The 2007 Federated Conference on
Rewriting, Deduction and Programming

Paris, France

June 25 – 29, 2007

HOR 2007

4th International Workshop on Higher-Order Rewriting

June 25, 2007

Proceedings

Editor:
Ralph Matthes

Preface

This is the proceedings volume of the 4th International Workshop on Higher-Order Rewriting (HOR 2007). 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. HOR 2002 was a part of FLoC 2002 in Copenhagen, Denmark; HOR 2004 was part of the RDP 2004 in Aachen, Germany; HOR 2006 was part of FLoC 2006 in Seattle, USA. HOR 2007 is part of RDP 2007 in Paris, France. This fourth installment of HOR enjoys additionally the status of a "small workshop" of the TYPES project.

The present volume provides final versions of six accepted contributed extended abstracts and short abstracts of all talks selected for the workshop. This also includes the talks by the invited speakers Carsten Schürmann (IT University of Copenhagen, Denmark) and Tarmo Uustalu (Institute of Cybernetics, Tallinn University of Technology, Estonia) and the system demo by Kristoffer Rose (IBM Thomas J. Watson Research Center, Yorktown Heights, USA).

Acknowledgements

I would like to thank the institutional sponsors of RDP'07 without whom it would not have been possible to organize RDP'07: the Conservatoire des Arts et Métiers (CNAM), the Centre National de la Recherche Scientifique (CNRS), the École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise (ENSIEE), the GDR Informatique Mathématique, the Institut National de Recherche en Informatique et Automatique (INRIA) unit Futurs, and the Région Île de France.

For the present workshop, I gratefully acknowledge the generous funding through the project TYPES (Coordination Action 510996 of the 6th Framework Programme of the EU), see `http://www.cs.chalmers.se/Cs/Research/Logic/Types/`, that additionally gave this fourth instance of the HOR workshop series the status of a "small workshop" and even funded the invited speaker that is not participating in the project. My personal thanks go to the project coordinator Bengt Nordström and the other members of the TYPES steering committee: Peter Aczel, Herman Geuvers, Zhaohui Luo, Christine Paulin-Mohring and Randy Pollack.

Many thanks to the program committee that I had the pleasure to chair, consisting of Herman Geuvers, Makoto Hamana, Albert Rubio and Mark-Oliver Stehr. The tasks were the decisions on the invited speakers, a timely refereeing of the submitted abstracts and the final decisions on the program. A heartfelt thank you to the HOR workshop series steering committee, Delia Kesner and Femke van Raamsdonk, who offered me to take this responsibility for the higher-order rewriting community and gave valuable advice throughout.

Thanks go to all the authors of abstract submissions, whether accepted or not. They were the raw material to shape this scientific meeting. A big thank you to the invited speakers Carsten Schürmann and Tarmo Uustalu that accepted this invitation to come to Paris despite heavy other obligations, and to Kristoffer Rose for his willingness to prepare a system demo that complements his talk.

I am much obliged to Andrej Voronkov and the support team of EasyChair for this marvellous tool free of charge that is very beneficial for the program

committee and in particular its chair already for small workshops such as HOR.

Finally, let me thank the organizers of the RDP'07 conference, Antonio Bucciarelli, Vincent Padovani, Ralf Treinen and Xavier Urbain who took care of all "physical" aspects of the organization, in particular Ralf Treinen in charge of the satellite workshops, even including the printing of these proceedings.

June 2007
Ralph Matthes, IRIT (CNRS and University Paul Sabatier, Toulouse, France)

Contents

On the formalization of logical relation arguments in Twelf

Carsten Schürmann
IT University of Copenhagen, Denmark

Abstract of HOR 2007 talk as invited speaker on June 25, 2007

Tait's method (a. k. a. proof by logical relations) is a powerful proof technique frequently used for showing foundational properties of languages based on typed lambda-calculi. Historically, these proofs have been difficult to formalize in Twelf, in part because logical relations are difficult to define judgementally.

In this talk I discuss how we did it, and what lessons we have learned about the relationship between logical frameworks, the consistency of logical systems, well-founded orderings, and proof theory.

This is joint work with Jeffrey Sarnat.

Circular proofs = Mendler in sequent form

Tarmo Uustalu
Institute of Cybernetics
Tallinn University of Technology, Estonia

Abstract of HOR 2007 talk as invited speaker on June 25, 2007

I present an interesting application of higher-order typed equational reasoning to structural proof theory. First I describe a simply typed lambda-calculus with inductive and coinductive types and guarded (co)recursion in the style of N. P. Mendler. This is a higher-order natural deduction system. Guardedness is expressed with type polymorphism. Then I show that remoulding this system as a higher-order sequent calculus yields a reconstruction with a solid foundation of Cockett and Santocanale's calculus of circular proofs (cf. Dam and Sprenger, Brotherston and Simpson).

Syntactical Strong Normalization for Intersection Types with Term Rewriting Rules

Andreas Abel*
Institut für Informatik
Ludwig-Maximilians-Universität München

June 4, 2007

Abstract

We investigate the intersection type system of Coquand and Spiwack with rewrite rules and natural numbers and give an elementary proof of strong normalization which can be formalized in a weak metatheory.

1 Introduction

For typed λ-calculi which are used as languages for theorem provers, such as Agda, Coq, LEGO or Isabelle, *normalization* is a crucial property; the consistency of these provers depend on it. Usually, normalization is proven by a model construction, but recently, syntactical normalization proofs have received some interest [Val01, Dav01, JM03]. One advantage of syntactical proofs is that they explain better why a calculus is normalizing; in such proofs one can see what actually decreases in each reduction step. Another advantage is that they can be formalized in weak logical theories. For instance, a syntactic normalization proof [Abe04] of the simply-typed λ-calculus (STL) can be carried out in Twelf, a logical framework supporting higher-order abstract syntax, whose proof-theoretic strength is probably ω^{ω^ω}, well below primitive recursive arithmetic. The insight that normalization of very weak languages, like the STL, can be proven using just lexicographic structural induction over Σ_1-sentences, has recently lead to a full formalization of an intermediate language for SML in Twelf [LCH07].

In this work, we consider a λ-calculus with simple and intersection types and term rewriting and show strong normalization by *structural* means, that is, no model construction, instead finitary inductive definitions and lexicographic induction. For intersection types without term rewriting, similar normalization proofs exist [Val01, Mat00]. The present system originates from work of Coquand and Spiwack [CS06]; there it serves as the basis of a filter λ model which ultimately shows normalization of a dependently-typed logical framework with bar recursion. The filter λ model "translates" a term of the logical framework into sets of finite types the term can receive in the intersection type system. If all intersection-typable terms are strongly normalizing, then so are all terms of

*Research partially supported by the EU coordination action *TYPES* (510996).

the logical framework whose denotation is not the empty set of types in the filter λ-model. Coquand and Spiwack prove normalization for the intersection type system using reducibility candidates; however, there should be a proof with weaker means. As Berger [Ber05] explained to me, the filter λ model which links the (strong) logical framework with the (presumably weak) intersection type systems uses proof-theoretically heavy tools, hence, the link between the intersection type system and strong normalization should be a lightweight one. This works tries to substantiate Berger's intuition.

2 Intersection Type System for Term Rewriting

As language, we consider the λ-calculus with constructors and functions defined by rewriting. For simplicity, we consider only the nullary constructor 0 and the unary constructor \$ (successor) for natural numbers and functions f with rewrite rules of the shape

$$\begin{array}{lcll} f(0) & \longrightarrow & \underline{z} & \underline{z} \text{ closed} \\ f(\$x) & \longrightarrow & \underline{s} & \mathsf{FV}(\underline{s}) \subseteq \{x\}. \end{array}$$

The right hand sides $\underline{z}$ and $\underline{s}$ may mention f and other defined functions, thus, recursion is *a priori* unrestricted. Although we only consider natural numbers in the following, the techniques extend to all first-order data types, such as lists or finitely branching trees. Higher-order data types, such as infinitely branching trees and tree ordinals pose yet some technical problem. Thus, we do not cover the full language of Coquand and Spiwack. Note however that first-order datatypes (natural numbers and lists) are sufficient to treat the main application of the Coquand and Spiwack's filter λ model, strong normalization of bar recursion.

The intersection type system does not know a type Nat of all natural numbers, however, it has one singleton type for each a natural number. We use the same constructors 0 and \$ for these singleton types. The type E is the least type in the subtyping relation, it is inhabited by terms blocking reduction, such as $f(\lambda xt)$.

Types. Let I, J, K denote non-empty finite index sets and let i, j, k range over indices.

$$\begin{array}{lcll} a, b, c & ::= & \mathsf{E} \mid 0 \mid \$a & \text{ground types} \\ A, B, C & ::= & a \mid \bigcap_{i\in I}(A_i \to B_i) & \text{types} \end{array}$$

The type $A \to B$ is a special case of the last alternative. A function type is a partial description of the graph of the function, for instance, the identity λxx could receive the type $(0 \to 0) \cap (\$0 \to \$0) \cap (\$\$0 \to \$\$0)$, or more generally the type $\bigcap_{i\in I}(\$^i 0 \to \$^i 0)$ for any finite set I of natural numbers.

A binary intersection $A \cap B$ is an associative commutative idempotent operation definable by induction on A and B.

$$\begin{array}{rcl} \mathsf{E} \cap A = \mathsf{E} & \qquad & 0 \cap 0 = 0 \\ 0 \cap \$a = \mathsf{E} & & \$a \cap \$a' = \$(a \cap a') \\ 0 \cap \bigcap_{i\in I}(A_i \to B_i) = \mathsf{E} & & (A \to B) \cap (A \to B') = A \to (B \cap B') \\ \$a \cap \bigcap_{i\in I}(A_i \to B_i) = \mathsf{E} & & \big(\bigcap_{i\in I}(A_i \to B_i)\big) \\ & & \cap \big(\bigcap_{i\in J}(A_i \to B_i)\big) = \bigcap_{i\in I\cup J}(A_i \to B_i) \end{array}$$

In the last clause, we assume all A_i for $i \in I \uplus J$ different. This invariant can be ensured using the but-last clause.

A measure on types $|A|$ is defined by $|a| = 0$ and $|\bigcap_{i\in I}(A_i \to B_i)| = \max\{|A_i|+1, |B_i| \mid i \in I\}$.

Subtyping $A \subseteq B$ is inductively given by the following rules [CS06].

$$\frac{}{\mathsf{E} \subseteq A} \qquad \frac{}{0 \subseteq 0} \qquad \frac{a \subseteq b}{\$a \subseteq \$b}$$

$$\frac{A \subseteq B_i \to C_i \text{ for all } i \in I}{A \subseteq \bigcap_{i\in I}(B_i \to C_i)} \qquad \frac{\left(\bigcap_{i\in J} B_i\right) \subseteq B}{\bigcap_{i\in I}(A_i \to B_i) \subseteq A \to B}\ J = \{i \mid A \subseteq A_i\} \neq \emptyset$$

This definition of subtyping is syntax-directed, however, it coincides with the usual axiomatic presentation: reflexivity, transitivity, and the usual rules for binary intersection, $A_1 \cap A_2 \subseteq A_i$ and $A \subseteq A_1 \;\&\; A \subseteq A_2 \implies A \subseteq A_1 \cap A_2$, are admissible; and the contravariant subtyping rule for function spaces is an instance of the last rule with $I = J = \{1\}$.

Typing The rules for the typing judgement $\Gamma \vdash t : A$ are taken from Coquand and Spiwack [CS06] and restricted to our set of constructors and function symbols. The first five rules are just intersection typing, the other five rules deal with constructors and functions.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x t : A \to B} \qquad \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\, s : B}$$

$$\frac{\Gamma \vdash r : A \qquad \Gamma \vdash r : B}{\Gamma \vdash r : A \cap B} \qquad \frac{\Gamma \vdash r : A \qquad A \subseteq B}{\Gamma \vdash r : B}$$

$$\frac{}{\Gamma \vdash 0 : 0} \qquad \frac{\Gamma \vdash r : a}{\Gamma \vdash \$r : \$a} \qquad \frac{\Gamma \vdash r : A}{\Gamma \vdash f(r) : \mathsf{E}}\ A \neq 0, \$a$$

$$\frac{\Gamma \vdash r : 0 \qquad \Gamma \vdash \underline{z} : C}{\Gamma \vdash f(r) : C}\ f(0) \longrightarrow \underline{z} \qquad \frac{\Gamma \vdash r : \$a \qquad \Gamma, x{:}a \vdash \underline{s} : C}{\Gamma \vdash f(r) : C}\ f(\$x) \longrightarrow \underline{s}$$

Observe that a recursive function f is typed "through its evaluation". For instance, if $f(0) \longrightarrow 0$ and $f(\$x) \longrightarrow \$(f(x))$, then f is the recursive identity, and to derive $y : \$^n 0 \vdash f(y) : \$^n 0$ we must derive $y : \$^m 0 \vdash f(y) : \$^m 0$ for all $m < n$ first. Hence, the whole computation tree of a recursive function application is already present in its typing. Therefore, a proof of strong normalization should be easy, in principle not harder as for the STL. In the following we substantiate this claim; although technically a bit involved, the proof has low proof-theoretic complexity.

3 Strong Normalization

In this section, we extend the normalization proof of Joachimski and Matthes [JM03] to the intersection type system of the last section. To this end, we introduce a judgement $\Gamma \vdash t \Uparrow C$ stating that $\Gamma \vdash t : C$ and t is strongly

normalizing. That this judgement is closed under substitution and application will be the main technical lemma; remember that closure under application is the difficult part in strong normalization proofs and usually requires a Tait-style logical relation argument. The basic idea behind the judgement is that typed weakly normalizing terms are closed under substitution: substituting the normal form v of a term of type $A = A_1 \to \cdots \to A_m \to B$ for x into the normal form w of another term can generate redexes if v contains subterms of the form $x\, v_1^{A_1} \ldots v_n^{A_n}$. Considering such a new β-redex $(\lambda x' w')\, v_i^{A_i}$, we observe that its degree A_i is smaller than A. Thus, the new substitution of v_i into w', which is necessary to reduce the redex, occurs with a smaller type; it might again create new redexes, however, with an even smaller type, so the whole process will eventually terminate. Watkins et. al. [WCPW03] coined the name *hereditary substitutions* for this process. Yet this normalization argument for the STL is quite old, Lévy [Lév76] attributes it to D. van Dalen.

SN: Atomic terms.

$$\frac{}{\Gamma \vdash x \downarrow \Gamma(x)} \qquad \frac{\Gamma \vdash r \downarrow \bigcap_{i\in I}(A_i \to B_i) \quad \Gamma \vdash s \Uparrow A_j \text{ for all } j \in J}{\Gamma \vdash r\, s \downarrow \bigcap_{j\in J} B_j}\ J \subseteq I$$

SN: Neutral terms.

$$\frac{\Gamma \vdash r \downarrow A \qquad A \subseteq B}{\Gamma \vdash r \Downarrow B} \qquad \frac{\Gamma \vdash r \Downarrow 0 \qquad \Gamma \vdash \underline{z}\, \vec{s} \Uparrow C}{\Gamma \vdash f(r)\, \vec{s} \Downarrow C}\ f(0) \longrightarrow \underline{z}$$

$$\frac{\Gamma \vdash r \Downarrow \$a \qquad \Gamma, x\!:\!a \vdash \underline{s}\, \vec{s} \Uparrow C}{\Gamma \vdash f(r)\, \vec{s} \Downarrow C}\ \begin{array}{l} f(\$x) \longrightarrow \underline{s} \\ x \notin \mathsf{FV}(\vec{s}) \end{array}$$

SN: Values, blocked terms, and weak head expansions.

$$\frac{\Gamma \vdash r \Downarrow A \quad A \subseteq B}{\Gamma \vdash r \Uparrow B} \qquad \frac{\Gamma, x\!:\!A_i \vdash t \Uparrow B_i \text{ for all } i \in I}{\Gamma \vdash \lambda x t \Uparrow \bigcap_{i\in I}(A_i \to B_i)} \qquad \frac{}{\Gamma \vdash 0 \Uparrow 0} \qquad \frac{\Gamma \vdash r \Uparrow a}{\Gamma \vdash \$r \Uparrow \$a}$$

$$\frac{\Gamma \vdash r \Uparrow A}{\Gamma \vdash f(r) \Uparrow \mathsf{E}}\ A \neq 0, \$a \qquad \frac{\Gamma \vdash r \Uparrow \mathsf{E} \qquad \Gamma \vdash s \Uparrow A}{\Gamma \vdash r\, s \Uparrow \mathsf{E}}$$

$$\frac{\Gamma \vdash s \Uparrow A \qquad \Gamma \vdash E[[s/x]t] \Uparrow C}{\Gamma \vdash E[(\lambda x t)\, s] \Uparrow C} \qquad \frac{\Gamma \vdash E[\underline{z}] \Uparrow C}{\Gamma \vdash E[f(0)] \Uparrow C}\ f(0) \longrightarrow \underline{z}$$

$$\frac{\Gamma \vdash r \Uparrow A \qquad \Gamma \vdash E[[r/x]\underline{s}] \Uparrow C}{\Gamma \vdash E[f(\$r)] \Uparrow C}\ f(\$x) \longrightarrow \underline{s}$$

Figure 1: Inductive characterization of SN.

The crucial property of STL which makes hereditary substitutions work is that in an *atomic* term $x\, s_1 \ldots s_n$, the types of all s_i are smaller than the type of x. In the presence of term rewriting, we can have terms like $f(x)\, s$ with a variable in the head, but now the type of s is no longer guaranteed to be smaller than the type of x. Let us call such terms *neutral*; they are of shape $E[x]$ where

$E[]$ is an *evaluation context* given by the grammar

$$E[] ::= [] \mid E[]\, s \mid f(E[]).$$

The termination argument of hereditary substitutions does no longer apply. However, we can salvage our substitution lemma by the following observation: For neutral terms of the shape $t = E[f(y\,\vec{s})]$, substitution for y might simplify the atomic part $y\,\vec{s}$ to 0 or $\$r$. In the first case t is s.n. if $E[\underline{z}]$ is, and in the second case, if $E[[r/x]\underline{s}]$ is.

Taking these thoughts into account we simultaneously define the three judgements (see Fig. 1):

$$\begin{array}{ll} \Gamma \vdash t \downarrow A & t \text{ is s.n. and atomic of type } A \\ \Gamma \vdash t \Downarrow A & t \text{ is s.n. and neutral of type } A \\ \Gamma \vdash t \Uparrow A & t \text{ is s.n. of type } A \end{array}$$

Lemma 1 (Weakening) *Let* $\Gamma' \subseteq \Gamma$.

1. *If* $\mathcal{D} :: \Gamma \vdash t \downarrow B$ *then* $\Gamma' \vdash t \downarrow A$ *for some* $A \subseteq B$.
2. *If* $\mathcal{D} :: \Gamma \vdash t \Downarrow B$ *then* $\Gamma' \vdash t \Downarrow B$.
3. *If* $\mathcal{D} :: \Gamma \vdash t \Uparrow B$ *and* $B \subseteq C$ *then* $\Gamma' \vdash t \Uparrow C$.

Proof. Simultaneously by induction on $\mathcal{D}$. □

The three judgements are closed under intersection: For $\mathcal{R} \in \{\downarrow, \Downarrow, \Uparrow\}$, $\Gamma \vdash t\ \mathcal{R}\ A$ and $\Gamma \vdash t\ \mathcal{R}\ B$ imply $\Gamma \vdash t\ \mathcal{R}\ A \cap B$, which can be proven simultaneously for all three $\mathcal{R}$ by induction.

Now we come to the crucial substitution lemma. Substitution for x in a derivation of

$$\frac{\Gamma, x{:}A \vdash r \Downarrow \$b \qquad \Gamma, x{:}A, y{:}b \vdash \underline{s}\,\vec{t} \Uparrow C}{\Gamma, x{:}A \vdash f(r)\,\vec{t} \Downarrow C}\ f(\$y) \longrightarrow \underline{s}$$

might trigger a substitution for y which in turn might trigger new substitutions. To establish termination of this process we require y to be of base type b. This is the reason why we disallow higher-order datatypes, which can have elements of higher types as arguments to their constructors. We generalize the substitution lemma of Joachimski and Matthes [JM03] to simultaneous substitution, but only one substituted variable may be of higher type.

Lemma 2 (Substitution and application) *Let* $\Gamma \vdash s \Uparrow A$ *and* $\Gamma \vdash s_i \Uparrow a_i$ *for* $i \in I$. *Let* $r' = [s/x][\vec{s}/\vec{x}]r$. *Let* $\mathcal{R} \in \{\Downarrow, \Uparrow\}$.

1. *If* $\mathcal{D} :: \Gamma, x{:}A, \vec{x}{:}\vec{a} \vdash r \downarrow C$ *then either* $\Gamma \vdash r' \downarrow C$, *or* $\Gamma \vdash r' \Uparrow C$ *and* $|C| \leq |A|$.
2. *If* $\mathcal{D} :: \Gamma, x{:}A, \vec{x}{:}\vec{a} \vdash r\ \mathcal{R}\ C$ *then* $\Gamma \vdash r' \Uparrow C$.
3. *If* $\mathcal{D} :: \Gamma \vdash r\ \mathcal{R} \bigcap_{i \in I}(A_i \to C_i)$ *and* $A = A_j$ *then* $\Gamma \vdash r\,s\ \mathcal{R}\ C_j$.

Proof. By lexicographic induction on $(|A|, \mathcal{D})$. For Prop. 1 consider the cases:

Case $\Gamma, x{:}A, \vec{x}{:}\vec{a} \vdash x \downarrow A$. Then $r' = s$ and $\Gamma \vdash r' \Uparrow A$ with $|A| \leq |A|$.

Case $\Gamma, x{:}A, \vec{x}{:}\vec{a} \vdash x_i \downarrow a_i$. Then $r' = s_i$ and $\Gamma \vdash r' \Uparrow a_i$, and $|a_i| = 0 \leq |A|$.

