1$ modulo a congruence relation $\simeq_c$ containing the pair $([\mathbf{S} \cdot \mathbf{I} \cdot e_1]_{\lambda\eta(E)}, [\mathbf{S} \cdot \mathbf{I} \cdot e_2]_{\lambda\eta(E)})$

4. Let ϕ be the homomorphism mapping an $a \in \mathcal{A}_1$ to its congruence class $\phi(a) \in \mathcal{A}_2$ so that $\phi([\mathbf{S}\cdot\mathbf{I}\cdot e_1]_{\lambda\eta(E)}) = \phi([\mathbf{S}\cdot\mathbf{I}\cdot e_2]_{\lambda\eta(E)})$.
5. Let $\mathcal{A}_3 := \partial\mathcal{A}_2$ be the *explicit* extension of $\mathcal{A}_2$ by a single indeterminate and ι_3 be the homomorphism ι associated with the explicit (see definition 4) extension from $\mathcal{A}_2$ to $\mathcal{A}_3$.

For the calculation below we write f_i for the interpretation of the function symbol f in $\mathcal{A}_i$. As $\mathcal{A}_3$ is the explicit extension $\partial\mathcal{A}_2$ of $\mathcal{A}_2$ we have $a \cdot_3 b = \mathbf{S}_2 \cdot_2 a \cdot_2 b$ for $a, b \in |\mathcal{A}_3| = |\mathcal{A}_2|$ and

$$\begin{array}{l}\mathbf{I}_2 \cdot_3 \phi([e_i]_{\lambda\eta(E)}) = \mathbf{S}_2 \cdot_2 \mathbf{I}_2 \cdot_2 \phi([e_i]_{\lambda\eta(E)}) = \phi(\mathbf{S}_1) \cdot_2 \phi(\mathbf{I}_1) \cdot_2 \phi([e_i]_{\lambda\eta(E)}) = \\ \phi([\mathbf{S}]_{\lambda\eta(E)}) \cdot_2 \phi([\mathbf{I}]_{\lambda\eta(E)}) \cdot_2 \phi([e_i]_{\lambda\eta(E)}) = \\ \phi([\mathbf{S}]_{\lambda\eta(E)} \cdot_1 [\mathbf{I}]_{\lambda\eta(E)} \cdot_1 [e_i]_{\lambda\eta(E)}) = \phi([\mathbf{S}\cdot\mathbf{I}\cdot e_i]_{\lambda\eta(E)})\end{array}$$

This implies

$$\mathcal{A}_3 \models \mathbf{I}_2 \cdot_3 \phi([e_1]_{\lambda\eta(E)}) = \mathbf{I}_2 \cdot_3 \phi([e_2]_{\lambda\eta(E)})$$

By construction of $\partial\mathcal{A}_2$ the element $\mathbf{I}_2$ is *the* indeterminate of $\mathcal{A}_3$. If we can prove that $\mathcal{A}_3 \not\models \phi([e_1]_{\lambda\eta(E)}) = \phi([e_2]_{\lambda\eta(E)})$ then $\mathbf{I}_2$ is not left cancellative. It is therefore sufficient to show that $[e_1]_{\lambda\eta(E)} \not\simeq_c [e_2]_{\lambda\eta(E)}$.

Left Invertibility as the Technical Key Concept From now on we will use the convention to identify equality classes $[M]_{\mathcal{T}}$ with their representing term M. This improves readability and we need to target $e_1 \not\simeq_c e_2$. In the presence of $\mathbf{S}\cdot\mathbf{I}\cdot e_1 \simeq_c \mathbf{S}\cdot\mathbf{I}\cdot e_2$ this can only work if there is no combinator L such that $\lambda\eta(E) \vdash L \cdot (\mathbf{S}\cdot\mathbf{I}\cdot e_i) = e_i$. That means there can be no L such that $\lambda\eta \vdash L(\mathbf{SI}x) = x$. Writing $M \circ N$ for $\mathbf{S}(\mathbf{K}M)N = \lambda x.M(Nx)$ there can be no L such that $\lambda\eta \vdash L \circ (\mathbf{SI}) = \mathbf{I}$. In other words $\mathbf{SI}$ can not be left invertible in the monoid $(\mathcal{M}(\lambda\eta), \circ)$. For more details about general, left and right invertibility the reader is referred to [1], Definition 9.2.

To complete the proof it is therefore sufficient to settle two things:

(LI1) The lambda term $\mathbf{SI}$ is not left invertible in $\mathcal{M}(\lambda\eta)$.

(LI2) There is a congruence relation $\simeq_c$ on $\mathcal{M}(\lambda\eta(E))$ such that $Fe_1 \simeq_c Fe_2$ for all combinators F that are not left invertible in $\mathcal{M}(\lambda\eta)$ while $e_1 \not\simeq_c e_2$.

This concludes the introduction. For further details and notations about the lambda calculus the reader is referred to [2]. Next, we will present explanations which kind of extended analysis of contexts is required before we then actually prove that $\mathbf{SI}$ is not left invertible. Finally, we will construct the congruence relation $\simeq_c$ on $\mathcal{M}(\lambda\eta(E))$ as required above.

2 Contexts and Bookkeeping of Bound Variables

Contexts of the lambda calculus are defined in [2], Definition 2.1.18. A more general treatise about contexts can be found in section 2.1.1 of [11]. Contexts help understand how occurrences of a term P in some other term M evolve when performing a reduction of M. They are the backbone for proofs of confluence as they help structure the cases to be considered. They are also important for the proof of (**LI1**) and (**LI2**) above. For (**LI1**) we need to be able to keep track of the order of bound variables capturing a hole $[\ ]$ of a context $C[\ ]$. As this can become very tedious for multi hole contexts we only consider single hole contexts $C[\ ]$.

Definition 6. *Let $C[\ ]$ be a context. Then $SBC(C[\ ])$ is the sequence of variables bound by the context $C[\ ]$.*

1. $SBC([\ ]) := \langle\rangle$
2. $SBC(\lambda x.C[\ ]) = \langle x\rangle * SBC(C[\ ])$.
3. $SBC(C[\ ]A) = SBC(C[\ ])$ *for* $C[\ ]$ *a context and* A *a lambda term.*
4. $SBC(AC[\ ]) = SBC(C[\ ])$ *for* $C[\ ]$ *a context and* A *a lambda term.*

This then helps answer the

First Context Challenge How can $C[M][x := N]$ be expressed as $C'[M']$ so that we can maintain control over the variables bound (capturing the hole of) by $C'[\ ]$?

Once this is mastered we can prove a vital lemma about the preservation of bindings:

Lemma 3. *Assume x does not occur in a context $C[\ ]$ and $C[xP]{\twoheadrightarrow_{\beta\eta}}x$.*
Then P reduces to a head normal form with a free head variable y among the sequence $SBC(C[\ ])$ of variables bound by $C[\ ]$.

