RWTH Aachen

Department of Computer Science
Technical Report

HOR 2004
2nd International Workshop on Higher-Order Rewriting

Delia Kesner and Femke van Raamsdonk and Joe Wells (eds.)

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2004- 03

RWTH Aachen · Department of Computer Science · June 2004

The publications of the Department of Computer Science of RWTH Aachen (*Aachen University of Technology*) are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Preface

The First Workshop on Higher-Order Rewriting (HOR 2002) was held in July 2002 in Copenhagen, Denmark, as part of the Federated Logic Conference (FLoC 2002).

This report contains the proceedings of the Second International Workshop on Higher-Order Rewriting (HOR 2004), which was held on Wednesday June 2, 2004, in Aachen, Germany. HOR 2004 was part of the Federated Conference on Rewriting, Deduction, and Programming (RDP 2004), which was held from May 31 through June 5, 2004, in Aachen. RDP 2004 consisted of the 15th International Conference on Rewriting Techniques and Applications (RTA 2004) and various workshops.

The aim of HOR is to provide an informal forum to discuss all aspects of higher-order rewriting. We encourage in particular the presentation of work in progress. The topics of the workshop include applications, foundations, frameworks, implementations, and semantics.

We are very grateful to Mariangiola Dezani-Ciancaglini (University of Torino, Italy) and Mark-Oliver Stehr (University of Illinois at Urbana-Champaign, USA) for kindly accepting to give invited talks at HOR 2004. We would like to thank them both for writing papers for these proceedings.

Finally, we would like to thank the organizing committee of RPD 2004, and in particular Jürgen Giesl, for all help in the preparation of the workshop.

May 2004
Delia Kesner (Université Denis Diderot Paris 7, Paris, France)
Femke van Raamsdonk (Vrije Universiteit, Amsterdam, The Netherlands)
Joe Wells (Heriot-Watt University, Edinburgh, Scotland)

Contents

Part I: Invited Talks

Inverse Limit Models as Filter Models

Fabio Alessi[1], Mariangiola Dezani-Ciancaglini[2⋆], and Furio Honsell[1]

[1] Dipartimento di Matematica e Informatica, Via delle Scienze, 206 33100 Udine (Italy)
alessi,honsell@dimi.uniud.it
[2] Dipartimento di Informatica, Corso Svizzera, 125 10149 Torino (Italy) dezani@di.unito.it

Abstract. Natural intersection type preorders are the type structures which agree with the plain intuition of intersection type constructor as set-theoretic intersection operation and arrow type constructor as set-theoretic function space constructor. In this paper we study the relation between natural intersection type preorders and natural λ-structures, i.e. ω-algebraic lattices $\mathcal{D}$ with Galois connections given by $F : \mathcal{D} \to [\mathcal{D} \to \mathcal{D}]$ and $G : [\mathcal{D} \to \mathcal{D}] \to \mathcal{D}$. We prove on one hand that natural intersection type preorders induces natural λ-structures, on the other hand that natural λ-structures admits presentations through intersection type preorders. Moreover we give a concise presentations of classical D_∞ λ-models of untyped λ-calculus through suitable natural intersection type preorders and prove that filter λ-models induced by them are isomorphic to D_∞.

1 Introduction

Intersection type preorders can be viewed as *domain logics* for ω-algerbraic lattices (see [CDCHL84], [Abr91]). That means that ω-algebraic lattices can be defined in a syntactic way through "axioms and rules" which involve intersection type preorders. This possibility brings a nice consequence. The classical way to interpret a statement of the shape $M \models \phi$ (the program M satisfies the property ϕ) in a semantic domain $\mathcal{D}$ is to view M as a point in $\mathcal{D}$, and ϕ as a (suitable) subset Φ of $\mathcal{D}$, obtaining a membership judgment in $\mathcal{D}$: i.e. $M \models \phi$ is translated into $[\![M]\!]^{\mathcal{D}} \in \Phi$, where the interpretation function $[\![\cdot]\!]$ maps programs to elements of $\mathcal{D}$. The Stone duality perspective uses intersection type preorders in order to "reverse" this point of view. Types are taken for setting up a basis for the topology of the space (in algebraic terms: the meet-semilattice of coprime compact open sets of the lattice under consideration). Points are not the building blocks of the semantic domains, rather they are recovered as *filters* of types. Following this view $M \models \phi$ is translated in an "opposite" membership judgment $A \in [\![M]\!]^{\mathcal{D}}$, that is: "the type A (corresponding to the property ϕ and interpreted as Φ) is a member of the filter (of properties) which sets up the whole interpretation of M".

This view is fruitful in the following sense: the interpretation of a program is fully determined when all the properties which the program satisfies are known. Since actually the syntactic way of defining lattices through intersection type preorders puts at disposal a machinery (the *type assignment system*) which allows to assign types/properties to programs in a finitary way, the gain consists in the possibility of defininig program

⋆ Partially supported by EU within the FET - Global Computing initiative, project DART ST-2001-33477, and by MURST Cofin'02 project McTati. The funding bodies are not responsible for any use that might be made of the results presented here.

interpretations by answering the question:"which types can be assigned to programs by the type assignment system?", whose answer can in turn exploit useful technical results on type assignment system (such as, for instance, the Generation Theorem at page 11).

Since, as mentioned, Stone duality is the mathematical framework where to settle the relationship between intersection type preorders and ω-algebraic complete lattices, we now recall shortly some basics facts concerning it. A complete treatment can be found in the milestone paper [Abr91].

Let X be a topological space with topology $\Omega(X)$ (we recall that $\Omega(X)$ is a frame, that is a complete distributive lattice).

Define a *completely prime filter* over X[1] as a subset $\xi \subseteq \Omega(X)$ such that (a, b range over $\Omega(X)$):

- $X \in \xi$;
- $a \in \xi$ and $a \subseteq b$ imply $b \in \xi$;
- $a \in \xi$ and $b \in \xi$ imply $a \cap b \in \xi$;
- $\bigcup_{i \in I} a_i \in \xi$ implies $a_i \in \xi$ for some $i \in I$.

Let $\mathsf{Pt}(\Omega(X))$ be the set of all completely prime filters over $\Omega(X)$. The fundamental result is that if we work in the category **Sob** of *sober* spaces, then we have bijections

$$(\dagger) \quad X \simeq \mathsf{Pt}(\Omega(X))$$

from which it follows an equivalence between the categories **Sob** and **Loc** (this last one is the opposite of the category of frames).

The importance of this result can be summarized as follows: given certain topological spaces (the sober ones), one can forget points, since topology allows to recover them completely.

Without entering the details of the rather involved definition of sober space (see [Joh86]), we just recall that all algebraic domains used in denotational semantics enjoy the property of being sober.

Intersection type preorders are particular structures which arise when restricting the equivalence (†) above to the case of the category **ALG** of ω-algebraic lattices endowed with their Scott topology. In such a case, it is possible to exploit the following property of the topology of ω-algebraic lattices: $\Omega(X)$ can be completely recoverd by the subsets $\mathsf{Cpr}(\Omega(X))$ of the *coprime* compact open sets (an open set a is coprime if $a \subseteq b \cup c$ implies $a \subseteq b$ or $a \subseteq c$). The domain $\mathsf{Cpr}(\Omega(X))$ turns out to be a meet-semilattice (whence the meet-semilattice structure of intersection types) and it satisfies

$$\mathsf{Pt}(\Omega(X)) \simeq \mathsf{Filt}(\mathsf{Cpr}(\Omega(X))),$$

where Filt is the operation of taking filters (defined by dropping the last condition in the definition above of completely prime filter). As a consequence of (†), any ω-algebraic lattice X satisfies

$$X \simeq \mathsf{Filt}(\mathsf{Cpr}(\Omega(X))).$$

[1] Actually one can take completely prime filters over any complete lattice $\mathcal{D}$, not just topologies.

A further step is to notice that $\mathsf{Filt}(\mathsf{Cpr}(\Omega(X)))$ is isomorphic to $\mathcal{K}^{op}(X)$, the subspace of compact elements of X, with the reverse ordering of X. Thus the final form which the "Stone duality" theory assumes when applied to ω-algebraic lattices is expressed by the isomorphism:

$$X \simeq \mathsf{Filt}(\mathcal{K}^{op}(X)).$$

This result is the foundation which guarantees the possibility of describing ω-algebraic lattices by means of intersection type preorders.

In the present paper we are mainly interested in a fine analysis of type preorders which agree with the intuition that arrow type constructor corresponds to the set-theoretic continuous function space constructor. We call *natural* this kind of type preorders. Our first result is to show that the semantic counterpart of natural type preorders are ω-algebraic lattices $\mathcal{D}$ endowed with pairs of continuous function $F : \mathcal{D} \to [\mathcal{D} \to \mathcal{D}]$, $G : [\mathcal{D} \to \mathcal{D}] \to \mathcal{D}$ which set up Galois connection:

$$F \circ G \sqsupseteq Id_{[\mathcal{D}\to\mathcal{D}]} \qquad G \circ F \sqsubseteq Id_{\mathcal{D}}.$$

We call *natural λ-structures* this kind of lattices. We prove on one hand that the space of filter on a natural type preorders is a natural λ-structure. On the other hand natural λ-structures can be presented via natural type preorders, that is

(iso) each natural λ-structure is isomorphic, both as lattice and as applicative structure, to the space of filters of a suitable natural type preorder.

Then we turn our attention to λ-models of untyped λ-calculus computed inside **ALG**, built through the classical inverse limit technique (see [Sco72]). As a consequence of *(iso)*, for any D_∞ it is possible to build a filter structure isomorphic to it, but the construction given in the proof of *(iso)* is not effective and uses a possibly countable amount of redundant types (since it introduces a constant type for any compact element of the domain). So we look for a more concise presentation of D_∞. Our second result is to prove that the natural type preorder which induces a filter λ-model isomorphic to D_∞, starting from D_0, is exactly the natural type preorder *freely generated* by a type preorder which induces D_0 together with the equalities which arise from encoding the initial projections.

This second isomorphism result could be obtained by adapting the technique of [Abr91], Section 4. Our approach does not use the complex Abramsky's machinery (tailored for more general domains, the SFP's ones) and allows to get a rather quick isomorphism proof.

Finally, the organization of the paper. In Section 2 we recall some standard facts on ω-algebraic lattices, and introduce natural λ-structures. Section 3 discusses type preorders, filter structures and type assignment systems. In Section 4 we prove the two isomorphism results which relate natural intersection type preorders with natural λ-structures. Finally, in Section 5, we give the effective and "concise" presentations of D_∞ λ-models via suitable natural intersection type preorders and show that the filter structures induced by them are isomorphic to D_∞'s.

2 Natural λ-structures

We start with a standard definition:

Definition 1. *1. If $\mathcal{D}$ is an ω-algebraic complete lattice, $[\mathcal{D} \to \mathcal{D}]$ denotes the set of continuous functions from $\mathcal{D}$ to $\mathcal{D}$, and $\mathcal{K}(\mathcal{D})$ the set of compact elements of $\mathcal{D}$.*
2. If $a, b \in \mathcal{D}$, then $a \Rightarrow b$ is the step function defined by

$$a \Rightarrow b\,(d) = \textit{ if } a \sqsubseteq d \textit{ then } b \textit{ else } \bot.$$

Recall that the compact elements in the domain of continuous functions are exactly the sups of finite sets of step functions between compact elements. Moreover we restate some well know properties of continuous functions [GHK$^+$80]. Let I be a finite set.

Proposition 1. *1. $c \Rightarrow d \sqsubseteq \bigsqcup_{i\in I}(a_i \Rightarrow b_i)$ iff $d \sqsubseteq \bigsqcup_{i\in J} b_i$ where $J = \{i \in I \mid a_i \sqsubseteq c\}$.*
2. Each continuous function f is the sup of the step functions between compact elements which are under f, i.e.

$$\begin{aligned} f &= \bigsqcup\{a \Rightarrow b \mid a \Rightarrow b \sqsubseteq f, a \textit{ and } b \textit{ compact}\} \\ &= \bigsqcup\{a \Rightarrow b \mid b \sqsubseteq f(a), a \textit{ and } b \textit{ compact}\}. \end{aligned}$$

Next definition introduces *natural λ-structures*. Natural λ-structures set up a bridge between domain theoretic λ-models and *filter structures*: more precisely, they are the semantic counterpart of those intersection type preorders (the *natural* ones, see Definition 6) whose axioms agree with the intuition that the arrow type constructor corresponds to the set-theoretic function space constructor.

Definition 2 (Natural λ-structure). *A* natural λ-structure *is a triple $\langle \mathcal{D}, F_{\mathcal{D}}, G_{\mathcal{D}}\rangle$, where $\mathcal{D}$ is an ω-algebraic complete lattice, and $F_{\mathcal{D}} : \mathcal{D} \to [\mathcal{D} \to \mathcal{D}]$, $G_{\mathcal{D}} : [\mathcal{D} \to \mathcal{D}] \to \mathcal{D}$ are Scott continuous functions such that $\langle F_{\mathcal{D}}, G_{\mathcal{D}}\rangle$ sets up a Galois connection, i.e.:*

1. $F_{\mathcal{D}} \circ G_{\mathcal{D}} \sqsupseteq Id_{[\mathcal{D}\to\mathcal{D}]}$;
2. $G_{\mathcal{D}} \circ F_{\mathcal{D}} \sqsubseteq Id_{\mathcal{D}}$.

Given a natural λ-structure $\langle \mathcal{D}, F_{\mathcal{D}}, G_{\mathcal{D}}\rangle$ and $a, b \in \mathcal{D}$, we will often write $a \cdot b$ as short for $F_{\mathcal{D}}(a)(b)$.

Example 1. An example of a natural λ-structure is $\langle \mathcal{D}^{\spadesuit}, F_{\spadesuit}, G_{\spadesuit}\rangle$, where

- $\mathcal{D}^{\spadesuit}$ is $\mathbb{N} \cup \{\bot, \top\}$, endowed with the order which is flat on natural numbers, and moreover $\bot = m \sqcap n$, $\top = m \sqcup n$, for any $m, n \in \mathbb{N}$, $m \neq n$;
- $F_{\spadesuit}(a) = (\bot \Rightarrow a)$ for any $a \in \mathcal{D}^{\spadesuit}$;
- $G_{\spadesuit}(f) = f(\top)$ for any $f \in [D^{\spadesuit} \to \mathcal{D}^{\spadesuit}]$.

$\langle \mathcal{D}^{\spadesuit}, F_{\spadesuit}, G_{\spadesuit}\rangle$ is a natural λ-structure. In fact

- $G_{\spadesuit}(F_{\spadesuit}(a)) = G_{\spadesuit}(\bot \Rightarrow a) = (\bot \Rightarrow a)(\top) = a$;
- $F_{\spadesuit}(G_{\spadesuit}(f)) = (\bot \Rightarrow f(\top)) \sqsupseteq f$,

hence $F_\spadesuit$ and $G_\spadesuit$ set up a Galois connection.

Natural λ-structures are λ-structures as defined in [Plo93], Section 3.

The following properties of natural λ-structures follow easily from their definitions. Although they are almost immediate consequence of the fact that, from a categorical point of view, $F_\mathcal{D}$ is left adjoint of $G_\mathcal{D}$, we will recall the direct proof.

Proposition 2. *Let $\langle \mathcal{D}, F_\mathcal{D}, G_\mathcal{D} \rangle$ be a natural λ-structure.*

1. *$G_\mathcal{D}$ maps always compact elements into compact elements.*
2. *$F_\mathcal{D}$ determines $G_\mathcal{D}$ by*

$$G_\mathcal{D}(f) = \sqcap\{d \mid f \sqsubseteq F_\mathcal{D}(d)\}$$

   for all continuous functions f.
3. *$G_\mathcal{D}$ is additive, $G_\mathcal{D}(f \sqcup g) = G_\mathcal{D}(f) \sqcup G_\mathcal{D}(g)$.*

Proof. Notice that, by condition (2) of Definition 2,

$$(*) \quad G_\mathcal{D}(f) \sqsubseteq G_\mathcal{D}(F_\mathcal{D}(d)) \text{ imply } G_\mathcal{D}(f) \sqsubseteq d.$$

1. We show that if f is compact then $G_\mathcal{D}(f)$ is compact, that is if $G_\mathcal{D}(f) \sqsubseteq \bigsqcup_{z\in Z} z$, where Z is directed, then $G_\mathcal{D}(f) \sqsubseteq z$ for some $z \in Z$.

$$\begin{aligned}
G_\mathcal{D}(f) \sqsubseteq \textstyle\bigsqcup_{z\in Z} z &\Rightarrow F_\mathcal{D}(G_\mathcal{D}(f)) \sqsubseteq \textstyle\bigsqcup_{z\in Z} F_\mathcal{D}(z) \\
&\quad \text{since } F_\mathcal{D} \text{ is continuous} \\
&\Rightarrow f \sqsubseteq \textstyle\bigsqcup_{z\in Z} F_\mathcal{D}(z) \\
&\quad \text{by condition (1) of Definition 2} \\
&\Rightarrow \exists z \in Z. f \sqsubseteq F_\mathcal{D}(z) \\
&\quad \text{since } f \text{ is compact and } \{F_\mathcal{D}(z) \mid z \in Z\} \text{ is directed} \\
&\Rightarrow \exists z \in Z. G_\mathcal{D}(f) \sqsubseteq G_\mathcal{D}(F_\mathcal{D}(z)) \\
&\quad \text{since } G_\mathcal{D} \text{ is monotone} \\
&\Rightarrow \exists z \in Z. G_\mathcal{D}(f) \sqsubseteq z \\
&\quad \text{by (*).}
\end{aligned}$$

2. It sufficies to show that $G_\mathcal{D}(f) \sqsubseteq d$ iff $f \sqsubseteq F_\mathcal{D}(d)$.

$$\begin{aligned}
G_\mathcal{D}(f) \sqsubseteq d &\Rightarrow F_\mathcal{D}(G_\mathcal{D}(f)) \sqsubseteq F_\mathcal{D}(d) \quad \text{since } F_\mathcal{D} \text{ is monotone} \\
&\Rightarrow f \sqsubseteq F_\mathcal{D}(d) \qquad\qquad \text{by condition (1) of Definition 2}
\end{aligned}$$

$$\begin{aligned}
f \sqsubseteq F_\mathcal{D}(d) &\Rightarrow G_\mathcal{D}(f) \sqsubseteq G_\mathcal{D}(F_\mathcal{D}(d)) \quad \text{since } G_\mathcal{D} \text{ is monotone} \\
&\Rightarrow G_\mathcal{D}(f) \sqsubseteq d \qquad\qquad \text{by (*).}
\end{aligned}$$

3. We have

$$\begin{aligned}
G_\mathcal{D}(f \sqcup g) &\sqsubseteq G_\mathcal{D}(F_\mathcal{D}(G_\mathcal{D}(f)) \sqcup F_\mathcal{D}(G_\mathcal{D}(g))) && \text{by condition (1) of Definition 2} \\
&\sqsubseteq G_\mathcal{D}(F_\mathcal{D}(G_\mathcal{D}(f) \sqcup G_\mathcal{D}(g))) && \text{since } F_\mathcal{D} \text{ is continuous} \\
&\sqsubseteq G_\mathcal{D}(f) \sqcup G_\mathcal{D}(g) && \text{by condition (2) of Definition 2.}
\end{aligned}$$

Natural λ-structures provide interpretation to terms of λ-calculus in a standard way: interpretation of application is obtained by applying $F_{\mathcal{D}}$ to the interpretation of the term M (in function position) in (MN); interpretation of abstraction is obtained by applying $G_{\mathcal{D}}$ to the function induced by $\lambda x.M$. Notice that the possibility of interpreting λ-terms just relies on the existence of $F_{\mathcal{D}}$ and $G_{\mathcal{D}}$, independently from the fact they set up a Galois connection.

In the following Λ denotes the set of λ-terms, Env_D denotes the set of functions *Var* $\to \mathcal{D}$ from term variables to $\mathcal{D}$ (term environments).

Definition 3. *Let $\langle \mathcal{D}, F_{\mathcal{D}}, G_{\mathcal{D}} \rangle$ be a natural λ-structure. The interpretation $[\![\ ]\!]^{\mathcal{D}} : \Lambda \times \mathsf{Env}_{\mathcal{D}} \to \mathcal{D}$ is defined inductively on λ-terms as follows:*

$$\begin{array}{ll} [\![x]\!]^{\mathcal{D}}_{\rho} & = \rho(x); \\ [\![MN]\!]^{\mathcal{D}}_{\rho} & = F_{\mathcal{D}}([\![M]\!]^{\mathcal{D}}_{\rho})([\![N]\!]^{\mathcal{D}}_{\rho}); \\ [\![\lambda x.M]\!]^{\mathcal{D}}_{\rho} & = G_{\mathcal{D}}(\boldsymbol{\lambda} d \in \mathcal{D}.[\![M]\!]^{\mathcal{D}}_{\rho[x:=d]}) \end{array}$$

where ρ ranges over the set of term environments Env_D.

Example 2. Consider the natural λ-structure $\mathcal{D}^{\spadesuit}$ defined in Example 1. Then for any $M \in \Lambda$, $[\![(\lambda x.x)M]\!]^{\mathcal{D}^{\spadesuit}}_{\rho} = \top$. In fact

$$\begin{array}{ll} [\![(\lambda x.x)]\!]^{\mathcal{D}^{\spadesuit}}_{\rho} & = G_{\spadesuit}(\boldsymbol{\lambda} d \in \mathcal{D}^{\spadesuit}.d) \\ & = G_{\spadesuit}(\bigsqcup\{a \Rightarrow a \mid a \in D^{\spadesuit}\}) \\ & = (\bigsqcup\{a \Rightarrow a \mid a \in D^{\spadesuit}\})(\top) \\ & = \top. \end{array}$$

Therefore

$$\begin{array}{ll} [\![(\lambda x.x)M]\!]^{\mathcal{D}^{\spadesuit}}_{\rho} & = F_{\spadesuit}(\top)([\![M]\!]^{\mathcal{D}^{\spadesuit}}_{\rho}) \\ & = (\bot \Rightarrow \top)([\![M]\!]^{\mathcal{D}^{\spadesuit}}_{\rho}) \\ & = \top. \end{array}$$

By the way notice that this proves that $\langle \mathcal{D}^{\spadesuit}, F_{\spadesuit}, G_{\spadesuit} \rangle$ is not a λ-model, since, for any y, ρ such that $\rho(y) = \bot$, it follows

$$\begin{array}{ll} [\![(\lambda x.x)y]\!]^{\mathcal{D}^{\spadesuit}}_{\rho} & = \top \\ & \neq \bot \\ & = [\![y]\!]^{\mathcal{D}^{\spadesuit}}_{\rho}. \end{array}$$

As well known, whenever $F_{\mathcal{D}} \circ G_{\mathcal{D}} = Id_{[\mathcal{D} \to \mathcal{D}]}$, the λ-structure $\langle \mathcal{D}, F_{\mathcal{D}}, G_{\mathcal{D}} \rangle$ is a λ-model, being a reflexive object in the cartesian closed category of ω-algebraic lattice and continuous functions.

The notion of isomorphism between λ-structures is as expected: a lattice isomorphism which "commutes" with F and G.