Case $\Gamma, x{:}A, \vec{x}{:}\vec{a} \vdash y \downarrow B$ for $y \notin \{x, \vec{x}\}$. Then $r' = y$ and $\Gamma \vdash r' \downarrow B$.

Case $J \subseteq I$ and

$$\frac{\Gamma, x{:}A, \vec{x}{:}\vec{a} \vdash t \downarrow \bigcap_{i\in I}(A_i \to B_i) \qquad \Gamma, x{:}A, \vec{x}{:}\vec{a} \vdash u \Uparrow A_i \text{ for all } i \in J}{\Gamma, x{:}A, \vec{x}{:}\vec{a} \vdash t\, u \downarrow \bigcap_{i\in J} B_i}$$

Let $t' = [s/x][\vec{s}/\vec{x}]t$ and $u' = [s/x][\vec{s}/\vec{x}]u$. By second induction hypothesis, $\Gamma \vdash u' \Uparrow A_i$ for all $i \in J$. We distinguish cases on the first induction hypothesis:

Subcase $\Gamma \vdash t' \downarrow \bigcap_{i\in I}(A_i \to B_i)$. Then $\Gamma \vdash t'\, u' \downarrow \bigcap_{i\in J} B_i$.

Subcase $\Gamma \vdash t' \Uparrow \bigcap_{i\in I}(A_i \to B_i)$ and $|\bigcap_{i\in I}(A_i \to B_i)| \leq |A|$. Then for each $i \in J$, we have $|A_j| < |A|$ and, thus, can apply induction hypothesis 3 to obtain $\Gamma \vdash t'\, u' \Uparrow B_i$ and $|B_i| \leq |A|$. Thus, $|\bigcap_{i\in J} B_i| \leq |A|$, and by closure unter intersection, $\Gamma \vdash t'\, u' \Uparrow \bigcap_{i\in J} B_i$.

The principal case of Proposition 2 is:

Case $f(\$y) \longrightarrow \underline{s}$, $y \notin \mathsf{FV}(\vec{t})$ and

$$\frac{\Gamma, x{:}A, \vec{x}{:}\vec{a} \vdash r \Downarrow \$b \qquad \Gamma, x{:}A, \vec{x}{:}\vec{a}, y{:}b \vdash \underline{s}\,\vec{t} \Uparrow C}{\Gamma, x{:}A, \vec{x}{:}\vec{a} \vdash f(r)\,\vec{t} \Downarrow C}$$

By induction hypothesis, $\mathcal{D}' :: \Gamma \vdash r' \Uparrow \$b$. Let $t'_j = [s/x][\vec{s}/\vec{x}]t_j$ for all j. We show $\Gamma \vdash f(r')\,\vec{t'} \Uparrow C$ by a local induction on $\mathcal{D}'$.

Subcase $r' = \$r''$ and $\Gamma \vdash r'' \Uparrow b$. Then by main induction hypothesis $\Gamma \vdash ([r''/y]\underline{s})\,\vec{t'} \Uparrow C$ which implies $\Gamma \vdash f(\$r'')\,\vec{t'} \Uparrow C$.

Subcase $r' = E[(\lambda z t)u]$ and

$$\frac{\Gamma \vdash u \Uparrow A' \qquad \Gamma \vdash E[[u/z]t] \Uparrow \$b}{\Gamma \vdash E[(\lambda z t)\,u] \Uparrow \$b}$$

Let $E'[] = f(E[])\,\vec{t'}$. By local induction hypothesis, $\Gamma \vdash E'[[u/z]t] \Uparrow C$, hence, $\Gamma \vdash E'[(\lambda z t)\,u] \Uparrow C$. The other cases of weak head expansions are treated analogously.

Subcase

$$\frac{\Gamma \vdash r' \Downarrow \$b' \qquad b' \subseteq b}{\Gamma \vdash r' \Uparrow \$b}$$

By main induction hypothesis, $\Gamma, y{:}b \vdash \underline{s}\,\vec{t'} \Uparrow C$. By the weakening lemma, $\Gamma, y{:}b' \vdash \underline{s}\,\vec{t'} \Uparrow C$, which entails $\Gamma \vdash f(r')\,\vec{t'} \Downarrow C$.

Subcase

$$\frac{\Gamma \vdash r' \Downarrow \mathsf{E} \qquad \mathsf{E} \subseteq \$b}{\Gamma \vdash r' \Uparrow \$b}$$

Then $\Gamma \vdash f(r') \Uparrow \mathsf{E}$, which implies $\Gamma \vdash f(r')\,\vec{t'} \Uparrow \mathsf{E}$ by iterated application. We conclude $\Gamma \vdash f(r')\,\vec{t'} \Uparrow C$ by weakening.

The principal case for Proposition 3 is:

Case

$$\frac{\Gamma, x{:}A_i \vdash t \Uparrow C_i \text{ for all } i \in I}{\Gamma \vdash \lambda x t \Uparrow \bigcap_{i\in I}(A_i \to C_i)}$$

By Prop. 2, $\Gamma \vdash [s/x]t \Uparrow C_j$. By weak head expansion, $\Gamma \vdash (\lambda x t)\, s \Uparrow C_j$. □

Lemma 3 (Recursion)

1. *If* $\mathcal{D} :: \Gamma \vdash r \Uparrow 0$ *and* $\Gamma \vdash \underline{z} \Uparrow C$ *then* $\Gamma \vdash f(r) \Uparrow C$.
2. *If* $\mathcal{D} :: \Gamma \vdash r \Uparrow \$a$ *and* $\Gamma, x{:}a \vdash \underline{s} \Uparrow C$ *then* $\Gamma \vdash f(r) \Uparrow C$.

Proof. Each by induction on $\mathcal{D}$. This essentially repeats proofs carried out for Proposition 2 in the previous lemma. □

Now we have shown that the judgement $\Uparrow$ is closed under all eliminations (intersection elim., application, recursion), hence, each well-typed term is in $\Uparrow$:

Theorem 4 *If* $\Gamma \vdash t : C$ *then* $\Gamma \vdash t \Uparrow C$.

Proof. By induction on $\Gamma \vdash t : C$. □

It remains to show that if $\Gamma \vdash t \Uparrow C$ then t is indeed strongly normalizing. Consider the following rule:

$$\frac{\Gamma \vdash r \Uparrow A}{\Gamma \vdash f(r) \Uparrow \mathsf{E}}\ A \neq 0, \$a.$$

To show strong normalization of $f(r)$ from s.n. of r we additionally need to know that r will never reduce to 0 or $\$r'$ for some r'.

Theorem 5 *Let* $\mathcal{R} \in \{\downarrow, \Downarrow, \Uparrow\}$ *and* $\Gamma \vdash t\ \mathcal{R}\ C$. *Then* t *is strongly normalizing. If* $\mathcal{R} \neq \Uparrow$, *then* t *is neutral. Otherwise,*

1. *if* $C \neq 0$, *then* $t \not\longrightarrow^* 0$,
2. *if* $C \neq \$c$, *then* $t \not\longrightarrow^* \$t'$ *for any* t', *and*
3. *if* $C \neq \bigcap_{i\in I}(A_i \to B_i)$, *then* $t \not\longrightarrow^* \lambda x t'$ *for any* t'.

Proof. By induction on $\Gamma \vdash t\ \mathcal{R}\ C$. For each rule we do a local Noetherian induction on the strong normalization of the terms in the premises. □

4 Conclusion

We have proven strong normalization for a λ-calculus with intersection types and term rewriting without the use of reducibility candidates or Tait-style saturated sets. The proof is technically involved, but can be formalized in a weak metatheory. Formalization in Twelf is not directly possible, not because Twelf's induction principles are not strong enough, but because we use constructs like $\bigcap_{i\in I}(A_i \to B_i)$ or simultaneous substitutions, which are not representable in Twelf, at least not directly.

We have partially answered our conjecture in the affirmative, that the intersection type system of Coquand and Spiwack [CS06] can be proven normalizing with simple means. What remains is an extension to higher-order datatypes. Also, our proof relies substantially on a *deterministic weak head reduction* relation, it is therefore not clear how we could handle overlapping patterns, an extension Coquand and Spiwack discuss in the conclusion of their article. Neither they nor we handle non-computational rewrite rules like $(x+y)+z \longrightarrow x+(y+z)$.

Riba [Rib07] considers a similar system, without datatypes yet with rewrite rules, and with union types. He shows that the elimination rule for union types can lead to diverging terms in some cases, and isolates conditions when these cases cannot occur. It would be interesting to see whether our proof could be extended to union types, and where Riba's conditions materialize in the proof.

Thanks to Thierry Coquand for interesting discussions.

References

[Abe04] Andreas Abel. Weak normalization for the simply-typed lambda-calculus in Twelf. In *LFM'04*, 2004.

[Ber05] Ulrich Berger. Continuous semantics for strong normalization. In *CiE'05*, volume 3526 of *LNCS*, pages 23–34. Springer, 2005.

[CS06] Thierry Coquand and Arnaud Spiwack. A proof of strong normalisation using domain theory. In *LICS'06*, pages 307–316. IEEE CS Press, 2006.

[Dav01] René David. Normalization without reducibility. *APAL*, 107(1–3):121–130, 2001.

[JM03] Felix Joachimski and Ralph Matthes. Short proofs of normalization. *AML*, 42(1):59–87, 2003.

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *POPL'07*, pages 173–184. ACM, 2007.

[Lév76] Jean-Jacques Lévy. An algebraic interpretation of the $\lambda\beta$K-calculus; and an application of a labelled λ-calculus. *TCS*, 2(1):97–114, 1976.

[Mat00] Ralph Matthes. Characterizing strongly normalizing terms of a calculus with generalized applications via intersection types. In *ITRS'00*, pages 339–354. Carleton Scientific, 2000.

[Rib07] Colin Riba. Strong normalization as safe interaction. In *Logics in Computer Science, LICS'07*, 2007. To appear.

[Val01] Silvio Valentini. An elementary proof of strong normalization for intersection types. *AML*, 40(7):475–488, October 2001.

[WCPW03] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgements and properties. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 2003.

Algorithmic equality in Heyting Arithmetic Modulo

Lisa Allali
LogiCal – Ecole polytechnique – INRIA,
www.lix.polytechnique.fr/Labo/Lisa.Allali/
allali@lix.polytechnique.fr

June 4, 2007

1 Introduction

We present in this paper a version of Heyting arithmetic where all the axioms are dropped and replaced by rewrite rules. A previous work has been done by Gilles Dowek and Benjamin Werner presenting Heyting Arithmetic in such a way [DW05], but where equality was defined by a "Leibniz rule" : a proposition of the form $x = y$ was rewritten in their system into $\forall p\ (x \in p \Rightarrow y \in p)$, that is provable if x and y are two equal closed terms, but not as simply as it could be expected. In this paper, in contrary, when x and y are closed terms, we consider checking equality between terms is just a computation : $x = y$ rewrites directly to $\top$ or $\bot$.
We followed a remark of Schwichtenberg, about how a set of rewrite rules could be (or not) enough to decide equality in Heyting Arithmetic. In the work we present here, we answer positively to this question and present a set of rewrite rules that define a new Heyting Arithmetic modulo $\mathrm{HA}_{\longrightarrow}$, that is

- an extension of axiomatic Heyting Arithmetic : all the theorems of arithmetic and, in particular, all instances of Leibniz' scheme can be proved in $\mathrm{HA}_{\longrightarrow}$
- this extension is conservative with respect to a very simple translation
- all proofs of $\mathrm{HA}_{\longrightarrow}$ strongly normalize.

This work suggests new ways to consider equality of inductive types in general, not anymore with Leibniz's axiom as it is the case in Coq for instance, but building specific rewrite rules to define equality in an algorithmic way.

2 Definitions

2.1 Deduction Modulo

Modern type theories feature a rule called *conversion rule* which allows to identify proposition which are *equal modulo beta-equivalence*. It is often presented as follows :

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T : Type \qquad \Gamma \vdash T' : Type}{\Gamma \vdash t : T'} \, T \equiv_\beta T'$$

where $T \equiv_\beta T'$ is read T *is convertible to* T'.
This convertibility is not checked by logical rules but by *computation* with the rule β. The idea of natural deduction modulo is to "import" this computation of convertibility inside the natural deduction but replacing $\equiv_\beta$ by an arbitrary congruence $\equiv$ defined by a confluent rewrite system. For instance, the axiom rule and the $\Rightarrow$ elimination rules are the following :

$$\frac{}{\Gamma \vdash_\equiv B} \text{ Ax if } A \in \Gamma \text{ and } A \equiv B$$

$$\frac{\Gamma \vdash_\equiv C \qquad \Gamma \vdash_\equiv A}{\Gamma \vdash_\equiv B} \Rightarrow\text{e if } C \equiv A \Rightarrow B$$

The other rules of natural deduction modulo are built the same way upon natural deduction [DHK03].
The convertibility $\equiv$ is not fixed. It can be any congruence defined by the reflexive, symmetric and transitive closure of a rewrite system which is confluent, rewrites term to term and atomic proposition to proposition.

2.2 Theories in natural deduction modulo

Definition 1 (Axiomatic theory)
An axiomatic theory *is a set of axioms.*

Definition 2 (Modulo theory)
A modulo theory *is a set of axioms and a congruence defined as the reflexive, transitive and symmetric closure of a set of rewrite rules.*

Definition 3 (Purely computational theory)
A purely computational theory *is a modulo theory where the set of axioms is empty.*

3 Heyting Arithmetic - from axioms to rewrite rules

3.1 The axiomatic presentation of Heyting Arithmetic

The language of arithmetic formed by the functional symbols 0 of arity 0, S of arity 1, $+$ and $\times$ of arity 2. The predicate symbol $=$ of arity 2. The axioms are structured in four groups as follow :

The axioms of equality

Reflexivity	*Leibniz' axiom scheme*
$\forall x\ x = x$	$\forall x\ \forall y\ x = y \Rightarrow P(x) \Leftrightarrow P(y)$

The axioms 3 and 4 of Peano

$\forall x\ \forall y\ (S(x) = S(y) \Rightarrow x = y)$ $\qquad$ $\forall x\ 0 = S(x) \Rightarrow \bot$

The induction scheme

$(P\{x := 0\} \wedge \forall y\ (P\{x := y\} \Rightarrow P\{x := S(y)\})) \Rightarrow \forall n\ P\{x := n\}$

The axioms of addition and multiplication.

$\forall y\ (0 + y = y)$	$\forall x\ \forall y\ (S(x) + y = S(x + y))$
$\forall y\ (0 \times y = 0)$	$\forall x\ \forall y\ (S(x) \times y = x \times y + y)$

3.2 The steps to go from an axiomatic theory of Heyting Arithmetic (HA) to a purely computational one

We shall introduce four successive theories to reach the final purely computational theory we aim at, each of them being an equivalent or conservative extension of HA.

3.2.1 $\mathbf{HA}_R$

Induction axiom scheme

$(P\{x := 0\} \wedge \forall y\ (P\{x := y\} \Rightarrow P\{x := S(y)\})) \Rightarrow \forall n\ P\{x := n\}$

Rewrite rules

$0 = 0 \longrightarrow \top$	$0 + y \longrightarrow y$
$0 = S(x) \longrightarrow \bot$	$S(x) + y \longrightarrow S(x + y)$
$S(x) = 0 \longrightarrow \bot$	$0 \times y \longrightarrow 0$
$S(x) = S(y) \longrightarrow x = y$	$S(x) \times y \longrightarrow x \times y + y$

The axioms of addition and multiplication are transformed in a very intuitive way into rewrite rules : for instance, the axiom $\forall x\ 0 + x = x$ becomes the rewrite rule $0 + x \longrightarrow x$. We keep the induction axiom scheme. We drop the Leibniz axiom and add 4 rewrite rules to define equality. We prove this theory is equivalent to HA.

3.2.2 $\mathbf{HA}_N$

Induction axiom scheme

$\forall n\, N(n) \Rightarrow (P\{x := 0\} \wedge \forall y\, (N(y) \Rightarrow P\{x := y\} \Rightarrow P\{x := S(y)\})) \Rightarrow P\{x := n\}$

Axioms for N

$N(0)$	$\forall x\ N(x) \Rightarrow N(S(x))$

Rewrite rules

$0 = 0 \longrightarrow \top$	$0 + y \longrightarrow y$
$0 = S(x) \longrightarrow \bot$	$S(x) + y \longrightarrow S(x + y)$
$S(x) = 0 \longrightarrow \bot$	$0 \times y \longrightarrow 0$
$S(x) = S(y) \longrightarrow x = y$	$S(x) \times y \longrightarrow x \times y + y$

We introduce a new predicate symbol N for the natural numbers and three axioms to define it. We prove this theory is a conservative extension of HA_R modulo a certain translation .

3.2.3 $\mathbf{HA}_K$

Comprehension scheme

$\forall x \forall y_1 ... \forall y_n\ (x \in f_{z,y_1,...,y_n,P}(y_1, \ldots, y_n) \Leftrightarrow P\{z := x\})$

Induction axiom

$\forall n\ (N(n) \Leftrightarrow \forall f\ (0 \in f \Rightarrow \forall y\ (N(y) \Rightarrow y \in f \Rightarrow S(y) \in f) \Rightarrow n \in f))$

Rewriting rules

$0 = 0 \longrightarrow \top$	$0 + y \longrightarrow y$
$0 = S(x) \longrightarrow \bot$	$S(x) + y \longrightarrow S(x + y)$
$S(x) = 0 \longrightarrow \bot$	$0 \times y \longrightarrow 0$
$S(x) = S(y) \longrightarrow x = y$	$S(x) \times y \longrightarrow x \times y + y$

We sort our theory with two sorts ι and κ. We add an infinite number of function symbols of the form $f_{z,y_1,...,y_n,P}$, one for each proposition P that can be expressed in the language of HA_N, where the free variables of P are $y_1, \ldots, y_n$. Each of those symbols is of rank $\langle \iota, ..., \iota, \kappa \rangle$. We add a symbol $\in$ of rank $\langle \iota, \kappa \rangle$. Finally we add an axiom scheme expressing the equivalence of the proposition $x \in f_{z,y_1,...,y_n,P}(y_1, \ldots, y_n)$ with P. Notice that the free variables of these propositions are the same. The induction axiom scheme is replaced by a single axiom. We prove this theory is a conservative extension of HA_N.

3.2.4 $\mathbf{HA}_{\longrightarrow}$

We transform the remaining axioms of HA_N into rewrite rules.
$\mathrm{HA}_{\longrightarrow}$ is a purely computational presentation of Heyting Arithmetic.

3.3 HA$_{\longrightarrow}$, a purely computational presentation of Heyting Arithmetic

Definition 4 (HA$_{\longrightarrow}$)

$$x \in f_{z,y_1,\ldots,y_n,P}(y_1,\ldots,y_n) \longrightarrow P\{z := x\}$$

$$N(n) \longrightarrow \forall f\ (0 \in f \Rightarrow \forall y\ (N(y) \Rightarrow y \in f \Rightarrow S(y) \in f) \Rightarrow n \in f)$$

$0 = 0 \longrightarrow \top$	$0 + y \longrightarrow y$
$0 = S(x) \longrightarrow \bot$	$S(x) + y \longrightarrow S(x + y)$
$S(x) = 0 \longrightarrow \bot$	$0 \times y \longrightarrow 0$
$S(x) = S(y) \longrightarrow x = y$	$S(x) \times y \longrightarrow x \times y + y$

We introduce the following **Translation** $|\cdot|$ **from HA to HA**$_{\longrightarrow}$
$|P| = P$, if P is atomic,
$|\top| = \top$,
$|\bot| = \bot$,
$|A \wedge B| = |A| \wedge |B|$,
$|A \vee B| = |A| \vee |B|$,
$|A \Rightarrow B| = |A| \Rightarrow |B|$,
$|\forall x\ A| = \forall x\ (N(x) \Rightarrow |A|)$,
$|\exists x\ A| = \exists x\ (N(x) \wedge |A|)$

Proposition 1
HA$_{\longrightarrow}$ is a conservative extension of HA, i.e. for all closed propositions A,

$$\vdash_{HA} A \qquad \textit{if and only if} \qquad \vdash_{HA_{\longrightarrow}} |A|$$

We prove that each of the theories HA, HA$_R$, HA$_N$, HA$_K$, HA$_{\longrightarrow}$ is a conservative extension of the previous one.
The main difficulty lies in the first step: proving that HA$_R$ is an extension of HA, and more specifically that the Leibniz's axiom scheme of HA is derivable in HA$_R$. This requires to prove successively the following properties of HA$_R$ equality:

$$\forall x\ (x = x)$$
$$\forall x\ \forall y\ (x = y \Rightarrow y = x)$$
$$\forall x\ \forall y\ \forall z\ (x = y \Rightarrow y = z \Rightarrow x = z)$$

$$\forall x\ \forall y\ \forall z\ x = y \Rightarrow x + z = y + z$$
$$\forall x\ \forall y\ \forall z\ x = y \Rightarrow z + x = z + y$$

$$\forall x\ \forall y\ \forall z\ x = y \Rightarrow x \times z = y \times z$$
$$\forall x\ \forall y\ \forall z\ x = y \Rightarrow z \times x = z \times y$$

$$\forall x\ x \times 0 = 0$$
$$\forall y\ \forall x\ (y \times S(x) = y \times x + y)$$
$$\forall x\ \forall y\ (x \times y = y \times x)$$

Thanks to those properties of equality in HA_R, we can prove by induction on t that for each term t, the proposition $\forall a\ \forall b\ (a = b \Rightarrow t\{y := a\} = t\{y := b\})$ is provable in HA_R.
This proposition is the basic case of an induction on P we made to prove that each instance of Leibniz' scheme

$$\forall x\ \forall y\ x = y \Rightarrow P(x) \Leftrightarrow P(y)$$

is provable in HA_R.

Finally we also prove two other results :

Proposition 2
The congruence defined by the rewrite rules of $HA_{\longrightarrow}$ is decidable.