Proof.
By induction on the length of the respective head reduction ending at $\lambda y_1 \dots y_k.xH_1 \dots H_k$. In the induction step we have the situation $C[xP] \equiv \lambda v_1 \dots v_n.(\lambda a.R)SQ_1 \dots Q_m$. If xP occurs within R we are in the scenario of the first context challenge and can proceed accordingly. Another interesting case is the occurrence of xP within S, where a single occurrence of xP can become multiplied within $R[a := S]$. That would lead to a multi hole context. But only one of the occurrences of xP within $R[a := S]$ can be the origin of the variable x in the head of $\lambda y_1 \dots y_k.xH_1 \dots H_k$. To properly deal with this case another context challenge needs to - and can - be mastered. □

Second Context Challenge How can we simply deal with the challenge that we do not consider multi hole contexts? The short answer is that the occurrences of a variable x that is substituted in but not bound by a context and which gets multiplied by substitution should be renamed to $x_1, \dots, x_n$. This way the distinct occurrences of x are marked by distinct terms x_i. If only one of the variables is relevant for a reduction under consideration we can move back to a single hole context scenario.

3 Showing that SI is not left invertible

Note that $\lambda\eta \vdash \mathbf{SI} = (\lambda xyz.xz(yz))\mathbf{I} = \lambda yz.z(yz)$. As in [6] let $\delta :\equiv \lambda yz.z(yz)$ and $\delta_y :\equiv \lambda z.z(yz)$. For a potential left inverse L of δ we get the equation $\lambda\eta \vdash L\delta_y = L(\lambda z.z(yz)) = y$. Then there would be (by [5], Corollary 2.5) a head reduction $L(\lambda z.z(yz)){\twoheadrightarrow_h}\lambda y_1 \dots y_k.yH_1 \dots H_k$ and $\lambda\eta \vdash H_i = y_i$. We want to prove that this is not possible, and we plan to do that by tracing the path of the occurrence (yz) in the associated head reduction. As the head reduction moves on the term (yz) may turn into a term (yP), but lemma 3 proves that P sticks to a free head variable z bound by the context around the (yP). To make the presentation below more compact we write $\overline{x}$ for terms of the form $\lambda y_1 \dots y_k.xH_1 \dots H_k$ with $\lambda\eta \vdash H_i = y_i$.

Lemma 4. *For x, y, P such that $y \notin FV(Px)$ we cannot have $P[x := \delta_y]{\twoheadrightarrow_h}\overline{y}$.*

Proof. By induction on the length of the head reduction. For the induction step assume $P[x := \delta_y]\overset{n+1}{\twoheadrightarrow_h}\overline{y}$. Then the length k of the head reduction path of P must be finite with $k \leq n + 1$.

Case 1: $k = n + 1$ Then there is a P' such that $P \to_h P' \xrightarrow{n}\!\!\!\!\twoheadrightarrow_h Q$ and consequently $P[x := \delta_y] \to_h P'[x := \delta_y] \xrightarrow{n}\!\!\!\!\twoheadrightarrow_h Q[x := \delta_y]$. We deduce $Q[x := \delta_y] \equiv \overline{y}$ and conclude by induction hypothesis - using $y \notin FV(P'x)$ - that this is not possible.

Case 2: $k \leq n$ Then x must be the head variable of $Q \equiv \lambda v_1 \dots v_p.xQ_1 \dots Q_r$.
Then $P[x := \delta_y] \xrightarrow{k}\!\!\!\!\twoheadrightarrow_h Q[x := \delta_y] \equiv \lambda v_1 \dots v_p.\delta_y Q_1[x := \delta_y] \dots Q_r[x := \delta_y]$. We now need to distinguish two cases:

1. The head variable in $\overline{y}$ traces back to the head occurrence of δ_y in $Q[x := \delta_y]$.
2. The head variable in $\overline{y}$ traces back to the other occurrences of δ_y in $Q[x := \delta_y]$.

The formal treatment of above case distinction leads us into the scenario of the second context challenge, can be processed accordingly and delivers two cases:

(head) $\lambda v_1 \dots v_l.\delta_{y_1} Q_1[x := \delta_{y_2}] \dots Q_r[x := \delta_{y_2}] \xrightarrow{n+1-k}\!\!\!\!\twoheadrightarrow_h \overline{y_1}$

(comp) $\lambda v_1 \dots v_l.\delta_{y_1} Q_1[x := \delta_{y_2}] \dots Q_r[x := \delta_{y_2}] \xrightarrow{n+1-k}\!\!\!\!\twoheadrightarrow_h \overline{y_2}$

Remember that $k \leq n$, so that $n + 1 - k > 0$ and $r > 0$. While (**comp**) relies on an easy induction step the (**head**) case is the core of the proof.

We have $\lambda v_1 \dots v_l.\delta_{y_1} Q_1[x := \delta_{y_2}] \dots Q_r[x := \delta_{y_2}] \xrightarrow{n+1-k}\!\!\!\!\twoheadrightarrow_h \overline{y_1}$. For $R_i :\equiv Q_i[x := \delta_{y_2}]$ we get $\lambda v_1 \dots v_l.(\lambda z.z(y_1 z))R_1 \dots R_r \to_h \lambda v_1 \dots v_l.(R_1(y_1 R_1))R_2 \dots R_r \xrightarrow{n-k}\!\!\!\!\twoheadrightarrow_h \overline{y_1}$ with y_1 not occurring in the R_i. Now use the context $C[\ ] :\equiv \lambda v_1 \dots v_l.(R_1([\ ])R_2 \dots R_r$ so that $C[y_1 R_1] \xrightarrow{n-k}\!\!\!\!\twoheadrightarrow_h \overline{y_1}$ and $SBC(C[\ ]) = \langle y_1, \dots, y_l \rangle$. Apply lemma 3 to see that R_1 reduces to a head normal form with a free head variable among the $v_1 \dots v_l$. But that means that $\lambda v_1 \dots v_l.(R_1(y_1 R_1))R_2 \dots R_r$ has a bound head variable among the $v_1 \dots v_l$. That is not possible as $\overline{y_1}$ has the free head variable y_1.

□

Corollary 1. SI *is not left invertible.*

Proof. Following the discussion at the beginning of this section assume there is an L such that $L \circ (\mathbf{SI}) = \mathbf{I}$. For new and distinct variables x, y apply above lemma for $P :\equiv Lx$ and δ_y. □

By CR it is easily seen that **SI** is left cancellative in $\mathcal{M}(\lambda\eta)$. Batenburg and Velmans were the first to present a term of this kind. They show in [3] that the left cancellative (injective) $F = \lambda xz.x(\lambda p.z(zp))$ does not possess a left inverse. Another trivial example is Ω.