Definition 4 (Isomorphism of natural λ-structures). *Two natural λ-structures $\langle \mathcal{D}, F_{\mathcal{D}}, G_{\mathcal{D}} \rangle$ and $\langle \mathcal{E}, F_{\mathcal{E}}, G_{\mathcal{E}} \rangle$ are isomorphic if there exists a lattice isomorphism $\mathsf{m} : \mathcal{D} \to \mathcal{E}$ such that for any $d \in \mathcal{D}$ and $f \in [\mathcal{D} \to \mathcal{D}]$:*

1. $F_{\mathcal{E}}(\mathsf{m}(d)) = \mathsf{m} \circ F_{\mathcal{D}}(d) \circ \mathsf{m}^{-1}$,
2. $\mathsf{m}(G_{\mathcal{D}}(f)) = G_{\mathcal{E}}(\mathsf{m} \circ f \circ \mathsf{m}^{-1})$.

It is easy to show that previous definition can be simplified.

Proposition 3. *Two natural λ-structures $\langle \mathcal{D}, F_{\mathcal{D}}, G_{\mathcal{D}} \rangle$ and $\langle \mathcal{E}, F_{\mathcal{E}}, G_{\mathcal{E}} \rangle$ are isomorphic iff there exists a lattice isomorphism $\mathsf{m} : \mathcal{D} \to \mathcal{E}$ such that*

$$\forall d, d' \in \mathcal{D}.\mathsf{m}(d \cdot d') = \mathsf{m}(d) \cdot \mathsf{m}(d').$$

Proof. First notice that condition (1) of Definition 4 is equivalent to the condition of Proposition 3. So it is enough to prove that condition (1) of Definition 4 implies condition (2) of the same definition.

Proof of $G_{\mathcal{E}}(\mathsf{m} \circ f \circ \mathsf{m}^{-1}) \sqsubseteq \mathsf{m}(G_{\mathcal{D}}(f))$.

$$\begin{array}{lll} G_{\mathcal{E}}(\mathsf{m} \circ f \circ \mathsf{m}^{-1}) \sqsubseteq G_{\mathcal{E}}(\mathsf{m} \circ (F_{\mathcal{D}}(G_{\mathcal{D}}(f)) \circ \mathsf{m}^{-1}) & \text{by condition (1) of Definition 2} \\ = G_{\mathcal{E}}(F_{\mathcal{E}}(\mathsf{m}(G_{\mathcal{D}}(f)))) & \text{by condition (1) of Definition 4} \\ \sqsubseteq \mathsf{m}(G_{\mathcal{D}}(f)) & \text{by condition (2) of Definition 2.} \end{array}$$

Before proving the other inequality, notice that in a symmetric way we can show

$$(\natural) \quad G_{\mathcal{D}}(\mathsf{m}^{-1} \circ f \circ \mathsf{m}) \sqsubseteq \mathsf{m}^{-1}(G_{\mathcal{E}}(f)).$$

Proof of $G_{\mathcal{E}}(\mathsf{m} \circ f \circ \mathsf{m}^{-1}) \sqsupseteq \mathsf{m}(G_{\mathcal{D}}(f))$.

$$\begin{array}{lll} \mathsf{m}(G_{\mathcal{D}}(f)) = \mathsf{m}(G_{\mathcal{D}}(\mathsf{m}^{-1} \circ \mathsf{m} \circ f \circ \mathsf{m}^{-1} \circ \mathsf{m})) & & \\ \sqsubseteq \mathsf{m}(G_{\mathcal{D}}(\mathsf{m}^{-1} \circ (F_{\mathcal{E}}(G_{\mathcal{E}}(\mathsf{m} \circ f \circ \mathsf{m}^{-1})) \circ \mathsf{m})) & \text{by condition (1) of Definition 2} \\ = \mathsf{m}(\mathsf{m}^{-1}(G_{\mathcal{E}}(F_{\mathcal{E}}(G_{\mathcal{E}}(\mathsf{m} \circ f \circ \mathsf{m}^{-1}))))) & \text{by } (\natural) \\ = G_{\mathcal{E}}(\mathsf{m} \circ f \circ \mathsf{m}^{-1}) & \text{by condition (2) of Definition 2.} \end{array}$$

3 Natural filter structures

Intersection types, the building blocks for the filter λ-models, are syntactical objects built by closing a given set $\mathbb{C}$ of *type atoms* (constants), which contains the universal type Ω, under the *function type* constructor $\to$ and the *intersection type* constructor $\cap$.

Definition 5 (Intersection type language). *The* intersection type language *over $\mathbb{C}$, denoted by $\mathbb{T} = \mathbb{T}(\mathbb{C})$, is defined by the following abstract syntax:*

$$\mathbb{T} = \mathbb{C} \mid \mathbb{T} \to \mathbb{T} \mid \mathbb{T} \cap \mathbb{T}.$$

Much of the expressive power of intersection type languages comes from the fact that they are endowed with a *preorder relation*, which induces, on the set of types, the structure of a meet semi-lattice with respect to intersection. We consider here a class of preorder relations we call natural, for the general definition see [ADCH03].

Definition 6 (Natural intersection type preorder).

1. *A* natural intersection type preorder (nitp) Σ *is a pair* $(\mathbb{C}^\Sigma, \leq_\Sigma)$ *where* $\mathbb{C}^\Sigma$ *is a set of type constants and* $\leq_\Sigma$ *is a binary relation over* $\mathbb{T}^\Sigma = \mathbb{T}(\mathbb{C}^\Sigma)$ *satisfying the following set* ∇^0 *("nabla-zero") of axioms and rules:*

$$(refl)\quad A \leq_\Sigma A \qquad (idem)\quad A \leq_\Sigma A \cap A$$

$$(incl_L)\quad A \cap B \leq_\Sigma A \qquad (incl_R)\quad A \cap B \leq_\Sigma B$$

$$(mon)\quad \frac{A \leq_\Sigma A' \quad B \leq_\Sigma B'}{A \cap B \leq_\Sigma A' \cap B'} \qquad (\textit{trans})\quad \frac{A \leq_\Sigma B \quad B \leq_\Sigma C}{A \leq_\Sigma C}$$

$$(\Omega)\quad A \leq_\Sigma \Omega \qquad (\Omega\text{-}\eta)\quad \Omega \leq_\Sigma \Omega \to \Omega$$

$$(\to\text{-}\cap)\quad (A \to B) \cap (A \to C) \leq_\Sigma A \to B \cap C \qquad (\eta)\quad \frac{A' \leq_\Sigma A \quad B \leq_\Sigma B'}{A \to B \leq_\Sigma A' \to B'}$$

2. *A recursive set* ∇ *of axioms and rules of the shape* $A \leq_\nabla B$ *over* $\mathbb{T}^\nabla = \mathbb{T}(\mathbb{C}^\nabla)$ *is said to generate the nitp* $\Sigma^\nabla = (\mathbb{C}^\nabla, \leq_\nabla)$ *if* $A \leq_\nabla B$ *holds iff it can be derived from the axioms and rules of* $\nabla \cup \nabla^0$.

Axiom (Ω) states that each nitp has a maximal element.

The meaning of the last three axioms and rules can be grasped if we consider types to denote subsets of a domain of discourse and we look at $\to$ as the function space constructor in the light of Curry-Scott semantics, see [Sco75]. Thus the type $A \to B$ denotes the set of *total* functions which map each element of A into an element of B. Axiom (Ω-η) expresses the fact that all the objects in our domain of discourse are total functions, i.e. that Ω is equal to $\Omega \to \Omega$ [BCDC83]. This is so since $\Omega \to \Omega$ is the set of functions which applied to an arbitrary element return again an arbitrary element.

The intended interpretation of arrow types motivates axiom ($\to$-$\cap$), which implies that if a function maps A into B, and the same function maps also A into C, then, actually, it maps the whole A into the intersection between B and C (i.e. into $B \cap C$), see [BCDC83].

Rule (η) is also very natural in view of the set-theoretic interpretation. It implies that the arrow constructor is contravariant in the first argument and covariant in the second one. It is clear that if a function maps A into B, and we take a subset A' of A and a superset B' of B, then this function will map also A' into B', see [BCDC83].

Notation.

- $A \sim_\Sigma B$ and $A \sim_\nabla B$ will be short for $A \leq_\Sigma B \leq_\Sigma A$ and $A \leq_\nabla B \leq_\nabla A$, respectively.
- Since $\cap$ is commutative and associative (modulo $\sim_\Sigma$), we shall write $\bigcap_{i \leq n} A_i$ for $A_1 \cap \ldots \cap A_n$. Similarly we shall write $\cap_{i \in I} A_i$, where I denotes always a finite set. Moreover we make the convention that $\cap_{i \in \emptyset} A_i$ is Ω.

Before going on, we give a simple lemma, whose proof is obtained combining rules ($\to$-$\cap$) and (η).

Lemma 1. *Let Σ be a nitp. Then, for any I, $A_i, B_i \in \mathbb{T}^\Sigma$ ($i \in I$), we have:*

$$\bigcap_{i\in I}(A_i \to B_i) \leq_\Sigma \bigcap_{i\in I} A_i \to \bigcap_{i\in I} B_i.$$

We can devise semantic domains out of intersection types by means of an appropriate notion of filter over a type preorder. This is a particular case of filter over a generic meet semi-lattice (see [Joh86]).

Definition 7 (Σ-filters). *A Σ-filter (or a filter over $\mathbb{T}^\Sigma$) is a set $X \subseteq \mathbb{T}^\Sigma$ such that*

1. *$\Omega \in X$;*
2. *if $A \leq_\Sigma B$ and $A \in X$, then $B \in X$;*
3. *if $A, B \in X$, then $A \cap B \in X$.*

$\mathcal{F}^\Sigma$ denotes the set of Σ-filters.

Given $X \subseteq \mathbb{T}^\Sigma$, $\uparrow X$ denotes the Σ-filter generated by X. For $A \in \mathbb{T}^\Sigma$, we write $\uparrow A$ instead of $\uparrow\{A\}$.

Proposition 4. *The set of Σ-filters $\mathcal{F}^\Sigma$, ordered by subset inclusion, is an ω-algebraic complete lattice, where $\uparrow \Omega$ is the bottom, and $\mathbb{T}^\Sigma$ is the top. Moreover if $X, Y \in \mathcal{F}^\Sigma$:*

$$X \sqcup Y = \uparrow (X \cup Y);$$
$$X \sqcap Y = X \cap Y.$$

If $\chi \subseteq \mathcal{F}^\Sigma$ is a directed set, then $\bigsqcup \chi = \bigcup \chi$.
The finite elements are exactly the principal filters.

It is possible to turn the space of filters into a natural λ-structure.

Definition 8 (Filter structures).

1. *Application $\_\cdot\_ : \mathcal{F}^\Sigma \times \mathcal{F}^\Sigma \to \mathcal{F}^\Sigma$ is defined as*

$$X \cdot Y = \uparrow\{B \mid \exists A \in Y. A \to B \in X\}.$$

2. *The maps $F^\Sigma : \mathcal{F}^\Sigma \to [\mathcal{F}^\Sigma \to \mathcal{F}^\Sigma]$ and $G^\Sigma : [\mathcal{F}^\Sigma \to \mathcal{F}^\Sigma] \to \mathcal{F}^\Sigma$ are defined by:*

$$F^\Sigma(X) = \boldsymbol{\lambda} Y \in \mathcal{F}^\Sigma. X \cdot Y;$$
$$G^\Sigma(f) = \uparrow \{A \to B \mid B \in f(\uparrow A)\}.$$

The triple $\langle \mathcal{F}^\Sigma, F^\Sigma, G^\Sigma\rangle$ is called the filter structure *induced by Σ.*

We now give a simple proposition whose results will be useful later on.

Proposition 5. *1. Each $f \in [\mathcal{F}^\Sigma \to \mathcal{F}^\Sigma]$ satisfies*

$$B \in f(\uparrow A) \iff \uparrow A \Rightarrow \uparrow B \sqsubseteq f$$

and

$$f = \bigsqcup\{\uparrow A \Rightarrow \uparrow B \mid B \in f(\uparrow A)\}.$$

2. *For all* $A, B \in \mathbb{T}^{\Sigma}$,

$$B \in X \cdot \uparrow A \textit{ iff } A \to B \in X.$$

Proof. (1) Immediate by Proposition 1(2), taking into account that $\uparrow A \Rightarrow \uparrow B$ are all and only the step functions in $[\mathcal{F}^{\Sigma} \to \mathcal{F}^{\Sigma}]$.
(2) ($\Rightarrow$) If $B \sim_{\Sigma} \Omega$ then $\Omega \to \Omega \leq_{\Sigma} A \to B$ by rule (η). So $A \to B \in X$ by definition of Σ-filter (Definition 7). Otherwise by definition of application (Definition 8(1)) $B \in X \cdot \uparrow A$ iff $B \in \uparrow \{D \mid \exists C \in \uparrow A.\ C \to D \in X\}$. Then there is I and types C_i, D_i such that $A \leq_{\Sigma} \bigcap_{i\in I} C_i$, $\bigcap_{i\in I} D_i \leq_{\Sigma} B$ and $C_i \to D_i \in X$ for all $i \in I$ by definition of Σ-filter (Definition 7). So we get $A \to B \in X$ by axiom ($\to$-$\cap$) and rule (η).
($\Leftarrow$) Trivial.

Arrow types allow to describe the functional behaviour of filters, as shown in the next proposition which relates them with step functions, F^{Σ} and G^{Σ}.

Proposition 6.

1. *For all* $X \in \mathcal{F}^{\Sigma}$ *we get* $F^{\Sigma}(X) = \bigsqcup\{\uparrow A \Rightarrow \uparrow B \mid A \to B \in X\}$.
2. *For all* $A, B \in \mathbb{T}^{\Sigma}$ *we get* $G^{\Sigma}(\uparrow A \Rightarrow \uparrow B) = \uparrow (A \to B)$.

Proof. (1) Let $\Xi = \bigsqcup\{\uparrow A \Rightarrow \uparrow B \mid A \to B \in X\}$. It suffices to show

$$D \in \Xi(\uparrow C) \;\Leftrightarrow\; D \in F^{\Sigma}(X)(\uparrow C).$$

We first prove ($\Leftarrow$). If $D \in F^{\Sigma}(X)(\uparrow C)$, then, by Proposition 5(2), it follows $C \to D \in X$. From this fact and $D \in (\uparrow C \Rightarrow \uparrow D)(\uparrow C)$, a fortiori we get immediately $D \in \Xi(\uparrow C)$.
($\Rightarrow$). If $D \in \Xi(\uparrow C)$, then, by definition of step function, we get $\uparrow C \Rightarrow \uparrow D \sqsubseteq \Xi$. By compactness of $\uparrow C \Rightarrow \uparrow D$ and Proposition 1(1), there exist I finite set and $A_i, B_i \in \mathbb{T}^{\Sigma}$, such that $\forall i \in I$, $A_i \to B_i \in X$, $\bigsqcup_{i\in I} \uparrow A_i \subseteq \uparrow C$, and $\uparrow D \subseteq \bigsqcup_{i\in I} \uparrow B_i$. We rewrite the previous three statements using the fact that X is a Σ-filter and Proposition 4 as follows:
(a) $\bigcap_{i\in I}(A_i \to B_i) \in X$;
(b) $C \leq_{\Sigma} \bigcap_{i\in I} A_i$;
(c) $\bigcap_{i\in I} B_i \leq_{\Sigma} D$.
Using rule (η) and (b), (c) above, we get $\bigcap_{i\in I} A_i \to \bigcap_{i\in I} B_i \leq_{\Sigma} C \to D$. This last judgment, along with rule (*trans*) and Lemma 1, imply $\bigcap_{i\in I}(A_i \to B_i) \leq_{\Sigma} C \to D$. By (a) above and the fact that X is a Σ-filter, we get $C \to D \in X$, hence $D \in F^{\Sigma}(X)(\uparrow C)$ by Proposition 5(2).
(2)

$$\begin{aligned} G^{\Sigma}(\uparrow A \Rightarrow \uparrow B) &= \uparrow \{C \to D \mid D \in (\uparrow A \Rightarrow \uparrow B)(\uparrow C)\} \text{ by definition of } G^{\Sigma} \\ &\supseteq \uparrow (A \to B). \end{aligned}$$

$$\begin{aligned} G^{\Sigma}(\uparrow A \Rightarrow \uparrow B) &= \uparrow \{C \to D \mid D \in (\uparrow A \Rightarrow \uparrow B)(\uparrow C)\} \text{ by definition of } G^{\Sigma} \\ &= \uparrow \{C \to D \mid C \leq_{\Sigma} A \text{ and } B \leq_{\Sigma} D\} \\ &\subseteq \uparrow \{C \to D \mid A \to B \leq_{\Sigma} C \to D\} \qquad \text{by rule } (\eta) \\ &= \uparrow (A \to B). \end{aligned}$$

3.1 Interpreting λ-terms in filter structures

Any filter structure $\mathcal{F}^\Sigma$, being endowed with the two mappings F^Σ and G^Σ, can be turned into a domain where to interpret λ-calculus by using the interpretation function $[\![\ ]\!]^{\mathcal{F}^\Sigma}$ as defined in Definition 3. In this subsection we will see how this interpretation can be built by means of a suitable *type assignemnt system*. The advantage of using type assignment systems consists in the possibility of calculating term interpretation in a finitary way, as filters of types that can be assigned to terms.

Definition 9 (Type assignment system). *The* intersection type assignment system *relative to the nitp Σ, notation $\lambda\cap^\Sigma$, is a formal system for deriving judgements of the form $\Gamma \vdash^\Sigma M : A$, where the* subject M *is an untyped λ-term, the* predicate A *is in $\mathbb{T}^\Sigma$, and Γ is a Σ-basis. Its axioms and rules are the following:*

$$(Ax)\ \frac{(x:A)\in\Gamma}{\Gamma\vdash^\Sigma x:A} \qquad (Ax\text{-}\Omega)\ \Gamma\vdash^\Sigma M:\Omega$$

$$(\to \mathrm{I})\ \frac{\Gamma, x:A\vdash^\Sigma M:B}{\Gamma\vdash^\Sigma \lambda x.M:A\to B} \qquad (\to \mathrm{E})\ \frac{\Gamma\vdash^\Sigma M:A\to B\ \ \Gamma\vdash^\Sigma N:A}{\Gamma\vdash^\Sigma MN:B}$$

$$(\cap\mathrm{I})\ \frac{\Gamma\vdash^\Sigma M:A\ \ \Gamma\vdash^\Sigma M:B}{\Gamma\vdash^\Sigma M:A\cap B} \qquad (\leq)\ \frac{\Gamma\vdash^\Sigma M:A\ \ A\leq_\Sigma B}{\Gamma\vdash^\Sigma M:B}$$

It is easy to verify that the following rules are admissible [2] :

$$(\leq \mathrm{L})\ \frac{\Gamma, x:A\vdash M:B\ \ A'\leq_\Sigma A}{\Gamma, x:A'\vdash M:B}$$

$$(\mathrm{W})\ \frac{\Gamma\vdash M:B\ \ x\notin\Gamma}{\Gamma, x:A\vdash M:B} \qquad (\mathrm{S})\ \frac{\Gamma, x:A\vdash M:B\ \ x\notin FV(M)}{\Gamma\vdash M:B}$$

We continue with a standard Generation Theorem, which is necessary for proving the main result of this subsection.

Theorem 1 (Generation Theorem).

1. *Assume $A\not\sim_\Sigma\Omega$. Then $\Gamma\vdash^\Sigma x:A$ iff $(x:B)\in\Gamma$ and $B\leq_\Sigma A$ for some $B\in\mathbb{T}^\Sigma$.*
2. *$\Gamma\vdash^\Sigma MN:A$ iff $\Gamma\vdash^\Sigma M:B\to A$, and $\Gamma\vdash^\Sigma N:B$ for some $B\in\mathbb{T}^\Sigma$.*
3. *$\Gamma\vdash^\Sigma \lambda x.M:A$ iff $\Gamma, x:B_i\vdash^\Sigma M:C_i$ and $\bigcap_{i\in I}(B_i\to C_i)\leq_\Sigma A$, for some I and $B_i, C_i\in\mathbb{T}^\Sigma$.*

Proof. The proof of each ($\Leftarrow$) is easy. So we only treat ($\Rightarrow$).

(1) Easy by induction on derivations, since only the axioms (Ax), (Ax-Ω), and the rules ($\cap$I), ($\leq$) can be applied. Notice that the condition $A\not\sim_\Sigma\Omega$ implies that $\Gamma\vdash^\Sigma x:A$ cannot be obtained just using axiom (Ax-Ω).

[2] Recall that a rule is *admissible* in a system if, for each instance of the rule, if its premises are derivable in the system then so is its conclusion.

(2) If $A \sim_\Sigma \Omega$ we can choose $B \sim_\Sigma \Omega$. Otherwise, the proof is by induction on derivations. The only interesting case is when $A \equiv A_1 \cap A_2$ and the last applied rule is ($\cap$I):

$$(\cap\text{I})\ \frac{\Gamma \vdash^\Sigma MN : A_1 \quad \Gamma \vdash^\Sigma MN : A_2}{\Gamma \vdash^\Sigma MN : A_1 \cap A_2}.$$

The condition $A \not\sim_\Sigma \Omega$ implies that we cannot have $A_1 \sim_\Sigma A_2 \sim_\Sigma \Omega$. We give the proof for $A_1 \not\sim_\Sigma \Omega$ and $A_2 \not\sim_\Sigma \Omega$, the other cases can be treated similarly. By induction there are B_1, B_2 such that

$$\begin{array}{c}\Gamma \vdash^\Sigma M : B_1 \to A_1,\ \Gamma \vdash^\Sigma N : B_1,\\ \Gamma \vdash^\Sigma M : B_2 \to A_2,\ \Gamma \vdash^\Sigma N : B_2.\end{array}$$

Then $\Gamma \vdash^\Sigma M : (B_1 \to A_1) \cap (B_2 \to A_2)$ and by rules (η), ($\to$-$\cap$):

$$(B_1 \to A_1) \cap (B_2 \to A_2) \leq_\Sigma B_1 \cap B_2 \to A_1 \cap A_2 \leq_\Sigma B_1 \cap B_2 \to A.$$

We are done, since $\Gamma \vdash^\Sigma N : B_1 \cap B_2$ by rule ($\cap$I) .

(3) The proof is very similar to the proof of Point (2). It is again by induction on derivations and again the only interesting case is when the last applied rule is ($\cap$I):

$$(\cap\text{I})\ \frac{\Gamma \vdash^\Sigma \lambda x.M : A_1 \quad \Gamma \vdash^\Sigma \lambda x.M : A_2}{\Gamma \vdash^\Sigma \lambda x.M : A_1 \cap A_2}.$$

By induction there are I, B_i, C_i, J, D_j, G_j such that

$$\begin{array}{l}\forall i \in I.\ \Gamma, x{:}B_i \vdash^\Sigma M : C_i, \forall j \in J.\ \Gamma, x{:}D_j \vdash^\Sigma M : G_j,\\ \bigcap_{i\in I}(B_i \to C_i) \leq_\Sigma A_1 \quad \& \quad \bigcap_{j\in J}(D_j \to G_j) \leq_\Sigma A_2.\end{array}$$

So we are done since $(\bigcap_{i\in I}(B_i \to C_i)) \cap (\bigcap_{j\in J}(D_j \to G_j)) \leq_\Sigma A$.