Proposition 3
$HA_{\longrightarrow}$ has cut elimination property.

4 Discussion

One can ask if this system is really efficient in practice: in one hand, the proof of $x = y$ are shorter, in the other hand the proof of $\forall x \forall y\ x = y \Rightarrow P(x) \Rightarrow P(y)$ is longer. There is no theoretical answer to that question, it's only by making tests that we would see how the size of proof terms would change. An good indication is that the way we manage to "simulate" an application of Leibniz principle with our rewrite rules (the way it is shown in [All]) is linear in the size of the proposition.

5 Conclusion

We have reached a presentation of Heyting Arithmetic without any axiom, simply defined by a rewrite rule system. A cornerstone of this presentation is that it makes use of the decidability of the equality in Heyting Arithmetic, indeed the equality is *defined* as a decision procedure, rather than as Leibniz's proposition which becomes a consequence of the congruence of the system.

References

[All] Lisa Allali, Memoire de DEA, `http://www.lix.polytechnique.fr/Labo/Lisa.Allali/rapport_MPRI.pdf`.

[DHK03] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31:32–72, 2003.

[DW05] Gilles Dowek and Benjamin Werner. Arithmetic as a theory modulo. J. Giesl (Ed.), *Term rewriting and applications* (RTA), Lecture Notes in Computer Science 3467, Springer-Verlag, 2005, pp. 423-437.

[Dow07] Gilles Dowek. Truth values algebras and normalization, to appear in TYPES 2006, 2007.

[Dow99] Gilles Dowek. La part du calcul. *Mémoire d'Habilitation à Diriger des Recherches*, Université Paris 7, 1999.

[vOvR94] Vincent van Oostrom, Femke van Raamsdonk. Weak Orthogonality Implies Confluence : The High-Order Case. *Technical Report: ISRL-94-5*, December, 1994.

[Coq06] The Coq Development Team . Manuel de Référence de Coq V8.0. `http://coq.inria.fr/doc/main.html`, LogiCal Project, 2004–2006.

Argument Filterings and Usable Rules for Simply Typed Dependency Pairs*
(extended abstract)

Takahito Aoto† Toshiyuki Yamada‡

June 4, 2007

1 Introduction

Simply typed term rewriting [Yam01] is a framework of higher-order term rewriting without bound variables. The authors extended the first-order dependency pair approach [AG00] to the case of simply typed term rewriting [AY05]. They gave a characterization of minimal non-terminating simply typed terms and incorporated the notions of dependency pairs, dependency graphs, and estimated dependency graphs into the simply typed framework. They extended the subterm criterion [HM04] of first-order dependency pairs and introduced the head instantiation technique to make the simply typed dependency pair method effectively applicable even in the presence of function variables.

In this paper, we incorporate termination criteria using reduction pairs and related refinements into the simply typed dependency pair framework. In particular, we extend the notions of argument filterings [AG00] and usable rules [HM04, TGSK04] of first-order dependency pairs to the case of simply typed term rewriting.

Refinements of dependency pair technique for higher-order systems with bound variables are studied in [Bla06, SK05], and an approach to deal within the framework of first-order dependency pairs is studied in [GTSK05]. In our framework the presence of simple types and higher-order variables/rules are reflected in more specific way comparing with [GTSK05]. On the other hand, since bound variables are not included in our framework, our dependency pair framework is simpler and thus easy to automate compared to the methods in [Bla06, SK05].

2 Preliminaries

A *simple type* is either the *base type* o or a *function type* $\tau_1 \times \cdots \times \tau_n \to \tau_0$. The set of simple types is denoted by ST. The sets of *constants*, *variables*, and

*The authors thank Jeroen Ketema and the referees for their comments.

†RIEC, Tohoku University, Japan. aoto@nue.riec.tohoku.ac.jp

‡Graduate School of Engineering, Mie University, Japan. toshi@cs.info.mie-u.ac.jp

simply typed terms are denoted by Σ, V, and $\mathrm{T}(\Sigma, V)$, respectively. The *head symbol* of a simply typed term is defined as follows: $\mathrm{head}(a) = a$ for $a \in \Sigma \cup V$; $\mathrm{head}((t_0\ t_1 \cdots t_n)) = \mathrm{head}(t_0)$. The set $\mathrm{PV}(t)$ of *primary variables* in a term t is defined as follows: $\mathrm{PV}(t) = \emptyset$ if $t \in \Sigma \cup V$; $\mathrm{PV}(t) = \{t_0\} \cup \bigcup_{i>0} \mathrm{PV}(t_i)$ if $t = (t_0\ t_1 \cdots\ t_n)$ and $t_0 \in V$; $\mathrm{PV}(t) = \bigcup_{i \geq 0} \mathrm{PV}(t_i)$ if $t = (t_0\ t_1 \cdots\ t_n)$ and $t_0 \notin V$. Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a *simply typed term rewriting system* (STTRS, for short). The set Σ_{d} of *defined symbols* of $\mathcal{R}$ is defined by $\Sigma_{\mathrm{d}} = \{\mathrm{head}(l) \mid l \to r \in R\}$.

Example 1 (simply typed term rewriting) Let $\mathcal{R} = \langle \Sigma, R \rangle$ be an STTRS where $\Sigma = \{\ \mathsf{0}^{\mathrm{o}},\ \mathsf{s}^{\mathrm{o}\to\mathrm{o}},\ [\,]^{\mathrm{o}},\ :^{\mathrm{o}\times\mathrm{o}\to\mathrm{o}},\ \mathsf{map}^{(\mathrm{o}\to\mathrm{o})\times\mathrm{o}\to\mathrm{o}},\ \circ^{(\mathrm{o}\to\mathrm{o})\times(\mathrm{o}\to\mathrm{o})\to\mathrm{o}\to\mathrm{o}},\ \mathsf{twice}^{(\mathrm{o}\to\mathrm{o})\to\mathrm{o}\to\mathrm{o}}\ \}$, and

$$R = \left\{ \begin{array}{lllll} (1) & \mathsf{map}\ F\ [\,] & \to & [\,] \\ (2) & \mathsf{map}\ F\ (:\ x\ xs) & \to & :\ (F\ x)\ (\mathsf{map}\ F\ xs) \\ (3) & (\circ\ F\ G)\ x & \to & F\ (G\ x) \\ (4) & \mathsf{twice}\ F & \to & \circ\ F\ F \end{array} \right\}.$$

Here is a rewrite sequence of $\mathcal{R}$:

$$\begin{array}{lll} \mathsf{map}\ (\mathsf{twice}\ \mathsf{s})\ (:\ \mathsf{0}\ [\,]) & \to_{\mathcal{R}} & \mathsf{map}\ (\circ\ \mathsf{s}\ \mathsf{s})\ (:\ \mathsf{0}\ [\,]) \\ & \to_{\mathcal{R}} & :\ ((\circ\ \mathsf{s}\ \mathsf{s})\ \mathsf{0})\ (\mathsf{map}\ (\circ\ \mathsf{s}\ \mathsf{s})\ [\,]) \\ & \to_{\mathcal{R}} & :\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{0}))\ (\mathsf{map}\ (\circ\ \mathsf{s}\ \mathsf{s})\ [\,]) \\ & \to_{\mathcal{R}} & :\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{0}))\ [\,]. \end{array}$$

3 Termination by reduction pairs

The *head rewrite step* $\xrightarrow{\mathrm{h}}$ is defined recursively as follows: $s \xrightarrow{\mathrm{h}} t$ if (1) $s = l\sigma$ and $t = r\sigma$ for some rewrite rule $l \to r$ and some substitution σ or (2) $s = (s_0\ u_1\ \cdots\ u_n)$, $t = (t_0\ u_1\ \cdots\ u_n)$, and $s_0 \xrightarrow{\mathrm{h}} t_0$. The *non-head rewrite step* is defined by $\xrightarrow{\mathrm{nh}} = \to \backslash \xrightarrow{\mathrm{h}}$. Let D be a set of *dependency pairs* of an STTRS $\mathcal{R}$. In simply typed term rewriting, a root rewrite step using a dependency pair is distinguished from the rewrite relation (since it is not in general type-preserving), and denoted by $\rightarrowtail_D$. A *dependency chain* of D is an infinite sequence $t_0, t_1, \ldots$ on $\mathrm{NT}_{\min}(\mathcal{R})$ such that $t_i \xrightarrow{\mathrm{nh}*} \cdot \rightarrowtail_D t_{i+1}$ for all $i \geq 0$. Here, $\mathrm{NT}_{\min}(\mathcal{R})$ is the set of minimal (with respect to the subterm relation $\trianglelefteq$) non-terminating terms. The family of all minimal (with respect to the set inclusion $\subseteq$) sets of dependency pairs that admit dependency chain is denoted by $\mathbf{DC}_{\min}(\mathcal{R})$. For $D \in \mathbf{DC}_{\min}(\mathcal{R})$, every element of D occurs infinitely often in its dependency chain.

Theorem 2 (termination by reduction pairs) Let $\mathcal{R} = \langle \Sigma, R \rangle$ be an STTRS and D a finite set of dependency pairs. If there exists a reduction pair $\langle \gtrsim, > \rangle$ such that $R \subseteq \gtrsim$, $D \subseteq \gtrsim$, and $D \cap > \neq \emptyset$, then $D \notin \mathbf{DC}_{\min}$.

In contrast to the first-order case, heads of rhs of dependency pairs need not be constants in general. Based on the *head instantiation* technique [AY05], however, it suffices to handle dependency pairs whose heads of rhs are constants. Such dependency pairs are referred to as *head-instantiated dependency pairs*.

4 Argument filterings

Since argument filtering may not preserve well-typedness, we need an underlying untyped calculus. For this, the framework of *S-expression reduction systems* (*SRSs* for short) [Toy04] is suitable. An S-expression is a first-order term with a special variadic function symbol @. The set $\mathrm{S}(\Sigma, V)$ of S-expressions is defined as: $\Sigma \cup V \subseteq \mathrm{S}(\Sigma, V)$; if $s_1, \ldots, s_n \in \mathrm{S}(\Sigma, V)$ $(n \geq 0)$ then $@(s_1, \ldots, s_n) \in \mathrm{S}(\Sigma, V)$. An S-expression $@(s_1, \ldots, s_n)$ is abbreviated as $(s_1 \ \cdots \ s_n)$. We note that $()$ and $((())())$ are also S-expressions. Each simply typed term can be regarded as an S-expression by forgetting its type information.

The first-order argument filtering is specified by function symbols, that is, $\pi(f(s_1, \ldots, s_n))$ is defined by the value of $\pi(f)$. In contrast, the head symbol of a simply typed term t is insufficient to specify filtering of t: e.g. $(f\ x\ y)$ and $((f\ x\ y)\ z)$ have the same head symbol but may have different filtering—the depth of head symbol occurrence needs to be considered additionally.

Definition 3 (filtering domain) Let X be a set of simply typed constants and simply typed variables. We define the *filtering domain* $\mathcal{D}(X) \subseteq X \times \mathbb{N}$ for X by $\mathcal{D}(X) = \bigcup_\tau \{\langle a, n\rangle \mid a \in X, a \text{ is of type } \tau, 0 \leq n < \mathrm{depth}(\tau)\}$. Here, the *depth* of a simple type τ is defined as follows: $\mathrm{depth}(\mathrm{o}) = 0$; $\mathrm{depth}(\tau_1 \times \cdots \times \tau_n \to \tau_0) = \mathrm{depth}(\tau_0) + 1$.

The *head depth* of a simply typed term t is defined as follows: $\mathrm{hdep}(a) = 0$ for $a \in \Sigma \cup V$; $\mathrm{hdep}((t_0\ t_1 \cdots t_n)) = \mathrm{hdep}(t_0) + 1$. The next lemma shows that mappings from the filtering domain are suitable to specify all argument filterings.

Lemma 4 Let X be a set of simply typed constants and simply typed variables. If s has a function type and $\mathrm{head}(s) \in X$, then $\langle \mathrm{head}(s), \mathrm{hdep}(s)\rangle \in \mathcal{D}(X)$.

The marking of head symbols similar to the first-order dependency pairs is useful to simplify the definition of argument filtering. For each $a \in \Sigma_\mathrm{d}$, let $a^\sharp$ be a new constant having the same type as a. Let $\Sigma_\mathrm{d}^\sharp = \{a^\sharp \mid a \in \Sigma_\mathrm{d}\}$ and $\Sigma^\sharp = \Sigma \cup \Sigma_\mathrm{d}^\sharp$.

For each $s \in \mathrm{T}(\Sigma, V)$ of type τ and $n \leq \mathrm{depth}(\tau)$, $\mathrm{type}(s, n)$ is defined as: $\mathrm{type}(s, 0) = \tau$; $\mathrm{type}(s, n+1) = \tau_0$ if $\mathrm{type}(s, n) = \tau_1 \times \cdots \times \tau_m \to \tau_0$. For any function type τ, $|\tau|$ is defined as: $|\tau_1 \times \cdots \times \tau_m \to \tau_0| = m$.

Definition 5 (argument filtering) Let $\mathbb{L}$ be the set of natural numbers and lists of natural numbers. An *argument filtering* π is a function from $\mathcal{D}(\Sigma^\sharp \cup V)$ to $\mathbb{L}$ such that for each $\langle f, n\rangle \in \mathcal{D}(\Sigma^\sharp \cup V)$, either $\pi(f, n) = [i_1, \ldots, i_k]$ for some $0 \leq i_1 < \cdots < i_k \leq |\mathrm{type}(f, n)|$ or $\pi(f, n) = i$ for some $0 \leq i \leq |\mathrm{type}(f, n)|$. Note that $k \geq 0$ and $k = 0$ means that the result is an empty list.

For a simply typed term t such that $\mathrm{head}(t) \in \Sigma_\mathrm{d}$, define $t^\sharp$ recursively as follows: $t^\sharp = a^\sharp$ if $t = a \in \Sigma_\mathrm{d}$; $t^\sharp = (t_0^\sharp\ t_1 \cdots t_n)$ if $t = (t_0\ t_1 \cdots t_n)$. The set of terms $\mathrm{T}(\Sigma, V) \cup \{t^\sharp \mid t \in \mathrm{T}(\Sigma, V), \mathrm{head}(t) \in \Sigma_\mathrm{d}\}$ is denoted by $\mathrm{T}^\sharp(\Sigma, V)$.

Definition 6 (application of argument filtering) Let π be an argument filtering. For each simply typed term $t \in \mathrm{T}^\sharp(\Sigma, V)$, an S-expressions $\pi(t)$ is defined as follows: (1) $\pi(a) = a$ for all $a \in \Sigma^\sharp \cup V$; (2) $\pi((t_0\ t_1 \cdots t_n)) = (\pi(t_{i_1})\ \cdots\ \pi(t_{i_k}))$ if $\pi(\mathrm{head}(t_0), \mathrm{hdep}(t_0)) = [i_1, \ldots, i_k]$; (3) $\pi((t_0\ t_1 \cdots t_n)) = \pi(t_i)$ if $\pi(\mathrm{head}(t_0), \mathrm{hdep}(t_0)) = i$.

Filtering functions should consistently select the same argument positions from both a term with head variable and its instance.

Example 7 (unsound filtering (1)) Let $\mathcal{R} = \langle \Sigma, R \rangle$ be an STTRS where $\Sigma = \{ \mathsf{0}^{\mathsf{o}}, \mathsf{f}^{\mathsf{o}\to\mathsf{o}}, \mathsf{s}^{\mathsf{o}\to\mathsf{o}} \}$ and

$$R = \left\{ \mathsf{f}\ (F\ x) \ \to \ \mathsf{f}\ (\mathsf{s}\ x) \right\}.$$

If π is an argument filtering such that $\pi(\mathsf{s}, 0) = 1$ and $\pi(\mathsf{f}, 0) = \pi(\mathsf{f}^\sharp, 0) = \pi(F, 0) = [0, 1]$, the following satisfiable set of constraints is obtained: $\{\mathsf{f}\ (F\ x) \geq \mathsf{f}\ x,\ \mathsf{f}^\sharp\ (F\ x) > \mathsf{f}^\sharp\ x\}$. Since $\mathcal{R}$ is not terminating, this argument filtering is unsound.

Filtering functions should consistently select the same argument positions from a term when its head is rewritten by a rule of function type.

Example 8 (unsound filtering (2)) Let $\mathcal{R} = \langle \Sigma, R \rangle$ be an STTRS where $\Sigma = \{ \mathsf{f}^{\mathsf{o}\to\mathsf{o}}, \mathsf{g}^{\mathsf{o}\to\mathsf{o}}, \mathsf{h}^{\mathsf{o}\to\mathsf{o}} \}$ and

$$R = \left\{ \begin{array}{lcl} \mathsf{f}\ (\mathsf{h}\ x) & \to & \mathsf{f}\ (\mathsf{g}\ x) \\ \mathsf{g} & \to & \mathsf{h} \end{array} \right\}.$$

Let $D = \{\mathsf{f}^\sharp\ (\mathsf{h}\ x) \rightarrowtail \mathsf{f}^\sharp\ (\mathsf{g}\ x)\}$. If π is an argument filtering such that $\pi(\mathsf{f}, 0) = \pi(\mathsf{g}, 0) = [\,]$, $\pi(\mathsf{f}^\sharp, 0) = [0, 1]$, and $\pi(\mathsf{h}, 0) = [1]$, the following satisfiable set of constraints is obtained: $\{() \geq (), \mathsf{g} \geq \mathsf{h}, (\mathsf{f}^\sharp\ (x)) > (\mathsf{f}^\sharp\ ())\}$. Since $\mathcal{R}$ is not terminating, this argument filtering is unsound.

Definition 9 (stabilization domain) Let π be an argument filtering. For any simply typed term $t \in \mathrm{T}^\sharp(\Sigma, V)$, the set $\mathrm{SDom}(t) \subseteq \mathcal{D}(\Sigma \cup V)$ of *stabilization domain* of t is defined as: $\mathrm{SDom}(a) = \emptyset$; $\mathrm{SDom}((t_0\ t_1 \cdots t_n)) = \bigcup\{\mathrm{SDom}(t_{i_j}) \mid 1 \leq j \leq k\}$ if $\pi(\mathrm{head}(t_0), \mathrm{hdep}(t_0)) = [i_1, \ldots, i_k]$ and $\mathrm{head}(t_0) \in \Sigma^\sharp_{\mathrm{d}}$; $\mathrm{SDom}((t_0\ t_1 \cdots t_n)) = \{\langle \mathrm{head}(t_0), \mathrm{hdep}(t_0)\rangle\} \cup \bigcup\{\mathrm{SDom}(t_{i_j}) \mid 1 \leq j \leq k\}$ if $\pi(\mathrm{head}(t_0), \mathrm{hdep}(t_0)) = [i_1, \ldots, i_k]$ and $\mathrm{head}(t_0) \in \Sigma \cup V$; $\mathrm{SDom}((t_0\ t_1 \cdots t_n)) = \mathrm{SDom}(t_i)$ if $\pi(\mathrm{head}(t_0), \mathrm{hdep}(t_0)) = i$ and $\mathrm{head}(t_0) \in \Sigma^\sharp_{\mathrm{d}}$; $\mathrm{SDom}((t_0\ t_1 \cdots t_n)) = \{\langle \mathrm{head}(t_0), \mathrm{hdep}(t_0)\rangle\} \cup \mathrm{SDom}(t_i)$ if $\pi(\mathrm{head}(t_0), \mathrm{hdep}(t_0)) = i$ and $\mathrm{head}(t_0) \in \Sigma \cup V$.

Definition 10 (stability) Let X be a set of simply typed constants and simply typed variables. Let f be a function from $\mathcal{D}(X)$ to $\mathbb{L}$. (1) f is stable w.r.t. a simple type τ if for any $\langle a, n\rangle, \langle b, m\rangle \in \mathcal{D}(X)$, $\mathrm{type}(a, n) = \mathrm{type}(b, m) = \tau$ implies $f(a, n) = f(b, m)$. (2) f is stable w.r.t. $A \subseteq \mathcal{D}(X)$ if f is stable w.r.t. any τ in the set $\{\mathrm{type}(a, n) \mid \langle a, n\rangle \in A, a \in V\}$.

Definition 11 (stability w.r.t. rules) Let π be an argument filtering and $\pi^{\mathrm{nh}} = \pi{\downarrow}\mathcal{D}(\Sigma, V)$. Here, $\downarrow$ denotes the operation of restricting the domain.

1. π is *stable* w.r.t. a set R of simply typed rewrite rules if π^{nh} is stable w.r.t. $\bigcup_{l\to r\in R} \mathrm{SDom}(l) \cup \mathrm{SDom}(r)$ and if R contains a simply typed rewrite rule of function type τ then π^{nh} is stable w.r.t. $\tau, \ldots, \tau + (\mathrm{depth}(\tau) - 1)$.

2. π is *stable* w.r.t. a set D of simply typed dependency pairs if π^{nh} is stable w.r.t. $\bigcup_{l\rightarrowtail r\in D} \mathrm{SDom}(l^\sharp) \cup \mathrm{SDom}(r^\sharp)$.

Theorem 12 (argument filtering refinement) Let $\mathcal{R} = \langle \Sigma, R \rangle$ be an STTRS and D a finite set of head-instantiated dependency pairs. If there exists a reduction pair $\langle \gtrsim, > \rangle$ on $\mathrm{S}(\Sigma, V)$ and an argument filtering π stable w.r.t. R and D, $\pi(R) \subseteq \gtrsim$, $\pi(D^\sharp) \subseteq \gtrsim$, and $\pi(D^\sharp) \cap > \neq \emptyset$, then $D \notin \mathbf{DC}_{\min}$.