4 Constructing the Congruence Relation

Following the discussion from the introduction we need to define a congruence relation $\simeq_c$ on $\mathcal{M}(\lambda\eta(E))$ satisfying $Fe_1 \simeq_c Fe_2$ for combinators F that are not left invertible. Note that we cannot in general allow $Fe_1 \simeq_c Fe_2$ for open terms F that are not left invertible. These terms do not possess good closure properties. Consider $F :\equiv \lambda x.xy$. Then F is not left invertible while $F' :\equiv \lambda xy.xy$ is (left) invertible. Allowing $Fe_1 \simeq_c Fe_2$ would deduce $e_1 = \lambda y.e_1 y = \lambda y.Fe_1 \simeq_c \lambda y.Fe_2 = \lambda y.e_2 y = e_2$ which we need to avoid. We want to introduce a new reduction $\to_c$ generating $\simeq_c$ so that $\beta\eta c$ is confluent. That means that we need to consider terms with free variables, and we are forced to introduce a new concept with better closure properties.

Definition 7. *(Inertness)*

1. *A lambda term F from $\Lambda(E)$ is inert if for a new variable x and arbitrary variables $y_1, \ldots, y_n$ the term $\lambda x y_1 \ldots y_n.Fx$ is not left invertible.*
2. *A context $C[\ ]$ from $\Lambda(E)$ is inert if the term $FUN_C :\equiv \lambda x.C[x]$ is inert for any new x.*

For closed terms F we have F is not left invertible if and only if F is inert. The context $\mathbf{SI}[\ ]$ is inert. The contexts $[\ ], [\ ]y$ and $y[\ ]$ are not inert while the contexts $[\ ]e_i$ are inert. We want to avoid $e_1 \simeq e_2$ while we will to allow $C[e_1] \simeq_c C[e_2]$ inert contexts $C[\ ]$. While $e_1 \simeq e_2$ is too strong for our needs we can allow $C[e_1] \simeq_c C[e_2]$ for inert contexts possessing only a limited functionality and shielding $e_1 \simeq e_2$ from being extracted. These contexts have the ability to weaken the unwanted strength of the *internal* relation $e_1 \simeq e_2$. In this paper we only deal with the simplistic *internal* relation $e_1 \simeq e_2$, but the methodology also works for more sophisticated *internal* relations.

We now introduce a reduction $\to_c$ on $\Lambda(E)$ generating the anticipated congruence relation $\simeq_c$.

Definition 8. $C[e_1] \to_c C[e_2]$ *if C is an inert context.*

Lemma 5. *Assume C is inert and let C' be some other context. Then $C'[C]$ is inert.*

Proof. The abstraction case follows from the built-in closure of inertness. The application cases are trivial. □

Lemma 6. *$\beta\eta c$ reductions are confluent and $e_1 \not\simeq_c e_2$ for the congruence generated by $\beta\eta c$.*

Proof. Use the Hindley-Rosen lemma ([11], 1.3.4) using that inert terms are closed under

1. exchanging constants - this implies the confluence of $\to_c$.
2. replacing constants by variables - this implies the commutativity of c and $\beta\eta$ reductions.

Then take into account that e_1 and e_2 are both in normal form. □

Theorem 1. *There is a Curry algebra $\mathcal{B}$ with indeterminate $u \in \mathcal{B}$ which is not left cancellative.*

Proof. Look back at the discussion in the introduction. Condition (**LI1**) is proved in corollary 1. Condition (**LI2**) is proved in lemma 6.

□

Final Remarks The idea of adding new variables and considering a quotient algebra generated by equations containing these new variables (or generators) is not new. The approach here takes some initial ideas from the presentation of groups as shown in [9].

We had expected to be able to deliver a fully algebraic proof of theorem 1, instead of that the proof relies on being able to work in term models, a deep analysis of the interplay of bound variables, contexts and reductions. It might be interesting to understand under which conditions the construction can be generalized to other types of models.

The introduction of this paper and the model finally presented might lead to the impression that indeterminates are not as well-behaved as variables. One might see the absolute interpretation of lambda terms as given in section 2.2 of [8] challenged. The author does not take this view. Considering an extension by indeterminates $\iota : \mathcal{A} \to \mathcal{B}$ by a single indeterminate x the behaviour

of x as indeterminate is determined by the model $\mathcal{A}$ and as good or bad as the behaviour of variables in $\mathcal{T}(\mathcal{A})$.

There are other research activities around the term **SI** and a bunch of papers [10],[6],[4],[7] attempt to show that **SI** is not a double fixed point combinator. There is no direct link from the present paper to this research activity.

Acknowledgements The author thanks the anonymous referees for their valuable input. Thanks also go to Femke van Raamsdonk, Giulio Manzonetto and Vincent van Oostrom for their kind guidance in the preparation phase of the paper.

References

[1] Henk Barendregt and Giulio Manzonetto. *A Lambda Calculus Satellite*. College Publications, 2022. URL: https://www.collegepublications.co.uk/logic/mlf/?00035.

[2] Henk Pieter Barendregt. *The lambda-calculus, its syntax and semantics*. Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, revised edition, 1984.

[3] A. Batenburg and J. Velmans. *Invertibility Properties of two λ-algebras*. Doctoraalskriptie, University of Utrecht, 1983.

[4] Jörg Endrullis, Dimitri Hendriks, Jan Willem Klop, and Andrew Polonsky. Clocked lambda calculus. *Math. Struct. Comput. Sci.*, 27(5):782–806, 2017. doi:10.1017/S0960129515000389.

[5] Enno Folkerts. Invertibility in $\lambda\eta$. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*, pages 418–429. IEEE Computer Society, 1998. doi:10.1109/LICS.1998.705676.

[6] Benedetto Intrigila. Non-existent Statman's double fixedpoint combinator does not exist, indeed. *Inf. Comput.*, 137(1):35–40, 1997. URL: https://doi.org/10.1006/inco.1997.2633, doi:10.1006/INCO.1997.2633.

[7] Giulio Manzonetto, Andrew Polonsky, Alexis Saurin, and Jakob Grue Simonsen. The fixed point property and a technique to harness double fixed point combinators. *J. Log. Comput.*, 29(5):831–880, 2019. URL: https://doi.org/10.1093/logcom/exz013, doi:10.1093/LOGCOM/EXZ013.

[8] Peter Selinger. The lambda calculus is algebraic. *J. Funct. Program.*, 12(6):549–566, 2002.

[9] Charles C. Sims. *Computation with finitely presented groups*, volume 48 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 1994.

[10] Richard Statman. Some examples of non-existent combinators. *Theor. Comput. Sci.*, 121(1&2):441–448, 1993. doi:10.1016/0304-3975(93)90096-C.