We are now in position for proving the main result of this subsection: in filter structures the interpretation of a term coincides with the set of types which are deducible for it.

Theorem 2. *Let $\langle \mathcal{F}^\Sigma, F^\Sigma, G^\Sigma\rangle$ be a filter structure. Then, for any λ-term M and environment ρ : Var $\to \mathcal{F}^\Sigma$,*

$$[\![M]\!]^\Sigma_\rho = \{A \in \mathbb{T}^\Sigma \mid \exists \Gamma \models \rho.\ \Gamma \vdash^\Sigma M : A\},$$

where $[\![\ ]\!]^\Sigma$ is the interpretation function $[\![\ ]\!]^{\mathcal{F}^\Sigma}$ and $\Gamma \models \rho$ iff $\rho(x : B) \in \Gamma$ implies $B \in \rho(x)$.

Proof. First notice that $\Gamma \models \rho$ and $\Gamma' \models \rho$ imply (by definitions of $\models$ and of filter) $\Gamma \uplus \Gamma' \models \rho$, where we use $\uplus$ to denote the union between bases defined by:

$$\begin{array}{rl}\Gamma_1 \uplus \Gamma_2 = & \{(x{:}\tau) \mid (x{:}\tau) \in \Gamma_1\ \ \& \ x \notin \Gamma_2\} \cup\\ & \{(x{:}\tau) \mid (x{:}\tau) \in \Gamma_2\ \ \& \ x \notin \Gamma_1\} \cup\\ & \{(x{:}\tau_1 \cap \tau_2) \mid (x{:}\tau_1) \in \Gamma_1\ \ \& \ (x{:}\tau_2) \in \Gamma_2\}.\end{array}$$

Moreover notice that by rules (W) and ($\leq$ L) if $\Gamma \vdash^{\Sigma} M : A$ then $\Gamma \uplus \Gamma' \vdash^{\Sigma} M : A$ for all Γ'. We can conclude that:

$$(\heartsuit) \quad \Gamma \models \rho, \Gamma' \models \rho, \text{ and } \Gamma \vdash^{\Sigma} M : A \text{ imply } \Gamma \uplus \Gamma' \models \rho \text{ and } \Gamma \uplus \Gamma' \vdash^{\Sigma} M : A.$$

We prove now the thesis by induction on M.
If $M \equiv x$, then

$$\begin{aligned}
[\![x]\!]^{\Sigma}_{\rho} &= \rho(x) \\
&= \{A \in \mathbb{T}^{\Sigma} \mid \exists B \in \rho(x).\ B \leq_{\Sigma} A\} \\
&= \{A \in \mathbb{T}^{\Sigma} \mid \exists B \in \rho(x).\ x : B \vdash^{\Sigma} x : A\} \text{ by Theorem 1(1)} \\
&= \{A \in \mathbb{T}^{\Sigma} \mid \exists \Gamma \models \rho.\ \Gamma \vdash^{\Sigma} x : A\}.
\end{aligned}$$

If $M \equiv NL$, then

$$\begin{aligned}
[\![NL]\!]^{\Sigma}_{\rho} &= [\![N]\!]^{\Sigma}_{\rho} \cdot [\![L]\!]^{\Sigma}_{\rho} \\
&= \uparrow \{C \in \mathbb{T}^{\Sigma} \mid \exists B \in [\![L]\!]^{\Sigma}_{\rho}. B \to C \in [\![N]\!]^{\Sigma}_{\rho}\} \\
&\quad \text{by definition of application} \\
&= \{A \in \mathbb{T}^{\Sigma} \mid \exists I, B_i, C_i.\ B_i \to C_i \in [\![N]\!]^{\Sigma}_{\rho}, B_i \in [\![L]\!]^{\Sigma}_{\rho}, \\
&\quad \textstyle\bigcap_{i\in I} C_i \leq_{\Sigma} A\} \\
&\quad \text{by definition of filter} \\
&= \{A \in \mathbb{T}^{\Sigma} \mid \exists \Gamma \models \rho, I, B_i, C_i.\ \Gamma \vdash^{\Sigma} N : B_i \to C_i, \\
&\quad \Gamma \vdash^{\Sigma} L : B_i, \textstyle\bigcap_{i\in I} C_i \leq_{\Sigma} A\} \\
&\quad \text{by induction and } (\heartsuit) \\
&= \{A \in \mathbb{T}^{\Sigma} \mid \exists \Gamma \models \rho.\ \Gamma \vdash^{\Sigma} NL : A\} \\
&\quad \text{by Theorem 1(2) and rule } (\leq).
\end{aligned}$$

If $M \equiv \lambda x.N$, then

$$\begin{aligned}
[\![\lambda x.N]\!]^{\Sigma}_{\rho} &= G^{\Sigma}(\boldsymbol{\lambda} X \in \mathcal{F}^{\Sigma}.[\![N]\!]^{\Sigma}_{\rho[x:=X]}) \\
&= \uparrow \{B \to C \in \mathbb{T}^{\Sigma} \mid C \in [\![N]\!]^{\Sigma}_{\rho[x:=\uparrow B]}\} \\
&\quad \text{by definition of } G^{\Sigma} \\
&= \{A \in \mathbb{T}^{\Sigma} \mid \exists \Gamma \models \rho, I, B_i, C_i.\ \Gamma, x : B_i \vdash^{\Sigma} N : C_i, \\
&\quad \textstyle\bigcap_{i\in I} (B_i \to C_i) \leq_{\Sigma} A\} \\
&\quad \text{by induction and } (\heartsuit) \\
&= \{A \in \mathbb{T}^{\Sigma} \mid \exists \Gamma \models \rho.\ \Gamma \vdash^{\Sigma} \lambda x.N : A\} \\
&\quad \text{by Theorem 1(3) and rule } (\leq).
\end{aligned}$$

4 Isomorphism results

In this section we will see that nitps are closely related to natural λ-structures. On one hand, any nitp induces a filter structure which is a natural λ-structure. On the other hand, for any natural λ-structure $\langle \mathcal{D}, F_{\mathcal{D}}, G_{\mathcal{D}} \rangle$, it is possible to find a presentation of it by means of a nitp Σ, i.e. $\langle \mathcal{F}^{\Sigma}, F^{\Sigma}, G^{\Sigma} \rangle$ and $\langle \mathcal{D}, F_{\mathcal{D}}, G_{\mathcal{D}} \rangle$ are isomorphic as natural λ-structures. This kind of presentation is not always given by means of a recursive set

of axioms and rules, but it will be so in the case of D_∞ λ-models as shown in the final section of the paper.

The correspondence between nitps and natural λ-structures can be refined in a categorical setting, showing that both natural λ-structures and nitps are objects of suitable categories, which turn out to be equivalent. In the present paper we give instead a direct proof.

We begin the present section by showing the first (easy) isomorphism result.

Theorem 3 (Isomorphism I). *Each $\langle \mathcal{F}^\Sigma, F^\Sigma, G^\Sigma \rangle$ is a natural λ-structure.*

Proof. We have just to prove that F^Σ and G^Σ set up a Galois connection, that is

$$\begin{array}{l} F^\Sigma \circ G^\Sigma \sqsupseteq Id_{[\mathcal{F}^\Sigma \to \mathcal{F}^\Sigma]} \\ G^\Sigma \circ F^\Sigma \sqsubseteq Id_{\mathcal{F}^\Sigma}. \end{array}$$

The first inequality is given by:

$$\begin{array}{rll} F^\Sigma(G^\Sigma(f)) = \bigsqcup\{\uparrow A \Rightarrow \uparrow B \mid A \to B \in G^\Sigma(f)\} & & \text{by Proposition 6(1)} \\ \sqsupseteq \bigsqcup\{\uparrow A \Rightarrow \uparrow B \mid B \in f(\uparrow A)\} & & \text{by definition of } G^\Sigma \\ = f & & \text{by Proposition 1(2).} \end{array}$$

For the second inequality we get

$$\begin{array}{rll} G^\Sigma(F^\Sigma(X)) = \uparrow \{A \to B \mid B \in F^\Sigma(X)(\uparrow A)\} & & \text{by definition of } G^\Sigma \\ = \uparrow \{A \to B \mid A \to B \in X\} & & \text{by Proposition 5(2)} \\ \subseteq X. & & \end{array}$$

In the remaining of the present subsection we will prove the vice versa, i.e. that each natural λ-structure can be generated by a suitable nitp.

To each λ-structure $\langle \mathcal{D}, F_\mathcal{D}, G_\mathcal{D} \rangle$ we associate a nitp $\Sigma^{\mathbb{D}}$. The preorder relation on types takes into account both the partial order between compact elements of $\mathcal{D}$ and the mapping $G_\mathcal{D}$.

Definition 10. *Let $\langle \mathcal{D}, F_\mathcal{D}, G_\mathcal{D} \rangle$ be a λ-structure. We define:*

1. $\mathbb{C}^{\mathbb{D}} = \{\psi_c \mid c \in \mathcal{K}(\mathcal{D})\}$, *where $\psi_\perp$ is Ω and ψ_c is a fresh constant for each other $c \in \mathcal{K}(\mathcal{D})$;*
2. $\leq_{\mathbb{D}} \subseteq \mathbb{T}^{\mathbb{D}} \times \mathbb{T}^{\mathbb{D}}$ *as the preorder relation generated by adding to ∇^0:*

$$\begin{array}{l} \mathbb{D} = \{\psi_c \leq_{\mathbb{D}} \psi_d \mid d \sqsubseteq c\} \cup \{\psi_c \cap \psi_d \leq_{\mathbb{D}} \psi_e \mid e = c \sqcup d\} \\ \qquad \cup \{\psi_c \to \psi_d \sim_{\mathbb{D}} \psi_e \mid e = G_\mathcal{D}(c \Rightarrow d)\} \end{array}$$

   where $\psi_c, \psi_d, \psi_e \in \mathbb{C}^{\mathbb{D}}$;
3. $\Sigma^{\mathbb{D}} = \langle \mathbb{C}^{\mathbb{D}}, \leq_{\mathbb{D}} \rangle$.

The nitp $\Sigma^{\mathbb{D}}$ enjoys some useful properties.

Proposition 7. *1. For all $A \in \mathbb{T}^{\mathbb{D}}$ there is $c \in \mathcal{K}(\mathcal{D})$ such that $A \sim_{\mathbb{D}} \psi_c$.*
2. For all $\psi_c, \psi_d \in \mathbb{C}^{\mathbb{D}}$: $\psi_c \leq_{\mathbb{D}} \psi_d$ iff $d \sqsubseteq c$;

3. *For all* $\psi_c, \psi_d, \psi_e \in \mathbb{CC}^{\mathbb{ID}}$*:* $\psi_e \leq_{\mathbb{ID}} \psi_c \to \psi_d$ *iff* $G_{\mathcal{D}}(c \Rightarrow d) \sqsubseteq e$.

Proof. (1) By induction on A. Let $B \sim_{\mathbb{ID}} \psi_b$ and $C \sim_{\mathbb{ID}} \psi_c$.
If $A \equiv B \cap C$ then $A \sim_{\mathbb{ID}} \psi_{b \sqcup c}$.
If $A \equiv B \to C$ then $A \sim_{\mathbb{ID}} \psi_d$ where $d = G_{\mathcal{D}}(b \Rightarrow c)$.
For (2) define $\mathsf{pp} : \mathbb{TT}^{\mathbb{ID}} \to \mathcal{K}(\mathcal{D})$ by:

$$\begin{array}{ll} \mathsf{pp}(\psi_c) & = c; \\ \mathsf{pp}(A \cap B) & = \mathsf{pp}(A) \sqcup \mathsf{pp}(B); \\ \mathsf{pp}(A \to B) & = G_{\mathcal{D}}(\mathsf{pp}(A) \Rightarrow \mathsf{pp}(B)). \end{array}$$

It is easy to verify by induction on $\leq_{\mathbb{ID}}$ that $A \leq_{\mathbb{ID}} B$ implies $\mathsf{pp}(B) \sqsubseteq \mathsf{pp}(A)$. This yields $\psi_c \leq_{\mathbb{ID}} \psi_d \Rightarrow d \sqsubseteq c$. The other implication is immediate by definition of $\mathbb{ID}$.

(3) follows from (2) since $\psi_c \to \psi_d \sim_{\mathbb{ID}} \psi_{G_{\mathcal{D}}(c \Rightarrow d)}$.

Notice that the first two points of the above proposition imply that for each type A in $\mathbb{TT}^{\mathbb{ID}}$ there is exactly one compact element c in $\mathcal{D}$ such that $A \sim_{\mathbb{ID}} \psi_c$.

We define now a lattice isomorphism between the set $\mathcal{F}^{\mathbb{ID}}$ of $\mathbb{ID}$-filters over $\mathbb{TT}^{\mathbb{ID}}$ and $\mathcal{D}$.

Definition 11. *The mapping* $\mathsf{m} : \mathcal{F}^{\mathbb{ID}} \to \mathcal{D}$ *is defined by*

$$\mathsf{m}(X) = \bigsqcup_{\psi_c \in X} c.$$

It is not difficult to verify that $\mathsf{m}(\uparrow \psi_c) = c$ and that m is a lattice isomorphism between $\mathcal{F}^{\mathbb{ID}}$ and $\mathcal{D}$.

We show that m commutes with application.

Lemma 2. $\mathsf{m}(X \cdot Y) = \mathsf{m}(X) \cdot \mathsf{m}(Y)$.

Proof. By the continuity of m and of application we need to consider only finite elements in $\mathcal{F}^{\mathbb{ID}}$, i.e. using also Proposition 7(1) we only need to show:

$$\mathsf{m}(\uparrow \psi_c \cdot \uparrow \psi_d) = \mathsf{m}(\uparrow \psi_c) \cdot \mathsf{m}(\uparrow \psi_d).$$

First notice that

$$\begin{array}{ll} \psi_c \leq_{\mathbb{ID}} \psi_d \to \psi_b \Leftrightarrow G_{\mathcal{D}}(d \Rightarrow b) \sqsubseteq c & \text{by Proposition 7(3)} \\ \Leftrightarrow d \Rightarrow b \sqsubseteq F_{\mathcal{D}}(c) & \text{by condition (1) of Definition 2} \\ \Leftrightarrow b \sqsubseteq F_{\mathcal{D}}(c)(d) & \text{by definition of step function} \\ \Leftrightarrow b \sqsubseteq c \cdot d & \text{by definition of application.} \end{array}$$

We get (using three times rule (η))

$$
\begin{aligned}
\mathsf{m}(\uparrow \psi_c \cdot \uparrow \psi_d) &= \mathsf{m}(\uparrow \{A \mid \psi_c \leq_{\mathbb{D}} \psi_d \to A\}) \\
&\quad \text{by definition of application} \\
&= \mathsf{m}(\uparrow \{\psi_b \mid b \in \mathcal{K}(\mathcal{D}) \text{ and } \psi_c \leq_{\mathbb{D}} \psi_d \to \psi_b\}) \\
&\quad \text{by Proposition 7(1)} \\
&= \bigsqcup\{b \in \mathcal{K}(\mathcal{D}) \mid \psi_c \leq_{\mathbb{D}} \psi_d \to \psi_b\} \\
&\quad \text{by definition of } \mathsf{m} \\
&= \bigsqcup\{b \in \mathcal{K}(\mathcal{D}) \mid b \sqsubseteq c \cdot d\} \\
&\quad \text{by above} \\
&= c \cdot d \\
&= \mathsf{m}(\uparrow \psi_c) \cdot \mathsf{m}(\uparrow \psi_d).
\end{aligned}
$$

Finally we can give the second isomorphism result, whose proof follows immediately from the previous lemma and Proposition 3.

Theorem 4 (Isomorphism II). *Let $\langle \mathcal{D}, F_{\mathcal{D}}, G_{\mathcal{D}} \rangle$ be a natural λ-structure, then the nitp $\Sigma^{\mathbb{D}}$ of Definition 10 is such that $\langle \mathcal{D}, F_{\mathcal{D}}, G_{\mathcal{D}} \rangle$ and $\langle \mathcal{F}^{\mathbb{D}}, F^{\mathbb{D}}, G^{\mathbb{D}} \rangle$ are isomorphic.*

5 D_∞-λ-models and filter λ-models

Since all ω-algebraic complete lattices which are extensional λ-models are clearly natural λ-structures, Theorem 4 implies that any such λ-model is isomorphic to a filter λ-model. However the finitary logical description provided by the proof of Theorem 4 is rather opaque. In this section we show that in the special case of D_∞ inverse limit λ-models, one can obtain far more concise type theoretic descriptions. Remarkably the nitp which induces a filter λ-model isomorphic to D_∞, starting from D_0, is exactly the nitp *freely generated* by a nitp which induces D_0 together with the equalities which arise from encoding the initial projections.

First of all we fix some notations and recall the standard D_∞ construction.

Definition 12. *1. Let D_0 be an ω-algebraic complete lattice and*

$$\langle \mathsf{i}_0, \mathsf{j}_0 \rangle : D_0 \to [D_0 \to D_0]$$

be an embedding-projection *pair, i.e. $\mathsf{i}_0 : D_0 \to [D_0 \to D_0]$ and $\mathsf{j}_0 : [D_0 \to D_0] \to D_0$ satisfy $\mathsf{i}_0 \circ \mathsf{j}_0 \sqsubseteq Id_{[D_0 \to D_0]}$ and $\mathsf{j}_0 \circ \mathsf{i}_0 = Id_{D_0}$.*

2. Define a tower *$\langle \mathsf{i}_n, \mathsf{j}_n \rangle : D_n \to D_{n+1}$ in the following way:*

- *$D_{n+1} = [D_n \to D_n]$;*
- *$\mathsf{i}_n(f) = \mathsf{i}_{n-1} \circ f \circ \mathsf{j}_{n-1}$ for any $f \in D_n$;*
- *$\mathsf{j}_n(g) = \mathsf{j}_{n-1} \circ g \circ \mathsf{i}_{n-1}$ for any $g \in D_{n+1}$.*

3. The set D_∞ is defined by

$$D_\infty = \{\langle d_n \rangle \mid \forall n.\ d_n \in D_n \text{ and } \mathsf{j}_n(d_{n+1}) = d_n\},$$

where $\langle d_n \rangle$ is short for $\langle d_n \rangle_{n \in \mathbb{N}}$.

4. The ordering on D_∞ is given by

$$\langle d_n\rangle \sqsubseteq \langle e_n\rangle \Leftrightarrow \forall k.\ d_k \sqsubseteq e_k.$$

5. *Let $\langle \Phi_{m\infty}, \Phi_{\infty m}\rangle$ denotes the standard embedding-projection pair from D_m to D_∞: for any $d \in D_m$, $\langle d_n\rangle \in D_\infty$,*
   $\Phi_{m\infty}(d) = \langle \ldots \mathsf{j}_{m-2}(\mathsf{j}_{m-1}(d)), \mathsf{j}_{m-1}(d), d, \mathsf{i}_m(d), \mathsf{i}_{m+1}(\mathsf{i}_m(d)) \ldots\rangle$,
   $\Phi_{\infty m}(\langle d_n\rangle) = d_m$.
6. *Let $\Phi_{mn} : D_m \to D_n$ be $\Phi_{\infty n} \circ \Phi_{m\infty}$.*
7. *Let $F_\infty : D_\infty \to [D_\infty \to D_\infty]$ be defined by*

$$F_\infty(\langle d_n\rangle)(\langle e_n\rangle) = \bigsqcup_{n\in\mathbf{N}} \Phi_{n\infty}(d_{n+1}(e_n)),$$

and $G_\infty : [D_\infty \to D_\infty] \to D_\infty$ by

$$G_\infty(f) = \bigsqcup_{n\in\mathbf{N}} \Phi_{(n+1)\infty}(\Phi_{\infty n} \circ f \circ \Phi_{n\infty}).$$

Remark 1. From previous definition it follows easily that, if $n \le p \le k$ and $d \in D_n$, $e \in D_p$, then $\Phi_{np}(d) \sqsubseteq e$ iff $\Phi_{nk}(d) \sqsubseteq \Phi_{pk}(e)$ iff $\Phi_{n\infty}(d) \sqsubseteq \Phi_{p\infty}(e)$.

Theorem 5. *([Sco72]) $\langle D_\infty, F_\infty, G_\infty\rangle$ is a λ-model.*

Next definition exhibits nitps which induce filter λ-models isomorphic to D_∞ λ-models. Notice the similarities with Definition 10. In particular, the equivalences between arrow types and constants are built in both cases by considering the action of the compact element preserving map ($G_{\mathcal{D}}$ in the case of Definition 10, i_0 here). A difference with respect to Definition 10 is that we are forced to define such equivalences by means of intersections and sups. The reason for this is that we do not have a constant for each compact function, which could lead to an apparently smoother definition such as in the case of Definition 10 (which actually yields a lot of redundant types), but rather we represent a compact function as a sup of suitable step functions. Dually, in the nitp, the compact function will be represented by the intersection of the arrow types which correspond to the involved step functions.

Definition 13. *Define:*

1. *$\mathbb{C}^\infty = \{\psi_c \mid c \in \mathcal{K}(D_0)\}$, where $\psi_\perp$ is Ω and ψ_c is a fresh constant for each other $c \in \mathcal{K}(D_0)$;*
2. *$\le_\infty$ as the preorder relation generated by adding to ∇^0:*

$$\begin{aligned}\infty = \{&\psi_c \le_\infty \psi_d \mid d \sqsubseteq c\} \cup \{\psi_c \cap \psi_d \sim_\infty \psi_e \mid e = c \sqcup d\}\\ &\cup\{\textstyle\bigcap_{j\in J}(\psi_{c_j} \to \psi_{d_j}) \sim_\infty \psi_d \mid \mathsf{i}_0(d) = \bigsqcup_{j\in J}(c_j \Rightarrow d_j)\}\end{aligned}$$

   where $\psi_c, \psi_d, \psi_e, c_j, d_j \in \mathbb{C}^\infty$;
3. *$\Sigma^\infty = \langle \mathbb{C}^\infty, \le_\infty\rangle$.*

The nitp Σ^∞ enjoys some useful properties.

Lemma 3. *1.* $\bigcap_{i\in I} \psi_{c_i} \sim_\infty \psi_{\bigsqcup_{i\in I} c_i}$.

2. $\bigcap_{i\in I}(C_i \to D_i) \leq_\infty A \to B$ *implies* $\bigcap_{i\in J} D_i \leq_\infty B$ *where* $J = \{i \in I \mid A \leq_\infty C_i\}$.

3. $\uparrow \bigcap_{i\in I}(C_i \to D_i)\cdot \uparrow A = \uparrow \bigcap_{i\in J} D_i$ *where* $J = \{i \in I \mid A \leq_\infty C_i\}$.

Proof. (1) follows easily from Definition 13.

For (2) notice that by definition for each constant $\alpha \in \mathbb{C}^\infty$ there is exactly one judgement of the shape $\alpha \sim_\infty \bigcap_{l\in L^{(\psi_d)}}(\gamma_l^{(\alpha)} \to \delta_l^{(\alpha)})$, where $\gamma_l^{(\alpha)}, \delta_l^{(\alpha)} \in \mathbb{C}^\infty$.