5 Usable rules

Definition 13 (usable rules) We write $f \blacktriangleright g$ when there exists a simply typed rewrite rule $l \to r \in R$ such that $\mathrm{head}(l) = f$ and $g \in \Sigma_{\mathrm{d}}(r)$. We denote the reflexive transitive closure of $\blacktriangleright$ by $\blacktriangleright^*$. For a set D of head-instantiated dependency pairs,

$$\mathcal{U}_R(D^\sharp) = \{l \to r \in R \mid f \blacktriangleright^* \mathrm{head}(l) \text{ for some } f \in \Sigma_{\mathrm{d}}(\mathrm{RHS}(D^\sharp))\}.$$

Let $\mathcal{C}_\mathcal{E} = \langle \{\mathsf{nil}, \mathsf{cons}\}, C_\mathcal{E} \rangle$ be an SRS where $C_\mathcal{E} = \{(\mathsf{cons}\ x\ y) \to x, (\mathsf{cons}\ x\ y) \to y\}$. Let us first explain that a naive extension of usual first-order usable rules criteria is not adapted to the higher-order setting.

Example 14 (counterexample) Let $\mathcal{R} = \langle \Sigma, R \rangle$ be an STTRS where $\Sigma = \{ \mathsf{0}^{\mathsf{o}}, \mathsf{f}^{(\mathsf{o}\to\mathsf{o})\times\mathsf{o}\to\mathsf{o}}, \mathsf{g}^{\mathsf{o}\to\mathsf{o}} \}$ and

$$R = \left\{ \begin{array}{lcl} \mathsf{f}\ F\ \mathsf{0} & \to & \mathsf{f}\ F\ (F\ \mathsf{0}) \\ \mathsf{g}\ \mathsf{0} & \to & \mathsf{0} \end{array} \right\}.$$

For $D = \{\mathsf{f}\ F\ \mathsf{0} \rightarrowtail \mathsf{f}\ F\ (F\ \mathsf{0})\}$, there is an infinite dependency chain $\mathsf{f}\ \mathsf{g}\ \mathsf{0} \rightarrowtail_D \mathsf{f}\ \mathsf{g}\ (\mathsf{g}\ \mathsf{0}) \to_{\mathcal{R}} \mathsf{f}\ \mathsf{g}\ \mathsf{0} \rightarrowtail_D \cdots$. However $\mathcal{U}_R(D^\sharp) = \emptyset$ and thus there is no infinite dependency chain on D and $C_\mathcal{E} \cup \mathcal{U}_R(D^\sharp)$.

Definition 15 (higher-order usable rules)

1. The *range-order* $\trianglerighteq$ is the smallest partial order on ST satisfying $\tau_1 \times \cdots \times \tau_n \to \tau_0 \trianglerighteq \tau_0$.
2. We write $f \blacktriangleright_{\mathrm{h}} g$ when (1) there exists a simply typed rewrite rule $l \to r \in R$ such that $\mathrm{head}(l) = f$ and $g \in \Sigma_{\mathrm{d}}(r)$, or (2) there exists a simply typed rewrite rule $l \to r \in R$ and $F^\tau \in \mathrm{PV}(r)$ such that $\mathrm{head}(l) = f$, $g^\rho \in \Sigma_{\mathrm{d}}$ for some $\rho \trianglerighteq \tau$. We denote the reflexive transitive closure of $\blacktriangleright_{\mathrm{h}}$ by $\blacktriangleright_{\mathrm{h}}^*$.
3. Let D be a set of head-instantiated dependency pairs. Then

$$\mathcal{U}_R^{\mathrm{h}}(D^\sharp) = \{l \to r \in R \mid f \blacktriangleright_{\mathrm{h}}^* \mathrm{head}(l) \text{ for some } f \in \Sigma_{\mathrm{d}}(\mathrm{RHS}(D^\sharp))\}$$

Theorem 16 (usable rules refinement) Let $\mathcal{R} = \langle \Sigma, R \rangle$ be an STTRS and D a finite set of head-instantiated dependency pairs. If there exists a reduction pair $\langle \gtrsim, > \rangle$ on $\mathrm{S}(\Sigma^\sharp \cup \{\mathsf{cons}, \mathsf{nil}\}, V)$ such that $C_\mathcal{E} \cup \mathcal{U}_R^{\mathrm{h}}(D^\sharp) \subseteq \gtrsim$, $D^\sharp \subseteq \gtrsim$, and $D^\sharp \cap > \neq \emptyset$, then $D \notin \mathbf{DC}_{\min}$.

Example 17 (termination proof) Let $\mathcal{R}$ be the STTRS of Example 1. The set of dependency pairs of $\mathcal{R}$ is as follows:

$$\left\{ \begin{array}{lllll} (5) & \mathsf{map}\ F\ (:\ x\ xs) & \rightarrowtail & F\ x \\ (6) & \mathsf{map}\ F\ (:\ x\ xs) & \rightarrowtail & \mathsf{map}\ F\ xs \\ (7) & (\circ\ F\ G)\ x & \rightarrowtail & F\ (G\ x) \\ (8) & (\circ\ F\ G)\ x & \rightarrowtail & G\ x \\ (9) & \mathsf{twice}\ F & \rightarrowtail & \circ\ F\ F \\ (10) & \mathsf{twice}\ F & \rightarrowtail & \circ \\ (11) & (\mathsf{twice}\ F)\ x & \rightarrowtail & (\circ\ F\ F)\ x \end{array} \right\}.$$

Two SCCs are obtained from its (approximated) dependency graph—namely, $\{(6)\}$ and $\{(7),(8),(9)\}$. Let $D = \{(7),(8),(9)\}$. The head instantiation and head marking of D yields the following set D' of simply typed dependency pairs:

$$\left\{\begin{array}{lll} (7a) & (\circ^\sharp\ (\circ\ U\ V)\ G)\ x & \rightarrowtail\ (\circ^\sharp\ U\ V)\ (G\ x) \\ (7b) & (\circ^\sharp\ (\mathsf{twice}\ U)\ G)\ x & \rightarrowtail\ (\mathsf{twice}^\sharp\ U)\ (G\ x) \\ (8a) & (\circ^\sharp\ F\ (\circ\ U\ V))\ x & \rightarrowtail\ (\circ^\sharp\ U\ V)\ x \\ (8b) & (\circ^\sharp\ F\ (\mathsf{twice}\ U))\ x & \rightarrowtail\ (\mathsf{twice}^\sharp\ U)\ x \\ (9) & (\mathsf{twice}^\sharp\ F)\ x & \rightarrowtail\ (\circ^\sharp\ F\ F)\ x \end{array}\right\}.$$

We have $\mathcal{U}^{\mathrm{h}}_R(D') = \{(3),(4)\}$. By taking a stable argument filtering π such that $\pi(F^{\mathrm{o}\to\mathrm{o}},0) = \pi(\circ,1) = \pi(\mathsf{twice},1) = 1$, $\pi(\mathsf{twice},0) = \pi(\mathsf{twice}^\sharp,0) = \pi(\circ^\sharp,1) = [0,1]$, and $\pi(\circ,0) = \pi(\circ^\sharp,0) = [0,1,2]$, we get the following set of constraints:

$$\left\{\begin{array}{lcl} x & \geq & x \\ \mathsf{twice}\ F & \geq & \circ\ F\ F \\ (\circ^\sharp\ (\circ\ U\ V)\ G)\ x & > & (\circ^\sharp\ U\ V)\ (G\ x) \\ (\circ^\sharp\ (\mathsf{twice}\ U)\ G)\ x & > & (\mathsf{twice}^\sharp\ U)\ (G\ x) \\ (\circ^\sharp\ F\ (\circ\ U\ V))\ x & > & (\circ^\sharp\ U\ V)\ x \\ (\circ^\sharp\ F\ (\mathsf{twice}\ U))\ x & > & (\mathsf{twice}^\sharp\ U)\ x \\ (\mathsf{twice}^\sharp\ F)\ x & > & (\circ^\sharp\ F\ F)\ x \end{array}\right\}.$$

All constraints are satisfied by the lexicographic path ordering for S-expressions [Toy04] with the precedence $\mathsf{twice} > \mathsf{twice}^\sharp > \circ^\sharp$ and $\mathsf{twice} > \circ > \circ^\sharp$. It is not hard to show $\{(6)\} \notin \mathbf{DC}_{\min}$ in a similar way. Thus $\mathcal{R}$ is terminating.

References

[AG00] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236(1–2):133–178, 2000.

[AY05] T. Aoto and T. Yamada. Dependency pairs for simply typed term rewriting. In *Proc. of RTA 2005*, volume 3467 of *LNCS*, pages 120–134. Springer-Verlag, 2005.

[Bla06] F. Blanqui. Higher-order dependency pairs. In *Proc. of WST 2006*, pages 22–26, 2006.

[GTSK05] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. of FroCoS 2005*, volume 3717 of *LNAI*, pages 216–231. Springer-Verlag, 2005.

[HM04] N. Hirokawa and A. Middeldorp. Dependency pairs revisited. In *Proc. of RTA 2004*, volume 3091 of *LNCS*, pages 249–268. Springer-Verlag, 2004.

[SK05] M. Sakai and K. Kusakari. On dependency pair method for proving termination of higher-order rewrite systems. *IEICE Trans. on Inf. & Sys.*, E88-D(3):583–593, 2005.

[TGSK04] R. Thiemann, J. Giesl, and P. Schneider-Kamp. Improved modular termination proofs using dependency pairs. In *Proc. of IJCAR 2004*, volume 3097 of *LNAI*, pages 75–90. Springer-Verlag, 2004.

[Toy04] Y. Toyama. Termination of S-expression rewriting systems: Lexicographic path ordering for higher-order terms. In *Proc. of RTA 2004*, volume 3091 of *LNCS*, pages 40–54. Springer-Verlag, 2004.

[Yam01] T. Yamada. Confluence and termination of simply typed term rewriting systems. In *Proc. of RTA 2001*, volume 2051 of *LNCS*, pages 338–352. Springer-Verlag, 2001.

Non-standard reductions in simply-typed, higher order and dependently-typed systems

Lionel Marie-Magdeleine and Serguei Soloviev
Institut de Recherche en Informatique de Toulouse
Université Paul Sabatier, Toulouse, France

Abstract of HOR 2007 talk on June 25, 2007

Earlier we studied some possibilities to add non-standard reductions to typed lambda-calculus with inductive types in such a way that SN and CR properties would be preserved. The aim was to open new possibilities for direct incorporation of computational algorithms in proof assistants. Some new methods to prove SN and CR were developped. In this talk we discuss their generalization to higher order and dependent type system cases.

CRSX – An Open Source Platform for Experiments with Higher Order Rewriting (Extended Abstract)

Kristoffer H. Rose
IBM Thomas J. Watson Research Center*

June 4, 2007

Abstract

The SourceForge "CRSX" project aims at implementing a generic higher order rewrite engine based on Klop's Combinatory Reduction Systems (CRS) formalism. The specific goals of the CRSX project are to

- provide a generic higher order rewrite engine that
- is easy to embed in other projects such as compiler optimizers,
- is simple to extend with experimental features, and
- runs on a universally available open source platform.

This paper summarizes how the current prototype implementation in Java (an open source platform) partially achieves these goals: the CRS abstraction that is actually implemented, the Java interfaces allowing using the engine on "foreign" terms, and some extensions.

1 Introduction

Higher order rewriting is well established as a useful generic mechanism for formalizing program transformation and evaluation when binding constructs are involved. Furthermore, mappings from most other formal mechanisms into higher order rewrite systems have been described in the literature. This would seem to make higher order rewriting an ideal mechanism for analysis and optimization in compilers in general, and in particular for compiler that use an internal "intermediate" or "core" language for analysis and optimizations. Some languages are even defined in terms of a core language [MTH90, FHPe92], which allows for discussing appropriate optimization analysis and transformation in the literature. However, it seems that every compiler writer invents her own "symbolic rewriting" engine to express these optimizations [RSF06]. We will argue that the techniques of higher-order rewriting in general (CRS [Klo80, KvOvR93] with just a few encoding tricks in particular) provide excellent expressive support for the *practical* analyses and transformations required for functional programs.

*P. O. Box 704, Yorktown Heights, NY 10598 (USA); http://www.research.ibm.com/people/k/krisrose.

The CRSX engine [Ros07, Ros06] supports expression of CRS rules directly over foreign expressions by having a rewrite engine defined entirely over an abstract or *virtual* notion of term expressed through a Java *interface* definition, which explains how expressions correspond to notions the CRS rewrite engine can understand, *i.e.*, in terms of "variables", "construction", "binding", *etc.* Furthermore, the CRSX interfaces provide abstract "hooks" designed to permit writing rule *schemas* that correspond to infinite enumerations of similar rules.

Below we outline the key definition of *virtualized CRS* in Sec. 2 and the Java interfaces realizing it in Sec. 3, including a simple example. In Sec. 4 we summarize the state of the CRSX project and suggest extensions for the SourceForge project [Ros07].

2 Virtualized CRS

To understand how CRS are virtualized we first present a (brief) traditional definition of CRS [Klo80, KvOvR93].

Definition 1 (CRS terms). The CRS *terms* over the *variables* $v \in V$, ranked *function symbols* $f^n \in F$, and ranked *metavariables* $z^n \in Z$, are the terms $t \in T$ generated by the grammar

$$t ::= v \mid f^n(b_1, \ldots, b_n) \mid z^n(t_1, \ldots, t_n) \qquad \text{(term)}$$
$$b ::= v.b \mid t \qquad \text{(binder)}$$

where the three term forms are called *variable occurrence*, *construction*, and *metaapplication*, respectively, and the purpose of $v.b$ is to *bind* all free occurrences of v in b as usual (but binders are restricted to occur as immediate subterms of constructions). The free variables of a term t are denoted $fv(t)$.

Definition 2 (CRS rewriting). A set of term pairs $R \subseteq (T \times T)$, each written $t_L \to t_R$, constitutes a set of *rewrite rules* if

- every t_L is a *pattern*, *i.e.*, a closed construction where all contained metaapplications have the form $z^n(v_1, \ldots, v_n)$ with distinct $v_1 \ldots v_n$, and
- every t_R is a *contractum* for the corresponding pattern, *i.e.*, all the metavariables must also occur in the pattern.

R generates the *rewrite relation* $\xrightarrow[R]{}$ over T defined by the following steps:

- A *valuation* σ is a map of type $Z \to (V^* \times T)_\perp$ where for all z^n either $\sigma(z^n) = \perp$ or $\sigma(z^n) = \langle\langle v_1, \ldots, v_n\rangle, t\rangle$ with distinct $v_1 \ldots v_n$ and $fv(t) \subseteq \{v_1, \ldots, v_n\}$.
- The *contraction* of a term t with the valuation σ is written as $\sigma(t)$ and defined as the homomorphic extension to terms of the rule that if $\sigma(z^n) = \langle\langle v_1, \ldots, v_n\rangle, t\rangle$ then $\sigma(z^n(t_1, \ldots, t_n)) = t[v_1 := \sigma(t_1), \ldots, v_n := \sigma(t_n)]$ (using usual simultaneous substitution).
- $\xrightarrow[R]{}$ relates all pairs of terms $C[\sigma(t_L)] \xrightarrow[R]{} C[\sigma(t_R)]$ for some rule $t_L \to t_R$ in R, context $C[]$, and valuation σ.

Remark 3 *(omissions).* We have here skipped over various "safety" constraints intended to avoid variable capture, *etc.*, as these are well covered in the literature [KvOvR93] and

will not interfere with virtualization below. We also assume standard set and domain notations such as implicit naming like $x, x', x_n, \ldots \in X$, $X_\perp$ for the usual lifting of X to contain a distinct $\perp$ member, and X^* for the set of sequences of X-members $\vec{x} = \langle x_1, \ldots, x_n \rangle$, including the empty sequence $\langle\rangle$.

The traditional definition relies on concrete comparison of terms with the notion of rewriting defined directly on terms. However, to realize a generic rewrite engine we need to separate the mechanics of rewriting from the concrete terms. The solution is to shift to an extensional representation where the set of function symbols F is not exposed except through a mechanism for function symbol matching.

The key to understanding how rewriting happens is manifest in the last part of Def. 2: "[rewriting] which relates all pairs of terms $C[\sigma(t_L)] \xrightarrow[R]{} C[\sigma(t_R)]$ for some rule $t_L \to t_R$, context $C[]$, and valuation σ." This means that the only extensionally observable properties of terms are to

- allow navigation to place in a term corresponding to the "hole" of the context $C[]$ – the (potential) *redex*,
- permit establishing whether a valuation σ can be created that maps the pattern of some rule to the redex, and
- construct a fresh copy of the term which is identical to the original term except for the context being filled with the result of contracting the rule right hand side with the valuation.

These properties are captured by the following.

Definition 4 (virtual CRS terms). Assume T, V, and Z, as in Def. 1, and the following operations:

$\#\colon T \to N$	with N the natural numbers from 0	(arity)
$v\colon T \to V_\perp$		(variable occurrence check)
$z\colon T \to Z_\perp$		(metavariable check)
$b\colon T \times N \to (V^*)_\perp$		(binder check)
$s\colon T \times N \to T_\perp$		(subterm check)
$m\colon T \times T \times \Sigma \to \Sigma_\perp$		(match)
$cc\colon T \times B^* \to T$	where $B = V^* \times T$	(copy constructor)
$cv\colon V \to T$		(copy variable occurrence)

The structure $\langle T, V, Z, \#, v, z, b, s, m, cc, cv \rangle$ is a *virtual CRS term structure* if the following constraints are satisfied:

- Subterms and bindings exist only where expected: $1 \le i \le \#(t) \iff s(t,i) \neq \perp \iff b(t,i) \neq \perp$,
- Variables are consistent, *i.e.*, $v(t) = v$ implies $\#(t) = 0$, $z(t) = \perp$, $m(t,t',\sigma) = \perp$, and $cc(t,\vec{b}) = \perp$.
- Metaapplications are consistent, *i.e.*, $z(t) = z^n$ implies $\#(t) = n$, $v(t) = \perp$, $b(t,i) = \langle\rangle$ for $1 \le i \le n$, $m(t,t',\sigma) = \perp$, and $cc(t,\vec{b}) = \perp$.

- Construction (identified by $\mathsf{v}(\mathsf{t}) = \mathsf{z}(\mathsf{t}) = \bot$ and with $\#(\mathsf{t}) = \mathfrak{n}$ subterms with optional binders $\vec{\mathsf{b}} = \langle\langle \mathsf{b}(\mathsf{t},1), \mathsf{s}(\mathsf{t},1)\rangle, \ldots, \langle \mathsf{b}(\mathsf{t},\mathfrak{n}), \mathsf{s}(\mathsf{t},\mathfrak{n})\rangle\rangle$) is consistent, *i.e.*, $\mathsf{s}(\mathsf{cc}(\mathsf{t},\vec{\mathsf{b}}),\mathsf{i}) = \mathsf{s}(\mathsf{t},\mathsf{i})$, $\mathsf{b}(\mathsf{cc}(\mathsf{t},\vec{\mathsf{b}}),\mathsf{i}) = \mathsf{b}(\mathsf{t},\mathsf{i})$, and $\mathsf{m}(\mathsf{cc}(\mathsf{t},\vec{\mathsf{b}}),\mathsf{t}',\sigma) = \mathsf{m}(\mathsf{t},\mathsf{t}',\sigma)$.
- Finally, variable creation works: $\mathsf{v}(\mathsf{cv}(\nu)) = \nu$.

Definition 5 (virtual CRS rewriting). Define rewrite rules R and valuations as in Def. 2. The rewrite relation $\xrightarrow[\mathsf{R}]{}$ over virtual CRS terms is defined as follows:

- The *contraction* of a virtual term with a valuation is defined relative to the operations.
- A *path* p is a sequence of integers corresponding to trail of s-indices; it is valid if it selects a subterm.
- A *virtual context* $\mathrm{C}[] = \langle \mathrm{t}_\mathrm{C}, \mathrm{p}_\mathrm{C}\rangle$ where p_C is valid into t_C and defined in terms of paths.
- Rewriting is defined just as in Def. 2.

Proposition 6. *Given a set of CRS terms* T *over* V, F, *and* Z *(Def. 1), and a virtual CRS term structure* $\langle \mathsf{T},\mathsf{V},\mathsf{Z},\#,\mathsf{v},\mathsf{z},\mathsf{b},\mathsf{s},\mathsf{m},\mathsf{cc},\mathsf{cv}\rangle$ *(Def. 4). Then the rewrite relations generated by the traditional (Def. 2) and virtualized (Def. 5) mechanism are equal.*

Proof sketch. We construct the "implementation" of the virtual CRS operations over real terms and show that (1) they obey the constraints and (2) the constraints create at least as many relations as the concrete rewrite relation has. □

3 CRSX

The CRSX code [Ros07] is based on virtual CRS terms realized by the Java interface CRSTerm, summarized in Fig. 1, with some supporting interfaces in net.sf.crsx, over which the rewrite engine CRS is implemented.

This realizes virtual CRS as discussed in the previous section with the following further extensions that we have not yet fully formalized:

- Matching is constrained by requiring that the two methods crsPreMatch and crsPostMatch succeed on *all* terms, which can be used to add *additional match constraints*.
- All variable occurrences are created based on the same unique variable object shared with the binding. This allows properties to be associated to variables *and accessed uniformly from all occurrences*.
- All contraction invokes methods on the contractum: crsCopyConstructor for constructors, crsCopyVariableOccurrence for variable occurrences, and crsMetaApplicationSubstitution for metaapplications.
- Destructive updating is supported to allow "in-place" updating of terms.