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

Proving Termination with CPO

Alejandro Díaz-Caro[1,2], Gilles Dowek[3], and Jean-Pierre Jouannaud[4]

[1] Université de Lorraine, CNRS, Inria, LORIA Nancy, France
[2] Universidad Nacional de Quilmes, Argentina. alejandro.diaz-caro@inria.fr
[3] Inria and ENS Paris-Saclay, France. gilles.dowek@ens-paris-saclay.fr
[4] ENS Paris-Saclay, France. jeanpierre.jouannaud@gmail.com

Abstract

In this paper, we show how to use the computability path ordering for proving termination of complex cut-elimination calculi aiming at providing a basis for quantum computations. A new CPO rule for CPO is introduced which allows to overcome the lack of transitivity of CPO –whose transitive closure is an order while CPO is not.

1 Introduction

A popular, effective simplification ordering for proving termination of rewrite rules in a first-order setting is the recursive path ordering (RPO, [2]). Its higher-order generalization, the computability path ordering (CPO, [1]) is as effective, but much less popular. In this paper, we show that it can be used in practice to save lots of complex termination arguments. Our examples originate from a higher-order calculus named $\mathcal{L}^{\mathcal{S}}$ modelling of quantum computations [3, 4].

The computability path ordering (CPO) is a well-founded relation over typed lambda terms that compares two terms by comparing their heads first, before recursing on their immediate subterms. As RPO, it uses a precedence on function symbols –including abstraction and application. Since the lambda calculus is non-terminating while its typed versions are terminating, CPO must also include type comparisons. Of course, CPO decreases when beta reducing a term, a rule common to all lambda calculi.

Applying CPO comparisons is easy once the precedence on function symbols and order on types are given, which makes it very easy to use. A difficulty, however, is that CPO is not transitive, hence is not an order, despite its name. In practice, it is sometimes necessary, when comparing two terms u, v to invent some *middle term* w and compare u with w and w with v. We propose here an additional CPO rule here that improves transitivity in practice.

2 CPO rules

We assume a simply typed λ-calculus generated by a set $\mathcal{F}$ of typed constants (the user's function symbols). We use $[x]u$ for abstractions, $@(u, v)$ for applications, and $\to$ for function types. Our examples describe lambda calculi, which will have their own abstraction, application and type constructors. We assume:

- a partial order on types $>_{\mathcal{T}}$, function types being strictly bigger than their arguments; RPO can be used here.
- a quasi-ordering $>_{\mathcal{F}}$ on $\mathcal{F}$, called *precedence*, whose strict part $>_{\mathcal{F}}$ is well-founded, and equivalence is denoted by $=_{\mathcal{F}}$;

- for every $f \in \mathcal{F}$, a status $stat(f) \in \{lex, mul\}$ such that symbols equivalent in $=_{\mathcal{F}}$ have the same status.

Definition 1 (core CPO). *CPO is the relation $\succ^{\varnothing}$ ($\succ$ for short) where:*

- *for any given finite set X of variables, $\succ^X$ is inductively defined in Figure 1;*
- *$t \succ^X_\tau u$ if $t \succ^X u$ and $\tau(t) \geq_{\mathcal{T}} \tau(u)$, also conveniently written in examples $t : \tau(t) \succ^X_\tau u : \tau(u)$.*

$$
\begin{array}{rl}
(\mathcal{F}\rhd) & f(\overline{t}) \succ^X v \text{ if } f \in \mathcal{F} \text{ and } s \succeq^X_\tau v \text{ for some } s \in \overline{t} \\
(\mathcal{F}=) & f(\overline{t}) \succ^X g(\overline{u}) \text{ if } f \in \mathcal{F},\ f =_{\mathcal{F}} g,\ (\forall i)\ f(\overline{t}) \succ^X u_i \text{ and } \overline{t}\,(\succ_\tau)_{stat(f)}\,\overline{u} \\
(\mathcal{F}>) & f(\overline{t}) \succ^X g(\overline{u}) \text{ if } f \in \mathcal{F},\ f >_{\mathcal{F}} g \text{ and } (\forall i)\ f(\overline{t}) \succ^X u_i \\
(\mathcal{F}@) & f(\overline{t}) \succ^X @(u,v) \text{ if } f \in \mathcal{F},\ f(\overline{t}) \succ^X u \text{ and } f(\overline{t}) \succ^X v \\
(\mathcal{F}\lambda) & f(\overline{t}) \succ^X [y]v \text{ if } f \in \mathcal{F},\ f(\overline{t}) \succ^{X\cup\{z\}} v^z_y,\ \tau(y) = \tau(z) \text{ and } z \text{ fresh} \\
(\mathcal{F}X) & f(\overline{t}) \succ^X y \text{ if } f \in \mathcal{F} \text{ and } y \in X \\
 & \\
(@\rhd) & @(t,u) \succ^X v \text{ if } t \succeq^X v \text{ or } u \succeq^X_\tau v \\
(@=) & @(t,u) \succ^X @(t',u') \text{ if } \left\{ \begin{array}{l} t = t' \text{ and } u \succ^X u', \text{ or} \\ @(t,u) \succ^X_@ t' \text{ and } @(t,u) \succ^X_@ u' \end{array} \right. \\
 & \text{where } @(t,u) \succ^X_@ v \text{ if } t \succ^X_\tau v \text{ or else } u \succeq^X_\tau v \text{ or} @(t,u) \succ^X_\tau v \\
(@\lambda) & @(t,u) \succ^X [y]v \text{ if } @(t,u) \succ^X v^z_y \text{ and } z \text{ fresh} \\
(@X) & @(t,u) \succ^X y \text{ if } y \in X \\
(@\beta) & @([x]t,u) \succ^X v \text{ if } t^u_x \succeq^X v \\
 & \\
(\lambda\rhd) & [x]t \succ^X v \text{ if } t^z_x \succeq^X_\tau v,\ \tau(x) = \tau(z) \text{ and } z \text{ fresh} \\
(\lambda\rhd X) & [x]t \succ^{X\cup\{z\}} v \text{ if } t^z_x \succeq^X_\tau v,\ \tau(x) = \tau(z) \text{ for some } z \in \mathcal{FV}ar(v) \\
(\lambda=) & [x]t \succ^X [y]v \text{ if } t^z_x \succ^X v^z_y,\ \tau(x) = \tau(y) = \tau(z) \text{ and } z \text{ fresh} \\
(\lambda\neq) & [x]t \succ^X [y]v \text{ if } [x]t \succ^{X\cup\{z\}} v^z_y,\ \tau(x) \neq \tau(y),\ \tau(y) = \tau(z) \text{ and } z \text{ fresh} \\
(\lambda X) & [x]t \succ^X y \text{ if } y \in X \\
(\lambda\eta) & [x]@(t,x) \succ^X v \text{ if } t \succeq^X v \text{ and } x \notin \mathcal{FV}ar(t)
\end{array}
$$

Figure 1: Core CPO

The parameter X serves as a meta-level binder to keep track of the variables that were previously bound in the right-hand side but have become free when destructuring a right-hand side abstraction. We say that a variable x is *fresh* with respect to a comparison $u \succ^X v$ if $x \notin X \cup FV(u,v)$.