We can prove by simultaneous induction on the definition of $\leq_\infty$ two statements, the first of which implies the thesis.

- if $(\bigcap_{i\in I}(A_i \to B_i)) \cap (\bigcap_{h\in H} \alpha_h) \leq_\infty (\bigcap_{j\in J}(C_j \to D_j)) \cap (\bigcap_{k\in K} \beta_k)$, then for each $j \in J$: $(\bigcap_{i\in I'} B_i) \cap (\bigcap_{h\in H'}(\bigcap_{l\in L^{(\alpha_h)'}} \delta_l^{(\alpha_h)})) \leq_\infty D_j$ where $I' = \{i \in I \mid C_j \leq_\infty A_i\}$, $H' = \{h \in H \mid \exists l \in L^{(\alpha_h)} C_j \leq_\infty \gamma_l^{(\alpha_h)}\}$, and ${L^{(\alpha_h)}}' = \{l \in L^{(\alpha_h)} \mid C_j \leq_\infty \gamma_l^{(\alpha_h)}\}$;
- if $(\bigcap_{i\in I}(A_i \to B_i)) \cap (\bigcap_{h\in H} \alpha_h) \leq_\infty (\bigcap_{j\in J}(C_j \to D_j)) \cap (\bigcap_{k\in K} \beta_k)$, then for each $k \in K, m \in L^{(\beta_k)}$ $(\bigcap_{i\in I'} B_i) \cap (\bigcap_{h\in H'}(\bigcap_{l\in L^{(\alpha_h)'}} \delta_l^{(\alpha_h)})) \leq_\infty \delta_m^{(\beta_k)}$ where $I' = \{i \in I \mid \gamma_m^{(\beta_k)} \leq_\infty A_i\}$, $H' = \{h \in H \mid \exists l \in L^{(\alpha_h)} \gamma_m^{(\beta_k)} \leq_\infty \gamma_l^{(\alpha_h)}\}$, and ${L^{(\alpha_h)}}' = \{l \in L^{(\alpha_h)} \mid \gamma_m^{(\beta_k)} \leq_\infty \gamma_l^{(\alpha_h)}\}$.

For (3) the inclusion $\subseteq$ follows immediately from the definition of filter application. We show the reverse inclusion.

$$
\begin{aligned}
B \in \uparrow \textstyle\bigcap_{i\in I}(C_i \to D_i)\cdot \uparrow A \Rightarrow\ & A \to B \in \uparrow \textstyle\bigcap_{i\in I}(C_i \to D_i) \\
& \text{by Proposition 5(2)} \\
\Rightarrow\ & \textstyle\bigcap_{i\in I}(C_i \to D_i) \leq_\infty A \to B \\
& \text{by definition of filter} \\
\Rightarrow\ & \textstyle\bigcap_{i\in J} D_i \leq_\infty B \text{ where } J = \{i \in I \mid A \leq_\infty C_i\} \\
& \text{by (2).}
\end{aligned}
$$

The proof of the isomorphism will be postponed because several preliminary results are needed. These are the subjects of Lemmata 4, 5 and 6.

First we classify the types in $\mathbb{T}^\infty$ according to the maximal number of nested arrow occurrences they may contain.

Definition 14. 1. *We define the map* rank *rk* : $\mathbb{T}^\infty \to \mathbb{N}$ *by:*

$$
\begin{aligned}
rk(\psi_c) &= 0; \\
rk(A \to B) &= max\{rk(A), rk(B)\} + 1; \\
rk(A \cap B) &= max\{rk(A), rk(B)\}.
\end{aligned}
$$

2. *Let* $\mathbb{T}_n^\infty = \{A \in \mathbb{T}^\infty \mid rk(A) \leq n\}$.

We can associate to each type in $\mathbb{T}_n^\infty$ an element in D_n: this will be crucial for defining the mapping which gives the desidered isomorphism (see Definition 16).

Definition 15. *We define, for each $n \in \mathbf{N}$, a map $w_n : \mathbb{T}_n^\infty \to D_n$ by a double induction on n and on the construction of types in $\mathbb{T}^\infty$:*

$$\begin{aligned} w_n(\psi_c) &= \Phi_{0n}(c); \\ w_n(A \cap B) &= w_n(A) \sqcup w_n(B); \\ w_n(A \to B) &= w_{n-1}(A) \Rightarrow w_{n-1}(B). \end{aligned}$$

The following property of w_n shows that no information is lost if we map a type into any D_n with n greater than the rank of the type.

Lemma 4. *For all $A \in \mathbb{T}_n^\infty$ and for all $m, p \geq n$ we have $\Phi_{m\infty}(w_m(A)) = \Phi_{p\infty}(w_p(A))$.*

Proof. We show by induction on the definition of w_n that $w_{n+1}(A) = \mathsf{i}_n(w_n(A))$. Then the desired equality follows from the definition of the function Φ. The only interesting case is when $A \equiv B \to C$. We get

$$\begin{aligned} w_{n+1}(B \to C) &= w_n(B) \Rightarrow w_n(C) && \text{by definition} \\ &= \mathsf{i}_{n-1}(w_{n-1}(B)) \Rightarrow \mathsf{i}_{n-1}(w_{n-1}(C)) && \text{by induction} \\ &= \mathsf{i}_n(w_{n-1}(B) \Rightarrow w_{n-1}(C)) && \text{by definition of } \mathsf{i}_n \\ & && \text{and of step function} \\ &= \mathsf{i}_n(w_n(B \to C)) && \text{by Definition 15.} \end{aligned}$$

The maps w_n reverse the order between types, as shown in the following lemma.

Lemma 5. *Let $n \geq rk(A \cap B)$. Then $A \leq_\infty B$ implies $w_n(B) \sqsubseteq w_n(A)$.*

Proof. The proof is by induction on the definition of $\leq_\infty$. We consider just the case of rule (η). Let $A \equiv C \to D$, $B \equiv E \to F$, with $E \leq_\infty C$, $D \leq_\infty F$. Then by induction $w_n(C) \sqsubseteq w_n(E)$ and $w_n(F) \sqsubseteq w_n(D)$, hence $w_n(E) \Rightarrow w_n(F) \sqsubseteq w_n(C) \Rightarrow w_n(D)$. Thus we get, by definition of w_n, $w_{n+1}(B) \sqsubseteq w_{n+1}(A)$, hence, by Lemma 4, $\mathsf{i}_n(w_n(B)) \sqsubseteq \mathsf{i}_n(w_n(A))$. By Remark 1 (since $\mathsf{i}_n = \Phi_{n(n+1)}$) the thesis follows.

Also the reverse implication of Lemma 5 holds.

Lemma 6. *Let $rk(A \cap B) \leq n$. Then $w_n(B) \sqsubseteq w_n(A)$ implies $A \leq_\infty B$.*

Proof. The proof is by induction on $rk(A \cap B)$.
If $rk(A \cap B) = 0$ we have $A \equiv \bigcap_{i \in I} \psi_{c_i}$, $B = \bigcap_{j \in J} \psi_{d_j}$. Then $w_n(B) \sqsubseteq w_n(A)$ implies $\bigsqcup_{j \in J} \Phi_{0n}(d_j) \sqsubseteq \bigsqcup_{i \in I} \Phi_{0n}(c_i)$, that is, by Remark 1, $\bigsqcup_{j \in J} d_j \sqsubseteq \bigsqcup_{i \in I} c_i$. By Definition 13 and Lemma 3(1) it follows $\bigcap_{i \in I} \psi_{c_i} \leq_\infty \bigcap_{j \in J} \psi_{d_j}$, hence $A \leq_\infty B$.
Otherwise, let

$$A \equiv (\bigcap_{i \in I} \psi_{c_i}) \cap (\bigcap_{l \in L} (C_l \to D_l)), B \equiv (\bigcap_{h \in H} \psi_{d_h}) \cap (\bigcap_{m \in M} (E_m \to F_m))$$

where $\psi_{c_i} \sim_\infty \bigcap_{j \in J_i} (\psi_{a_j} \to \psi_{b_j})$, $\psi_{d_h} \sim_\infty \bigcap_{k \in K_h} (\psi_{e_k} \to \psi_{f_k})$. The last two equivalences imply by Lemma 5 that for all $n \geq 1$

$$w_n(\psi_{c_i}) = w_n(\bigsqcup_{j \in J_i} (\psi_{a_j} \Rightarrow \psi_{b_j})), w_n(\psi_{d_h}) = w_n(\bigsqcup_{k \in K_h} (\psi_{e_k} \Rightarrow \psi_{f_k})).$$

So we get

$$\bigsqcup_{h\in H}(\bigsqcup_{k\in K_h} w_n(\psi_{e_k}) \Rightarrow w_n(\psi_{f_k})) \sqcup (\bigsqcup_{m\in M} w_n(E_m) \Rightarrow w_n(F_m)) \sqsubseteq \bigsqcup_{i\in I}(\bigsqcup_{j\in J_i} w_n(\psi_{a_j}) \Rightarrow w_n(\psi_{b_j})) \sqcup (\bigsqcup_{l\in L} w_n(C_l) \Rightarrow w_n(D_l)).$$

Now by definition of step function this implies that for each $h \in H$, $k \in K_h$,

$$w_n(\psi_{f_k}) \sqsubseteq \bigsqcup_{i\in I'}(\bigsqcup_{j\in J'_i} w_n(\psi_{b_j})) \sqcup (\bigsqcup_{l\in L'} w_n(D_l))$$

where $I' = \{i \in I \mid \exists j \in J_i \ \& \ w_n(\psi_{a_j}) \sqsubseteq w_n(\psi_{e_k})\}$, $J'_i = \{j \in J_i \mid w_n(\psi_{a_j}) \sqsubseteq w_n(\psi_{e_k})\}$, $L' = \{l \in L \mid w_n(C_l) \sqsubseteq w_n(\psi_{e_k})\}$.
Since all types involved in the two above judgments have ranks strictly less than $rk(A \cap B)$, by induction and by Lemma 3 we obtain

$$\psi_{e_k} \leq_\infty \bigcap_{i\in I'}(\bigcap_{j\in J'_i} \psi_{a_j}) \cap \bigcap_{l\in L'} C_l,$$
$$\bigcap_{i\in I'}(\bigcap_{j\in J'_i} \psi_{b_j}) \cap \bigcap_{l\in L'} D_l \leq_\infty \psi_{f_k}.$$

Therefore we have $A \leq_\infty \psi_{e_k} \to \psi_{f_k}$ for each $h \in H$, $k \in K_h$. In a similar way we can prove that $A \leq_\infty E_m \to F_m$, for any $m \in M$. Putting together these results we get $A \leq_\infty B$.

We can now prove the isomorphism between $\langle \mathcal{D}_\infty, F_\infty, G_\infty\rangle$ and $\langle \mathcal{F}^\infty, F^\infty, G^\infty\rangle$. First we give the isomorphism map.

Definition 16. *Let $\hat{\mathsf{m}}$ be the unique continuous extension of the mapping $\mathsf{m} : \mathcal{K}(\mathcal{F}^\infty) \to \mathcal{K}(D_\infty)$ defined by*

$$\mathsf{m}(\uparrow A) = \Phi_{r\infty}(w_r(A)),$$

where $r = rk(A)$.

Notice that by Lemma 4 we have $\mathsf{m}(\uparrow A) = \Phi_{n\infty}(w_n(A))$ for all $n \geq rk(A)$. This will be freely used in the proof of Theorem 6.

We recall that (see [Sco72])

1. $F_\infty \circ G_\infty = Id_{[\mathcal{D}_\infty \to \mathcal{D}_\infty]}$;
2. $G_\infty \circ F_\infty = Id_{\mathcal{D}_\infty}$.

On the other hand, $\langle \mathcal{F}^\infty, F^\infty, G^\infty\rangle$ is a natural λ-structure. So both $\langle \mathcal{D}_\infty, F_\infty, G_\infty\rangle$ and $\langle \mathcal{F}^\infty, F^\infty, G^\infty\rangle$ are natural λ-structures.

Theorem 6. *The natural λ-structures $\langle \mathcal{D}_\infty, F_\infty, G_\infty\rangle$ and $\langle \mathcal{F}^\infty, F^\infty, G^\infty\rangle$ are isomorphic.*

Proof. The mapping m is monotone and injective by Lemmas 5 and 6, hence $\hat{\mathsf{m}}$ is so. We prove surjection over D_∞ by induction on n, by showing that each w_n is surjective on D_n. Surjection of w_0 is obvious by definition of w_0 and of the nitp Σ^∞.

Let $f \in D_{n+1}$. By Proposition 1(2) there exist I and $a_i, b_i \in D_n$ such that $f = \bigsqcup_{i\in I}(a_i \Rightarrow b_i)$. By induction there exist types A_i, B_i such that for all $i \in I$, $w_n(A_i) = a_i$ and $w_n(B_i) = b_i$. Therefore

$$\begin{aligned} w_{n+1}(\textstyle\bigcap_{i\in I}(A_i \to B_i)) &= \textstyle\bigsqcup_{i\in I}(w_n(A_i) \Rightarrow w_n(B_i)) \\ &= \textstyle\bigsqcup_{i\in I}(a_i \Rightarrow b_i) \\ &= f \end{aligned}$$

We have so proved that each w_n is surjective. This implies m is surjective onto compact elements of D_∞, hence $\hat{\mathsf{m}} : \mathcal{F}^\infty \to D_\infty$ is surjective.

From Lemma 6 it follows that m^{-1} is monotone, hence $\hat{\mathsf{m}}^{-1}$ is continuous (by definition). We have finally to prove that $\hat{\mathsf{m}}$ commutes with application. Since it is enough to prove the thesis on compact elements, that is on principal filters of $\mathcal{F}^\infty$, we are left to prove that for any $A, B \in \mathbb{T}^\infty$

$$\mathsf{m}(\uparrow A\cdot \uparrow B) = \mathsf{m}(\uparrow A) \cdot \mathsf{m}(\uparrow B).$$

Let $A, B \in \mathbb{T}^\infty$, $A \sim_\infty \bigcap_{i\in I}(C_i \to D_i)$, $J = \{i \in I \mid B \leq_\infty C_i\}$, and n any natural number greater than $rk(A \cap B)$. Then by the definition of $\Phi_{n\infty}$ and Lemmas 5, 6 we get

$$(\flat) J = \{i \in I \mid \Phi_{n\infty}(w_n(C_i)) \sqsubseteq \Phi_{n\infty}(w_n(B))\}.$$

$$\begin{aligned} \mathsf{m}(\uparrow A\cdot \uparrow B) &= \mathsf{m}(\uparrow \textstyle\bigcap_{i\in J} D_i) \text{ by Lemma 3(3)} \\ &= \Phi_{n\infty}(w_n(\textstyle\bigcap_{i\in J} D_i)) \text{ by definition of } \mathsf{m} \\ &= \textstyle\bigsqcup_{i\in J} \Phi_{n\infty}(w_n(D_i)) \\ &\quad \text{by definition of } w_n \text{ and addittivity of } \Phi_{n\infty} \\ &= \textstyle\bigsqcup_{i\in I}(\Phi_{n\infty}(w_n(C_i)) \Rightarrow \Phi_{n\infty}(w_n(D_i))) \cdot \Phi_{n\infty}(w_n(B)) \\ &\quad \text{by definition of step function and } (\flat) \\ &= \textstyle\bigsqcup_{i\in I} \Phi_{(n+1)\infty}(w_n(C_i) \Rightarrow w_n(D_i)) \cdot \Phi_{n\infty}(w_n(B)) \\ &\quad \text{by definition of } \Phi_{n\infty} \\ &= \Phi_{(n+1)\infty}(w_{n+1}(\textstyle\bigcap_{i\in I}(C_i \to D_i))) \cdot \Phi_{n\infty}(w_n(B)) \\ &\quad \text{by definition of } w_n \\ &= \mathsf{m}(\uparrow A) \cdot \mathsf{m}(\uparrow B) \text{ by definition of } \mathsf{m}. \end{aligned}$$

This completes the proof that $\langle \mathcal{D}_\infty, F_\infty, G_\infty\rangle$ and $\langle \mathcal{F}^\infty, F^\infty, G^\infty\rangle$ are isomorphic as natural λ-structures, hence as λ-models.

(ω-*Scott*) $\Omega \to \omega \sim \omega$	(ω-*Park*) $\omega \to \omega \sim \omega$
$(\omega\varphi)$ $\omega \leq_\Sigma \varphi$	$(\varphi \to \omega)$ $\varphi \to \omega \sim \omega$
$(\omega \to \varphi)$ $\omega \to \varphi \sim \varphi$	(I) $(\varphi \to \varphi) \cap (\omega \to \omega) \sim \varphi$

Fig. 1. Possible Axioms and Rules concerning $\leq_\Sigma$.

$\mathfrak{C}^{\mathcal{S}c}$	$= \{\Omega, \omega\}$	$\mathcal{S}c$	$= \{(\omega\text{-}Scott)\}$
$\mathfrak{C}^{\mathcal{P}a}$	$= \{\Omega, \omega\}$	$\mathcal{P}a$	$= \{(\omega\text{-}Park)\}$
$\mathfrak{C}^{\mathcal{CDZ}}$	$= \{\Omega, \varphi, \omega\}$	$\mathcal{CDZ}$	$= \{(\omega\varphi), (\varphi \to \omega), (\omega \to \varphi)\}$
$\mathfrak{C}^{\mathcal{HR}}$	$= \{\Omega, \varphi, \omega\}$	$\mathcal{HR}$	$= \{(\omega\varphi), (I), (\omega \to \varphi)\}$

Fig. 2. Type Theories: constants, axioms and rules.

Figure 1 lists axioms and rules used in Figure 2 to define nitps which induce filter λ-models isomorphic to well known inverse limit λ-models. We shall denote such theories as Σ^{∇}, with various different names ∇ corresponding to the initials of the authors who have first considered the λ-model induced by such a theory. For each such Σ^{∇} we specify in Figure 2 the nitp $\Sigma^{\nabla} = (\mathfrak{C}, \leq_{\nabla})$ by giving the set of constants $\mathfrak{C}^{\nabla}$ and the set ∇ of extra axioms and rules.

As particular cases of Theorem 6 we get that Scott λ-model as defined in [Sco72] is isomorphic to the filter λ-model induced by the nitp $\Sigma^{\mathcal{S}c}$ and Park λ-model as defined in [Par76] is isomorphic to the filter λ-model induced by the nitp $\Sigma^{\mathcal{P}a}$.

The construction of Theorem 6 was first discussed in [CDCHL84]. Other relevant references are [CDCZ87], which presents the filter λ-model induced by the nitp $\Sigma^{\mathcal{CDZ}}$, [HRDR92], where the filter λ-models induced by the nitps $\Sigma^{\mathcal{P}a}, \Sigma^{\mathcal{HR}}$ and other λ-models are considered, and [Ale91], [DGH93], [Plo93], where the relation between λ-structures and nitps is studied.

Results similar to Theorem 6 can be given also for other, non-extensional, inverse limit λ-models, obtained as solutions of domain equations involving also other functors. For instance one can consider the *lifted space of functions* $[\ \to\ ]_{\perp}$, the space of *strict functions* $[\ \to_{\perp}\ ]$, a *product* $[\ \to\ ] \times A$, or a *sum* $[\ \to\ ] + A$ with a set A of *atoms*, and so on. In all such cases one gets concise type theoretic descriptions of the λ-models obtained as fixed points of such functors corresponding to suitable choices of G [CDL83]. At least the following result is worthwhile mentioning in this respect, see [CDCHL84] for a proof. We define [BCDC83]

$$\mathfrak{C}^{\mathcal{BCD}} = \{\Omega\} \cup \mathfrak{C}_{\infty} \qquad \mathcal{BCD} = \{(\Omega\text{-}\eta)\}$$

where $\mathfrak{C}_{\infty}$ is an infinite set of fresh (i.e. different from Ω, ϕ, ω) constants.

Proposition 8. *The filter λ-model induced by $\Sigma^{\mathcal{BCD}}$ is isomorphic to $\langle \mathcal{D}, F, G \rangle$, where $\mathcal{D}$ is the initial solution of the domain equation $[\mathcal{D} \to \mathcal{D}] \times \mathsf{P}(\mathfrak{C}_{\infty}) \equiv \mathcal{D}$, the pair $\langle F, G \rangle$ set up a Galois connection and G is the map which picks always the minimal element in the extensionality classes of all functions.*

References

[Abr91] Samson Abramsky. Domain theory in logical form. *Ann. Pure Appl. Logic*, 51(1-2):1–77, 1991.

[ADCH03] Fabio Alessi, Mariangiola Dezani-Ciancaglini, and Furio Honsell. A complete characterization of complete intersection-type preorders. *ACM TOCL*, 4(1):120–147, 2003.

[Ale91] Fabio Alessi. *Strutture di tipi, teoria dei domini e modelli del lambda calcolo*. PhD thesis, Torino University, 1991.

[BCDC83] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48(4):931–940 (1984), 1983.

[CDCHL84] Mario Coppo, Mariangiola Dezani-Ciancaglini, Furio Honsell, and Giuseppe Longo. Extended type structures and filter lambda models. In G.Lolli, G.Longo, and A.Marcja, editors, *Logic Colloquium '82*, pages 241–262, Amsterdam, 1984. North-Holland.

[CDCZ87] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Maddalena Zacchi. Type theories, normal forms, and D_∞-lambda-models. *Inform. and Comput.*, 72(2):85–116, 1987.

[CDL83] Mario Coppo, Mariangiola Dezani, and Giuseppe Longo. Applicative information systems. In G.Ausiello and M.Protasi, editors, *CAAP'83, Trees in Algebra and Programming*, pages 35–64. Springer-Verlag, Berlin, 1983.

[DGH93] Pietro Di Gianantonio and Furio Honsell. An abstract notion of application. In Marc Bezem and Jan F. Groote, editors, *TLCA'93, Typed lambda calculi and applications*, number 664 in LNCS, pages 124–138. Springer-Verlag, 1993.

[GHK$^+$80] Gerhard K. Gierz, Karl Heinrich Hofmann, Klaus Keimel, Lawson Jimmie D., Michael W. Mislove, and Dana S. Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, Berlin, 1980.

[HRDR92] Furio Honsell and Simona Ronchi Della Rocca. An approximation theorem for topological lambda models and the topological incompleteness of lambda calculus. *J. Comput. System Sci.*, 45(1):49–75, 1992.

[Joh86] Peter T. Johnstone. *Stone Spaces*. Cambridge University Press, Cambridge, 1986. Reprint of the 1982 edition.

[Par76] David Park. The $\mathbf{Y}$-combinator in Scott's λ-calculus models (revised version). Theory of Computation Report 13, Department of Computer Science, University of Warick, 1976.

[Plo93] Gordon D. Plotkin. Set-theoretical and other elementary models of the λ-calculus. *Theoret. Comput. Sci.*, 121(1-2):351–409, 1993.

[Sco72] Dana S. Scott. Continuous lattices. In F.W.Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, volume 274 of *Lecture Notes in Mathematics*, pages 97–136, Berlin, 1972. Springer-Verlag.

[Sco75] Dana S. Scott. Open problem. In C. Böhm, editor, *Lambda Calculus and Computer Science Theory*, volume 37 of *Lecture Notes in Computer Science*, page 369. Springer-Verlag, Berlin, 1975.