These additional conventions effectively allow working with rule "schemas" (finite descriptions of an infinity of simple rules).

```
interface CRSTerm // Term t.
{
 enum CRSKind {CONSTRUCTOR,VARIABLE_OCCURRENCE,META_APPLICATION};
 public CRSKind crsKind(); // fast term kind dispatch

 // Semantic abstract terms.
 int crsArity(); // #(t)
 CRSVariable crsVariable(); // v(t) using null for ⊥
 String crsMetaVariable(); // z(t) using null for ⊥
 CRSVariable[] crsBinders(int i); // b(t,i)
 CRSTerm crsSub(int i); // s(t,i)
 boolean crsPreMatch(CRSTerm other, CRS crs); // m(t, t_other, σ)
 boolean crsPostMatch(CRSTerm other, CRSMatching matching);
 CRSTerm crsCopyConstructor(CRSVariable[][] subbinders, CRSTerm[] subterms); // cc(t, b⃗)
 CRSTerm crsCopyVariableOccurrence(CRSVariable variable); // cv(v)

 // Destructive update support.
 void crsReplaceSub(int i, CRSTerm subterm);

 // Rule schema support.
 CRSTerm crsMetaApplicationSubstitution(CRSValuation valuation, int sequence,
              CRSRenaming renaming, CRSTerm copy);
}
```

Figure 1: CRSX (net.sf.crsx) CRSTerm interface highlights.

Example 7 (XQuery with annotations). We use CRSX to implement the optimizer of our "Virtual XML" XQuery compiler [RV+06]. The compiler comes with an existing notion of internal "core language" with an abstract syntax tree (AST) implementation in Java. To make CRSX work over these, the Java AST implementation was extended to implement CRSTerm by

- mapping all the Java AST constructs to CRS constructors with appropriate binders, and
- inserting a dummy "properties" constructor to hold annotations.

With this encoding, rewrite rules like the following can be used directly; R: is the prefix used for pseudo-notations (so they can be parsed as XQuery expressions).

```
typeswitch ( R:type(R:Expr0() instance of R:Type) )
  case $c as R:Type return R:YesCase1($c)
  default $d return R:NoCase($d)
→
let $c as R:Type := R:Expr0() return R:YesCase1($c)
```

The trick is to encode the pseudo-terms in such a way that the pre- and post-matching takes care of any additional match constraints. In this example, the pattern matches if the expression is a **typeswitch** expression where the selection expression has been shown to have a type, R:Type, which also appears in the type switch case. If so then the expression rewrites to a simple **let** declaration.

It is also possible to encode analyses and inference rules.

4 Conclusion

The CRSX engine is available on SourceForge [Ros07] and already works well enough to be used by our XQuery compiler [RV+06]. It has extensions for simple "rule schema" as summarized above but these are not yet formally founded. We hope to improve this through a "CRSX community" to get a great CRS engine that can be embedded easily by other projects like we have done with ours.

Future work. The next steps will be to implement much more of the CRS culture in CRSX: first of all classical formal CRS tests such as critical pair search (and orthogonality), configurable rewrite strategies, and a more precise notion of rule schema to support, for example, pattern calculi [JK06]. Other obvious experiments include providing a CRS term model for XML [BPSM+06] data and for CRS systems themselves. Finally, we shall have to implement a proper compiler for CRS to avoid having to load the CRS rules on every run: an interesting aspect of this is what parts of code generation should be delegated to the term representation.

References

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.0 (fourth edition). Recommendation, W3C, August 2006.

[FHPe92] Joseph H. Fasel, Paul Hudak, Simon Peyton Jones, and Philip Wadler (editors). Haskell special issue. *SIGPLAN Notices*, 27(5), May 1992.

[JK06] C. Barry Jay and Delia Kesner. Pure pattern calculus. In *Proceedings of the European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, pages 100–114, Vienna, Austria, March-April 2006. Springer-Verlag.

[Klo80] Jan Willem Klop. *Combinatory Reduction Systems*. Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

[KvOvR93] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theoretical Computer Science*, 121(1-2):279–308, 1993.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Ros06] Kristoffer Rose. Stand-alone use of higher-order rewriting. Blog entry on http://domino.research.ibm.com/comm/research_people.nsf/pages/krisrose.blog.html, December 2006.

[Ros07] Kristoffer Rose. Combinatory reduction systems extended. SourceForge project http://crsx.sf.net, April 2007.

[RSF06] Christopher Ré, Jérôme Siméon, and Mary Fernández. A complete and efficient algebraic compiler for XQuery. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, Atlanta, Georgia, April 2006.

[RV+06] Kristoffer Rose, Lionel Villard, et al. Virtual XML. http://www.research.ibm.com/virtualxml, November 2006.

Demonstration of CRSX

Kristoffer H. Rose
IBM Thomas J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598 (USA)
http://www.research.ibm.com/people/k/krisrose

Abstract of HOR 2007 system demo on June 25, 2007

I'll demonstrate the capabilities of the CRSX engine:

- Using "traditional CRS", http://crsx.sourceforge.net/crsx-howto.pdf
- Use of CRSX in the Virtual XML Query compiler, http://www.alphaworks.ibm.com/tech/virtualxml

Everything is done with general normalization using CRS rewrite rules.

Elements of a Categorical Semantics for the Open Calculus of Constructions

Max Schäfer*
Institute of Information Science
Academia Sinica, Taipei 115, Taiwan

June 1st, 2007

The Open Calculus of Constructions [Ste02] is a type theory in the style of the Calculus of Constructions [CH88] incorporating concepts from Rewriting Logic [OM93]. Based on our exposition in [Sch07], we describe a categorical semantics for a subsystem of OCC, which is obtained quite straightforwardly by enriching a D-category based semantics with a 2-category structure. The resulting semantics is sound and conceivably encompasses a wide variety of different models for the calculus; thus it could serve as a basis for further meta-theoretical investigations of OCC and perhaps also related systems.

1 The System OCC_3

The variant of the Open Calculus of Constructions described here, which for consistency with [Sch07] is called OCC_3, provides four judgement forms:

- Typing judgements of the form $\Gamma \vdash M : A$: Such a judgement expresses that in the context Γ, which declares the types of all used variables, the term M has type A.
- Structural equality judgements of the form $\Gamma \vdash ||(M = N) : A$: Such a judgement expresses that in the context Γ the terms M and N are both of type A and operationally indistinguishable (i.e., they should be treated as interchangeable).
- Computational equality judgements of the form $\Gamma \vdash !!(M = N) : A$: Such a judgement expresses that in the context Γ the terms M and N are both of type A, and M can be reduced to N (but not the other way around).
- Rewriting judgements of the form $\Gamma \vdash M \to N : A$: Such a judgement expresses that in the context Γ the terms M and N are both of type A, and there exists an (abstract) rewrite from the former to the latter.

Notice that this last form of judgement does not appear in the original formulation of OCC. On the other hand, several of its other judgements involving

*This work was partially supported by the iCAST project sponsored by the National Science Council, Taiwan, under grants no. NSC95-3114-P-001-001-Y02 and NSC95-3114-P-001-002-Y02.

features like assertional equality and subtyping, which are not considered in our semantics, are omitted.

In general, OCC_3's syntax has more typing annotations than the original OCC, which are needed for defining the semantic interpretation functions. This is a common phenomenon when describing categorical semantics for type theories; see, for example, [Str91] for a discussion of this issue.

2 Inference Rules and Semantics

For space reasons, we cannot give all the inference rules of OCC_3 here (see [Sch07] for a complete list); according to the judgement form of the conclusion, however, we can roughly distinguish four groups of rules, which correspond to different aspects of the semantics.

2.1 Modelling Contexts, Universes, and Products

The rules for deriving typing judgements are mostly similar to those seen in the Calculus of Constructions and this part of the calculus can indeed be modelled using D-categories in a similar way as it was done for CC by Ehrhard [Ehr88]:

Contexts and their morphisms (which can be seen as generalized substitutions) are interpreted as the objects and morphisms of a base category $\mathcal{C}$. Another category $\mathcal{E}$, which is fibred over $\mathcal{C}$ through a fibration functor $p\colon \mathcal{E} \to \mathcal{C}$, contains the semantic representations of types and terms.

Thus, a context Γ is modelled as an object C of $\mathcal{C}$. The fiber over C is the subcategory $\mathcal{E}_C$ of $\mathcal{E}$ consisting of all objects $X \in \mathrm{Ob}(\mathcal{E})$ such that $p(X) = C$ and all arrows f such that $p(f) = \mathrm{id}_C$. Types and terms in context Γ are interpreted inside this fiber: For a derivable judgement $\Gamma \vdash M\colon A$, the interpretation functions yield an object a representing type A and a morphism $m\colon \mathbf{1}_C \to a$ representing term M, where $\mathbf{1}_C$ is the terminal object of $\mathcal{E}_C$.

Substitutions are modelled as context morphisms, i.e. arrows in $\mathcal{C}$. Since p is a (split) fibration, every such morphism $\vartheta\colon D \to C$ induces a reindexing functor $\vartheta^*\colon \mathcal{E}_C \to \mathcal{E}_D$ modelling the application of a substitution to terms and types. An example scenario is depicted in the diagram in Figure 1, where we follow the convention of Jacobs [Jac91] depicting the objects and morphisms of a fiber vertically above the corresponding object of the base category.

As usual, product types are modelled as right adjoints to weakening functors plus a coherence condition. The product forming operation of OCC_3 (like the original OCC) is very powerful, since it allows product formation over a user-definable universe hierarchy similar to pure type systems [Bar92]. These universes are specified as a set of universe constants $\mathcal{S}$, with two disjoint subsets $\mathcal{S}_i$ and $\mathcal{S}_p$ of impredicative and predicative universes, respectively. The product forming rule depends, as with pure type systems, on a ternary relation $\mathcal{R} \subseteq \mathcal{S}^3$ (which is required to be functional in its first two arguments):

$$\text{(Pi)}\ \frac{\Gamma \vdash S\colon s_1 \qquad \Gamma, x\colon S \vdash T\colon s_2}{\Gamma \vdash \Pi x\colon S.T\colon s_3}\ , \mathcal{R}(s_1, s_2) = s_3$$

It is well known that unbridled use of this feature can lead to inconsistent systems (see, e.g., [Bar92]), so OCC poses a number of technical constraints to

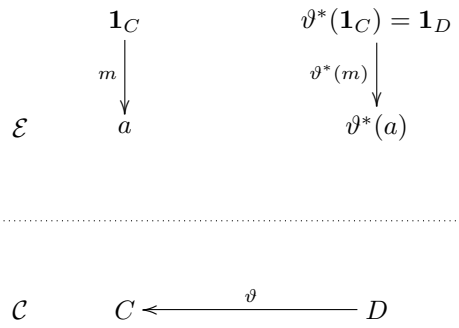

Figure 1: Contexts, context morphisms, and reindexing functors

ensure that potentially dangerous systems are weeded out. For details we refer to [Ste02] and [Sch07].

In the categorical semantics, the universes are modelled as designated objects of $\mathcal{E}$ with a mapping $\mathcal{U}$ that converts between their categorical elements and objects of $\mathcal{E}$. This reflects the double role of types in the calculus, which can be seen both as terms belonging to a universe, and as types to which other terms belong.

For instance, a type `Nat` of natural numbers could inhabit a universe `Type` of types, and could itself be inhabited by the term `42`. In its first role, `Nat` would be modelled as an object N of category $\mathcal{E}$, and `42` as its categorical element. In its second role, on the other hand, `Nat` would be modelled as a categorical element n of another object representing universe `Type`. The mapping $\mathcal{U}$ maps the arrow n to the object N, thus corresponding to the (syntactic) constructor `Proof(-)` found in some formulations of CC.

2.2 Modelling Equality and Rewriting

The rules for deriving equality and rewriting judgements form the bulk of the inference system. Following [Mes05], we model rewriting by a 2-category structure. Briefly, a 2-category is a category in which every homset has itself a category structure; in particular, there are "morphisms between (parallel) morphisms", which are called 2-cells.

Since terms are modelled as morphisms of $\mathcal{E}$, it seems natural to model rewrites between terms as 2-cells between the corresponding morphisms. We then need to require that all categories, functors, and adjunctions in the model are in fact 2-categories, 2-functors, and 2-adjunctions (i.e., they also act on 2-cells).

Using this approach, we can neatly model the equality/rewriting hierarchy of OCC by corresponding classes of 2-cells:

- Structural equality is modelled by the class of identity 2-cells: If two terms are structurally equal, then there should be an identity 2-cell between their interpretations, i.e. their interpretations should be equal.
- Computational equality is modelled by a class of *reduction 2-cells*: If two

terms are computationally equal, then there should be such a reduction 2-cell between their interpretations.

- Rewriting is modelled by the class of all 2-cells: If one term can be rewritten to another, there should be a 2-cell between the former and the latter's interpretation.

The class of reduction 2-cells is closed under identity and vertical composition, mirroring the reflexivity and transitivity of computational equality. The same is trivially true of the class of identity 2-cells which is additionally closed under "change of direction": for every identity 2-cell $\alpha\colon f \Rightarrow g$, there is another identity 2-cell $\beta\colon g \Rightarrow f$ (in fact, $f = g$ and $\beta = \alpha$). This is, of course, needed to model symmetry of structural equality. Further flavors of equality could likewise be modelled by defining a corresponding class of 2-cells.

The rules for deriving structural equality judgements describe the basic properties of structural equality, and also the structural equality type constructor: For two terms M and N of type A, there is a type $\mathrm{StrEq}_A(M, N)$ of proofs of structural equality for M and N. Following an approach outlined by Streicher [Str91], we model this type constructor as a family of objects in $\mathcal{E}$ with two *extraction morphisms* that allow access to the terms M and N being compared.

This approach readily generalizes to computational equality types $\mathrm{CompEq}_A(M, N)$ and rewriting types $\mathrm{Rw}_A(M, N)$ by requiring that the extraction morphisms be connected by a reduction 2-cell resp. any 2-cell at all. Also, the existence of an inhabitant of a structural or computational equality or rewriting type implies that the two terms compared are actually structurally equal/computationally equal/rewritable, thus these type constructors are in fact strong.

The rules for the computational equality and rewriting judgements, finally, closely parallel the structural equality rules. But while structural equality exhibits congruence closure (roughly speaking, if the individual parts of two terms are equal, then so are the terms themselves), computational equality does not. Computational equality can be used for type reduction: If a term M has type A and type A is computationally equal to type A', then term M also has type A'. The rewriting judgement, however, is completely passive in this respect: the existence of a rewrite between two terms does not affect their typing or operational behavior.

This completes our overview of the semantics. While the precise definition of the interpretation functions is a bit technical, it is not very difficult, and one readily obtains a soundness proof.

3 Future Directions and Related Work

We are confident that it will not be too hard to find examples of categorical models for OCC_3: Although we have not yet finished all the detailed verifications, it seems that a set-based model as well as a term model can be defined in a more or less standard way. One might hope that the term model could then be used to obtain a completeness proof of the inference system.

Another important use of example models would be the construction of counter-examples; realizability models in particular have proved to be a fertile source of independence proofs for the Calculus of Constructions [Str91].

Finally, OCC is not the only system aiming at a unification of type theory and rewriting. The same two ingredients can be found in the Rho-Calculus [CLW03], which however pursues quite different goals and uses a different approach. Although it is wider in scope and also more powerful than OCC, it might be interesting to see if the ideas outlined here could be applied to it.

References

[Bar92] Henk Barendregt. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Clarendon Press, Oxford, 1992.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2–3), 1988.

[CLW03] Horatiu Cirstea, Luigi Liquori, and Benjamin Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. In *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[Ehr88] Thomas Ehrhard. A Categorical Semantics of Constructions. In *Proceedings of LICS '88*, pages 264–273. IEEE, 1988.

[Jac91] Bart Jacobs. *Categorical Type Theory*. PhD thesis, University of Nijmegen, The Netherlands, September 1991.

[Mes05] José Meseguer. Functorial semantics of rewrite theories. *Lecture Notes in Computer Science : Formal Methods in Software and Systems Modeling*, pages 220–235, 2005.

[OM93] Martí N. Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework, 1993.

[Sch07] Max Schäfer. Towards a Categorical Semantics for the Open Calculus of Constructions. Master's thesis, TU Dresden, 2007. Available online at http://www.iis.sinica.edu.tw/∼xiemaisi/publications.html.

[Ste02] Mark-Oliver Stehr. *Programming, Specification, and Interactive Theorem Proving*. PhD thesis, University of Hamburg, September 2002.

[Str91] Thomas Streicher. *Semantics of Type Theory: Correctness and Completeness*. Progress in Theoretical Computer Science. Birkhäuser, December 1991.

Principal Typings for Explicit Substitutions Calculi*

Daniel Lima Ventura[1†] and Mauricio Ayala-Rincón[1‡] and Fairouz Kamareddine[2]

[1]Grupo de Teoria da Computação, Dep. de Matemática
Universidade de Brasília, Brasília D.F., Brasil

[2] School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, Scotland

{ventura,ayala}@mat.unb.br, fairouz@macs.hw.ac.uk

June 4, 2007

Abstract

Having principal typings (for short PT) is an important property of type systems. This property guarantees the possibility of type deduction which means it is possible to develop a complete and terminating type inference mechanism. It is well-known that the simply typed λ-calculus has this property, but recently, J. Wells has introduced a system-independent definition of PT which allows to prove that some type systems, e.g. the Hindley/Milner type system, do not satisfy PT. The main computational drawback of the λ-calculus is the implicitness of the notion of substitution, a problem which in the last years gave rise to a number of extensions of the λ-calculus where the operation of substitution is treated explicitly. Unfortunately, some of these extensions do not necessarily preserve basic properties of the simply typed λ-calculus such as preservation of strong normalization. We consider two systems of explicit substitutions ($\lambda\sigma$ and λs_e) and we show that they can be accommodated with an adequate notion of PT. Specifically, our results can be summarized as follows:

• We introduce PT notions for the simply typed versions of the $\lambda\sigma$ and the λs_e-calculus that are proved to agree with Wells' notion of PT.

• We show that these versions of the $\lambda\sigma$ and the λs_e satisfy PT by revisiting previously introduced type inference algorithms.

1 Introduction

The development of well-behaved calculi of explicit substitutions is of great interest in order to bridge the formal study of the λ-calculus and its real implementations. Since β contraction depends on the definition of the operation of

*Research supported by the CNPq Brazilian Research Council.

†Corresponding author, currently supported by a PhD scholarship of the CNPq at the Heriot-Watt University.

‡Author partially supported by the CNPq.

substitution, which is informally given in the theory of λ-calculus, substitutions are in fact made explicit, but obscurely developed (in and *ad hoc* manner), when most computational environments based on the λ-calculus are implemented. A remarkable exception is λProlog, for which its explicit substitutions calculus, the suspension calculus, has been extracted and formally studied [NaWi98].

In the study of making substitutions explicit, several alternatives rose out and all of them are directed to guarantee essential properties such as simulating beta-reduction, confluence, noetherianity (of the associated substitution calculus), subject reduction, having principal typings (for short PT), preservation of strong normalization etc. This is a non trivial task; for instance, the $\lambda\sigma$-calculus [ACCL91], that is one of the first proposed calculi of explicit substitutions, was reported to break the latter property after some years of its introduction [Mel95]: this implies that infinite derivations starting from well-typed λ-terms are possible in this calculus, which is at least questionable for any mechanism supposed to simulate the λ-calculus explicitly. Here the focus is on the PT property, which means that for any typable term a, there exists a type judgment $\Gamma \vdash a : A$, representing all possible typings for a, where for a typing of a one understands the pair (Γ, A). In the simply typed λ-calculus this corresponds to the existence of *more representative* typings. PT guarantees compositional type inference helping in making a complete/terminating type inference algorithm.

We assume the reader familiar with λ-calculus in de Bruijn notation and with the type-free and simply typed versions of the $\lambda\sigma$ [ACCL91] and λs_e-calculi [KR97]. The proofs are included in an extended version of this work available at `www.mat.unb/~ayala/publications.html`. In section 2 we present the type assignment systems background and then we present simply typed systems for each calculus. Following type systems presentation, we discuss the general notion of principal typings defined in [We2002] and present notions of principal typings for λ-calculus in de Bruijn notation, $\lambda\sigma$ and λs_e and prove they are adequate ones. Then we conclude and present future work.

2 The Type Systems

Definition 1. *The syntax of the* **simple types** *and* **contexts***:*
Types $A ::= K \mid A \to A$ ***Contexts*** $\Gamma ::= nil \mid A.\Gamma$

K ranges over **type variables**. A **type assignment system** S is a set of rules which allows some terms of a given system be associated with a type. A **context** gives the necessary information used by S rules to associate a type to a term. In the simply typed λ-calculus[Hi97], the typable terms are strongly normalizing. The ordered pair (Γ, A), of a context and a type, is called a **typing in** S. For a term a, $\Gamma \vdash a : A$ denotes that a has type A in context Γ, and (Γ, A) is called a **typing of** a. Let $\tau = (\Gamma, A)$ be a typing in S. $S \rhd a : \tau$ denotes that τ is a typing of a in S.

The contexts for λ-terms in de Bruijn notation are sequences of types. Let Γ be some context and $n \in \mathbb{N}$. Then $\Gamma_{<n}$ denotes the first $n-1$ types of Γ. Similarly we define $\Gamma_{>n}$, $\Gamma_{\leq n}$ and $\Gamma_{\geq n}$. Note that, for $\Gamma_{>n}$ and $\Gamma_{\geq n}$ the final nil element is included. For n=0, $\Gamma_{\leq 0}.\Gamma$=$\Gamma_{<0}.\Gamma$=Γ. The length of Γ is defined as $|nil|$=0 and, if Γ is not nil, $|\Gamma|$=1+$|\Gamma_{>1}|$. The addition of some type A at the end of a context Γ is defined as $\Gamma.A$=$\Gamma_{\leq m}.A.nil$, where $|\Gamma|$=m.

Given a term a, an interesting question is whether it is typable in S or not. Note that, we are using the so-called Curry-style or implicit typing, where in terms of the form $\lambda.a$ we did not specify the type of the bound variable($\underline{1}$). Such terms have many types, depending on the context. Another important question is whether given a term, its so-called most general typing can be found. An answer to this question, which represents in some sense any other answer, is called **principal typing**. Principal typing(which is context independent) is not to be confused with a principal type(which is context dependent). Let τ be a typing in S and $\mathbf{Terms_S}(\tau)=\{a|S \rhd a{:}\tau\}$. J. Wells introduced in [We2002] a system-independent definition of PT and proved that it generalizes previous system-specific definitions.