Explicit variable renamings and the associated freshness conditions are used to make the relation invariant by α-equivalence and by appropriate renaming of the variables in X. Note that we must assume here that the expressions u and u^z_x have the same type, this is the substitution Lemma of the formalism.

This version of CPO is called core CPO. More elaborated versions allow for *small symbols* that behave differently from the *big symbols* used here, and for richer type theories, including in particular inductive types [1] and dependent types [6].

Theorem 2 ([1]). *The computability path ordering $\succ^X$ is (i) stable under instantiation, under α-conversion and under renaming of the variables in X away from the free variables of the compared terms; and (ii) the computability path ordering $\succ^{\varnothing}$ is monotonic and well-founded.*

Rule $(\lambda \triangleright X)$is new. Its proof is easily integrated to the CPO proof, see [5]. It serves resynchronizing abstractions when a previous use of rule $(\mathcal{F}\lambda)$ deconstructed an abstraction on the right-hand side while keeping the entire left-hand one. In practice, it allows us to shortcut some proofs requiring middle terms to overcome the lack of transitivity.

In the following, we use CPO to show termination of two rewrite systems, one for natural deduction [3], and one for the quantum in-left-right calculus [4].

2.1 Propositional natural deduction

We start with propositional natural deduction for which we give successively:

$[x]u$	:	$A \to B \to (A \to B)$		$@(u,v)$	:	$(A \to B) \to A \to B$	(core)
$\lambda(u)$	:	$(A \to B) \to (A \Rightarrow B)$		$u\ v$	:	$(A \Rightarrow B) \to A \to B$	(user's)
$\delta^1_\wedge(u,v)$	:	$(A \vee B) \to (A \to C) \to C$		$\delta^2_\wedge(u,v)$	:	$(A \vee B) \to (B \to C) \to C$	
$\delta_\vee(u,v,w)$	:	$(A \vee B) \to$ $(A \to C) \to (B \to C) \to C$		$\langle u,v\rangle$	:	$A \to B \to (A \wedge B)$	
$inl(u)$	:	$A \to A \vee B$		$inr(u)$	:	$B \to A \vee B$	

Figure 2: Signature

$$\frac{}{\Gamma \vdash x : A}\ \text{axiom } x : A \in \Gamma \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t,u\rangle : A \wedge B}\ \wedge\text{-i}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda([x]t) : A \Rightarrow B}\ \Rightarrow\text{-i} \qquad \frac{\Gamma \vdash t : A \Rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B}\ \Rightarrow\text{-e}$$

$$\frac{\Gamma \vdash t : A \wedge B \quad \Gamma, x : A \vdash u : C}{\Gamma \vdash \delta^1_\wedge(t,[x]u) : C}\ \wedge\text{-e1} \qquad \frac{\Gamma \vdash t : A \wedge B \quad \Gamma, x : B \vdash u : C}{\Gamma \vdash \delta^2_\wedge(t,[x]u) : C}\ \wedge\text{-e2}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash inl(t) : A \vee B}\ \vee\text{-i1} \qquad \frac{\Gamma \vdash u : B}{\Gamma \vdash inr(u) : A \vee B}\ \vee\text{-i2}$$

$$\frac{\Gamma \vdash t : A \vee B \quad \Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C}{\Gamma \vdash \delta_\vee(t,[x]u,[y]v) : C}\ \vee\text{-e}$$

Figure 3: Typing rules

$\delta_\top(\star,t)$	$\longrightarrow$	t		$\lambda([x]t)\ u$	$\longrightarrow$	t^u_x
$\delta^1_\wedge(\langle t,u\rangle,[x]v)$	$\longrightarrow$	v^t_x		$\delta^2_\wedge(\langle t,u\rangle,[x]v)$	$\longrightarrow$	v^u_x
$\delta_\vee(inl(t),[x]v,[y]w)$	$\longrightarrow$	v^t_x		$\delta_\vee(inr(u),[x]v,[y]w)$	$\longrightarrow$	w^u_y

Figure 4: Reduction rules

To show termination, we choose: for the order on types the recursive path ordering; for the precedence, the empty one; and no status is needed.

The first rule decreases by case $(\mathcal{F}\triangleright)$, since $t \succeq^{\varnothing}_{\tau} t$.

$a + b : \mathcal{S} \to \mathcal{S} \to \mathcal{S}$ $a \cdot \star : \mathcal{S} \to \mathtt{1}$	$a \times b : \mathcal{S} \to \mathcal{S} \to \mathcal{S}$ $\delta_{\mathtt{1}}(e, u) : \mathtt{1} \to A \to A$	$a \bullet u : \mathcal{S} \to A \to A$
$u \mathbin{+\!\!+} v : A \to A \to A$	$\lambda([x]u) : A \to B \to (A \multimap B)$	$u\ v : (A \multimap B) \to A \to B$
$inl(u) : A \to (A \oplus B)$	$inr(u) : B \to (A \oplus B)$	$inlr(u) : A \to B \to (A \oplus B)$
$\delta_{\oplus}(u, v, w) : (A \oplus B) \to (A \to C) \to (B \to C) \to C$ $\delta^{nd}_{\oplus}(u, v, w) : (A \oplus B) \to (A \to C) \to (B \to C) \to C$		

Figure 5: Signature

For the second rule, we have to prove that $\lambda([x]t)\ u \succ t^u_x$, for which we consider the middle term $@([x]t, u)$ which yields two goals, of which the second $@([x]t, u) \succ t^u_x$ is case $(@\beta)$. We are therefore left with the goal ($\succeq$ is enough since we have a strict comparison already):

$\lambda([x]t)\ u \succeq @([x]t, u)$, which yields two subgoals by case $(\mathcal{F}@)$:

1. $\lambda([x]t)\ u \succ [x]t$, which yields by case $(\mathcal{F}\triangleright)$ the subgoal
   1.1. $\lambda([x]t) : A \Rightarrow B \succeq_\tau [x]t : A \to B$, for which the type comparison $A \Rightarrow B >_\mathcal{T} A \to B$ succeeds since $\Rightarrow >_\mathcal{F} \to$, while the subgoal $\lambda([x]t) \succeq [x]t$ succeeds by $(\mathcal{F}\triangleright)$.
2. $\lambda([x]t)\ u \succ u$, which yields by $(\mathcal{F}\triangleright)$
   2.1. $u \succeq_\tau u$, which succeeds trivially.