Higher-Order Rewriting via Conditional First-Order Rewriting in the Open Calculus of Constructions

Mark-Oliver Stehr*

University of Illinois at Urbana-Champaign
The Thomas M. Siebel Center for Computer Science
Urbana, IL 61801-2302, USA
stehr@uiuc.edu

Abstract. Although higher-order rewrite systems (HRS) seem to have a first-order flavor, the direct translation into first-order rewrite systems, using e.g. explicit substitutions, is by no means trivial. In this paper, we explore a two-stage approach, by showing how higher-order pattern rewrite systems, and in fact a somewhat more general class, can be expressed by conditional first-order rewriting in the open calculus of constructions (OCC), which itself has been presented and implemented using explicit substitutions. The key feature of OCC that we exploit is that conditions are allowed to contain quantifiers and equations which can be solved using first-order matching. The way we express HRS works in spite of the fact that structural equality of OCC does not subsume α-conversion. Another topic that we touch upon in this paper is the use of higher-order abstract syntax in a classical framework like OCC, because it is often used in connection with higher-order rewriting.

1 Introduction

Higher-order rewriting in the general sense aims to generalize first-order rewriting to the higher-order case, that is to allow patterns that contain binders and to perform rewriting modulo a substitution calculus that correctly handles instantiation of such binders [33]. Two natural frameworks have evolved, namely combinatory reduction systems (CRS) [15] and higher-order rewrite systems (HRS) [23], called pattern rewrite systems in [19,24], and both formalisms have been shown to have essentially the same expressive power [32]. In the case of higher-order rewrite systems (HRS), the underlying substitution calculus is simply typed λ-calculus with β-reduction and restricted η-expansion.

The main objective of this paper is to explore how HRS can be expressed by conditional first-order rewriting in the open calculus of constructions (OCC) [29]. OCC is an equational type theory with dependent types, but its operational semantics is based on first-order rewriting and goal-oriented proof search. Built-in higher-order features, such as β-reduction are mapped to first-order rewriting using CINNI [29,28], a calculus of substitutions with named and indexed variables. OCC was designed to integrate two lines of research, which are: (1) λ-calculi with dependent types in the style of Martin-Löf's type theory [25] and the calculus of constructions [7], and (2) algebraic specification languages in the style of membership equational logic [4] and rewriting logic [20] as implemented in Maude [6].

The simulation of HRS in OCC is possible thanks the operational semantics of conditional computational equations in OCC, but we will see that there is still a gap between

* Support by ONR Grant N00014-02-1-0715 is gratefully acknowledged.

the two approaches, which is caused by the more extensional nature of our representation, and requires assumptions about the strategy used to solve conditions. We decided to make such assumptions syntactically explicit using the concept of matching equations [5]. There is some related work in this direction, namely [3,10], where explicit substitutions calculi based on de Bruijn indices are directly used to represent higher-order concepts. Our approach, on the other hand, proceeds in two stages: HRS are represented in OCC, which provides a notion of essentially first-order rewriting (with β-reduction, but without η- and even without α-conversion), and OCC has itself been presented and implemented using CINNI, that is explicit substitutions with names. Another closely related line of work, from which the idea of higher-order patterns originated, is the work on higher-order logic programming, see e.g. [21], which is based on the more powerful concept of (higher-order) unification (modulo α-conversion), rather than matching (without α-conversion) which we use in this paper.

Last but not least, we address how the operational semantics of OCC, which supports both equations and predicates, can be used to overcome certain semantic issues concerned with the use of higher-order abstract syntax, which is often used together with higher-order rewriting. In this paper we use a particular *predicative* instance of OCC,[1] that is an instance with a fixed predicative hierarchy of universes and a straightforward set-theoretic and operational semantics. To illustrate the capabilities of OCC we employ a new version of the OCC prototype that supports all the features needed in this paper (especially conditions with quantifiers and matching equations) and will be made publicly available in the near future.

2 Preliminaries

For a uniform way to deal with binding in the presentation of OCC we use CINNI [29,28], a generic calculus of explicit substitutions that generalizes Lescanne's λv [2] and can be instantiated to the syntax of nearly arbitrary object languages. We say that the syntax of an object language is a CINNI syntax if there is a distinguished sort of names and all variables, i.e. referencing occurrences of names, are of the form X_i for a name X and an index $i \in \mathbb{N}$. The idea is that X_i refers to the X-binder that can be reached on the way towards the outermost position after skipping i X-binders. Hence, X_0 (wich we simply write as X if there is no danger of confusion) refers to the innermost encompassing X-binder. Given the CINNI syntax $\mathcal{L}$ of a language we use $\text{CINNI}_\mathcal{L}$ to denote the instantiation of CINNI to the syntax of $\mathcal{L}$. The language of $\text{CINNI}_\mathcal{L}$ extends $\mathcal{L}$ by simple substitutions $[X{:=}M]$, shift substitutions $\uparrow_X$, and lift substitutions $\Uparrow_X S$, and a notion of substitution application, written $S\ M$, assuming that S ranges over substitutions and M ranges over terms. The equations of $\text{CINNI}_\mathcal{L}$ define the semantics of these substitutions and have been shown to be strongly normalizing. Since substitutions can always be eliminated using these equations, each $\text{CINNI}_\mathcal{L}$ term is equivalent to a unique $\mathcal{L}$ term. We use this as a justification for introducing the *general convention* to identify $\text{CINNI}_\mathcal{L}$ terms that are equivalent by virtue of the equational theory of $\text{CINNI}_\mathcal{L}$ throughout the present paper. Note, however, that this

[1] A slight difference to [29] is that we allow reduction under binders, a feature that is implemented in the new version of the OCC prototype.

convention does *not* imply that α-equivalent terms are identified. In fact, a key point in the design of OCC is that in spite of the use of names there is no need for the assumption of α-equality to define a notion of reduction.

We use [] to denote the empty list and a comma to denote list concatenation. We sometimes indicate the length of a list with a superscript. Therefore, X^n denotes a list of n elements. We write X_i (or X_i^n) for the i-th element of X. We abbreviate constructions over all elements in a list as constructions over the list itself: for example, we may write $(M\ N^n)$ for $(M\ N_1 \dots N_n)$, and $\{X^n : U^n\}$ for $\{X_1^n : U_1^n\} \dots \{X_n^n : U_n^n\}$.

3 The Open Calculus of Constructions

The *open calculus of constructions (OCC)* is a family of type theories that are concerned with three classes of terms: *elements*, *types* and *universes*. Types serve as an abstraction for collections of elements, and universes as an abstraction for collections of types.

OCC is parameterized by OCC signatures defining the universe structure. In this paper we use a fixed signature $\Sigma = (\mathcal{S}, \mathcal{S}^p, \mathtt{Prop}, :, \mathcal{R}, \leq)$ with *predicative* universes $\mathcal{S}^p = \{\mathtt{Type}, \mathtt{Type}_1, \mathtt{Type}_2, \dots\}$, which form a cumulative predicative hierarchy, and we add a propositional universe at the bottom by defining $\mathcal{S} = \{\mathtt{Prop}\} \cup \mathcal{S}^p$. This means that we have $\mathtt{Prop} : \mathtt{Type} : \mathtt{Type}_1 : \mathtt{Type}_2 \dots$, a subtyping relation $\mathtt{Prop} \leq \mathtt{Type} \leq \mathtt{Type}_1 \leq \mathtt{Type}_2 \dots$ (also called subuniverse relation), and $(s, s', s'') \in \mathcal{R}$ for all $s, s' \in \mathcal{S}$, where s'' is the least upper bound w.r.t. $\leq$ in $\mathcal{S}^p$ (not in $\mathcal{S}$).[2]

The formal system of OCC is designed to make sense under the *propositions-as-types interpretation*, where propositions are interpreted as types and proofs are interpreted as elements of these types. Since in OCC there is no a priori distinction between *terms* and *types*, and furthermore between *types* and *propositions*, we use all these notions synonymously.

OCC has the standard constructs known from pure type systems (cf. [1,29,30]) and a few additional ones. An *OCC term* can be one of the following: a *universe* s, a *variable* X_i (CINNI syntax), a typed λ-*abstraction* $[X : S]M$, a *dependent function type* $\{X : S\}T$, a *type assertion* $M : T$, an ϵ-*construct* $\epsilon\ A$ to denote an irrelevant proof of a proposition A, a *propositional equality* $M = N$, or one of three flavors of *operational propositions*, written as $||\ A$, $!!\ A$, or $??\ A$. Here and in following we usually use M, N, P, Q, S, T, U, V, A, and B to range over OCC terms, and X, Y, Z to range over names. Operational propositions can either be *structural propositions* designated by $||$ (built into the term representation), *computational propositions* designated by $!!$ (used for reduction), or an *assertional propositions* designated by $??$ (used for goal-directed proof search). Subsequently, we use τ to range over these three flavors $\{||, !!, ??\}$.

OCC contexts are lists of *declarations* of the form $X : S$. The empty context is written as []. Typically, we use Γ to range over OCC contexts. An *OCC specification* is simply an OCC context Γ in this paper.

[2] The effect of this choice of $\mathcal{R}$, a standard parameter for pure type systems [1], is that for arbitrary types $S : s$ (in a context Γ) and $T : s'$ (in a context $\Gamma, X : S$) with $s, s' \in \mathcal{S}$ we can form the dependent type $\{X : S\}T : s''$ (in Γ) for $s'' = s \sqcup s'$. Note that $\mathtt{Prop}$ is not closed under formation of dependent types, and hence it is not impredicative in this paper !

3.1 Model-theoretic Semantics

Since we are working with a predicative instance of OCC, it is straightforward to define a model-theoretic semantics based on classical set theory with suitable universes [29]. A similar semantics has been used for Martin-Löf's type theory [8,9] and for predicative universes of the extended calculus of constructions (ECC) [17] and the calculus of inductive constructions CIC [34].

A *set-theoretic universe*[3] is a set that is closed under the standard constructions of Zermelo-Fraenkel set theory with the axiom of choice (ZFC) [22,27].

In this paper we work with a fixed but arbitrary *OCC frame* for Σ, i.e. a family of sets $(\mathcal{U}_s)_{s\in\mathcal{S}}$ and satisfies the following conditions: For the OCC universe $s = \texttt{Prop}$ we require $\mathcal{U}_s = \{\emptyset, \{\bullet\}\}$, that is the propositional universe is simply interpreted as the Boolean set $\{\mathbb{F}, \mathbb{T}\}$, with $\mathbb{F} = \emptyset$ and $\mathbb{T} = \{\bullet\}$ ($\mathbb{T}$ is inhabited by $\bullet$ as a proof). Additionally, for each predicative OCC universe $s \in \mathcal{S}^p$ we require a classical set-theoretic universe $\mathcal{U}_s$, such that $s \prec s'$ implies $\mathcal{U}_s \in \mathcal{U}_{s'}$ for all OCC universes $s, s' \in \mathcal{S}$. We immediately observe that in each OCC frame $s : s'$ implies $\mathcal{U}_s \in \mathcal{U}_{s'}$, and that $s \leq s'$ implies $\mathcal{U}_s \subseteq \mathcal{U}_{s'}$ for all $s, s' \in \mathcal{S}$, which means that the cumulative hierarchy of OCC universes is strictly reflected in the set-theoretic semantics, i.e. by set-theoretic membership and inclusion. We furthermore define the set $\mathcal{U} = \bigcup\{\mathcal{U}_s \mid s \in \mathcal{S}\}$. In the semantics of OCC we will use the sets in $\mathcal{U}$ to interprete types, and the elements in $\bigcup\mathcal{U}$ to interprete arbitrary terms. To construct an OCC frame for a fixed OCC signature we can use well-known axiomatic extensions of ZFC which are sufficiently strong to interprete all OCC universes (see [29] for details).

Given an OCC frame we have to define the interpretation of OCC terms, OCC contexts, and finally OCC judgements. To this end, we first extend the original definition of OCC terms/context to enriched OCC terms/context by adding a new term constructor σ with $\bigcup\mathcal{U}$ as its domain so that *enriched OCC terms* are defined by the same syntax as OCC terms but extended by constructs $\sigma(\alpha)$ for all $\alpha \in \bigcup\mathcal{U}$. We usually leave σ implicit, writing α instead of $\sigma(\alpha)$. Also, we define enriched $\mathrm{CINNI}_{\mathrm{OCC}}$ terms/contexts in complete analogy to $\mathrm{CINNI}_{\mathrm{OCC}}$ terms/contexts, again with the *general convention* to identify them if they are equivalent by virtue of $\mathrm{CINNI}_{\mathrm{OCC}}$.

We first define a partial interpretation $[\![-]\!]$. It is a partial function on closed enriched terms which is subsequently extended to enriched contexts, and we generally assume that it is undefined for all cases not covered by the definitions below.[4] We begin with the definition of $[\![-]\!]$ for closed enriched OCC terms:

$$
\begin{aligned}
[\![\alpha]\!] &= \alpha \\
[\![s]\!] &= \mathcal{U}_s \\
[\![M\ N]\!] &= [\![M]\!]([\![N]\!]) \\
[\![[X : S]M]\!] &= \lambda\, \alpha \in [\![S]\!]\,.\, [\![[X{:=}\alpha]M]\!] \ \text{ if } \ [\![S]\!] \in \mathcal{U} \\
[\![\{X : S\}T]\!] &= \Pi\, \alpha \in [\![S]\!]\,.\, [\![[X{:=}\alpha]T]\!] \ \text{ if } \ [\![S]\!] \in \mathcal{U}
\end{aligned}
$$

[3] called *Z-F universe* in [22] and just *universe* in [18]

[4] We also make use of the standard convention that a set-theoretic formula can only hold and a set-theoretic expression can only be defined if all its subexpressions are defined.

$$
\begin{aligned}
[\![M : S]\!] &= [\![M]\!] \text{ if } [\![M]\!] \in [\![S]\!] \in \mathcal{U} \\
[\![M = N]\!] &= \mathbb{T} \text{ if } [\![M]\!] = [\![N]\!] \text{ and } [\![M]\!], [\![N]\!] \in \bigcup \mathcal{U} \\
[\![M = N]\!] &= \mathbb{F} \text{ if } [\![M]\!] \neq [\![N]\!] \text{ and } [\![M]\!], [\![N]\!] \in \bigcup \mathcal{U} \\
[\![\epsilon\ A]\!] &= \text{a unique element of } [\![A]\!] \text{ if } [\![A]\!] \neq \emptyset \text{ and } [\![A]\!] \in \mathcal{U} \\
[\![\tau\ P]\!] &= [\![P]\!] \text{ if } [\![P]\!] \in \mathcal{U}
\end{aligned}
$$

On the right hand side of the equations for $[\![[X : S]M]\!]$ and $[\![\{X : S\}T]\!]$ we have used the set-theoretic λ-abstraction and the set-theoretic dependent function space: $\lambda\, x \in S\,.\, t(x)$ denotes the total function f with domain S defined by $f(x) = t(x)$ for all $x \in S$, and $\Pi\, x \in S\,.\, T(x)$ denotes the set of all functions f with domain S with the property that $f(x) \in T(x)$ for all $x \in S$.

The uniqueness requirement in the definition of $\epsilon\ A$ can be realized by choosing an element that is minimal w.r.t. a fixed well-founded total order. The term ϵA is used to denote an unspecified proof of A if A can be proved by means of the operational semantics. Concerning the interpretation $[\![\tau\ P]\!]$ of operational propositions, we can see that the set-theoretic semantics abstracts from their operational nature which is specified by τ.

The partial interpretation $[\![-]\!]$ of closed enriched OCC contexts is defined by sets of tuples of the form $(\alpha_1, \alpha_2, \ldots, \alpha_n) = (\alpha_1, (\alpha_2, (\ldots, (\alpha_n, ()))))$ as follows.

$$
\begin{aligned}
[\![[]]\!] &= \{()\} \\
[\![X : S,\ \Gamma]\!] &= \Sigma\, \alpha \in [\![S]\!]\,.\, [\![[X{:=}\alpha]\,\Gamma]\!] \text{ if } [\![S]\!] \in \mathcal{U}
\end{aligned}
$$

Here we use the set-theoretic *dependent sum* $\Sigma\, x \in S\,.\, T(x)$ to denote the set of pairs (x, y) satisfying $x \in S$ and $y \in T(x)$. Given an *OCC specification* $\Gamma = Z^n : S^n$, a *model* of Γ is simply a tuple $\gamma^n \in [\![\Gamma]\!]$.

3.2 Formal System and Operational Semantics

The operational semantics of OCC is explained in terms of the formal system of OCC. It is a direct generalization of the operational semantics of membership equational logic [4] as implemented in Maude [6].

The *formal system of OCC* defines *derivability* of OCC judgements $\Gamma \vdash J$. For brevity we only give an informal explanation of all judgements and their intuitive operational meaning.

- The *type inference judgement* $\Gamma \vdash M \rightarrow: S$ asserts that the term M is an *element* of the *inferred type* S in the context Γ. Operationally, Γ and M are given and S is obtained by syntax-directed type inference and possible reduction using computational equations modulo the structural equations of Γ.
- The *typing judgement* $\Gamma \vdash M : S$ asserts that M is an *element* of *type* S in the context Γ. Operationally, Γ, M and S are given and verifying $\Gamma \vdash M : S$ amounts to type checking. Type checking is always reduced to type inference and the verification of an assertional subtyping judgement.

- The *structural equality judgement* $\Gamma \vdash || (M = N)$ is used to express that M and N are considered to be structurally equal elements in the context Γ. Operationally, structural equality is realized by a suitable term representation so that structurally equal terms cannot be distinguished when they participate in computations.
- The *computational equality judgement* $\Gamma \vdash !! (M = N)$ is the judgement that defines the notion of reduction for the simplification of terms. The judgement states that the element M can be reduced to the element N in the context Γ. Operationally, Γ and M are given and N is the result of reducing M using the computational equations in Γ modulo the structural equations in Γ. Computational equality subsumes β-reduction (CINNI-based) and as an extension of [29] we add rules for reduction under binders.
- The *assertional judgement* $\Gamma \vdash ?? A$ states that A is provable by means of the operational semantics in the context Γ. Operationally, Γ and A are given and the judgement is verified by a combination of reduction using the computational equations and exhaustive goal-oriented search using the assertional propositions in Γ. Both processes take place modulo the structural equations in Γ. It is important to point out that assertional equality subsumes α-conversion, but structural and computational equality do not.
- The *assertional equality judgement* $\Gamma \vdash ?? (M = N)$ states that M and N are assertionally equal in Γ, a notion that treats equality as a predicate and subsumes the structural and computational equality judgements. Operationally, Γ, M and N are given and the judgement is verified like other assertional judgements in a goal-oriented fashion.
- The *assertional subtyping judgement* $\Gamma \vdash ?? (S \leq T)$ subsumes the assertional equality judgement and states that S is a subtype of T in Γ as a consequence of the cumulativity of the universe hierarchy. Operationally, Γ, S and T are given and the judgement is verified like other assertional judgements in a goal-oriented fashion.

All these judgements are interdependent. For instance, computational equations can have conditions giving rise to assertional judgements. Conversely, solving assertional judgements may apart from exhaustive proof search involve simplification using computational equations. Both computational equations and assertional propositions can be universally quantified, and like in higher-order logic programming [21] their conditions can again involve universal quantifiers. A universally quantified condition $\{X : A\}B$ gives rise to a goal $\Gamma \vdash ??\{X : A\}B$. To solve such goals, OCC has an inference rule (corresponding the the introduction rule for universal quantifiers) which reduces the goal $\Gamma \vdash ??\{X : A\}B$ to $\Gamma, X : A \vdash ??B$, that is the subject B needs to be proved for a symbolic $X : A$.

Conditions can contain other logical operators that can be specified inside OCC, such as conjunction or existential quantifiers. We assume that this is done in an initial context that we do not make explicit in the reminder of this paper. It should contain at least the following logical constants, operators, and introduction rules:

```
False : Prop .
True : Prop .
True_intro : ?? True .

Not : Prop -> Prop .
Not_intro : {A : Prop}(A -> False) -> (Not A) .
```

```
And : Prop -> Prop -> Prop .
And_intro : ?? {A,B : Prop} A -> B -> (And A B) .

Ex : {T : Type} (T -> Prop) -> Prop .
Ex_intro : ?? {T : Type}{x : T}{P : (T -> Prop)} (P x) -> (Ex T P) .
```

To cover a wide range of implementations, the formal system of OCC leaves open how operational propositions are instantiated. Typically, this will be done by matching, but variables that cannot be determined in this way are instantiated by metavariables with the hope that they can be solved later.

The formal system has been shown to be sound w.r.t. to the classical set-theoretic semantics given earlier [29], a result that immediately implies consistency of the formal system as a logic, i.e. under the propositions-as-types interpretation. To formulate the soundness result, we first define *validity* of OCC judgements, based on the partial interpretation $[\![-]\!]$ of enriched OCC terms. For arbitrary set-theoretic expressions α, we use the abbreviation $\downarrow\alpha$ to express that α is defined. We assume that the validity predicate is false in all cases not covered by the following definition.

$$
\begin{aligned}
\models M \to: S &\text{ if } [\![M]\!] \in [\![S]\!] \\
\models M : S &\text{ if } \downarrow[\![S]\!] \text{ implies } [\![M]\!] \in [\![S]\!] \\
\models ??\ A &\text{ if } \downarrow[\![A]\!] \text{ implies } [\![A]\!] \neq \emptyset \\
\models ??\ (M = N) &\text{ if } (\downarrow[\![M]\!] \text{ and } \downarrow[\![N]\!]) \text{ implies } [\![M]\!] = [\![N]\!] \\
\models ??\ (S \leq T) &\text{ if } (\downarrow[\![S]\!] \text{ and } \downarrow[\![T]\!]) \text{ implies } [\![S]\!] \subseteq [\![T]\!] \\
\models ||\ (M = N) &\text{ if } (\downarrow[\![M]\!] \text{ or } \downarrow[\![N]\!]) \text{ implies } [\![M]\!] = [\![N]\!] \\
\models !!\ (M = N) &\text{ if } \downarrow[\![M]\!] \text{ implies } [\![M]\!] = [\![N]\!] \\
\Gamma \models J &\text{ if } \downarrow[\![\Gamma]\!] \text{ implies } \models [Z^n{:=}\gamma^n]\ J \text{ for all } \gamma^n \in [\![\Gamma]\!]
\end{aligned}
$$

where Γ is of the form $Z^n : S^n$.

Theorem 1 (Soundness).
For each derivable OCC judgement $\Gamma \vdash J$ we have $\Gamma \models J$.

As a simple corollary we obtain consistency of the formal system, i.e., that we cannot prove the absurd proposition $\bot_s = \{X : s\}X$ for any universe s in the empty context.

Corollary 1 (Consistency). *The formal system of OCC is consistent, i.e., $[] \vdash M : \bot_s$ is not derivable for any M, s.*

By fixing the signature at the beginning of this section, we have introduced a particular instance of OCC. The full version of OCC [29] includes (nonequational) rewrite judgements and generalizes rewriting logic [20], a logic for the specification of distributed systems. We have omitted these judgements in our presentation of OCC, because we focus on purely equational rewriting in this paper.