Definition 2 ([We2002])**.** *A typing τ in system S is principal for some term a if $S \rhd a : \tau$ and for any τ' such that $S \rhd a : \tau'$ we have that $\tau \leq_S \tau'$, where $\tau_1 \leq_S \tau_2 \iff Terms_S(\tau_1) \subseteq Terms_S(\tau_2)$.*

In simply typed systems the principal typing notion is tied to type substitution and weakening. **Weakening** allows one to add unnecessary information to contexts. **Type substitution** maps type variables to types. Given a type substitution s, the extension for functional types is straightforward as $s(A{\rightarrow}B){=}s(A){\rightarrow}s(B)$ and the extension for sequential contexts as $s(nil){=}nil$ and $s(A.\Gamma){=}s(A).s(\Gamma)$. The extension for typings is given by $s(\tau){=}(s(\Gamma), s(A))$.

2.1 Principal typings for the simply typed λ-calculus in de Bruijn notation $TA_{\lambda dB}$

Definition 3. ***(The System $TA_{\lambda dB}$)*** *The $TA_{\lambda dB}$ typing rules are given by*

$$\textit{(}\lambda dB\textit{-var)} \quad A.\Gamma \vdash \underline{1} : A \qquad \textit{(}\lambda dB\textit{-varn)} \quad \frac{\Gamma \vdash \underline{n} : B}{A.\Gamma \vdash \underline{n+1} : B}$$

$$\textit{(}\lambda dB\textit{-lambda)} \quad \frac{A.\Gamma \vdash b : B}{\Gamma \vdash \lambda.b : A \rightarrow B} \qquad \textit{(}\lambda dB\textit{-app)} \quad \frac{\Gamma \vdash a : A \rightarrow B \quad \Gamma \vdash b : A}{\Gamma \vdash (a\ b) : B}$$

This system is similar to TA_{λ}([Hi97]).

Lemma 1. *Let a be a λ-term in de Bruijn notation. If $\Gamma \vdash_{TA_{\lambda dB}} a : A$, then $\Gamma.B \vdash_{TA_{\lambda dB}} a : A$. Hence, the rule ($\lambda dB$-weak) is admissible in the system $TA_{\lambda dB}$, where $\dfrac{\Gamma \vdash a : A}{\Gamma.B \vdash a : A}(\lambda dB\text{-weak})$.*

Using the rule (λdB-weak) and type substitution, we can define principal typing for the λ-calculus in de Bruijn notation similarly to the definition of [We2002] for Hindley's Principal Typing.

Definition 4. *A **principal typing** in $TA_{\lambda dB}$ of a λ-term a is the typing $\tau = (\Gamma, B)$ such that*

1. *$TA_{\lambda dB} \rhd a : \tau$*
2. *If $TA_{\lambda dB} \rhd a : \tau'$ for any typing $\tau' = (\Gamma', B')$, then exists some substitution s such that $s(\Gamma) = \Gamma'_{\leq|\Gamma|}.nil$ and $s(B) = B'$.*

Observe that, given the principal typing (Γ, A) of a, the context Γ is the shortest context where a can be typable. This property corresponds to the property of principal typing in the simply typed λ-calculus with names, where the context of a principal typing is the smallest set as well[We2002]. As is

the case for the simply typed λ-calculus with names, the best way to assure that Definition 4 is the correct translation of the PT concept, is to verify that Definition 4 corresponds to Definition 2.

Theorem 1. *A typing τ is principal in $TA_{\lambda dB}$ according to Definition 4 iff τ is principal in $TA_{\lambda dB}$ according to Definition 2.*

A type inference algorithm for terms from $TA_{\lambda dB}$ is presented, similar to the one in [AyMu2000] for λs_e. Given any term a, decorate each subterm with a new type variable as subscript and a new context variable as superscript, obtaining a new term denoted as a'. For example, for term $\lambda.(\underline{2}\ \ \underline{1})$ we have the decorated term $(\lambda.(\underline{2}\ {}^{\Gamma_1}_{A_1}\ \ \underline{1}\ {}^{\Gamma_1}_{A_2})^{\Gamma_3}_{A_3})^{\Gamma_4}_{A_4}$. Then, rules from Table 1 are applied to pairs of the form $\langle R, E\rangle$, where R is a set of decorated terms and E a set of equations on type and context variables.

(Var)	$\langle R \cup \{\underline{1}^{\Gamma}_{A}\}, E\rangle$	$\rightarrow \langle R, E \cup \{\Gamma = A.\Gamma'\}\rangle$,where Γ' is a fresh context variable;
(Varn)	$\langle R \cup \{\underline{n}^{\Gamma}_{A}\}, E\rangle$	$\rightarrow \langle R, E \cup \{\Gamma = A'_1.\cdots.A'_{n-1}.A.\Gamma'\}\rangle$,where Γ' and $A'_1, \ldots, A'_{n-1}$ are fresh context and type variables;
(Lambda)	$\langle R \cup \{(\lambda.a^{\Gamma_1}_{A_1})^{\Gamma_2}_{A_2}\}, E\rangle$	$\rightarrow \langle R, E \cup \{A_2 = A^* \rightarrow A_1, \Gamma_1 = A^*.\Gamma_2\}\rangle$, where A^* is a fresh type variable;
(App)	$\langle R \cup \{(a^{\Gamma_1}_{A_1}\ b^{\Gamma_2}_{A_2})^{\Gamma_3}_{A_3}\}, E\rangle$	$\rightarrow \langle R, E \cup \{\Gamma_1 = \Gamma_2, \Gamma_2 = \Gamma_3, A_1 = A_2 \rightarrow A_3\}\rangle$

Table 1: Rules for Type Inference in System $TA_{\lambda dB}$

Type inference for a starts with $\langle R_0, \varnothing\rangle$, where R_0 is the set of all a' subterms. The rules from Table 1 are applied until reaches $\langle \varnothing, E_f\rangle$, where E_f is a set of first-order equations over context and type variables.

Example 1. *Let $a = \lambda.(\underline{2}\ \ \underline{1})$. Then $a' = (\lambda.(\underline{2}\ {}^{\Gamma_1}_{A_1}\ \ \underline{1}\ {}^{\Gamma_2}_{A_2})^{\Gamma_3}_{A_3})^{\Gamma_4}_{A_4}$ and $R_0 = \{\underline{2}\ {}^{\Gamma_1}_{A_1}, \underline{1}^{\Gamma_2}_{A_2}, (\underline{2}\ {}^{\Gamma_1}_{A_1}\ \ \underline{1}\ {}^{\Gamma_2}_{A_2})^{\Gamma_3}_{A_3}, (\lambda.(\underline{2}\ {}^{\Gamma_1}_{A_1}\ \ \underline{1}\ {}^{\Gamma_2}_{A_2})^{\Gamma_3}_{A_3})^{\Gamma_4}_{A_4}\}$. Using the rules in Table 1 we have the following reduction:*

$\langle R_0, \emptyset\rangle \rightarrow_{\text{Varn}}$

$\langle R_1 = R_0 \setminus \{\underline{2}\ {}^{\Gamma_1}_{A_1}\}, E_1 = \{\Gamma_1 = A'_1.A_1.\Gamma'_1\}\rangle \rightarrow_{\text{Var}}$

$\langle R_2 = R_1 \setminus \{\underline{1}^{\Gamma_2}_{A_2}\}, E_2 = E_1 \cup \{\Gamma_2 = A_2.\Gamma'_2\}\rangle \rightarrow_{\text{App}}$

$\langle R_3 = R_2 \setminus \{(\underline{2}\ {}^{\Gamma_1}_{A_1}\ \ \underline{1}\ {}^{\Gamma_2}_{A_2})^{\Gamma_3}_{A_3}\}, E_3 = E_2 \cup \{\Gamma_1 = \Gamma_2, \Gamma_2 = \Gamma_3, A_1 = A_2 \rightarrow A_3\}\rangle \rightarrow_{\text{Lambda}}$

$\langle R_4 = R_3 \setminus \{(\lambda.(\underline{2}\ {}^{\Gamma_1}_{A_1}\ \ \underline{1}\ {}^{\Gamma_2}_{A_2})^{\Gamma_3}_{A_3})^{\Gamma_4}_{A_4}\}, E_4 = E_3 \cup \{A_4 = A^*_1 \rightarrow A_3, \Gamma_3 = A^*_1.\Gamma_4\}\rangle$

Thus, $E_4 = E_f$. Solving the trivial equation over context variables, i.e. $\Gamma_1 = \Gamma_2 = \Gamma_3$, and using variables of smaller subscripts, one gets $\{A_1 = A_2 \rightarrow A_3, A_4 = A^*_1 \rightarrow A_3, \Gamma_1 = A'_1.A_1.\Gamma'_1, \Gamma_1 = A_2.\Gamma'_2, \Gamma_1 = A^*_1.\Gamma_4\}$. Thus, simplifying one gets $\{A_1 = A_2 \rightarrow A_3, A_4 = A^*_1 \rightarrow A_3, A'_1.A_1.\Gamma'_1 = A_2.\Gamma'_2 = A^*_1.\Gamma_4\}$. From these equations one gets the most general unifier (mgu for short) $A_4 = A_2 \rightarrow A_3$ and $\Gamma_4 = (A_2 \rightarrow A_3).\Gamma'_1$, for the variables of interest. Since the context must be the shortest one, $\Gamma'_1 = nil$ and $(A_2 \rightarrow A_3, (A_2 \rightarrow A_3).nil)$ is the principal typing of a.

From Definition 4 and by the uniqueness of the solutions of the type inference algorithm, one deduces that $TA_{\lambda dB}$ satisfies PT. The next theorem says that every typable term has a principal typing.

Theorem 2 (Principal Typings for $TA_{\lambda dB}$)**.** *$TA_{\lambda dB}$ satisfies the property of having principal typings.*

2.2 Principal typings for $TA_{\lambda\sigma}$, the simply typed $\lambda\sigma$

The typed version is presented in Curry style, instead of Church style presented in [DoHaKi2000]. Thus, the syntax of $\lambda\sigma$-terms and the rules are the same as the untyped version.

The typing rules of the $\lambda\sigma$-calculus provide types for objects of sort term as well as for objects of sort substitution. An object of sort substitution, due to its semantics, can be viewed as a list of terms. Consequently, its type is a context. $s \rhd \Gamma$ denotes that the object of sort substitution s has type Γ.

Definition 5 (The System $TA_{\lambda\sigma}$)**.** *$TA_{\lambda\sigma}$ is given by the following typing rules.*

(var)	$A.\Gamma \vdash \underline{1} : A$	*(lambda)*	$\dfrac{A.\Gamma \vdash b : B}{\Gamma \vdash \lambda.b : A \to B}$
(app)	$\dfrac{\Gamma \vdash a : A \to B \quad \Gamma \vdash b : A}{\Gamma \vdash (a\ b) : B}$	*(clos)*	$\dfrac{\Gamma \vdash s \rhd \Gamma' \qquad \Gamma' \vdash a : A}{\Gamma \vdash a[s] : A}$
(id)	$\Gamma \vdash id \rhd \Gamma$	*(shift)*	$A.\Gamma \vdash \uparrow \rhd \Gamma$
(cons)	$\dfrac{\Gamma \vdash a : A \qquad \Gamma \vdash s \rhd \Gamma'}{\Gamma \vdash a.s \rhd A.\Gamma'}$	*(comp)*	$\dfrac{\Gamma \vdash s'' \rhd \Gamma'' \quad \Gamma'' \vdash s' \rhd \Gamma'}{\Gamma \vdash s' \circ s'' \rhd \Gamma'}$

Observe that the name of the typing rules begin with lower-case letters, while the rewriting rules with upper-case letters. We have verified that this version of $\lambda\sigma$ in Curry style has the same properties as the version of $\lambda\sigma$ in Church style given in [DoHaKi2000].

Since subterms of $\lambda\sigma$-terms can be of sort either term or substitution, we will enclose both sorts by the denomination $\lambda\sigma$-expression (sub-expression). For $TA_{\lambda\sigma}$ the notion of typing has to be adapted since the $\lambda\sigma$-expression of sort substitution is decorated with contexts variables as types and as contexts. Thus, one may say that $\tau = (\Gamma, \mathbb{T})$ is a typing of a $\lambda\sigma$-expression in $TA_{\lambda\sigma}$, where $\mathbb{T}$ can be either a type or a context. If the analysed expression belongs to the λ-calculus, the notion of typing corresponds to that of $TA_{\lambda dB}$.

Lemma 2 (Weakening for $\lambda\sigma$)**.** *Let a be a $\lambda\sigma$-term and s a $\lambda\sigma$-substitution. If $\Gamma \vdash a : A$, then $\Gamma.B \vdash a : A$, for any type B. Similarly, if $\Gamma \vdash s \rhd \Gamma'$, then $\Gamma.B \vdash s \rhd \Gamma'.B$. Hence, the rules ($\lambda\sigma$-tweak) and ($\lambda\sigma$-sweak) are admissible in System $TA_{\lambda\sigma}$, where*

$$\frac{\Gamma \vdash a : A}{\Gamma.B \vdash a : A}(\lambda\sigma\text{-}tweak) \qquad \frac{\Gamma \vdash s \rhd \Gamma'}{\Gamma.B \vdash s \rhd \Gamma'.B}(\lambda\sigma\text{-}sweak)$$

The rules given in Lemma 2 and the type substitution allow us present a definition for PT in $TA_{\lambda\sigma}$.

Definition 6 (Principal Typings in $TA_{\lambda\sigma}$)**.** *A **principal typing** of an expression a in $TA_{\lambda\sigma}$ is a typing $\tau = (\Gamma, \mathbb{T})$ such that*

1. *$TA_{\lambda\sigma} \rhd a : \tau$*
2. *If $TA_{\lambda\sigma} \rhd a : \tau'$ for any typing $\tau' = (\Gamma', \mathbb{T}')$, then there exists a substitution s such that $s(\Gamma) = \Gamma'_{\leq|\Gamma|}.nil$ and if $\mathbb{T}$ is a type, $s(\mathbb{T}) = \mathbb{T}'$, otherwise we have that $s(\mathbb{T}) = \mathbb{T}'_{\leq|\mathbb{T}|}.nil$.*

We might verify if this PT definition has a correspondence with Wells' system-independent definition [We2002].

Theorem 3. *A typing τ is principal in $TA_{\lambda\sigma}$ according to Definition 6 iff τ is principal in $TA_{\lambda\sigma}$ according to Definition 2.*

An algorithm for type inference is presented, to verify if $\mathrm{TA}_{\lambda\sigma}$ has PT according to Definition 6. Thus, given an expression a, we will work with the decorated expression a' but the type for substitutions is a context as well. We use the same syntax for decorated expressions as in [Bo95].

The inference rules presented in the Table 2 are given according to the typing rules of the system $TA_{\lambda\sigma}$ presented in the Definition 5. The rules are applied to pairs $\langle R, E\rangle$, starting from the pair $\langle R_0, \emptyset\rangle$, as was done to $TA_{\lambda dB}$.

(Var)	$\langle R \cup \{\underline{1}_A^\Gamma\}, E\rangle$	$\rightarrow \langle R, E \cup \{\Gamma = A.\Gamma'\}\rangle$,where Γ' is a fresh context variable;
(Lambda)	$\langle R \cup \{(\lambda.a_{A_1}^{\Gamma_1})_{A_2}^{\Gamma_2}\}, E\rangle$	$\rightarrow \langle R, E \cup \{A_2 = A^* \rightarrow A_1, \Gamma_1 = A^*.\Gamma_2\}\rangle$, where A^* is a fresh type variable;
(App)	$\langle R \cup \{(a_{A_1}^{\Gamma_1}\ b_{A_2}^{\Gamma_2})_{A_3}^{\Gamma_3}\}, E\rangle$	$\rightarrow \langle R, E \cup \{\Gamma_1 = \Gamma_2, \Gamma_2 = \Gamma_3, A_1 = A_2 \rightarrow A_3\}\rangle$
(Clos)	$\langle R \cup \{(a_{A_1}^{\Gamma_1}[s_{\Gamma_3}^{\Gamma_2}])_{A_2}^{\Gamma_4}\}, E\rangle$	$\rightarrow \langle R, E \cup \{\Gamma_1 = \Gamma_3, \Gamma_2 = \Gamma_4, A_1 = A_2\}\rangle$
(Id)	$\langle R \cup \{id_{\Gamma_2}^{\Gamma_1}\}, E\rangle$	$\rightarrow \langle R, E \cup \{\Gamma_1 = \Gamma_2\}\rangle$
(Shift)	$\langle R \cup \{\uparrow_{\Gamma_2}^{\Gamma_1}\}, E\rangle$	$\rightarrow \langle R, E \cup \{\Gamma_1 = A'.\Gamma_2\}\rangle$,where A' is a fresh type variable;
(Cons)	$\langle R \cup \{(a_{A_1}^{\Gamma_1}.s_{\Gamma_3}^{\Gamma_2})_{\Gamma_5}^{\Gamma_4}\}, E\rangle$	$\rightarrow \langle R, E \cup \{\Gamma_1 = \Gamma_2, \Gamma_2 = \Gamma_4, \Gamma_5 = A_1.\Gamma_3\}\rangle$
(Comp)	$\langle R \cup \{(s_{\Gamma_2}^{\Gamma_1} \circ t_{\Gamma_4}^{\Gamma_3})_{\Gamma_6}^{\Gamma_5}\}, E\rangle$	$\rightarrow \langle R, E \cup \{\Gamma_1 = \Gamma_4, \Gamma_2 = \Gamma_6, \Gamma_3 = \Gamma_5\}\rangle$

Table 2: Type inference rules for the $\lambda\sigma$-calculus

Example 2. *For* $a = (\underline{2}.id) \circ \uparrow$ *one has* $a' = (((\underline{1}_{A_1}^{\Gamma_1}[\uparrow_{\Gamma_3}^{\Gamma_2}])_{A_2}^{\Gamma_4}.id_{\Gamma_6}^{\Gamma_5})_{\Gamma_8}^{\Gamma_7} \circ \uparrow_{\Gamma_{10}}^{\Gamma_9})_{\Gamma_{12}}^{\Gamma_{11}}$. *Then* $R_0 = \{(\underline{1}_{A_1}^{\Gamma_1}[\uparrow_{\Gamma_3}^{\Gamma_2}])_{A_2}^{\Gamma_4}, ((\underline{1}_{A_1}^{\Gamma_1}[\uparrow_{\Gamma_3}^{\Gamma_2}])_{A_2}^{\Gamma_4}.id_{\Gamma_6}^{\Gamma_5})_{\Gamma_8}^{\Gamma_7}, (((\underline{1}_{A_1}^{\Gamma_1}[\uparrow_{\Gamma_3}^{\Gamma_2}])_{A_2}^{\Gamma_4}.id_{\Gamma_6}^{\Gamma_5})_{\Gamma_8}^{\Gamma_7} \circ \uparrow_{\Gamma_{10}}^{\Gamma_9})_{\Gamma_{12}}^{\Gamma_{11}}$, $\underline{1}_{A_1}^{\Gamma_1}, \uparrow_{\Gamma_3}^{\Gamma_2}, id_{\Gamma_6}^{\Gamma_5}, \uparrow_{\Gamma_{10}}^{\Gamma_9}\}$. *Applying the rules from Table 2 to the pair* $\langle R_0, \emptyset\rangle$ *until obtain the pair* $\langle \emptyset, E_f\rangle$ *and simplifying* E_f*, as in example 1, one obtains the set of equations* $\{A_1 = A_2, \Gamma_{11} = \Gamma_{12} = A_2.\Gamma_2, \Gamma_2 = A_1'.\Gamma_1, \Gamma_1 = A_1.\Gamma_1'\}$. *From this equational system one obtains the mgu* $\Gamma_{11}{=}\Gamma_{12}{=}A_1.A_1'.A_1.\Gamma_1'$*, for the variables of interest. Thus,* $(A_1.A_1'.A_1.nil, A_1.A_1'.A_1.nil)$ *is the principal typing of* a.

Theorem 4 (Principal Typings for $TA_{\lambda\sigma}$)**.** *$TA_{\lambda\sigma}$ satisfies the property of having principal typings.*

2.3 Principal typings for $TA_{\lambda s_e}$, the simply typed λs_e

Definition 7 (The System $TA_{\lambda s_e}$)**.** *$TA_{\lambda s_e}$ is given by the following typing rules.*

$$\textit{(Var)}\quad A.\Gamma \vdash \underline{1} : A \qquad \textit{(Varn)}\quad \frac{\Gamma \vdash \underline{n} : B}{A.\Gamma \vdash \underline{n+1} : B}$$

$$\textit{(Lambda)}\quad \frac{A.\Gamma \vdash b : B}{\Gamma \vdash \lambda.b : A \rightarrow B} \qquad \textit{(App)}\quad \frac{\Gamma \vdash a : A \rightarrow B \quad \Gamma \vdash b : A}{\Gamma \vdash (a\ b) : B}$$

$$\textit{(Sigma)}\quad \frac{\Gamma_{\geq i} \vdash b : B \quad \Gamma_{<i}.B.\Gamma_{\geq i} \vdash a : A}{\Gamma \vdash a\,\sigma^i b : A} \qquad \textit{(Phi)}\quad \frac{\Gamma_{\leq k}.\Gamma_{\geq k+i} \vdash a : A}{\Gamma \vdash \varphi_k^i\, a : A}$$

As for $\lambda\sigma$, the typed version of λs_e-calculus presented is in Curry style, which we have verified that has the same properties of the version in Church style presented in [ARKa2001a].

Lemma 3 (Weakening for λs_e)**.** *Let a be a λs_e-term. If $\Gamma \vdash a : A$, then $\Gamma.B \vdash a : A$, for any type B. Hence, the rule (λs_e-weak) is admissible in System $TA_{\lambda s_e}$, where* $\dfrac{\Gamma \vdash a : A}{\Gamma.B \vdash a : A}(\lambda s_e\text{-}weak)$.