The last four are all similar, using with the middle term $@([x]t, u)$. We carry out the last one only:

$\delta_\vee(inr(u), [x]v, [y]w) \succ @([y]w, u)$, which yields by $(\mathcal{F}>)$:

1. $\delta_\vee(inr(u), [x]v, [y]w) \succeq [y]w$, which succeeds by $(\mathcal{F}\triangleright)$.
2. $\delta_\vee(inr(u), [x]v, [y]w) \succ u$, which yields by $(\mathcal{F}\triangleright)$:
   2.1. $inr(u) : A \vee B \succeq_\tau u : A$, for which the comparison $A \vee B >_\mathcal{T} A$ succeeds as expected, while the subgoal $inr(u) \succeq u$ succeeds by $(\mathcal{F}\triangleright)$.

2.2 The quantum calculus $\mathcal{L}^\mathcal{S}$[4]

We describe the signature, precedence and statuses, before giving the rules.

Expressions of the calculus $\mathcal{L}^\mathcal{S}$ belong to four different syntactic categories, called sorts: $term, type, unit$, and $scalar$, where $type$ is the type of expressions of sort $term$. Note that $unit$ and $scalar$ expressions are both needed to express $type$ expressions. In the sequel, by type expression, we mean an expression of sort $type, unit$, or $scalar$.

We use a, b for arbitrary $scalar$ expressions, e for $unit$ expressions, $\mathtt{1}$ for the $unit$ sort, u, v, w, t for $term$ expressions, and A, B, C for expressions of sort $type$. We use the column to express membership to a sort, as in $e : \mathtt{1}$. We assume an order $>_s$ on sort expressions defined by $A >_s e >_s a$ if $A : type, e : \mathtt{1}$ and $a : scalar$.

We will use:

- the smallest order $>_\mathcal{T}$ on type expressions containing the order $>_s$ on sort expressions and the order $>_r$ such that $T \to U >_r U$, and such that $V >_\mathcal{T} V'$ implies $U \to V >_\mathcal{T} U \to V'$. This order is a well-founded order on type expressions that can be used to generate CPO (Lemma 2.3 in [4]);
- a multiset status for all function symbols;
- the signature given at Figure 5;

$\delta_1(a.\star, t)$	$\longrightarrow$	$a \bullet t$	$\lambda([x]t)\ u$	$\longrightarrow$	t^u_x
$\delta_\oplus(inl(t), [x]v, [y]w)$	$\longrightarrow$	v^t_x	$\delta_\oplus(inr(u), [x]v, [y]w)$	$\longrightarrow$	w^u_y
$\delta_\oplus(inlr(t,u), [x]v, [y]w)$	$\longrightarrow$	$v^t_x + w^u_y$	$\delta^{nd}_\oplus(inl(t), [x]v, [y]w)$	$\longrightarrow$	v^t_x
$\delta^{nd}_\oplus(inr(u), [x]v, [y]w)$	$\longrightarrow$	w^u_y	$\delta^{nd}_\oplus(inlr(t,u), [x]v, [y]w)$	$\longrightarrow$	v^t_x
$\delta^{nd}_\oplus(inlr(t,u), [x]v, [y]w)$	$\longrightarrow$	w^u_y			
			$a.\star + b.\star$	$\longrightarrow$	$(a+b).\star$
$\lambda([x]t) + \lambda([x]u)$	$\longrightarrow$	$\lambda([x](t+u))$	$inl(t) + inl(v)$	$\longrightarrow$	$inl(t+v)$
$inl(t) + inr(w)$	$\longrightarrow$	$inlr(t,w)$	$inl(t) + inlr(v,w)$	$\longrightarrow$	$inlr(t+v,w)$
$inr(u) + inl(v)$	$\longrightarrow$	$inlr(v,u)$	$inr(u) + inr(w)$	$\longrightarrow$	$inr(u+w)$
$inr(u) + inlr(v,w)$	$\longrightarrow$	$inlr(v,u+w)$	$inlr(t,u) + inl(v)$	$\longrightarrow$	$inlr(t+v,u)$
$inlr(t,u) + inr(w)$	$\longrightarrow$	$inlr(t,u+w)$	$inlr(t,u) + inlr(v,w)$	$\longrightarrow$	$inlr(t+v,u+w)$
$a \bullet b.\star$	$\longrightarrow$	$(a \times b).\star$	$a \bullet \lambda([x]t)$	$\longrightarrow$	$\lambda([x](a \bullet t))$
$a \bullet inl(t)$	$\longrightarrow$	$inl(a \bullet t)$	$a \bullet inr(t)$	$\longrightarrow$	$inr(a \bullet t)$
$a \bullet inlr(t,u)$	$\longrightarrow$	$inlr(a \bullet t, a \bullet u)$			

Figure 6: Rules

- the following precedence on function symbols:
  $\delta_1 >_\mathcal{F} \bullet >_\mathcal{F} \cdot\star >_\mathcal{F} \times$ and $\{\delta_\oplus, \delta^{nd}_\oplus\} >_\mathcal{F} + >_\mathcal{F} \{inl, inr, inlr, \cdot\star\} >_\mathcal{F} \lambda$;
- the rules given at Figure 6.

This set of rules is conjectured to be terminating in [4]. We claim here that it is indeed terminating, and that its proof can be entirely carried out with CPO, using $>_s$ for the order on type expressions, $>_\mathcal{F}$ for the order on function on the signature, and $\succ$ for the order CPO itself. We give a few comparisons below.

$\delta_1(a.\star, t) \succ a \bullet t$. By $(\mathcal{F}>)$, we get $\delta_1(a.\star, t) \succ a$ and $\delta_1(a.\star, t) \succ t$. The first subgoal yields $a.\star \succ_\tau a$ by $(\mathcal{F}\triangleright)$, which succeeds by $(\mathcal{F}\triangleright)$again, using the fact that units are bigger then scalars in $\geq_\mathcal{T}$. The second succeeds by $(\mathcal{F}\triangleright)$.

$\lambda([x]t)\ u \succ u^t_x$. As for the similar second rule of the first set, we use the middle term $@([x]t, u)$, and Rule $(\mathcal{F}@)$ to compare $\lambda([x]t)\ u$ with $@([x]t, u)$.

$\delta_\oplus(inl(t), [x]v, [y]w) \succ v^t_x$, and $\delta_\oplus(inr(u), [x]v, [y]w) \succ w^u_y$ are similar.

$\delta_\oplus(inlr(t,u), [x]v, [y]w) \succ v^t_x + w^u_y$. Starts with $(\mathcal{F}>)$, then similar.

$\delta^{nd}_\oplus(inl(t), [x]v, [y]w) \succ v^t_x$, $\delta^{nd}_\oplus(inr(u), [x]v, [y]w) \succ w^u_y$, $\delta^{nd}_\oplus(inlr(t,u), [x]v, [y]w) \succ v^t_x$, and $\delta^{nd}_\oplus(inlr(t,u), [x]v, [y]w) \succ w^u_y$ are similar again.

$a.\star + b.\star \succ (a+b).\star$. By $(\mathcal{F}>)$ twice, then $(\mathcal{F}\triangleright)$ twice.