4 Higher-Order Rewriting via First Order-Rewriting

A simple example of higher-order rewriting that can be treated entirely as first order rewriting is the following. We introduce a parameterized type of finite multisets with three constructors and the standard structural axioms of a commutative monoid. For better readability, we write `{T | Type}` instead of `{T : Type}`, a way of expressing that this argument is implicit and should be inferred.

```
fms : Type -> Type .

empty : {T | Type} (fms T) .
single : {T | Type} T -> (fms T) .
union : {T | Type} (fms T) * (fms T) -> (fms T) .

assoc : || {T : Type}{l1,l2,l3 : (fms T)}
  (union (l1,(union (l2,l3)))) = (union ((union (l1,l2)),l3)) .
comm : || {T : Type}{l1,l2 : (fms T)}
  (union (l1,l2)) = (union (l2,l1)) .
id : || {T : Type}{l : (fms T)}
  (union (l,empty)) = l .
```

Now the higher-order function `map`, which applies a function to each element of a given multiset, can be naturally specified using computational equations:

```
map : {U,V | Type} (U -> V) -> (fms U) ->(fms V) .

map-empty : !! {U,V : Type}{f : (U -> V)}
  (map f empty) = empty .
map-cons : !! {U,V : Type}{f : (U -> V)}{x : U}{l : (fms U)}
  (map f (union (l,(single x)))) = (union ((map f l),(single (f x)))) .
```

With `red` the OCC prototype reduces the given term in the current context according to the computational equality judgement until it cannot be further reduced:

```
red (map ([x : nat] (suc x))
         (union ((single 0),union((single 1),(single 2))))) .
result: (union ((single 1),(single 2),(single 3)))
```

Although these features are not the main topic of this paper, this examples also shows how OCC allows us to express polymorphism and rewriting modulo structural axioms.

4.1 Conditional Higher-Order Rewriting

An example using the expressive power of conditional higher-order rewriting is the following function `select` which selects elements from a multiset satisfying a certain predicate, i.e. an element of type `(T -> Prop)`.

```
select : {T | Type} (T -> Prop) -> (fms T) -> (fms T) .

select-empty : !! {T : Type}{P : (T -> Prop)}
```

```
  (select P empty) = empty .

select-single-1 : !! {T : Type}{P : (T -> Prop)}{x : T}{l : (fms T)}
  (P x) -> (select P (union (l,(single x)))) =
           (union ((select P l),(single x))) .

select-single-2 : !! {T : Type}{P : (T -> Prop)}{x : T}{l : (fms T)}
  (Not (P x)) -> (select P (union (l,(single x)))) =
                 (select P l) .
```

Note how the predicate `P` is passed as an argument and used in the condition. The following execution shows that β-reduction can be involved in solving conditions, apart from the exhaustive goal-oriented search, which takes into account the assertional inequality axiom.

```
A : Type .  a : A .  b : A .

not-eq-a-b : ?? (Not (a = b)) .

red (select ([x : A](x = b)) (union ((union ((single a),(single b))),(single a)))) .
result: (single b)
```

4.2 Dealing with Binders in Patterns

Since computational equations do not operate modulo α-equality, binders in patterns, like in the following equation, are useless. For instance, the following *naive representation* of a higher-order rewrite rule in OCC would take the name of the bound variable on the left-hand side seriously:

```
map-id : !! {T : Type}{l : (fms T)}(map ([x : T] x) l) = l .

red (map ([x : nat] x) empty) .
result: empty

red (map ([y : nat] y) empty) .
result: (map ([y : nat] y) empty)
```

Fortunately, we can express this equation in a semantically equivalent way (in the model-theoretic sense) by using the power of universally quantified conditions:

```
map-id : !! {T : Type}{id : T -> T}{l : (fms T)}
            ({X : T} ((id X) = X)) -> (map id l) = l .

l : (fms nat) .
red (map ([x : nat] x) l) .
result: l
```

The universal quantifier in the condition operationally amounts to a fresh variable `X` that can be used to symbolically verify `(id X) = X`, in this case using the built-in β-reduction.

4.3 Rewriting Higher-Order Abstract Syntax

Higher-order abstract syntax [26] has been proposed as a systematic approach to represent the abstract syntax of languages with binding constructs in logical frameworks such as LF [12]. The main idea is to represent the binding constructs of the object language using binding constructs of the metalanguage, i.e. to aim at a shallow embedding. The advantage of using higher-order abstract syntax is that α-equality is inherited from the metalanguage and substitution corresponds to β-reduction.

In the following we use terms of the untyped λ-calculus with a single constant as an example. Using the original approach [26] a type of terms would be specified as follows:

```
term : Type .
const : term .
app : term -> term -> term .
abs : (term -> term) -> term .
```

For instance, the untyped λ-term $\lambda x.\lambda y.x$ would be represented as

```
(abs ([x : term] (abs ([y : term] x))))
```

in the context of this specification.

To give a first example how this representation can be used operationally, we specify an equality predicate by the following conditional assertional axioms:

```
eq : term -> term -> Prop .

eq_0 : ?? {M : term} (M = M) -> (eq M M) .
eq_1 : ?? {M,N,M',N' : term} (eq M M') -> (eq N N') -> (eq (app M N) (app M' N')) .
eq_2 : ?? {B,B' : term -> term} ({M : term}(eq (B M) (B' M))) -> (eq (abs B) (abs B')) .
```

With `ver` the OCC prototype tries to solve the assertional judgement corresponding to the given proposition in the current context:

```
ver (eq (abs ([y : term] y)) (abs ([x : term] x))) .
all goals solved
```

To accommodate for the equational nature of the λ-calculus, the specification can be extended by a computational equation for β-reduction and by an assertional equality for η-conversion:

```
beta : !! {B : (term -> term)}{M : term} (app (abs B) M) = (B M) .

red (app (abs ([x : term] x)) (abs ([x : term] x)) ) .
result: (abs ([x : term] x))

eta : ?? {B : (term -> term)}{F : term} ({X : term} ((app F X) = (B X))) -> (abs B) = F .

ver ((abs ([x : term] (app const x))) = const) .
all goals solved
```

Alternatively, we could specify η-conversion using a computational equation rather then using an assertional equality:

```
eta : !! {B : (term -> term)}{F : term} ({X : term} ((app F X) = (B X))) -> (abs B) = F .

red (abs ([x : term] (app const x))) .
result: const

red (abs ([x : term] (app x x))) .
result: (abs ([x : term] ((app x) x)))
```

Notice that the conclusion `(abs B) = F` of `eta` has a variable `F` on the right which does not appear on the left hand side of the conclusion. Operationally, `F` can be determined by the condition `(app F X) = (B X)` using pattern matching, i.e. by reducing `(B X)` and matching against the pattern `(app F X)`. In general, solving goals with metavariables variables (like `F` in this case) may require unification (or more generally narrowing), but for all purposes of this paper matching is sufficient. Note also that nothing happened in the last reduction, because it would require binding `F` to `X` which is not in the scope of `F`.

4.4 Rewriting with Nested Binders

The binding structure of patterns is not always as simple as in the above example of λ-calculus. Consider the following fragment of classical higher-order logic:

```
implies : bool -> bool -> bool .
all : (bool -> bool) -> bool .
ex : (bool -> bool) -> bool .
```

We want to formulate the higher-order rewrite rule, naively represented in OCC as

```
eq : !! {P : bool -> bool -> bool}
          (implies (ex [X : bool] (all [Y : bool] (P X Y)))
                   (all [Y : bool] (ex [X : bool] (P X Y)))) = true .
```

in a way that captures its actual meaning in HRS. Obviously, we need to get rid of all binders in the pattern on the left hand side. A naive attempt, which introduces local variables `X` and `Y` using quantifiers in the condition and decomposes the formula using matching equations, might look like this:

```
eq : !! {L,L',R,R' : (bool -> bool)}
   ({X,Y : bool} (And (And ((all L') = (L X))
                           ((ex R') = (R Y)))
                      ((L' b) = (R' X)))) ->
   (implies (ex L) (all R)) = true .

red (implies (ex [x : bool] (all ([y : bool] true)))
             (all [y : bool] (ex ([x : bool] true)))) .
result: true

red (implies (ex [x : bool] (all ([y : bool] x)))
             (all [y : bool] (ex ([x : bool] x)))) .
result: ((implies (ex ([x : bool] (all ([y : bool] x)))))
                  (all ([y : bool] (ex ([x : bool] x)))))
```

Unfortunately, the second reduction does not give the desired result. The core of the problem is that the only potential solution for `L'` in `(all L') = (L X)` requires `L'` to contain `X`, but this is impossible, because `L'` is outside of the scope in which `X` is known. The correct representation needs to allow the dependency of `L'` on `X`, which can be achieved using an existential quantifier inside the condition:

```
eq : !! {L,R : (bool -> bool)}
   ({X,Y : bool}
     (Ex (bool -> bool) [L' : (bool -> bool)]
     (Ex (bool -> bool) [R' : (bool -> bool)]
        (And (And ((all L') = (L X))
                  ((ex R') = (R Y)))
                  ((L' Y) = (R' X)))))) ->
   (implies (ex L) (all R)) = true .

red (implies (ex [x : bool] (all ([y : bool] true)))
             (all [y : bool] (ex ([x : bool] true)))) .
result: true

red (implies (ex [x : bool] (all ([y : bool] x)))
             (all [y : bool] (ex ([x : bool] x)))) .
result: true
```

4.5 Higher-Order Rewrite Systems

Generalizing the ad hoc approach of the previous section, we now define a translation from HRS into OCC. Since HRS are based on simply typed λ-calculus, we assume in the following that all term variables are restricted to terms or types (depending on the context) of the simply typed λ-calculus, which both constitute terms in OCC.

A *higher-order rewrite system* (HRS) is a set of rewrite rule of the form $LHS \to RHS$, where LHS and RHS are long $\beta\eta$-normal forms, and LHS is a higher order-pattern, and not a free variable. A *higher-order pattern* is a term in β-normal form satisfying the condition that in each *free variable application*, i.e. in a subterm term $(P\ M_1\ \ldots M_n)$ with P free in LHS, all $M_1 \ldots, M_n$ are η-equivalent to distinct bound variables of LHS. For simplicity we assume that all bound variables are distinct. We write $M \to_{\mathcal{R}} M'$ to express that M and M' are terms in $\beta\eta$-normal form and there is a rule $LHS \to RHS$ in $\mathcal{R}$, a position p in M, and a substitution θ such that $M/p = \theta(LHS)$, $M' = RHS[\theta(RHS)]_p$. Here, substitution application is defined by $\theta(N) = (([X^n : S^n]N)P^n)\uparrow^{\eta}_{\beta}$ for a substitution θ mapping typed variables $X^n : S^n$ to terms P^n. For further details we refer to [24].

We start with the *naive representation* of a HRS in OCC, i.e. a context where each HRS rewrite rule $LHS \to RHS$ is represented as a universally quantified unconditional equation of the form

$$eq : !!\ \{Z^z : W^z\}\ LHS = RHS,$$

where LHS and RHS are of the same base type, the free variables of RHS are a subset of the free variables Z^z (which have types W^z) of LHS, and LHS is a higher-order pattern.

Obviously, the requirement that LHS is of base type implies that LHS cannot be a λ-abstraction $[X : S]M$ itself. Furthermore, by the definition of higher-order patterns a free variable application can occur only under a binder.

The translation proceeds in three steps. The first step brings the HRS into left-linear form, and the remaining two steps successively eliminate binders in patterns. Each equation of the naive representation is translated into a conditional equation of the form

$$eq : !! \ \{Z^z : W^z\} \ \{P^p : U^p\} \ C \to D \to LHS = RHS,$$

where the condition C is of the form $\{X^x : S^x\} \ \exists[Q^q : V^q] \ (C_1 \ \wedge \ C_2 \ \wedge \ \ldots \ \wedge \ C_c)$ with $C_1, \ldots, C_c$ being equations (used to express matching constraints), and D is of the form $(D_1 \ \wedge \ D_2 \ \wedge \ \ldots \ \wedge \ D_d)$ with $D_1, \ldots, D_d$ being possibly universally quantified equations (used to express left-linearity constraints). The order of quantifiers is not essential, except that in the condition C the existential quantifiers should be in the scope of the universal quantifiers.

1. As long as there is an free variable P occurring more than once in LHS do the following:
   - Choose a fresh variable P'.
   - Replace the second leftmost P in LHS by P'.
   - Add a universally quantified condition $\{Y^y : T^y\}(P \ Y^y) = (P' \ Y^y)$, assuming that P has the type $T^y \to T$, with T being a base type.
2. As long as there is a λ-abstraction in LHS do the following:
   - Choose a fresh variable P.
   - Replace the leftmost subterm $[X : S]M$ in LHS by P.
   - Add an outer universal quantifier $\{P : U\}$, with U being the type of $[X : S]M$.
   - Add an equation $[X : S]M = P$ to the condition.
3. As long as the condition contains an equation $PAT = N$ such that PAT contains a λ-abstraction choose the leftmost such equation and do the following:
   (a) If PAT is a λ-abstraction then:
      - Replace the leftmost condition of the form $[X : S]M = N$ by $M = (N \ X)$.
      - Add an outer universal quantifier $\{X : S\}$ to the condition.
   (b) If PAT contains a λ-abstraction as a proper subterm then:
      - Choose a fresh variable Q.
      - Replace the leftmost subterm $[X : S]M$ in PAT by Q.
      - Add an inner existential quantifier $\exists[Q : V]$ to the condition, with V being the type of $[X : S]M$.
      - Add an equation $[X : S]M = Q$ to the condition.

Let us apply this translation to our previous example. For brevity, we sometimes combine two steps in one. Note that the first step makes the rule left-linear:

```
eq : !! {P : bool -> bool -> bool}
          (implies (ex [X : bool] (all [Y : bool] (P X Y)))
                   (all [Y : bool] (ex [X : bool] (P X Y)))) = true .

eq : !! {P,P' : bool -> bool -> bool}
          ({X,Y : bool} (P X Y) = (P' X Y)) ->
          (implies (ex [X : bool] (all [Y : bool] (P X Y)))
                   (all [Y : bool] (ex [X : bool] (P' X Y)))) = true .

eq : !! {P,P' : bool -> bool -> bool}{L,R : bool ->bool}
          (And (([X : bool] (all [Y : bool] (P X Y))) = L)
               (([Y : bool] (ex [X : bool] (P' X Y))) = R)) ->
          ({X,Y : bool} (P X Y) = (P' X Y)) ->
          (implies (ex L) (all R)) = true .

eq : !! {P,P' : bool -> bool -> bool}{L,R : bool ->bool}
          ({X : bool}{Y : bool}
            (And ((all [Y : bool] (P X Y)) = (L X))
                 ((ex [X : bool] (P' X Y)) = (R Y)))) ->
          ({X,Y : bool} (P X Y) = (P' X Y)) ->
          (implies (ex L) (all R)) = true .

eq : !! {P,P' : bool -> bool -> bool}{L,R : bool -> bool}
          ({X : bool}{Y : bool}
          (Ex (bool -> bool) [L' : bool -> bool]
          (Ex (bool -> bool) [R' : bool -> bool]
            (And (And ((all L') = (L X))
                      ((ex R') = (R Y)))
                 (And (([Y : bool] (P X Y)) = L')
                      (([X : bool] (P' X Y)) = R')))))) ->
          ({X,Y : bool} (P X Y) = (P' X Y)) ->
          (implies (ex L) (all R)) = true .

eq : !! {P,P' : bool -> bool -> bool}{L,R : bool -> bool}
         ({X : bool}{Y : bool}
           (Ex (bool -> bool) [L' : bool -> bool]
           (Ex (bool -> bool) [R' : bool -> bool]
             (And (And ((all L') = (L X))
                       ((ex R') = (R Y)))
                  (And ((P X Y) = (L' Y))
                       ((P' X Y) = (R' X)))))))) ->
         ({X,Y : bool} (P X Y) = (P' X Y)) ->
         (implies (ex L) (all R)) = true .
```

The last equation is operationally equivalent to our running ad hoc solution of Section 4.4, but it is more general, because it would also allow the use of `P` on the right hand side. In the goals generated by the last two conditions `(P X Y) = (L' Y)` and `(P' X Y) = (R' X)` the variables `P` and `P'` become metavariables, and they have trivial solutions, namely `[X : bool][X : bool] (L' Y)` and `[X : bool][X : bool] (R' X)`. So in cases like this, where the pattern is a free variable application, there is not even a need for matching.

The general way to express this is to perform matching modulo β_0, a trivial form of β-conversion generated by $([X:T]M)X = M$. The more general idea of unification modulo β_0 to deal with higher-order patterns is well-studied [21], but here we use β_0 in an even more restricted sense that does not admit α-conversion. In general, β_0-conversion is needed to solve matching equations of the form $(P\ X_1\ \ldots X_n) := M$ with a free variable application as the pattern.[5] The solution in this case is simply $P = [X^x : S^x]M$ for suitable types S^x. Because of the trivial nature of β_0 we consider it as a structural equation for the reminder of this paper, that is *we identify β_0-equivalent terms.*

Another example that illustrates the translation is the following:

```
f : ((nat -> nat) -> nat) -> nat .

eq : !! {F : (nat -> nat) -> nat}
        (f ([X : nat -> nat] (F ([Z : nat] (X Z))))) = (F ([Y : nat] Y)) .

eq : !! {F : (nat -> nat) -> nat}{L : ((nat -> nat) -> nat)}
        (([X : nat -> nat] (F ([Z : nat] (X Z)))) = L) ->
        (f L) = (F ([Y : nat] Y)) .

eq : !! {F : (nat -> nat) -> nat}{L : ((nat -> nat) -> nat)}
        ({X : nat -> nat} (((F ([Z : nat] (X Z)))) = (L X))) ->
        (f L) = (F ([Y : nat] Y)) .

eq : !! {F : (nat -> nat) -> nat}{L : ((nat -> nat) -> nat)}
        ({X : nat -> nat} (Ex (nat -> nat) [L' : nat -> nat]
          (And (((F L')) = (L X))
               (([Z : nat] (X Z)) = L')))) ->
        (f L) = (F ([Y : nat] Y)) .

eq : !! {F : (nat -> nat) -> nat}{L : ((nat -> nat) -> nat)}
        ({X : nat -> nat}{Z : nat} (Ex (nat -> nat) [L' : nat -> nat]
          (And (((F L')) = (L X))
               ((X Z) = (L' Z))))) ->
        (f L) = (F ([Y : nat] Y)) .

F : (nat -> nat) -> nat .

red (f ([x : nat -> nat] (F ([z : nat] (x z))))) .
result: (F ([Y : nat] Y))
```

Note that the goal generated by the last condition `(X Z) = (L' Z)` will not contain any metavariables.

The next theorem makes precise the statement that HRS can be simulated in OCC. Since HRS require us to go to long $\beta\eta$-normal form (which may require the traditional implicit use of α-conversion) before applying a rule,[6] and OCC does not have α- and η-conversion

[5] In the goals generated by matching equations P is a metavariable possibly subject to certain substitutions, but due to the structure of our representation of HRS in OCC all dependencies are explicit in $X_1, \ldots, X_n$ so that these substitutions can be eliminated.

[6] To be precise we should distinguish between two notions of β-reduction: HRS uses the standard one which allows α-conversion, and OCC uses CINNI and hence preserves names [30].

(at the structural and computational level), we allow for some extra $\alpha\eta$-conversion after the simulation step. We do not explicitly mention the trivial β_0-conversion, because according to our convention it is always part of the structural equality of OCC.

Theorem 2. *Let $\mathcal{R}$ be an arbitrary HRS and $\Gamma_\mathcal{R}$ the OCC context obtained by our translation of $\mathcal{R}$, and let M and M' be terms in the simply typed λ-calculus. Then*

$$M \updownarrow_\beta^\eta \to_\mathcal{R} M' \Rightarrow \exists M'' \,.\, \Gamma_\mathcal{R} \vdash !!(M =_\mathcal{R} M'') \text{ in OCC} \wedge M'' \equiv_{\alpha\eta} M',$$

where $\Gamma_\mathcal{R} \vdash !!(M =_\mathcal{R} M')$ means $\Gamma_\mathcal{R} \vdash !!(M = M')$ with the restriction that (disregarding goals arising from conditions) it is generated by an application of one of the computational equations in $\Gamma_\mathcal{R}$, preceeded or followed by any number of OCC β-reduction steps.

Proof Sketch. Consider an extension $\text{OCC}_{\alpha\eta}$ of OCC with structural α- and η-equality. Let Γ_1 be the naive representation of $\mathcal{R}$ in OCC, and let $\Gamma_1, \Gamma_2, \ldots, \Gamma_n{=}\Gamma_\mathcal{R}$ be the sequence of OCC contexts obtained by our translation. Since β_0-equivalent terms are identified we can show that

$$M \updownarrow_\beta^\eta \to_\mathcal{R} M' \Rightarrow \Gamma_1 \vdash !!(M =_\mathcal{R} M') \text{ in } \text{OCC}_{\alpha\eta}.$$

Furthermore, it can be easily verified that each of the steps of our translation, namely (1), (2), (3a), and (3b) preserves computational equality in the sense that for all $i, i{+}1 \in \{1, \ldots, n\}$ we have

$$\Gamma_i \vdash !!(M =_\mathcal{R} M') \text{ in } \text{OCC}_{\alpha\eta} \Rightarrow \Gamma_{i+1} \vdash !!(M =_\mathcal{R} M') \text{ in } \text{OCC}_{\alpha\eta}.$$

As a consequence we obtain

$$M \updownarrow_\beta^\eta \to_\mathcal{R} M' \Rightarrow \Gamma_\mathcal{R} \vdash !!(M =_\mathcal{R} M') \text{ in } \text{OCC}_{\alpha\eta}.$$

However, in the final result $\Gamma_\mathcal{R}$ of the translation we do not make use of binders on the left hand side of the computational equations so that, disregarding the process of solving conditions, in the derivation of $\Gamma_\mathcal{R} \vdash !!(M =_\mathcal{R} M')$ all α- and η-conversions can be postponed. Furthermore, for each solution of a condition of $\Gamma_\mathcal{R}$ in $\text{OCC}_{\alpha\eta}$ there is an $\alpha\eta$-equivalent solution that does not require structural α- or η-conversion and hence is a solution in OCC. Therefore we have

$$\Gamma_\mathcal{R} \vdash !!(M =_\mathcal{R} M') \text{ in } \text{OCC}_{\alpha\eta} \Rightarrow \exists M'' \,.\, \Gamma_\mathcal{R} \vdash !!(M =_\mathcal{R} M'') \text{ in OCC and } M'' \equiv_{\alpha\eta} M'.$$

4.6 Matching Equations

Given the previous theorem an obvious question is whether each application of a computational equation in the OCC representation to a simply typed λ-term (in long $\beta\eta$-normal form) can be simulated by the original HRS in some reasonable sense, but it can be easily shown that this is not the case. Consider the following example:

```
T : Type . a, b, c : T .

h : (T -> T) -> T .

eq-1 : !! a = b .
eq-2 : !! (h ([x : T] a)) = c .
```

In a HRS with these two rules the term `(h ([x : T] b))` does not reduce, but if we apply our translation, which only affects `eq-2`, a reduction is possible:

```
eq-2 : !! {P : T -> T} ({X : T} (a = (P X))) -> (h P) = c .

red (h ([x : T] b)) .
result: c
```

This example might suggest that it would be sufficient to assume that the left hand sides of all equations of the HRS are normalized w.r.t. the HRS itself, but a slightly modified example shows that this is not sufficient:

```
f : T -> T .  g : T -> T .

eq-1 : !! (f a) = (g b) .
eq-2 : !! {Z : T} (h ([x : T] (f Z))) = c .
```

The term `(h ([x : T] (g b)))` is not reducible in this HRS, but applying our translation replaces `eq-2` by the following equation, which can be potentially (according to the formal system of OCC) instantiated with `Z` bound to `a`, so that `(h ([x : T] (g b)))` becomes reducible.

```
eq-2 : !! {P : T -> T} {Z : T} ({X : T} (f Z) = (P X)) -> (h P) = c .
```

For applications in theorem proving it seems actually beneficial to have a less constrained reduction relation as long as it is not the source of nontermination and consistent with the model-theoretic semantics, which in some sense serves as an upper bound for computational equality and operational propositions in general. For instance, in an OCC based proof assistant the theorem `(h ([x : T] b)) = c` can be automatically proved in our first OCC representation of out first counterexample but not in the original HRS, although it obviously holds. On the other hand, to transfer termination arguments from HRS to their OCC representation we need some form of simulation in the opposite direction. Unfortunately, there does not seem to be an easy way to strengthen the theorem without *restrictions on the strategy* used to solve conditions. For instance, in the second conterexample, solving the condition by matching cannot result in `Z` being bound to `a`, but the use of narrowing would allow this solution.