Since λs_e remains close to the λ-calculus in de Bruijn notation, the definition of principal typings in λs_e is the same as that for $TA_{\lambda dB}$. For the sake of completeness we repeat it here.

Definition 8 (Principal Typings in $TA_{\lambda s_e}$)**.** *A **principal typing** of a term a in $TA_{\lambda s_e}$ is a typing $\tau = (\Gamma, B)$ such that*

1. *$TA_{\lambda s_e} \rhd a : \tau$*
2. *If $TA_{\lambda s_e} \rhd a : \tau'$ for any typing $\tau' = (\Gamma', B')$, then there exists a substitution s such that $s(\Gamma) = \Gamma'_{\leq|\Gamma|}.nil$ and $s(B) = B'$.*

Theorem 5. *A typing τ is principal in $TA_{\lambda s_e}$ according to Definition 8 iff τ is principal in $TA_{\lambda s_e}$ according to Definition 2.*

A type inference algorithm for the λs_e-calculus is presented, similarly to that of [AyMu2000]. The decorated term associated with a, denoted as a', has syntax closer to the one of decorated λ-terms: any subterm is decorated with its type and its context variables.

(Var)	$\langle R \cup \{\underline{1}_A^\Gamma\}, E\rangle \rightarrow$ $\langle R, E \cup \{\Gamma = A.\Gamma'\}\rangle$, where Γ' is a fresh context variable;
(Varn)	$\langle R \cup \{\underline{n}_A^\Gamma\}, E\rangle \rightarrow$ $\langle R, E \cup \{\Gamma = A'_1.\cdots.A'_{n-1}.A.\Gamma'\}\rangle$,where Γ' and $A'_1, \ldots, A'_{n-1}$ are fresh context and type variables;
(Lambda)	$\langle R \cup \{(\lambda.a_{A_1}^{\Gamma_1})_{A_2}^{\Gamma_2}\}, E\rangle \rightarrow$ $\langle R, E \cup \{A_2 = A^* \rightarrow A_1, \Gamma_1 = A^*.\Gamma_2\}\rangle$, where A^* is a fresh type variable;
(App)	$\langle R \cup \{(a_{A_1}^{\Gamma_1}\ b_{A_2}^{\Gamma_2})_{A_3}^{\Gamma_3}\}, E\rangle \rightarrow$ $\langle R, E \cup \{\Gamma_1 = \Gamma_2, \Gamma_2 = \Gamma_3, A_1 = A_2 \rightarrow A_3\}\rangle$
(Sigma)	$\langle R \cup \{(a_{A_1}^{\Gamma_1}\ \sigma^i b_{A_2}^{\Gamma_2})_{A_3}^{\Gamma_3}\}, E\rangle \rightarrow$ $\langle R, E \cup \{A_1{=}A_3, \Gamma_1{=}A'_1.\cdots.A'_{i-1}.A_2.\Gamma_2, \Gamma_3{=}A'_1.\cdots.A'_{i-1}.\Gamma_2\}\rangle$, where $A'_1, \ldots, A'_{i-1}$ are new type variables and the sequence is empty if $i = 1$;
(Phi)	$\langle R \cup \{(\varphi_k^i\ a_{A_1}^{\Gamma_1})_{A_2}^{\Gamma_2}\}, E\rangle \rightarrow$ $\langle R, E \cup \{A_1 = A_2, \Gamma_2 = A'_1.\cdots.A'_{k+i-1}.\Gamma', \Gamma_1 = A'_1.\cdots.A'_k.\Gamma'\}\rangle$, where Γ' and $A'_1, \ldots, A'_{k+i-1}$ are new context and type variables and if $k + i - 1, k = 0$ then the sequences $A'_1, \ldots, A'_{k+i-1}$ and $A'_1, \ldots, A'_k$, respectively, are empty.

Table 3: Type inference rules for the λs_e-Calculus

Similarly to the previous algorithm, the rules of the Table 3, developed according to the rules of Definition 7, are applied to pairs $\langle R, E\rangle$, where R is a set of decorated subterms of a' and E a set of equations over type and context variables.

Example 3. *For the λs_e-term $a = \lambda.((\underline{1}\,\sigma^2\underline{2})\ (\varphi_0^2\,\underline{2}))$, one obtains the corresponding R_0 from $a' = (\lambda.((\underline{1}_{A_1}^{\Gamma_1}\,\sigma^2\underline{2}_{A_2}^{\Gamma_2})_{A_3}^{\Gamma_3}\ (\varphi_0^2\,\underline{2}_{A_4}^{\Gamma_4})_{A_5}^{\Gamma_5})_{A_6}^{\Gamma_6})_{A_7}^{\Gamma_7}$. Then, applying the rules in Table 3 to the pair $\langle R_0, \emptyset\rangle$, obtaining the pair $\langle \emptyset, E_f\rangle$, and simplifying E_f, similarly to the example 1, one obtains the system of equations $\{\, A_1 = A_4 \rightarrow A_6\ ,\ A_7 = A_1^* \rightarrow A_6\ ,\ A_1.\Gamma'_1 = A'_2.A_2.\Gamma_2\ ,\ A'_2.\Gamma_2 = A'_4.A'_3.A_4.\Gamma'_3 = A_1^*.\Gamma_7\ ,\ \Gamma_2 = A'_1.A_2.\Gamma'_2\,\}$ from which one has the mgu $A_7 = (A_2 \rightarrow A_6) \rightarrow A_6$ and $\Gamma_7 = A'_1.A_2.\Gamma'_2$ for variables of interest.*

Theorem 6 (Principal Typings for $TA_{\lambda s_e}$)**.** *$TA_{\lambda s_e}$ satisfies the property of having principal typings.*

3 Conclusions and Future Work

We consider for $\lambda\sigma$ and λs_e particular notions of principal typings and presented respective definitions which are proved to agree with the system-independent notion introduced by Wells in [We2002]. The adaptation of this general notion of principal typings for the $\lambda\sigma$ requires special attention, since this calculus enlarges the language of the λ-calculus by introducing a new sort of *substitution* objects, whose types are contexts. then the provided PT notion has to deal with the principality of substitution objects as well. Then, the property of having principal typings is straightforwardly proved by revisiting type inference algorithms for the $\lambda\sigma$ and the λs_e, previously presented in [Bo95] and [AyMu2000],

respectively. The result is based on the correctness, completeness and uniqueness of solutions given by adequate first-order unification algorithms(e.g. see the unification algorithm given in [Hi97]).

Investigation of this property for more elaborated typing systems of explicit substitutions is an interesting work to be done.

References

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *J. of Functional Programming*, 1(4):375–416, 1991.

[ARMoKa2005] M. Ayala-Rincón, F. de Moura, and F. Kamareddine. Comparing and Implementing Calculi of Explicit Substitutions with Eta-Reduction. *Annals of Pure and Applied Logic*, 134:5–41, 2005.

[ARKa2001a] M. Ayala-Rincón and F. Kamareddine. Unification via the λs_e-Style of Explicit Substitution. *The Logical Journal of the Interest Group in Pure and Applied Logics*, 9(4):489–523, 2001.

[AyMu2000] M. Ayala-Rincón and C. Muñoz. Explicit Substitutions and All That. *Revista Colombiana de Computación*, 1(1):47–71, 2000.

[Bo95] P. Borovanský. Implementation of Higher-Order Unification Based on Calculus of Explicit Substitutions. In M. Bartošek, J. Staudek, and J. Wiedermann, editors, *Proceedings of the SOFSEM'95: Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 363–368. Springer Verlag, 1995.

[deBru72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[DoHaKi2000] G. Dowek, T. Hardin, and C. Kirchner. Higher-order Unification via Explicit Substitutions. *Information and Computation*, 157(1/2):183–235, 2000.

[Hi97] J. R. Hindley. *Basic Simple Type Theory.* Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1997.

[KR97] F. Kamareddine and A. Ríos. Extending a λ-calculus with Explicit Substitution which Preserves Strong Normalisation into a Confluent Calculus on Open Terms. *J. of Func. Programming*, 7:395–420, 1997.

[Mel95] P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. In Proc. of TLCA'95, volume 902 of *LNCS*, pages 328–334. Springer Verlag, 1995.

[NaWi98] G. Nadathur and D. S. Wilson. A Notation for Lambda Terms A Generalization of Environments. *Theoretical Computer Science*, 198:49–98, 1998.

[We2002] J. Wells. The essence of principal typings. In *Proc. 29th International Colloquium on Automata, Languages and Programming, ICALP 2002*, volume 2380 of *LNCS*, pages 913–925. Springer Verlag, 2002.

A The type free calculi

A.1 λ-calculus in de Bruijn notation

Definition 9. *The set Λ_{dB} of* ***λ-terms in de Bruijn notation*** *is defined inductively as*

Terms $a ::= \underline{n} \,|\, (a\ a) \,|\, \lambda.a$ *where* $n \in \mathbb{N}^* = \mathbb{N} \setminus \{0\}$

Definition 10. *Let a be a λ-term. A subterm a_1 of a is* ***n-deep in*** *a, if the least possible index of any free variable in a_1 is greater than n. In other words, a_1 is in between n abstractors.*

We say that $\underline{i}$ **occurs as free index** in a term a if any occurrence of $\underline{i+n}$ is n-deep in a. Terms like $((((a_1\ a_2)\ a_3)\dots)\ a_n)$ are written as usual $(a_1\, a_2\, \dots\, a_n)$. The β-contraction definition in this notation needs a mechanism which detects and update free indices of terms. It follows an operator similar to the one presented in [ARKa2001a].

Definition 11. *Let $a \in \Lambda_{dB}$ and $i \in \mathbb{N}$. The* ***i-lift*** *of a, denoted as a^{+i}, is defined inductively as*

1. $(a_1\, a_2)^{+i} = (a_1^{+i}\, a_2^{+i})$
2. $(\lambda.a_1)^{+i} = \lambda.a_1^{+(i+1)}$
3. $\underline{n}^{+i} = \begin{cases} \underline{n+1}, & \text{if } n > i \\ \underline{n}, & \text{if } n \leq i. \end{cases}$

The **lift** of a term a is its 0-lift, denoted as a^+. Intuitively, the lift of a corresponds to increment by 1 all free indices occurring in a. Using the i-lift, we are able to present the definition of the substitution used by β-contractions, similar to the one presented in [ARKa2001a].

Definition 12. *Let $m, n \in \mathbb{N}^*$. The* ***β-substitution*** *for free ocurrences of $\underline{n}$ in $a \in \Lambda_{dB}$ by term b, denoted as $\{\underline{n}/b\}a$, is defined inductively as*

1. $\{\underline{n}/b\}(a_1\ a_2) = (\{\underline{n}/b\}a_1\ \{\underline{n}/b\}a_2)$
2. $\{\underline{n}/b\}\lambda.a_1 = \lambda.\{\underline{n+1}/b^+\}a_1$
3. $\{\underline{n}/b\}\underline{m} = \begin{cases} \underline{m-1}, & \text{if } m > n \\ b, & \text{if } m = n \\ \underline{m}, & \text{if } m < n \end{cases}$

Observe that in item 2 of Def. 12, the lift operator is used to avoid captures of free indices in b. We present the β-contraction as defined in [ARKa2001a].

Definition 13. ***β-contraction*** *of λ-terms in de Bruijn notation is defined as $(\lambda.a\ b) \to_\beta \{\underline{1}/b\}a$.*

Notice that item 3 in Definition 12, for $n = 1$, is the mechanism which does the substitution and updates the free indices in a as consequence of the lead abstractor elimination.

A.2 The $\lambda\sigma$-Calculus

The $\lambda\sigma$-calculus is given by a first-order rewriting system, which makes substitutions explicit by extending the language with two sorts of objects: **terms** and **substitutions**.

Definition 14. *The syntax of the type free $\lambda\sigma$-calculus is given by*

Terms $a ::= \underline{1} \,|\, (a\ a) \,|\, \lambda.a \,|\, a[s]$ ***Substitutions*** $s ::= id \,|\, \uparrow \,|\, a.s \,|\, s \circ s$

Substitutions are lists of the form $b/\underline{i}$ indicating that the index $\underline{i}$ should be changed to the term b. id represents a substitution of the form $\{\underline{1}/\underline{1}, \underline{2}/\underline{2}, \dots\}$ and $\uparrow$ is the substitution $\{\underline{i+1}/\underline{i} \,|\, i \in \mathbb{N}^*\}$. $s \circ s$ represents the composition of

substitutions. $\underline{1}\,[\uparrow^n]$, where $n \in \mathbb{N}^*$, codifies the de Bruijn index $\underline{n+1}$. $\underline{i}\,[s]$ represents the value of $\underline{i}$ through the substitution s, which can be seen as a function $s(i)$. The substitution $a.s$ has the form $\{a/\underline{1}\,, s(i)/\underline{i+1}\,\}$, called the **cons of** a **in** s. $a[b.id]$ starts the simulation of the β-reduction of $(\lambda.a\ b)$ in $\lambda\sigma$. Thus, in addition to the substitution of the free occurrences of the index $\underline{1}$ by the corresponding term, free occurrences of indices should be decremented because of the elimination of the abstractor. The Table 4 includes the rewriting system of the $\lambda\sigma$-calculus, as presented in [DoHaKi2000].

$(\lambda.a\ b)$	$\longrightarrow$	$a[b.id]$	$(Beta)$
$(a\ b)[s]$	$\longrightarrow$	$(a[s]\ b[s])$	(App)
$1[a.s]$	$\longrightarrow$	a	$(VarCons)$
$a[id]$	$\longrightarrow$	a	(Id)
$(\lambda.a)[s]$	$\longrightarrow$	$\lambda.(a[1.(s\circ\uparrow)])$	(Abs)
$(a[s])[t]$	$\longrightarrow$	$a[s\circ t]$	$(Clos)$
$id\circ s$	$\longrightarrow$	s	(IdL)
$\uparrow\circ(a.s)$	$\longrightarrow$	s	$(ShiftCons)$
$(s_1\circ s_2)\circ s_3$	$\longrightarrow$	$s_1\circ(s_2\circ s_3)$	$(AssEnv)$
$(a.s)\circ t$	$\longrightarrow$	$a[t].(s\circ t)$	$(MapEnv)$
$s\circ id$	$\longrightarrow$	s	(IdR)
$1.\uparrow$	$\longrightarrow$	id	$(VarShift)$
$1[s].(\uparrow\circ s)$	$\longrightarrow$	s	$(Scons)$
$\lambda.(a\ \underline{1}\,)$	$\longrightarrow$	b if $a{=_\sigma}b[\uparrow]$	(Eta)

Table 4: The rewriting system for the $\lambda\sigma$-calculus with Eta rule

This system without (Eta) is equivalent to that of [ACCL91]. The associated substitution calculus, denoted as σ, is the one induced by all the rules except (Beta) and (Eta), and its equality is denoted as $=_\sigma$.

A.3 The λs_e-Calculus

In contrast with $\lambda\sigma$, The λs_e-calculus has a sole sort of objects maintaining its syntax closer to the λ-calculus. The λs_e-calculus controls the atomization of the substitution operation by introducing the use of arithmetic constraints through two operators σ and φ, for substitution and updating, respectively.

Definition 15. *The syntax of the untyped λs_e-calculus, where $n, i, j \in \mathbb{N}^*$ and $k \in \mathbb{N}$ is given as*

Terms $\quad a ::= \underline{n} \mid (a\ a) \mid \lambda.a \mid a\,\sigma^i a \mid \varphi_k^j\, a$

The term $a\,\sigma^i b$ represents the term $\{\underline{i}/b\}a$; i.e., substitution of free occurrences of $\underline{i}$ in a by b, updating free variables in a (and in b). The term $\varphi_k^j\, a$ represents $j-1$ applications of the k-lift to the term a; i.e., $a^{+k^{(j-1)}}$. Table 5 contains the rewriting rules of the λs_e-calculus and the rule (Eta), as given in [ARKa2001a]. $=_{s_e}$ denotes the equality for the associated substitution calculus, denoted as s_e, induced by all the rules except (σ-generation) and (Eta).

B Proofs

The proofs are divided in three parts: B.1, where the proof of weakening for each type system are placed; B.2, where the three proofs of the correspondence

$$
\begin{array}{lcll}
(\lambda.a\ b) & \longrightarrow & a\,\sigma^1 b & (\sigma\text{-generation}) \\
(\lambda.a)\,\sigma^i b & \longrightarrow & \lambda.(a\,\sigma^{i+1} b) & (\sigma\text{-}\lambda\text{-transition}) \\
(a_1\ a_2)\,\sigma^i b & \longrightarrow & ((a_1\,\sigma^i b)\ (a_2\,\sigma^i b)) & (\sigma\text{-app-trans.}) \\
\underline{n}\,\sigma^i b & \longrightarrow & \begin{cases} \underline{n-1} & \text{if } n > i \\ \varphi_0^i\, b & \text{if } n = i \\ \underline{n} & \text{if } n < i \end{cases} & (\sigma\text{-destruction}) \\
\varphi_k^i\,(\lambda.a) & \longrightarrow & \lambda.(\varphi_{k+1}^i\, a) & (\varphi\text{-}\lambda\text{-trans.}) \\
\varphi_k^i\,(a_1\ a_2) & \longrightarrow & ((\varphi_k^i\, a_1)\ (\varphi_k^i\, a_2)) & (\varphi\text{-app-trans.}) \\
\varphi_k^i\,\underline{n} & \longrightarrow & \begin{cases} \underline{n+i-1} & \text{if } n > k \\ \underline{n} & \text{if } n \leq k \end{cases} & (\varphi\text{-destruction}) \\
(a_1\,\sigma^i a_2)\,\sigma^j b & \longrightarrow & (a_1\,\sigma^{j+1} b)\,\sigma^i (a_2\,\sigma^{j-i+1} b) \quad \text{if } i \leq j & (\sigma\text{-}\sigma\text{-trans.}) \\
(\varphi_k^i\, a)\,\sigma^j b & \longrightarrow & \varphi_k^{i-1}\, a \quad \text{if } k < j < k+i & (\sigma\text{-}\varphi\text{-trans. 1}) \\
(\varphi_k^i\, a)\,\sigma^j b & \longrightarrow & \varphi_k^i\,(a\,\sigma^{j-i+1} b) \quad \text{if } k+i \leq j & (\sigma\text{-}\varphi\text{-trans. 2}) \\
\varphi_k^i\,(a\,\sigma^j b) & \longrightarrow & (\varphi_{k+1}^i\, a)\,\sigma^j (\varphi_{k+1-j}^i\, b) \quad \text{if } j \leq k+1 & (\varphi\text{-}\sigma\text{-trans.}) \\
\varphi_k^i\,(\varphi_l^j\, a) & \longrightarrow & \varphi_l^j\,(\varphi_{k+1-j}^i\, a) \quad \text{if } l+j \leq k & (\varphi\text{-}\varphi\text{-trans. 1}) \\
\varphi_k^i\,(\varphi_l^j\, a) & \longrightarrow & \varphi_l^{j+i-1}\, a \quad \text{if } l \leq k < l+j & (\varphi\text{-}\varphi\text{-trans. 2}) \\
\lambda.(a\ \underline{1}\,) & \longrightarrow & b \quad \text{if } a =_{s_e} \varphi_0^2\, b & (Eta)
\end{array}
$$

Table 5: The rewriting system of the λs_e-calculus with Eta rule

between system-independent and system-specific definition of PT are merged in one; and B.3, where the three proofs of PT are also merged.

B.1 Proofs of weakening

Proof of Lemma 1(Weakening for $TA_{\lambda dB}$). Let $\Gamma \vdash a : A$. We will prove a more general result stating, for $i \in \mathbb{N}$, that $\Gamma_{\leq i}.B.\Gamma_{>i} \vdash a^{+i} : A$. The proof is done by induction on a structure. Note that if $i \geq m$, where $m = |\Gamma|$, then B is added at the end of Γ.

1) $a = \underline{n}$: Suppose $\Gamma \vdash \underline{n} : A$. If $n \leq i$, then $\underline{n}^{+i} = \underline{n}$. The B addition at i+1-th position changes only types of indices greater or equal to $\underline{i+1}$, thus one has trivially that $\Gamma_{\leq i}.B.\Gamma_{>i} \vdash \underline{n} : A$. If $n > i$, then $\underline{n}^{+i} = \underline{n+1}$. By ($TA_{dB}$-varn) i times one has $\Gamma_{>i} \vdash \underline{n-i} : A$. Thus, by ($TA_{dB}$-varn) applied $i+1$ times, one has that $\Gamma_{\leq i}.B.\Gamma_{>i} \vdash \underline{n+1} : A$.
2) $a = (b\,c)$: Suppose $\Gamma \vdash (b\,c) : A$. By (TA_{dB}-app) one has that $\Gamma \vdash b : C \to A$ and $\Gamma \vdash c : C$. By IH one has $\Gamma_{\leq i}.B.\Gamma_{>i} \vdash b^{+i} : C \to A$ and $\Gamma_{\leq i}.B.\Gamma_{>i} \vdash c^{+i} : C$. Thus, by ($TA_{dB}$-app), $\Gamma_{\leq i}.B.\Gamma_{>i} \vdash (b^{+i}\, c^{+i}) : A$.
3) $a = \lambda.b$: Suppose $\Gamma \vdash \lambda.b : A$. By ($TA_{dB}$-lambda) one has that $C.\Gamma \vdash b : D$, where $A = C \to D$. By IH one has $C.\Gamma_{\leq i}.B.\Gamma_{>i} \vdash b^{+(i+1)} : A$. Thus, by ($TA_{dB}$-lambda), $\Gamma_{\leq i}.B.\Gamma_{>i} \vdash \lambda.b^{+(i+1)} : C \to D = A$.