$\lambda([x]t) + \lambda([x]u) \succ \lambda([x](t+u))$. That's where we will need the new rule $(\lambda\triangleright X)$. By $(\mathcal{F}\lambda)$, we get

1. $\lambda([x]t) + \lambda([x]u) \succ^z (t+u)^z_x = t^z_x + u^z_x$. By $(\mathcal{F}=)$, we get
   - 1.1. $\{\lambda([x]t), \lambda([x]u)\} \succ^z_\tau \{t^z_x, u^z_x\}$ which reduces to
     - 1.1.1. $\lambda([x]t) \succ^z_\tau t^z_x$, and by $(\lambda\triangleright X)$, we get
       - 1.1.1.1. $t^z_x \succeq_\tau t^z_x$, which succeeds.
     - 1.1.2. $\lambda([x]u) \succ^z_\tau u^z_x$, which is similar.

$inl(t) + inl(v) \succ inl(t+v)$. By $(\mathcal{F}>)$, then $(\mathcal{F}=)$, then $(\mathcal{F}\triangleright)$twice.

$inl(t) + inr(w) \succ inlr(t,w)$, $inl(t) + inlr(v,w) \succ inlr(t+v,w)$, $inr(u) + inl(v) \succ inlr(v,u)$, $inr(u) + inr(w) \succ inr(u+w)$, $inr(u) + inlr(v,w) \succ inlr(v,u+w)$, $inlr(t,u) + inl(v) \succ inlr(t+$

$v, u)$, $inlr(t, u) + inr(w) \succ inlr(t, u + w)$, and $inlr(t, u) + inlr(v, w) \succ inlr(t + v, u + w)$ are then all similar.

$a \bullet (b.\star) \succ ((a \times b).\star)$. By $(\mathcal{F}>)$ twice, then $(\mathcal{F}\triangleright)$ twice.

$a \bullet \lambda([x]t) \succ \lambda([x](a \bullet t))$. By $(\mathcal{F}>)$, we get

1. $a \bullet \lambda([x]t) \succ [x](a \bullet t)$, and by $(\mathcal{F}\lambda)$:
   1.1. $a \bullet \lambda([x]t) \succ^z a \bullet t^z_x$. By $(\mathcal{F}=)$, we get:
      1.1.1. $\{a, \lambda([x]t)\} \succ^z_\tau \{a, t^z_x\}$, which reduces to
         1.1.1.1. $\lambda([x]t) \succ^z_\tau t^z_x$, wich suceeds with $@([x]t, z)$ as middle term. Note that we could use $(\lambda\triangleright X)$as we did before. Both do succeed in this special case.

$a \bullet inl(t) \succ inl(a \bullet t)$. By successively $(\mathcal{F}>)$, $(\mathcal{F}=)$and $(\mathcal{F}\triangleright)$.

$a \bullet inr(t) \succ inr(a \bullet t)$ and $a \bullet inlr(t, u) \succ inlr(a \bullet t, a \bullet u)$ are similar.

3 Conclusion

In many cases, we have used a middle term to overcome the lack of transitivity of CPO. These cases are all similar, their shape is discussed in [1], where a tactic is suggested that would cover our uses of a middle term here. We have also used the new (re-synchronization) rule to carry out the termination proof of the calculus $\mathcal{L}^{\mathcal{S}}$, but we could have used a middle term as well in those specific cases, as we pointed out.

The calculus $\mathcal{L}^{\mathcal{S}}$ is a very interesting example which shows very well both indeed the flexibility and strength of CPO.

The well-foundedness proof of CPO is based on Girard's reducibility candidates, hence its name. It is therefore no wonder that CPO is so efficient to prove termination of sets of rules that describe various forms of cut elimination. CPO is implemented in Wanda [7], and possibly in other termination tools as well. We believe that it should be a must to implement CPO in all termination tools that exist on the market and participate to the termination competition.

Acknowledgements: To Cynthia Kop who suggested the use of a middle term in order to simplify one of the termination proofs.

This work has been partially supported by the European Union through the MSCA SE project QCOMICAL (Grant Agreement ID: 101182520), by the Plan France 2030 through the PEPR integrated project EPiQ (ANR-22-PETQ- 0007), and by the Uruguayan CSIC grant 22520220100073UD.

References

[1] Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. The computability path ordering. *Log. Methods Comput. Sci.*, 11(4), 2015. doi:10.2168/LMCS-11(4:3)2015.

[2] Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982. doi:10.1016/0304-3975(82)90026-3.

[3] Alejandro Diaz-Caro and Gilles Dowek. A linear linear lambda calculus. *Mathematical Structures in Computer Science*, 34(10):1103–1133, 2024. doi:10:1017/S096012952400019.

[4] Alejandro Diaz-Caro and Gilles Dowek. A new introduction rule for disjunctions. arXiv:2502.19172, 2025. URL: https://archiv.org/abs/2502.19172.

[5] Alejandro Díaz-Caro, Gilles Dowek, and Jean-Pierre Jouannaud. Termination. Extended abstract to be prsented at Workshop HOR, Birmingham, UK, July 14, 2025, 2025. URL: https://inria.hal.science/hal-05113173.

[6] Jean-Pierre Jouannaud and Jianqi Li. Termination of dependently typed rewrite rules. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 257–272. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. URL: https://doi.org/10.4230/LIPIcs.TLCA.2015.257, doi:10.4230/LIPICS.TLCA.2015.257.

[7] 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. URL: https://doi.org/10.4230/LIPIcs.FSCD.2020.36, doi:10.4230/LIPICS.FSCD.2020.36.

Towards the type safety of Pure Subtype Systems

Valentin Pasquale[1] and Álvaro García-Pérez[2]

[1] CEA List, Université Paris-Saclay, Palaiseau, France
valentin.pasquale@cea.fr
[2] CEA List, Université Paris-Saclay, Palaiseau, France
alvaro.garciaperez@cea.fr

Abstract

We reformulate the open problem of type safety in Hutchins' Pure Subtype Systems (PSS) by proving it under a different conjecture. PSS (hereafter in the singular) harmoniously mixes terms and types, thus enabling a number of advanced language features that combine dependent types with higher-order subtyping. In PSS, terms and types belong to the same kind (everything is a subtype) and the resulting theory is based on subtyping. However, conflating terms and types results in high impredicativity and complicates PSS meta-theory. The crucial property of type safety in PSS hinges on the well-known problem of transitivity elimination. Despite Hutchins' attempts, he failed to prove a key commutativity property, and type safety has remained an open problem for more than a decade. We introduce a reformulation of Hutchins' PSS, which we name Machine-Based Pure Subtype System (MPSS), which is based on a mechanism reminiscent of the Krivine Abstract Machine, where the proof of transitivity elimination is direct. Alas, in our MPSS, subtyping is scope-dependent and type safety rests on the conjecture that subtyping is congruent with covariant contexts. Our reformulation of type safety in PSS uncovers a dilemma about a commutativity/congruence problem, which sheds new light on possible solutions and evidences the difficulty of the challenge proposed by Hutchins.