Since such restrictions are not desirable for all purposes we introduce the concept of a matching equation $PAT := M$, which is simply an ordinary equality $PAT = M$, but with the operational restriction that reduction is never applied on its left hand side (except for trivial β_0-steps that we included in our structural equality).

Now we *redefine our translation* such that each equation $PAT = M$ introduced in the condition by the steps (2) or (3) becomes a matching equation $PAT := M$. The following theorem states: (1) that our previous result remains valid, and (2) that a simulation in the opposite direction is possible as well. Regarding (2), it is noteworthy that a single step in OCC can give rise to several steps in HRS, because an arbitrary number of reductions can be hidden in the process of solving conditions. Furthermore, we need to be more relaxed by allowing extra β-conversion steps after the HRS reduction, because HRS force us to go to long $\beta\eta$-normal form before a rule can be applied.

Theorem 3. *Let $\mathcal{R}$ be an arbitrary HRS and $\Gamma_\mathcal{R}$ the OCC context obtained by our (redefined) translation of $\mathcal{R}$, and let M and M' be terms in the simply typed λ-calculus. Then*

1. $M \updownarrow_\beta^\eta \to_\mathcal{R} M' \Rightarrow \exists M''\,.\,\Gamma_\mathcal{R} \vdash !!(M =_\mathcal{R} M'')$ *in OCC* $\wedge\ M'' \equiv_{\alpha\eta} M'$, *and*
2. $\Gamma_\mathcal{R} \vdash !!(M =_\mathcal{R} M')$ *in OCC* $\Rightarrow \exists M''\,.\,M \updownarrow_\beta^\eta \to_\mathcal{R}^+ M'' \wedge M'' \equiv_{\alpha\beta\eta} M'$.

Proof Sketch. The proof of (1) is the same as the proof of the previous theorem, except that the implication

$$\Gamma_i \vdash !!(M =_\mathcal{R} M') \text{ in } \mathrm{OCC}_{\alpha\eta} \Rightarrow \Gamma_{i+1} \vdash !!(M =_\mathcal{R} M') \text{ in } \mathrm{OCC}_{\alpha\eta}.$$

needs to be reverified for the three steps of our translation, taking into account that reduction is not possible on the left hand side of goals corresponding to matching equations.

The proof of (2) essentially reverses the reasoning of (1). Obviously,

$$\Gamma_\mathcal{R} \vdash !!(M =_\mathcal{R} M') \text{ in } \mathrm{OCC} \Rightarrow \Gamma_\mathcal{R} \vdash !!(M =_\mathcal{R} M') \text{ in } \mathrm{OCC}_{\alpha\eta}.$$

Furthermore, it can be easily verified that each of the steps of our translation, namely (1), (2), (3a), and (3b) preserves the computational equality in the sense that for all $i, i{-}1 \in \{n, \ldots, 1\}$ we have

$$\Gamma_i \vdash !!(M =_\mathcal{R} M') \text{ in } \mathrm{OCC}_{\alpha\eta} \Rightarrow \Gamma_{i-1} \vdash !!(M =_\mathcal{R}^+ M') \text{ in } \mathrm{OCC}_{\alpha\eta}.$$

The use of $=_\mathcal{R}^+$ instead of $=_\mathcal{R}$ on the right hand side is needed only for the steps (1) and (2), because terms that appear in the condition are moved to the left hand side of the computational equation so that computations required to solve the condition need to be performed before the equation is applied. Furthermore, it is easy to see that

$$\Gamma_1 \vdash !!(M =_\mathcal{R} M') \text{ in } \mathrm{OCC}_{\alpha\eta} \Rightarrow \exists M''\,.\,M \updownarrow_\beta^\eta \to_\mathcal{R} M'' \wedge M'' \equiv_{\alpha\beta\eta} M',$$

and by postponing α- and η-conversion we obtain

$$\Gamma_1 \vdash !!(M =_\mathcal{R}^+ M') \text{ in } \mathrm{OCC}_{\alpha\eta} \Rightarrow \exists M''\,.\,M \updownarrow_\beta^\eta \to_\mathcal{R}^+ M'' \wedge M'' \equiv_{\alpha\beta\eta} M'.$$

Although this theorem does not require confluence or termination properties it should be clear that each implementation of OCC assumes (equational) confluence and termination for terms involved in actual computations. Recall, however, that the consistency of OCC is established completely independent of such assumptions.

4.7 Beyond Higher-Order Patterns

Last but not least, we show that our translation cannot only be applied to HRS, but actually covers a more general class. The following equation does not fit into the restricted form of HRS or CRS, because `H` is applied to a pattern which is not simply a variable. As pointed out in e.g. [16] it might be desirable to extend HRS/CRS to include rules like this.

```
f : (nat -> nat) -> nat .  g : (nat -> nat) -> nat .

eq : !! {H : nat -> nat}
        (f ([X : nat] (H (suc X)))) = (g ([X : nat] (H X))) .

eq : !! {H : nat -> nat}{L : nat -> nat}
        ([X : nat] (H (suc X))) := L ->
        (f L) = (g ([X : nat] (H X))) .

eq : !! {H : nat -> nat}{L : nat -> nat}
        ({X : nat} (H (suc X)) := (L X)) ->
        (f L) = (g ([X : nat] (H X))) .

red (f ([x : nat] (suc (suc x)))) .
result: (g ([X : nat] (suc X)))
```

5 Higher-Order Abstract Syntax in a Classical Setting

We cannot consider the higher-order abstract syntax specification of Section 4.3 as an inductive definition, because of the negative occurrence of `term` in the argument type of `abs`. Another well-known problem with the specification above is that the assumption of a standard elimination principle would allows us to construct elements in `term` that do not represent terms in the λ-calculus, and the same problem occurs with the induction principle if we assume the axiom of unique choice [13]. An even more serious problem with that specification appears if we need β-equality for the represented λ-terms. Our addition of β-equality as an axiom requires `abs` to be injective, which is clearly impossible, or more precisely the specification does not have a model under a classical set-theoretic semantics.

In view of these difficulties it is not immediately clear how the idea of higher-order abstract syntax could be fruitfully used under a classical set-theoretic semantics. Obviously the full set-theoretic function space `(term -> term)`, which is used as a domain of `abs`, contains functions that do not represent the body of a λ-abstraction. If we wish to maintain the computational benefits of higher-order abstract syntax, a possible solution is to refine the previous specification and to be precise about what elements of `term` and `term -> term` are suitable representatives of terms and abstraction bodies. To capture this information we specify two predicates `term?` and `body?` as follows. The main idea behind the last axiom below is that a function in `term -> term` represents a body if it is uniform in the sense that it respects each conceivable (partial) congruence `eq`.

```
term?_ax_1 : ?? (term? const) .
term?_ax_2 : ?? {M,N : term} (term? M) -> (term? N) -> (term? (app M N)) .
term?_ax_3 : ?? {B : term -> term} (body? B) -> (term? (abs B)) .

body?_eq : ?? {B : term -> term}
  ({X,Y : term}{eq : term -> term -> Prop}
    (?? (eq const const)) ->
    (?? {M,N,M',N' : term}
        (eq M M') -> (eq N N') ->
        (eq (app M N) (app M' N'))) ->
```

```
    (?? {B,B' : term -> term}
        ({X,Y : term}(eq X Y) -> (eq (B X) (B' Y))) ->
        (eq (abs B) (abs B'))) ->
    (?? (eq X Y)) -> (eq (B X) (B Y))) ->
  (body? B) .

ver (term? (abs ([x : term] (abs ([y : term] x))))) .
all goals solved
```

Notice that the condition `({X,Y : term}{eq : term -> term -> Prop} ...)` of the operational assertion `body?_ax` has a universal quantifier which operationally amounts to the generation of fresh variables `X`,`Y`, and `eq`, for which the condition `...` is verified.

The only change w.r.t. to our previous representation of β- and η-equality is the need for a few extra conditions. For instance, the computational equation for β-reduction becomes:

```
beta : !! {B : (term -> term)}{M : term} (body? B) -> (term? M) ->
   ((app (abs B) M) := (B M)) .

red (app (abs ([x : term] x)) (abs ([x : term] x))) .
result: (abs ([x : term] x))
```

Initial models of this specification can be constructed, e.g. using an approach similar to [11] based on de Bruijn indices.

6 Conclusion

In all our examples the use of conditional equations allows us to avoid binding constructs in patterns. The patterns that we use on the left hand side of computational and assertional equations are essentially first-order, meaning that they constitute a very simple form of higher-order patterns that for all nontrivial cases can be solved by first-order matching. It seems that this rather modest approach constitutes an interesting intermediate stage between purely first-order rewriting and the notationally more convenient higher-order approaches to rewriting. We have shown that the most commonly used HRS can be simulated in OCC, and a simulation in the opposite direction is possible under some natural restriction on the strategy that is employed to solve conditions. This restriction corresponds to the concept of matching equations, which can be found in the recent version of Maude [5] and have also been implemented in the new version of the OCC prototype. We furthermore found that our translation applies to a class that is somewhat more general than HRS, but the precise characterization of this class is left as future work.

We have also illustrated that rewriting in OCC is more general for various other reasons, the increased generality of the type system, the possible use of structural equations, the use of conditions giving rise to assertional propositions with an operational semantics based on goal-oriented proof search and equational simplification. The latter has been used to deal with higher-order abstract syntax in a classical framework in a way that seems to be beyond the capabilities of HRS.

The underlying assumption of this paper is that HRS are interpreted as the specification of an equational theory, similar to the spirit of higher-order algebraic specifications [14]. A less abstract interpretation of HRS is to consider them as the specification of a transition system, similar to the rewrite rules of rewriting logic [20]. Some recent work on representing π-calculus using higher-order abstract syntax [31] suggests that this interpretation might be useful for symbolic analysis of systems at a level more abstract (because it does not distinguish α-equivalent states) than a naive first-order representation. In OCC this would amount to using computational rewrite axioms rather than computational equations. Furthermore, in this case the higher-order rewrite rules cannot impact the solutions of equational conditions, which should make the correspondence to HRS even easier to establish.[7]

References

1. H. P. Barendregt. Lambda-calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*. Claredon Press, Oxford, 1992.
2. Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, September 1996.
3. E. Bonelli, D. Kesner, and A. Ríos. From higher-order to first-order rewriting. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA'01), Utrecht, The Netherlands, June 2001*, volume 2051 of *LNCS*, pages 47–62. Springer-Verlag, 2001.
4. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Towards Maude 2.0. In K. Futatsugi, editor, *The 3rd International Workshop on Rewriting Logic and its Applications, Kanazawa City Cultural Hall, Kanzawa, Japan, September 18–20, 2000, Proceedings*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 297 – 318. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.
6. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.
7. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.
8. P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.
9. P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In J.-Y. Girard, editor, *Typed Lambda Calculi and Applications. Proceedings of TLCA'99, L'Aquila, Italy*, volume 1581 of *LNCS*, pages 129 – 146. Springer-Verlag, 1999.
10. T. Hardin G. Dowek, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming, Bonn, Germany, September 1996*, pages 259–273. MIT Press, 1996.
11. A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*, volume 1125, pages 173–190. Springer-Verlag, 1996.
12. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science, Ithaca, New York, 22–25 June 1987, Proceedings*, pages 193–204. IEEE, 1987.

[7] Suitable congruence rules for computational rewrite judgements in OCC are needed, however, to allow the comparison with HRS on an equal footing.

13. M. Hoffmann. Semantical analysis of higher-order abstract syntax. In *14th Annual Symposium on Logic in Computer Science, Trento, Italy, 2–5 July 1999, Proceedings*, pages 214–224. IEEE, July 19901999.
14. J.-P. Jouannaud. Executable higher-order algebraic specifications. In C. Choffrut and M. Jantzen, editors, *STACS 91, 8th Annual Symposium on Theoretical Aspects of Computer Science, Hamburg, Germany, February 14-16, 1991, Proceedings*, volume 480 of *LNCS*. Springer, 1991.
15. J. W. Klop. *Combinatory Reduction Systems*. PhD thesis, Rijksuniversiteit Utrecht, June 1980. Mathematical Centre Tracts 127.
16. J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1–2):279–308, December 1993.
17. Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. International Series of Monographs on Computer Science. Oxford University Press, 1994.
18. S. MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.
19. R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(1):3–29, 1998.
20. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
21. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497 – 536, 1991.
22. Y. Moschovakis. *Notes on set theory*. Springer-Verlag, 1994.
23. T. Nipkow. Higher-order critical pairs. In *Sixth Annual IEEE Symposium on Logic in Computer Science, Amsterdam, The Netherlands, 15–18 July 1991, Proceedings*. IEEE, 1991.
24. Tobias Nipkow and Christian Prehofer. Higher-order rewriting and equational reasoning. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications. Volume I: Foundations*, volume 8 of *Applied Logic Series*, pages 399–430. Kluwer, 1998.
25. K. Petersson, J. Smith, and B. Nordstroem. *Programming in Martin-Löf's Type Theory. An Introduction*. International Series of Monographs on Computer Science. Oxford: Clarendon Press, 1990.
26. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, 22-24 June 1988*, SIGPLAN Notices 23(7), pages 199–208, 1988.
27. J. R. Shoenfield. Axioms of set theory. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 321–344. North-Holland, 1977.
28. M.-O. Stehr. CINNI – A Generic Calculus of Explicit Substitutions and its Application to λ-, σ- and π-calculi. In K. Futatsugi, editor, *The 3rd International Workshop on Rewriting Logic and its Applications, Kanazawa City Cultural Hall, Kanzawa, Japan, September 18–20, 2000, Proceedings*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71 – 92. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.
29. M.-O. Stehr. Programming, Specification, and Interactive Theorem Proving — Towards a Unified Language based on Equational Logic, Rewriting Logic, and Type Theory. Doctoral Thesis, Universität Hamburg, Fachbereich Informatik, Germany, 2002. `http://www.sub.uni-hamburg.de/disse/810/`.
30. M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic. In *From Object-Orientation to Formal Methods: Dedicated to The Memory of Ole-Johan Dahl*, volume 2635 of *LNCS*. Springer-Varlag, 2004.
31. A. Tiu and D. Miller. A proof search specification of the π-calculus. April 2004.
32. V. van Oostrom and F. van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In *Proceedings of the International Workshop on Higher-Order Algebra, Logic, and Term Rewriting (HOA '93), Amsterdam, September 1993*, volume 816 of *LNCS*, pages 276 – 304. Springer-Verlag, 1993.
33. F. van Raamsdonk. *Confluence and normalization for higher order rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, 1996.
34. B. Werner. Sets in types, types in sets. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS'97, Sendai, Japan, September 23–26, 1997, Proceedings*, volume 1281 of *LNCS*. Springer-Verlag, 1997.

Part II: Regular Talks

FD à la Melliès

Vincent van Oostrom

Department of Philosophy, Universiteit Utrecht, The Netherlands
Vincent.vanOostrom@phil.uu.nl

Abstract. As observed by Hyland, the Finite Developments Theorem follows from the fact that the residuals of a redex will never *nest* one another along a development. We generalise this method to combinatory reduction systems by replacing nesting by Melliès' *holding* relation.

1 Introduction

A development is a rewrite sequence along which only residuals of redexes in its source are contracted. The Finite Developments Theorem (FD) states that all developments terminate. Various proofs of FD for various rewriting formats can be found in the literature. In this note, we analyse a proof due to Hyland [1] for the λ-calculus and extend it to Klop's combinatory reduction systems.

The problem in proving FD is to control the way in which residuals may replicate one another. To that end, we introduce a 'may replicate' relation between residuals, which should be preserved by rewriting. In particular, the residuals of one and the same redex should never be related to one another.

Definition 1 *A binary relation on redex(occurrenc)es is called* parting *for a class of rewriting systems, if it can never be satisfied by the residuals of any redex along any rewrite sequence.*

Once an appropriate such relation has been found, the idea is to weigh a redex by the redexes which it may replicate, and show that the total weight decreases along any rewrite step.

We assume the reader to be familiar with the basic syntactic rewrite theory of first-order term rewriting systems (TRSs), the lambda calculus (Λ), combinatory reduction systems (CRSs), and higher-order pattern rewriting systems (PRSs), in particular with residuals and developments [6].

2 λ-calculus

For the λ-calculus, the above can be instantiated with the nesting relation. A redex is said to *nest* another redex if the latter is a proper subterm of the former. Parting of nesting is known as the disjointness property (DP [2, Def. I.4.3.1]).

Remark 2 *1. Nesting is trivially parting for TRSs since subterms can only be duplicated, not nested, by performing a rewrite step [2, Rem. I.4.3.2].*

2. As remarked there as well, nesting is non-parting for the lambda calculus, hence for PRSs, as witnessed by the rewrite sequence $(\lambda x.xx)\lambda y.R(y) \to_\beta (\lambda y.R(y))\lambda y.R(y) \to_\beta R(\lambda y.R(y))$*, where* $R(M) =^{\mathrm{def}} (\lambda x.M)N$*. Note that in the second step of this sequence a redex* created *in the first step has been contracted.*

As shown in [1] contraction of a created redex, as in the second item of the remark, is necessary in order to nest residuals of a redex. We present a simple proof of this fact, employing the following well-known formalisation of developments for the λ-calculus.

Definition 3 *The set* $\underline{\Lambda}$ *of* underlined *lambda terms is defined by:*

($\underline{var}$*)* $x \in \underline{\Lambda}$*, for all (countable many) variables* x*,*
($\underline{app}$*)* $M, N \in \underline{\Lambda} \implies MN \in \underline{\Lambda}$*,*
($\underline{abs}$*)* $M \in \underline{\Lambda} \implies \lambda x.M \in \underline{\Lambda}$*,*
($\underline{beta}$*)* $M, N \in \underline{\Lambda} \implies (\underline{\lambda} x.M)N \in \underline{\Lambda}$*.*

The rewrite relation on $\underline{\Lambda}$ *is generated by the rule:* $\underline{\boldsymbol{\beta}}\colon (\underline{\lambda} y.P)Q \to P^{[y \mapsto Q]}$*.*

A development in the ordinary lambda calculus Λ is defined as a rewrite sequence in the underlined lambda calculus $\underline{\Lambda}$ from which the underlining has been removed. We want to prove that nesting is parting for $\underline{\Lambda}$. Let P denote the subset of $\underline{\Lambda}$ for which nesting is parting.

Lemma 4 ([1]) $\underline{\Lambda} = \mathrm{P}$.

Proof One proves *for all* $\underline{\Lambda}$*-terms* M *and substitutions* σ *mapping the free variables of* M *to terms in* P*, it holds that* $M^\sigma \in \mathrm{P}$, by induction on the derivation of $M \in \underline{\Lambda}$, from which the lemma follows taking the identity for σ (variables are in P). The only interesting case is for underlined beta redexes.

($\underline{beta}$) $((\underline{\lambda} x.M)N)^\sigma = (\underline{\lambda} x.M^\sigma)N^\sigma$. M^σ and N^σ are in P by the induction hypothesis, so a non-P rewrite sequence must be of the form $(\underline{\lambda} x.M^\sigma)N^\sigma \twoheadrightarrow_{\underline{\beta}} (\underline{\lambda} x.M')N' \to_{\underline{\beta}} M'^{[x \mapsto N']} \twoheadrightarrow \ldots$. Since (by the substitution lemma) $(\underline{\lambda} x.M^\sigma)N^\sigma \to_{\underline{\beta}} M^{\sigma[x \mapsto N^\sigma]} \twoheadrightarrow_{\underline{\beta}} M'^{[x \mapsto N']}$ and the middle term is in P by the induction hypothesis, it suffices to show P is reflected by the first step. For this, the only problem might be that redexes in distinct copies of N^σ eventually get nested, but since no variable in any copy is bound by a lambda outside it the copies remain non-nested, hence the residuals in them as well. (Formally one shows that if $M^\sigma \to_{\underline{\beta}} \hat{M}$, then $\hat{M} = M'^{\sigma'}$ and either $M \to_{\underline{\beta}} M'$, $\sigma = \sigma'$ or $M = M'^\rho$, $\rho; \sigma \to_{\underline{\beta}} \sigma'$ for renaming ρ.) $\square$

Remark that substituting SN (strong normalisation) for P yields the proof of FD à la Tait for Λ as presented in [7].[1] Here we focus, as outlined in the introduction, on an alternative way to derive FD, using the lemma as a black box. The proof is originally due to Micali (see [2, Lem. I.4.3.3]). We give an account of that proof replacing his multiset argument by an appeal to recursive path orders (RPO).

Proof (of FD) $\underline{\Lambda}$-terms are interpreted as first-order terms via the mapping T defined by

$$\begin{aligned} T(x) &=^{\mathrm{def}} c \\ T(MN) &=^{\mathrm{def}} @(T(M), T(N)) \\ T(\lambda x.M) &=^{\mathrm{def}} T(M) \\ T((\underline{\lambda} x.M)N) &=^{\mathrm{def}} i(T(M), T(N)) \end{aligned}$$

where the *weight* i of an underlined beta redex $(\underline{\lambda} x.M)N$ is the number of distinct residuals (i.e. residuals of initially distinct redexes) it nests. By choosing the precedence $j > i > @, c$, for all $j > i$, we have $M \to_\beta N \implies T(M) >_{RPO} T(N)$, since by P the weight of a redex is always larger than the weight of all redexes it nests, from which FD follows. $\square$

[1] The proof of FD is actually simpler than the proof of DP since the reflection argument is not needed. It is the shortest proof we know of.

3 Combinatory reduction systems

The problem we are faced with is to find a relation which is parting for developments in combinatory reduction systems. We will show that the notion of gripping as introduced by Melliès, in his abstract approach to FD [3], meets the requirements.