Since all information about a free varibles is in context Γ, one has that a maximum value for a free index occurrence, at 0-deep in a, is $m = |\Gamma|$. Consequently, $a^{+j} = a$ for any $j \geq m$. Thus, for $i = m$, we have that $\Gamma.B \vdash a : A$, for any type B. Then a weak rule for $TA_{\lambda dB}$ is admissible, adding types at the end of the context. A type addition in any other position of context Γ would require updating some free indices, then a^{+i} would correspond to a different function from the one to which term a corresponds. □

The proof of Lemma 2 needs some auxiliar definitions and lemmas.

Definition 16. *Let a be a $\lambda\sigma$-object. Define $\|\cdot\| : \Lambda\sigma \to \mathbb{N}$ as*

$$\begin{array}{rclcrcl} \|(a\ b)\| & = & \|a\| + \|b\| & & \|\underline{1}\| & = & 0 \\ \|\lambda.a\| & = & \|a\| & & \|id\| & = & 0 \\ \|a[s]\| & = & \|a\| + \|s\| & & \|\uparrow\| & = & 0 \\ \|s \circ t\| & = & \|s\| + \|t\| & & \|a.t\| & = & 1 + \|a\| + \|t\| \end{array}$$

Lemma 4. *Let s be a $\lambda\sigma$-substitution such that $\|s\| = 0$. If $\Gamma \vdash s \rhd \Gamma'$, then $\Gamma.B \vdash s \rhd \Gamma'.B$*

Proof. Induction on s structure.

1) $s = id$: By (id) it has $\Gamma.B \vdash id \rhd \Gamma'.B$, trivially.
2) $s = \uparrow$: Let $\Gamma \vdash\uparrow \rhd\Gamma'$ where, by (shift), $\Gamma = A.\Gamma'$. Thus $\Gamma.B \vdash\uparrow \rhd\Gamma'.B$.
3) $s = u\circ t$: Let $\Gamma \vdash u\circ t\rhd\Gamma'$. By (comp), it has that $\Gamma \vdash t\rhd\Gamma''$ and $\Gamma'' \vdash u\rhd\Gamma'$, for some Γ''. By induction hypothesis(IH) it has $\Gamma.B \vdash t \rhd \Gamma''.B$ and $\Gamma''.B \vdash u \rhd \Gamma'.B$. Thus, by (comp), $\Gamma.B \vdash u \circ t \rhd \Gamma'.B$.

□

Lemma 5. *Let a be a $\lambda\sigma$-term such that $\|a\| = 0$. If $\Gamma \vdash a : A$, then $\Gamma.B \vdash a : A$.*

Proof. Induction on a structure.

1) $a = \underline{1}$: Let $\Gamma \vdash \underline{1} : A$. By (var) it has that $\Gamma = A.\Gamma'$, for some Γ'. Thus it has $\Gamma.B \vdash \underline{1} : A$, trivially.
2) $a = (b\ c)$: Let $\Gamma \vdash (b\ c) : A$. By (app) it has that $\Gamma \vdash b : C \to A$ and $\Gamma \vdash c : C$, for some C. By IH it has $\Gamma.B \vdash b : C \to A$ and $\Gamma.B \vdash c : C$. Thus, by (app), $\Gamma.B \vdash (b\ c) : A$.
3) $a = \lambda.b$: Let $\Gamma \vdash \lambda.b : A$. By (lambda) it has that $C.\Gamma \vdash b : D$, where $A = C \to D$. By IH it has $C.\Gamma.B \vdash b : D$. Thus, by (lambda), $\Gamma.B \vdash \lambda.b : A$.
4) $a = b[s]$: Let $\Gamma \vdash b[s] : A$. By (clos) is has that $\Gamma \vdash s\rhd\Gamma'$ and $\Gamma' \vdash b : A$, for some Γ'. It has $\|b[s]\| = \|b\|+\|s\| = 0$. Then, by Lemma 4, $\Gamma.B \vdash s\rhd\Gamma'.B$. By IH it has that $\Gamma'.B \vdash b : A$. Thus, by (clos), $\Gamma.B \vdash b[s] : A$.

□

Proof of Lemma 2(Weakening for $TA_{\lambda\sigma}$). Induction on a structure with subinduction on $\|\cdot\|$, having Lemmas 4 and 5 as induction base(IB).

1) $a = \underline{1}$: Let $\Gamma \vdash \underline{1} : A$. By (var) it has that $\Gamma = A.\Gamma'$, for some Γ'. Thus it has $\Gamma.B \vdash \underline{1} : A$, trivially.
2) $a = (b\ c)$: Let $\Gamma \vdash (b\ c) : A$. By (app) it has that $\Gamma \vdash b : C \to A$ and $\Gamma \vdash c : C$, for some C. By IH on structure it has $\Gamma.B \vdash b : C \to A$ and $\Gamma.B \vdash c : C$. Thus, by (app), $\Gamma.B \vdash (b\ c) : A$.
3) $a = \lambda.b$: Let $\Gamma \vdash \lambda.b : A$. By (lambda) it has that $C.\Gamma \vdash b : D$, where $A = C \to D$. By IH on structure it has $C.\Gamma.B \vdash b : D$. Thus, by (lambda), $\Gamma.B \vdash \lambda.b : A$.

4) $a = b[s]$: Let $\Gamma \vdash b[s] : A$. By (clos) is has that $\Gamma \vdash s \rhd \Gamma'$ and $\Gamma' \vdash b : A$, for some Γ'. By IH on structure it has $\Gamma'.B \vdash b : A$. Substitution s has to be examined. If $\|b\| > 0$, then by IH on $\|\cdot\|$, as $\|b[s]\| > \|s\|$, it has that $\Gamma.B \vdash s \rhd \Gamma'.B$.

   Else, if $\|b\| = 0$:

   - If $\|s\| = 0$, then Lemma 4 can be applied.
   - Otherwise, $s = c.t$ or $s = u \circ t$. If $s = c.t$, then by (cons) it has that $\Gamma \vdash c : C$ and $\Gamma \vdash t \rhd \Gamma''$, where $\Gamma' = C.\Gamma''$. As $\|c\|, \|t\| < \|s\| = \|b[s]\|$, by IH on $\|\cdot\|$ it has $\Gamma.B \vdash c : C$ and $\Gamma.B \vdash t \rhd \Gamma''.B$. Thus, by (cons), $\Gamma.B \vdash c.t \rhd \Gamma'.B$. If $s = u \circ t$, then by (comp) it has that $\Gamma \vdash t \rhd \Gamma''$ and $\Gamma'' \vdash u \rhd \Gamma'$, for some Γ''. If $\|u\|, \|t\| > 0$, the result holds by IH on $\|\cdot\|$. Otherwise, at least one of the substitutions has $\|\cdot\|$ greater than 0. Using induction on substitution s structure, where $\|s\| > 0$, the result holds. Then, it has that $\Gamma.B \vdash t \rhd \Gamma''.B$ and $\Gamma''.B \vdash u \rhd \Gamma'.B$. Thus, by (comp), $\Gamma.B \vdash u \circ t \rhd \Gamma'.B$.

   Finally, by (clos), it has that $\Gamma.B \vdash b[s] : A$.

□

Proof of Lemma 3(Weakening for $TA_{\lambda s_e}$). Induction on a structure.

1) $a = \underline{n}$: Let $\Gamma \vdash \underline{n} : A$. Since the type addition at the end of Γ do not change any free index type, one has trivially that $\Gamma.B \vdash \underline{n} : A$.
2) $a = (b\ c)$: Let $\Gamma \vdash (b\ c) : A$. By (App) one has that $\Gamma \vdash b : C \to A$ and $\Gamma \vdash c : C$, for some C. By IH one has $\Gamma.B \vdash b : C \to A$ and $\Gamma.B \vdash c : C$. Thus, by (App), $\Gamma.B \vdash (b\ c) : A$.
3) $a = \lambda.b$: Let $\Gamma \vdash \lambda.b : A$. By $(Lambda)$ one has that $C.\Gamma \vdash b : D$, where $A = C \to D$. By IH one has $C.\Gamma.B \vdash b : D$. Thus, by $(Lambda)$, $\Gamma.B \vdash \lambda.b : A$.
4) $a = b\,\sigma^i c$: Let $\Gamma \vdash b\,\sigma^i c : A$. By $(Sigma)$ one has that $\Gamma_{\geq i} \vdash c : C$ and $\Gamma_{<i}.C.\Gamma_{\geq i} \vdash b : A$. By IH one has $\Gamma_{\geq i}.B \vdash c : C$ and $\Gamma_{<i}.C.\Gamma_{\geq i}.B \vdash b : A$. Thus, by $(Sigma)$, $\Gamma.B \vdash b\,\sigma^i c : A$.
5) $a = \varphi^i_k b$: Let $\Gamma \vdash \varphi^i_k b : A$. By (Phi) one has that $\Gamma_{\leq k}.\Gamma_{\geq k+i} \vdash b : A$. By IH one has $\Gamma_{\leq k}.\Gamma_{\geq k+i}.B \vdash b : A$. Thus, by (Phi), $\Gamma.B \vdash \varphi^i_k b : A$.

□

B.2 Proof of Correspondence

Proof of Theorems 1, 3 and 5. The proofs are an adapted version of that given by Wells in [We2002]. Our adaptation deals with de Bruijn indices rather than variables and the proof for $\lambda\sigma$ has an adaptation to deal with substitutions too. Let $u \in \{\lambda dB, \lambda\sigma, \lambda s_e\}$ and $\mathcal{O}_u$ be the index updating operator of each calculus. In other words, $\mathcal{O}_{\lambda dB}(a) = a^+$, $\mathcal{O}_{\lambda\sigma}(a) = a[\uparrow]$ and $\mathcal{O}_{\lambda s_e}(a) = \varphi^2_0\, a$. Let $\mathcal{O}^1_u = \mathcal{O}_u$ and $\mathcal{O}^{n+1}_u(a) = \mathcal{O}_u(\mathcal{O}^n_u(a))$. For a type A, let $\mathcal{T}(A)$ be the set of type variables ocurring in A. For brevity, $\underline{1}\,[\uparrow^n]$ is denoted as $\underline{n+1}$.
⇒ **proof:** Let $\tau_u = (\Gamma_u, B_u)$ be a PT of some term a_u, according to Definitions 4, 6 and 8, and $\tau'_u = (\Gamma'_u, B'_u)$ be a typing of a_u. By PT definition for each type system, we have that exists a type substitution s such that $s(\Gamma_u) = (\Gamma'_u)_{\leq|\Gamma_u|}.nil$

and $s(B_u) = B'_u$. By the property which says that if $TA_u \rhd a : \tau_u$, then $TA_u \rhd a : s(\tau_u)$, for any type substitution s, we have $\tau_u \leq_{TA_u} s(\tau_u)$. By the weakening admissible rule for each type system ((λdB-weak), ($\lambda\sigma$-tweak) and (λs_e-weak)), we have that $s(\tau_u) \leq_{TA_u} \tau'_u$. Thus, τ_u is PT of a_u, according to Definition 2.

The proof for a $\lambda\sigma$-substitution t with PT $\tau = (\Gamma, \Delta)$ according to Definiton 6 and typing $\tau' = (\Gamma, \Delta)$ is similar to the proof for $\lambda\sigma$-terms, using the proper weakening rule ($\lambda\sigma$-sweak).

$\Leftarrow$ **proof:** Let $\tau_u = (\Gamma_u, B_u)$ be a PT of some term a_u, according to Definitions 4, 6 and 8, and $\tau'_u = (\Gamma'_u, B'_u)$ be a typing of a_u which is not PT according to these definitions. Then, exists a type substitution s such that $s(\Gamma_u) = (\Gamma'_u)_{\leq|\Gamma_u|}.nil$ and $s(B_u) = B'_u$ and does not exist any substitution s' such that $s'(\Gamma'_u) = (\Gamma_u)_{\leq|\Gamma'_u|}.nil$ and $s'(B'_u) = B_u$.

1. Suppose $s(\Gamma_u) \neq \Gamma'_u$. Then, $m_u = |\Gamma_u| < |\Gamma'_u|$. Let $b_u = (\lambda.\mathcal{O}_u(a_u)\ \underline{m_u{+}1}\,)$.
2. Otherwise, $s(\Gamma_u) = \Gamma'_u$. Let K be a type variable. Define the functions ϕ_1^u and ϕ_2^u as:

$$\begin{array}{rcl}
\phi_1^u(K,K) & = & \lambda\lambda.\big(\underline{1}\ (\underline{2}\ \ \underline{4})\,(\underline{2}\ \ \underline{3})\big) \\
\phi_1^u(A\to B,K) & = & \begin{cases} \lambda\lambda.\big(\underline{1}\ (\underline{3}\ \ \underline{2})\,(\mathcal{O}_u^3(\lambda.\phi_1^u(A,K))\ \ \underline{2})\big), & \text{if } K\in\mathcal{T}(A) \\ \lambda.\big(\mathcal{O}_u^2(\lambda.\phi_1^u(B,K))\ (\underline{2}\ \ \underline{1})\big), & \text{otherwise} \end{cases} \\
\phi_2^u(K,K) & = & \lambda\lambda.\big(\underline{1}\ (\underline{2}\ \ \underline{3})\,(\underline{2}\ \ \underline{4})\big) \\
\phi_2^u(A\to B,K) & = & \begin{cases} \lambda\lambda.\big(\underline{1}\ (\underline{4}\ \ \underline{2})\,(\mathcal{O}_u^2(\lambda.\phi_1^u(A,K))\ \ \underline{2})\big), & \text{if } K\in\mathcal{T}(A) \\ \lambda.\big(\mathcal{O}_u(\lambda.\phi_1^u(B,K))\ (\underline{3}\ \ \underline{1})\big), & \text{otherwise} \end{cases}
\end{array}$$

   (a) Suppose $s(K_u)$ is not a type variable for $K_u \in \mathcal{T}(\tau_u)$
      i. Suppose $K_u \in \mathcal{T}(B_u)$.
      Let $b_u = \Big(\lambda.\big(\lambda.\underline{2}\ \ \lambda.(\mathcal{O}_u(\lambda.\phi_2^u(B_u,K_u))\ \lambda.\underline{2})\big)\ a_u\Big)$.
      ii. Suppose $K_u \in \mathcal{T}((\Gamma_u)_{i_u})$.
      Let $b_u = \big(\lambda.\mathcal{O}_u(a)\ \lambda.(\lambda.\lambda.\phi_2^u((\Gamma_u)_{i_u},K_u)\ \underline{i_u{+}1}\ \lambda.\underline{2})\big)$.
   (b) Suppose $s(K_u^1) = s(K_u^2) = L$ for distinct $K_u^1, K_u^2 \in \mathcal{T}(\tau_u)$
      i. Suppose $K_u^j \in \mathcal{T}((\Gamma_u)_{i_{u,j}})$ for $j \in \{1,2\}$.
      Let $p_u^j = \big(\lambda.\phi_1^u((\Gamma_u)_{i_{u,j}},K_u^j)\ \ \underline{i_{u,j}{+}1}\,\big)$
      and $p_u = \lambda\lambda.\big(\underline{1}\ \mathcal{O}_u(p_u^1)\,\mathcal{O}_u(p_u^2)\big)$. Let $b_u = \big(\lambda\lambda.\underline{2}\ a_u\,p_u\big)$.
      ii. Suppose $K_u^1 \in \mathcal{T}((\Gamma_u)_{i_u})$ and $K_u^2 \in \mathcal{T}(B_u)$.
      Let $p_u = \lambda\lambda.\big(\underline{1}\ \big(\mathcal{O}_u(\lambda.\phi_1^u((\Gamma_u)_{i_u},K_u^1))\ \ \underline{i_u{+}3}\,\big)\,\mathcal{O}_u(\phi_2^u(B_u,K_u^2))\big)$
      and $b_u = \big(\lambda.(\lambda.\underline{2}\ \ p_u)\ a_u\big)$.
      iii. Suppose $K_u^i \in \mathcal{T}(B_u)$ for $i \in \{1,2\}$.
      Let $p_u = \lambda\lambda.\big(\underline{1}\ \mathcal{O}_u(\phi_2^u(B_u,K_u^1))\ \mathcal{O}_u(\phi_2^u(B_u,K_u^2))\big)$ and $b_u = \big(\lambda.(\lambda.\underline{2}\ \ p_u)\ a_u\big)$.

   Then, $b_u \in Terms_{TA_u}(\tau'_u) \setminus Terms_{TA_u}(\tau_u)$. Thus, $\tau'_u \not\leq_{TA_u} \tau_u$.

As consequence, if τ'_u is not PT according to Definitions 4, 6 and 8, τ'_u is not PT according to Definition 2.

Let a be a $\lambda\sigma$-substitution t and $\tau = (\Gamma, \Delta)$ be a PT of t, according to Definition 6, and $\tau' = (\Gamma', \Delta')$ be a typing of t which is not PT according to

this definition. Then, exists a type substitution s such that $s(\Gamma) = \Gamma'_{\leq|\Gamma|}.nil$ and $s(\Delta) = \Delta'_{\leq|\Delta|}.nil$ and does not exist any substitution s' such that $s'(\Gamma') = \Gamma_{\leq|\Gamma'|}.nil$ and $s'(\Delta') = \Delta_{\leq|\Delta'|}.nil$.

1. Suppose $s(\Gamma) \neq \Gamma'$. Then, $m = |\Gamma| < |\Gamma'|$. Let $s_i{=}(\underline{1}\,.\,\underline{2}\,.\cdots.\,\underline{m{+}1}\,.\uparrow^{m+1})$ and $r = t \circ s_i$.

2. Otherwise, $s(\Gamma) = \Gamma'$. Define the functions ϕ_1 as $\phi_1^{\lambda\sigma}$ and ϕ_2 as $\phi_2^{\lambda\sigma}$ defined above.

   (a) Suppose $s(K)$ is not a type variable for $K \in \mathcal{T}(\tau)$

      i. Suppose $K{\in}\mathcal{T}(\Delta_i)$. Let $b = \big(\lambda.\big(\lambda.\,\underline{2}\;\lambda.((\lambda.\phi_2(\Delta_i, K))[\uparrow]\;\lambda.\,\underline{2})\big)\;\underline{i}\big)$ and let $s'_i = (\,\underline{1}\,.\,\underline{2}\,.\cdots.\,\underline{i{-}1}\,.b.\uparrow^i)$. Let $r = s'_i \circ t$.

      ii. Suppose $K \in \mathcal{T}(\Gamma_i)$. Let b and s'_i be as above. Let $r = t \circ s'_i$.

   (b) Suppose $s(K_1) = s(K_2) = L$ for distinct $K_1, K_2 \in \mathcal{T}(\tau)$

      i. Suppose $K_j \in \mathcal{T}(\Gamma_{i_j})$ for $j{\in}\{1,2\}$. Let $p_j{=}\big(\lambda.\phi_1(\Gamma_{i_j}, K_j)\;\;\underline{i_j{+}1}\big)$ and $p{=}\lambda.\lambda.\big(\underline{1}\;\;p_1[\uparrow]\;\;p_2[\uparrow]\big)$. Let $b_j{=}\big(\lambda.\lambda.\,\underline{2}\;\;\underline{i_j}\;\;p\big)$, where j can be either 1 or 2 and let $s_{i_j}{=}(\,\underline{1}\,.\,\underline{2}\,.\cdots.\,\underline{i_j{-}1}\,.b_j.\uparrow^{i_j})$. Let $r = t \circ s_{i_j}$.

      ii. Suppose $K_j{\in}\mathcal{T}(\Delta_{i_j})$, $j{\in}\{1,2\}$. Let $p_j{=}\big(\lambda.\phi_1(\Delta_{i_j}, K_j)\;\;\underline{i_j{+}1}\big)$. Then, for p, b_j and s_{i_j} as above, let $r = s_{i_j} \circ t$.

      iii. Suppose $K_1 \in \mathcal{T}(\Gamma_i)$ and $K_2 \in \mathcal{T}(\Delta_j)$. Let $b = \big(\lambda.(\lambda.\,\underline{2}\;\;p)\;\;\underline{j}\,[t]\big)$, where $p = \lambda.\lambda.\big(\underline{1}\;\big((\lambda.\phi_1(\Gamma_i, K_1))[\uparrow]\;\;\underline{i{+}3}\big)\;\phi_2(\Delta_j, K_2)[\uparrow]\big)$. Let $r = \big(\underline{1}\,[t].\underline{2}\,[t].\cdots.\,\underline{j-1}\,[t].b.(\uparrow^j \circ t)\big)$.

   Then, $r \in Terms_{TA_{\lambda\sigma}}(\tau') \setminus Terms_{TA_{\lambda\sigma}}(\tau)$. Thus, $\tau' \not\leq_{TA_{\lambda\sigma}} \tau$

As consequence, if a typing τ' of some $\lambda\sigma$-substitution is not PT according to Definition 6, τ' is not PT according to Definition 2. □

B.3 Proof of PT

Proof of Theorems 2, 4 and 6. Let a be any term (expression in $\lambda\sigma$) and a' its decorated version. Let R_0 be the set of all sub-terms(sub-expression) of a'. Starting with the pair $\langle R_0, \emptyset\rangle$ and applying the rules of the type inference algorithm in the Table 1, 2 or 3 one obtains a final pair after a finite number of steps, because after each step the number of elements in the set of decorated sub-terms(sub-expressions) of the pair is decremented. By the uniqueness in the decomposition of the sub-terms(sub-expressions) in each calculus, one has that a unique rule can be applied to each element of R_0. Thus, the process finishes with a pair $\langle\emptyset, E_f\rangle$, where E_f is a set of first-order equations over context and type variables, according to the rules of the type systems $TA_{\lambda dB}$, $TA_{\lambda\sigma}$ and $TA_{\lambda s_e}$ respectively. An adequate first-order unification algorithm, e.g. see [Hi97], is then applied. And by the correctness, completeness and uniqueness of first-order unification, one has that the algorithm will find a mgu in the case that a is typable. Otherwise, the algorithm will report that there are no unifier. Consequently, the typing systems $TA_{\lambda dB}$, $TA_{\lambda\sigma}$ and $TA_{\lambda s_e}$ satisfy PT. □