Hutchins' Pure Subtype Systems (PSS for short) [3, 2] have been proposed as a novel approach to type theory that enables a number of advanced language features for extensibility and genericity which, among others, harmoniously solve the expression problem in both the functional and object-oriented (OO) programming styles [7, 8, 2]. PSS (hereafter in the singular) blurs the distinction between types and terms, and thus it naturally combines dependent types and higher-order subtyping, which makes the approach very promising as a basis for generic, modular, and extensible programming. However, a proof of type safety in PSS is lacking, and this crucial property is only shown to hold under the conjecture that two key reduction relations in the system commute [2]. Commutativity of the two reductions is an instance of a recurrent problem in higher-order subtyping known as *transitivity elimination*. Type safety in PSS has remained an open problem for more than a decade.

We introduce a reformulation of Hutchins' PSS that is based on an mechanism reminiscent of the Krivine Abstract Machine, where the proof of transitivity eliminaiton is direct. The paragraphs below elaborate on PSS and on how our contribution could pave the way for a proof of PSS type safety.

Contrary to traditional type systems, PSS harmoniously mixes terms and types. Consider the natural number 3 and the type *Nat* of natural numbers. In PSS one can have a term that encodes $Nat + 3$ with the naive Church encoding of naturals, which type-checks to *Nat*. Terms can also be used instead of types: so for instance the function $\lambda x \leq 3.x$, whose argument type is 3 (we write $\leq$ for the subtyping relation, this is, the substitute of typing in our calculus), is a valid expression. The latter example is an instance of *bounded quantification* [5] where the term 3 is the singleton type $\{3\}$ and the subtyping relation is akin to the subset relation.

PSS replaces the typing relation with a subtyping relation, which greatly increases expressivity. Since terms and types belong to the same kind, PSS naturally subsumes dependent types and higher-order subtyping, which enables a number of advanced language features for extensibility, genericity, and efficiency that include virtual types, recursive types, deep mixin composition, feature-oriented programming, bounded quantification, and partial evaluation [3, 8, 2].

In the original work on PSS, Hutchins introduces a *declarative* system with two relations, one for equivalence (akin to β-equivalence) and the other for subtyping, both of which are closed by transitivity. Type safety states a *type preservation* property that prescribes that the type of a program must be preserved by the evaluation of the program. Type preservation hinges on an inversion lemma that states that any two functions in the subtyping relation must take operands of equivalent subtypes. This inversion lemma cannot be proven directly when the subtyping derivation relating the two functions contains transitive steps, because of the intermediary terms that may not be structurally related to the two functions, and type preservation amounts to showing that transitivity is admissible in the declarative system.

In order to prove transitivity elimination, Hutchins proposes an equivalent syntax-directed, *algorithmic* system with two different notions of reduction: an *equivalence reduction* that models β-reduction, and a *subtyping reduction* that models small-step subtyping, where subtyping reduction subsumes equivalence reduction. For short, we may write "reduction" for the β-reduction, and "promotion" for the subtyping reduction proper that goes beyond β-reduction. The declarative subtyping relation is shown to be equivalent to the combination of subtyping and equivalence reduction sequences as depicted below.

$$u \leq t \quad \text{iff} \quad \text{exists } v \text{ such that} \qquad \begin{array}{ccc} t & \xrightarrow{\equiv} & v \\ & & \leq \uparrow \\ & & u \end{array}$$

In the algorithmic setting, transitivity can be shown to be admissible if these two notions of reduction commute. Diagrammatically, commutativity ensures that a transitive derivation comprising a "stair" with several steps as the "angle" above can be flattened into a single angle.

Despite his efforts, Hutchins failed to show that commutativity holds in his algorithmic system, which prevented him from proving transitivity elimination and type safety. The culprit of this failure is exemplified by a very elementary case (depicted below), which involves the reduction of a redex and the promotion of the formal parameter in the redex's abstraction to its annotation.

$$\begin{array}{ccc} (\lambda x \leq t.t)\, v & \xrightarrow{\equiv} & t \\ \leq \uparrow & & ? \vdots \\ (\lambda x \leq t.x)\, v & \xrightarrow{\equiv} & v \end{array}$$

The redex $(\lambda x \leq t.x)v$ on the bottom-left corner of the diagram is reduced (in the horizontal) to v, and promoted (in the vertical) to $(\lambda x \leq t.t)v$. Under the assumption of well-formedness, $v \leq^* t$ holds in the declarative system, but that judgement may entail transitive steps, which become a path in the shape of a stair when translated to the algorithmic system, and the right edge of the diagram cannot be completed with elementary subtyping reduction, thus preventing a direct proof of the diamond property.

To tackle this problem, Hutchins resorted to simultaneous reduction and to the *decreasing diagrams* technique of [6], with the aim of proving weak commutativity (a result weaker than the diamond property, but still enough to prove transitivity elimination). But he failed to assign

depths to each reduction step so as to show that reduction decreases in the way prescribed by the decreasing diagrams technique, because β-reduction increases this depth. Hutchins himself explains this failure in detail in his PhD thesis (Section 2.7: Confluence and commutativity)[2].

We reformulate the PSS theory by providing an alternative version of the system with similar expressive power but with a more fine-grained notion of subtyping, in which we can now prove the commutativity conjecture assumed in Hutchins' works. Our version of PSS, which we dub *Machine-Based PSS* (MPSS for short), uses a continuation stack reminiscent of the Krivine Abstract Machine (KAM) [4, 1] to keep track of arguments that have been passed to abstractions, which enables a direct proof of Hutchins' conjecture on commutativity. Indeed, thanks to the stack that keeps track of arguments that have been passed at a given scope, the subtyping relation in our MPSS exposes some intermediary terms that are absent in Hutchins' formulation. These intermediary terms allow us to complete a direct proof of commutativity without the decreasing diagrams technique.

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of International Conference on Principles and Practice of Declarative Programming*, pages 8–19, 2003.

[2] DeLesley Hutchins. Pure subtype systems: A type theory for extensible software. 2009.

[3] DeLesley S. Hutchins. Pure subtype systems. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 287–298. Association for Computing Machinery, 2010.

[4] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[5] Benjamin Pierce and Martin Steffen. Higher-order subtyping. *Theoretical computer science*, 176(1-2):235–282, 1997.

[6] Vincent Van Oostrom. Confluence by decreasing diagrams. *Theoretical computer science*, 126(2):259–280, 1994.

[7] Philip Wadler. The expression problem, 1998. Posted to Java Genericity internet mailing list.

[8] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *Workshop on Foundations of Object-Oriented Languages*, 2005.