Remark 5 *In the case of CRSs nesting is non-parting even for developments as remarked in [2, Rem. II.2.13], and witnessed by the CRS:*

$$\begin{aligned}\mu([x]Z(x)) &\to Z(\mu([x]Z(x)))\\ a(x) &\to b(x)\end{aligned}$$

After the step $\underline{\mu}([x]\underline{a}(x)) \to \underline{a}(\mu([x]\underline{a}(x)))$ *the residual of the* a*-redex on the left nests the one on the right.*

A redex is said to *grip* another redex if the latter nests a variable bound by the (pattern of the) former. *Holding* is the transitive closure of gripping. (In the initial term of previous example, the μ-redex holds the a-redex.)

Remark 6 *Holding is trivially parting for TRSs, but is non-parting for* Λ*, hence for PRSs, as witnessed by the reduction* $(\lambda x.xx)\lambda y.R(y) \to_\beta (\lambda y.R(y))\lambda y.R(y) \to_\beta R(\lambda y.R(y)) \to_\beta (\lambda x.R(x))N$*, where* $R(M) =^{\text{def}} (\lambda x.Mx)N$*.*

We prove that holding is parting for developments in CRSs. To that end, the definition of underlined system is generalised to Nipkow's pattern rewrite systems (PRSs), hence applies to CRSs as the latter are a special case of the former.

Definition 7 *The underlined version* $\underline{\mathcal{P}}$ *of a PRS* $\mathcal{P}$ *is:*

1. *The alphabet of* $\underline{\mathcal{P}}$ *is the alphabet of* $\mathcal{P}$ *extended with a function symbol* $\underline{l \to r}$ *of type* θ *for any rule* $l \to r$ *of type* θ *in* $\mathcal{P}$*.*
2. *For rule* $l \to r$ *in* $\mathcal{P}$*, there's a rule* $\underline{l \to r} \to r$ *in* $\underline{\mathcal{P}}$*.*

Let P denote the subset of $\underline{\mathcal{P}}$ for which holding is parting, and $\mathrm{P}^{\to}$ its subset consisting of terms of the form $\boldsymbol{X}.\mathtt{a}(\boldsymbol{M})$ such that $\mathtt{a}(\boldsymbol{M})^\sigma \in \mathrm{P}$ for every substitution $\sigma \in \mathrm{P}^{\to}$ (on a subset of $\boldsymbol{X}$). This is well-defined since the types of the terms in σ are smaller than the type of $\boldsymbol{X}.\mathtt{a}(\boldsymbol{M})$. The set P^n is obtained by bounding substitutions by order n in the definition of $\mathrm{P}^{\to}$.

Lemma 8 *Let* $\mathcal{P}$ *be a CRS, then every* $\underline{\mathcal{P}}$ *term is in* P*.*

Proof We prove that *every* $\boldsymbol{X}.\mathtt{a}(\boldsymbol{M})$ *such that* $\boldsymbol{X}$ *is a vector of* metavariables *and* $\mathtt{a}(\boldsymbol{M})$ *a* metaterm *(in CRS-jargon) is in* P^2*, i.e. that for any substitution (*substitute *in CRS jargon)* $\sigma \in \mathrm{P}^1$*,* $\mathtt{a}(\boldsymbol{M})^\sigma \in \mathrm{P}$ by induction, first on the number of underlined symbols and then on the number of head symbols in the term, from which the lemma follows by taking the identity for σ (variables are trivially in P^1, hence we may also assume that $\boldsymbol{X}$ is the domain of σ)[2]. By induction hypothesis we know that $\boldsymbol{X}.M_i \in \mathrm{P}^2$ for every M_i among $\boldsymbol{M}$. The proof is by cases on $\mathtt{a}$, the only interesting case being the rule case.

[2] In fact, this shows P for underlined CRS metaterms implying P for underlined CRS terms.

(rule) If $\mathtt{a}$ is an underlined symbol $\underline{l \to r}$ for some rule $l \to r$ of $\mathcal{P}$, then since its arguments are in P^2 by the induction hypothesis, a rewrite sequence witnessing non-P must look like $\underline{l \to r}(\boldsymbol{M}^\sigma) \twoheadrightarrow \underline{l \to r}(\boldsymbol{M'}) \to r(\boldsymbol{M'}) \to \ldots$. Since (by the substitution lemma) $\underline{l \to r}(\boldsymbol{M}^\sigma) \to r(\boldsymbol{M}^\sigma) \twoheadrightarrow r(\boldsymbol{M'})$ and the middle term is in P by the induction hypothesis, it suffices to show P is reflected by the first step. For this, the only problem might be that redexes in distinct copies of some $M_i{}^\sigma$ eventually hold one another, but since no second-order variable in $M_i{}^\sigma$ is bound by a λ outside it,[3] the copies remain non-holding, hence the residuals in them as well. (Formally one shows that if $M^\sigma \to \hat{M}$ for non-underlined metaterm M, substitute σ, and (underlined) term $\hat{M}$, then $\hat{M} = M'^{\sigma'}$ with $M = M'^{\rho}$, and $\rho;\sigma \to \sigma'$ for renaming ρ.) $\square$

Again, substituting SN for P (and omitting the reflection argument) yields a proof of FD for CRSs (cf. [6]). Here we focus, as outlined in the introduction, on an alternative way to derive FD for CRSs, using the lemma as a $\blacksquare$,[4] which is due to Melliès [3]. We present a slight modification of that proof replacing his multiset argument by an appeal to RPO.

Proof (of FD) $\underline{\mathcal{P}}$-terms are interpreted as first-order terms via the mapping T.

$$\begin{aligned} T(\boldsymbol{X}.Y(\boldsymbol{M})) &=^{\mathrm{def}} c(T(\boldsymbol{M})) \\ T(\boldsymbol{X}.f(\boldsymbol{M})) &=^{\mathrm{def}} f(T(\boldsymbol{M})) \\ T(\boldsymbol{X}.\underline{l \to r}(\boldsymbol{M})) &=^{\mathrm{def}} i(T(\boldsymbol{M})) \end{aligned}$$

where the *weight* i is the number of distinct redexes the redex $\underline{l \to r}(\boldsymbol{M})$ holds[5]. Taking the precedence $j > i > f, c$, for all $j > i$, we show that $M \to N \implies T(M) >_{RPO} T(N)$. The correspondence between this proof and the proof in [4, Ch. 3], based on a number of axioms (A,B,C,fd-1,fd-2,fd-3,fd-4,Z-1, and Z-2), is illustrated by decorating the proof steps with the corresponding axioms.[6]

One shows: $M = C[l^\sigma] \to C[r^\sigma] = N \implies C_T[l_T^{\sigma_T}] >_{RPO} C'_T[r_T^{\sigma'_T}]$, *for some* $C_T \geq_{RPO} C'_T$, $l_T >_{RPO} r_T$, *and* $\sigma_T \geq_{RPO} \sigma'_T$. This decomposition is achieved as follows:

(Ctx) We can take for C_T and C'_T just the parts of the first-order terms ($T(M)$ and $T(N)$) corresponding to the context C, in which weights can only decrease by rewriting due to P (Axiom fd-3), hence $C_T \geq_{RPO} C'_T$.

(Rule) For l_T, we take that part of the first-order term which corresponds to the redexes which are held by l, and for r_T the descendant of that part. Firstly, we show that the so-constructed parts are convex. For l_T this is clear since redexes inbetween the contracted redex and a redex held by it, are held as well (Axiom fd-4). For r_T this holds since for a residual u' of u above a residual v' of a redex v in l_T, we must have that either u itself is held by the contracted redex and belongs to l_T so u' belongs to r_T (Axiom fd-2), or u belongs to C_T and u' belongs to C'_T. By P (Axiom Z-1), the weights of all residuals in r_T are strictly dominated by the weight of the left-hand side l_T, hence $l_T >_{RPO} r_T$.

(Sub) For σ_T and σ'_T, the remaining parts of the first-order terms are collected. Since none of the subterms in σ_T is held by the contracted redex, its descendants can only

[3] This is exactly what fails to be true in third-order PRSs.

[4] Actually, the FD proof for CRSs in [4, Sec. 7.3.3] is based on the (weaker) property that gripping is acyclic.

[5] Formally c and the is have to be indexed by the number of arguments they take.

[6] The correspondence is not exact (the axioms are much more abstract, e.g. they only concern redexes and their residuals, but the RPO proof employs terms containing non-redex function symbols as well), hence some proofsteps here are not essential, and are not backed up by an axiom in the abstract approach.

have been duplicated (Axiom FD-1), and the weights of the redexes in them has not changed, hence $\sigma_T \geq_{RPO} \sigma'_T$. □

4 Pattern rewrite systems?

Surprisingly, holding need not be parting for developments in orthogonal PRSs in general. Consider the underlined version of the following (orthogonal) PRS:

$$g(M.N.X(x.M(x), N)) \to_\gamma X(y.X(x.y, I), I)$$
$$(\lambda x.M(x))N \to_\beta M(N)$$

(where λ and g are unary function symbols and the binary application operator @ is dropped from the left-hand side of the β-rule as in combinatory logic.) After the step $\underline{g}(M.N.(\underline{\lambda}x.M(x)N)) \to_\gamma (\underline{\lambda}x.(\underline{\lambda}y.x)I)I$ the residual of the β-redex on the left holds the one on the right. This example shows at the same time that the axioms for FD of [4, Ch. 3], are not verified in general by (non-second-order) PRSs.[7]

Since the direct inductive proof of FD above generalises without effort to PRSs [6], one could expect to lift the typical (combinatorial) second-order property of gripping to a (logical) property which holds for all orders, and show that that property is parting for developments in any orthogonal PRS. We leave this to further research.

5 Conclusion

We've indicated some relationships between different approaches to the finiteness of developments theorem. The axiomatic approach due to Melliès which is known to apply to CRSs [4], was shown not to apply (without further ado) to PRSs. This is somewhat surprising considering the close correspondence between CRSs and HRSs as established in [5], but as remarked there as well, although CRSs and PRSs have the same 'matching power', the latter have more 'rewriting power'. From a technical point of view, properties need to be defined (and checked) for CRSs only up to order 2 (their substitution calculus is second-order),[8] whereas for PRSs they need to be defined for any order (their substitution calculus is simply typed lambda calculus).

References

1. J.M.E. Hyland. A simple proof of the Church-Rosser theorem, 1973. Typescript.
2. J.W. Klop. *Combinatory Reduction Systems.* Proefschrift, Rijksuniversiteit Utrecht, 1980.
3. P.-A. Melliès. An abstract finite developments theorem. Unpublished, 1993.
4. P.-A. Melliès. *Description Abstraite des Systèmes de Réécriture.* Thèse de doctorat, Université Paris VII, 1996.
5. V. van Oostrom and F. van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. pp. 276–304, HOA'93, 1993.
6. Terese. *Term Rewriting Systems.* Cambridge University Press, 2003.
7. V. van Oostrom. Take five. Technical Report IR-406, Vrije Universiteit, Amsterdam, June 1996.

[7] To be precise, the axioms fd-3 and either acyclicity of gripping or axiom Z-1 cannot hold at the same time.

[8] CRSs are second-order in the precise sense that embedding them into PRSs as in [5] yields rules having variables ranging over first-order functions (of type $o \to \cdots \to o$) at the highest.

Strong Normalization in the ρ-cube: the Simply-Typed System

Benjamin Wack

LORIA & Université Henri Poincaré, Nancy, France
Benjamin.Wack@loria.fr

1 Introduction

The *rewriting calculus* (or ρ-calculus) embeds the λ-calculus and the rewriting in a single formalism. It endows all the basic ingredients of the rewriting with a status of first-class citizens. In [4], a collection of type systems for the ρ-calculus was presented, extending Barendregt's λ-cube to a ρ-cube. For these type systems and for the similar formalism of P^2TS, the basic typing properties have been studied in [2,5]. Still, strong normalization did remain an open problem for the type systems of the ρ-cube.

In this paper, we give a first positive answer to this problem. Together with the consistency of normalizing terms, already proved in [2], this result makes the ρ-calculus a good candidate for a proof-term language integrating deduction and computation.

Conventions and notations Generally, the reader can assume that every capital letter denotes an object belonging to the ρ-calculus, and every small letter denotes an object belonging to the λ-calculus. Syntactic equivalence of terms will be denoted by $\equiv$. If a substitution θ has domain $X_1 \dots X_n$ and $\forall i, \theta(X_i) \equiv A_i$, we will also write it $[X_1 := A_1 \dots X_n := A_n]$. To denote a tuple of terms $B_1 \dots B_n$, we will use the vector notation $\overrightarrow{B}$. To avoid confusion between arrows, we will use the symbol $\twoheadrightarrow$ for the abstraction arrow of the ρ-calculus (which is the same for terms and types).

2 The languages

2.1 The rewriting calculus and the ρ-cube

The syntax of the ρ-calculus extends that of the typed λ-calculus with structures and patterns [4]. Several choices can be made for the set of patterns P: in this paper, *we only consider algebraic patterns*, whose shape is defined below.

$$
\begin{array}{ll}
\mathit{Signature} & \Sigma ::= \emptyset \mid \Sigma, f : A \\
\mathit{Context} & \Gamma ::= \emptyset \mid \Gamma, X : A \\
\mathit{Pattern} & P ::= X \mid f\bullet\overrightarrow{P} \\
\mathit{Term} & A ::= f \mid X \mid P \twoheadrightarrow_\Delta A \mid [P \ll_\Delta A]A \mid A\bullet A \mid A; A
\end{array}
$$

The notion of free variables can be adapted for the ρ-calculus: in an abstraction $P \twoheadrightarrow_\Delta B$, all the variables appearing in the pattern P are bound. Similarly, in $[P \ll_\Delta B]A$, the variables of P are bound and may occur in A. Extending Church's notation, the context Δ

contains the type declarations of the variables appearing in P. As usual, we work modulo α-*conversion* and we adopt Barendregt's "*hygiene-convention*" [1], *i.e.* free and bound variables have different names.

The ρ-calculus features pattern abstractions whose application requires solving matching problems. For the purpose of this paper, we consider only syntactic matching:

$$\begin{array}{lrll}(\rho) & (P \twoheadrightarrow_\Delta A)\bullet B & \to_\rho & [P \ll_\Delta B]A \\ (\sigma) & [P \ll_\Delta B]A & \to_\sigma & A\theta_{(P \ll B)} \text{ if } P\theta_{(P \ll B)} \equiv B \\ (\delta) & (A;B)\bullet C & \to_\delta & A\bullet C; B\bullet C\end{array}$$

Let us briefly recall the various type systems that were proposed so far for the ρ-calculus:

- In [3], a first strongly normalizing type system was introduced; however, the proof of normalization is based on a heavy restriction over the types of constants.
- In [6], we studied another type system allowing a broader class of constants and enforcing subject reduction, but also allowing to typecheck some terms with infinite reductions.
- The type systems of [4,2] were inspired from Barendregt's λ-cube, and were designed for logical purposes. Until now, strong normalization was an open problem for these systems. Here, we show this property for a slight variation of [4]. The simply-typed system is recalled in Fig. 1. However, this is not a standard simply-typed system, in the sense that patterns occur in the (left-hand sides of the) types; it would not be equivalent to write "arrow-types" like in the simply-typed λ-calculus.

$$\frac{\Sigma, \Gamma \vdash_\rho A : \Phi \quad \Sigma, \Gamma \vdash_\rho B : \Phi}{\Sigma, \Gamma \vdash_\rho A; B : \Phi}\ (\textsc{Struct}) \qquad \frac{\Sigma, \Gamma \vdash_\rho A : \Psi \quad \Sigma, \Gamma \vdash_\rho \Phi : * \quad \Phi =_{\rho\delta} \Psi}{\Sigma, \Gamma \vdash_\rho A : \Phi}\ (\textsc{Conv})$$

$$\frac{\Sigma, \Gamma \vdash_\rho \Phi : * \quad \alpha \notin \mathcal{D}om(\Sigma, \Gamma)}{\Sigma, \Gamma, \alpha{:}\Phi \vdash_\rho \alpha : \Phi}\ (\textsc{Atom}) \qquad \frac{\Sigma, \Gamma \vdash_\rho A : \Phi \quad \Sigma, \Gamma \vdash_\rho \Psi : * \quad \alpha \notin \mathcal{D}om(\Sigma, \Gamma)}{\Sigma, \Gamma, \alpha{:}\Psi \vdash_\rho A : \Phi}\ (\textsc{Weak})$$

$$\frac{\Sigma, \Gamma, \Delta \vdash_\rho A : \Phi \qquad \Sigma, \Gamma \vdash_\rho P \twoheadrightarrow_\Delta \Phi : *}{\Sigma, \Gamma \vdash_\rho P \twoheadrightarrow_\Delta A : P \twoheadrightarrow_\Delta \Phi}\ (\textsc{Abs}) \qquad (\text{above } \alpha \text{ can be } X \text{ or } f)$$

$$\frac{\Sigma, \Gamma \vdash_\rho A : P \twoheadrightarrow_\Delta \Phi \quad \Sigma, \Gamma \vdash_\rho [P \ll_\Delta B]\Phi : *}{\Sigma, \Gamma \vdash_\rho A\bullet B : [P \ll_\Delta B]\Phi}\ (\textsc{App}) \qquad \frac{\Sigma, \Gamma, \Delta \vdash_\rho A : \Phi \quad \Sigma, \Gamma \vdash_\rho [P \ll_\Delta B]\Phi : *}{\Sigma, \Gamma \vdash_\rho [P \ll_\Delta B]A : [P \ll_\Delta B]\Phi}\ (\text{DMC})$$

$$\frac{\forall (X{:}\Psi) \in \Delta, \quad \Sigma, \Gamma, \Delta \vdash_\rho \Psi : * \qquad \Sigma, \Gamma, \Delta \vdash_\rho P : \Psi_0 \qquad \Sigma, \Gamma, \Delta \vdash_\rho \Phi : *}{\Sigma, \Gamma \vdash_\rho P \twoheadrightarrow_\Delta \Phi : *}\ (\textsc{Prod})$$

$$\frac{\forall (X{:}\Psi) \in \Delta, \ \Sigma, \Gamma, \Delta \vdash_\rho \Psi : * \quad \Sigma, \Gamma, \Delta \vdash_\rho P : \Psi_0 \quad \Sigma, \Gamma \vdash_\rho B : \Psi_0 \quad \Sigma, \Gamma, \Delta \vdash_\rho \Phi : *}{\Sigma, \Gamma \vdash_\rho [P \ll_\Delta B]\Phi : *}\ (\text{DMCSort})$$

Fig. 1. The simply-typed system of the ρ-cube

Theorem 1 (Subject reduction) *If $\Gamma \vdash_\rho A : \Phi$ and $A \mapsto\!\!\!\!\!\rightarrow_{\rho\delta} A'$, then $\Gamma \vdash_\rho A' : \Phi$.*

Notice that the λ-calculus is directly embedded in the ρ-calculus, by considering only variables for patterns. The systems of the ρ-cube are also a conservative extension of the λ-cube.

2.2 The System Fω

The type system Fω was first introduced and studied by Girard; it allows one to define terms depending on terms, terms depending on types and types depending on types. For a detailed presentation, the interested reader can refer to [1].

$$\begin{array}{ll} \mathit{Kinds} & k ::= * \mid k \to k \\ \mathit{Types} & \tau ::= \beta \mid \Pi(x:\tau).\tau \mid \lambda(\beta:k).\tau \mid \tau\tau \\ \mathit{Terms} & M ::= x \mid \lambda(x:\tau).M \mid M\,M \mid \lambda(\beta:k).M \mid M\tau \end{array}$$

Theorem 2 (Subject reduction) *If $\Gamma \vdash_{\mathsf{F}\omega} t : \sigma$ and $t \mapsto\!\!\!\!\rightarrow_{\beta} t'$, then $\Gamma \vdash_{\mathsf{F}\omega} t' : \sigma$.*

Theorem 3 (Strong normalization) *If $\Gamma \vdash_{\mathsf{F}\omega} t : \sigma$, then t is strongly normalizing.*

3 Untyped encoding

We begin with translating the untyped ρ-terms with algebraic patterns into untyped λ-terms. We use the following notations: S is the number of symbols appearing in the (arbitrarily ordered) signature. The ith symbol of Σ is denoted by f_i. To build the encoding of pattern matching, we need three conditions about constants and structures:

1. each constant f_i has a "maximal" arity α_i, in the sense that f_i has at most α_i arguments (used when translating a constant);
2. in every matching equation $f_i \bullet P_1 \bullet \ldots \bullet P_p \ll f_j \bullet B_1 \bullet \ldots \bullet B_q$, we have $\alpha_i - p = \alpha_j - q$ (used when translating an abstraction);
3. each term $(A; B)$ has a maximal arity α (used when translating a structure).

This notion of arity can be properly defined using types: as in the λ-calculus, the type of a term bounds the number of arguments it can take. The translation is given in Fig. 2.

$$\begin{array}{rcl} [\![X]\!] & \triangleq & X \\ [\![f_i]\!] & \triangleq & \lambda x_1 \ldots \lambda x_{\alpha_i}.\ (\lambda z_1 \ldots \lambda z_S.(z_i\ x_1 \ldots x_{\alpha_i})) \\ [\![A; B]\!] & \triangleq & \lambda x_1 \ldots \lambda x_\alpha.\ \Big((\lambda z.([\![A]\!] x_1 \ldots x_\alpha))([\![B]\!] x_1 \ldots x_\alpha)\Big) \\ [\![X \twoheadrightarrow A]\!] & \triangleq & \lambda X.[\![A]\!] \\ [\![(f_i \bullet P_1 \bullet \ldots \bullet P_p) \twoheadrightarrow A]\!] & \triangleq & \lambda y.(y \underbrace{x_\perp \ldots x_\perp}_{(\alpha_i - p)}\ \underbrace{x_\perp \ldots x_\perp}_{(i-1)}\ [\![P_1 \twoheadrightarrow \ldots P_p \twoheadrightarrow x'_{p+1} \twoheadrightarrow \ldots x'_{\alpha_i} \twoheadrightarrow A]\!]\ \underbrace{x_\perp \ldots x_\perp}_{(S-i)}) \\ [\![A \bullet B]\!] & \triangleq & [\![A]\!][\![B]\!] \\ [\![[P \ll B]A]\!] & \triangleq & \text{the term obtained by head-}\beta\text{-reducing } [\![(P \twoheadrightarrow A) \bullet B]\!] \end{array}$$

Fig. 2. Untyped term translation

Theorem 4 (Faithful reductions) *For any ρ-terms A and B, if $A \mapsto\!\!\!\!\rightarrow_{\rho\delta} B$, then $[\![A]\!] \mapsto\!\!\!\!\rightarrow_{\beta} [\![B]\!]$ in at least one step.*

The most interesting case is the simulation of a successful σ-reduction. An example is given in Fig. 3. The translated reduction is:

$$\begin{array}{rl}(Y \twoheadrightarrow \underline{(f\bullet X \twoheadrightarrow X)\bullet Y})\bullet(f\bullet a) & \mapsto_{\mathcal{P}} \underline{(Y \twoheadrightarrow [f\bullet X \ll Y]X)\bullet(f\bullet a)} \\ & \mapsto_{\mathcal{P}} \underline{[Y \ll f\bullet a][f\bullet X \ll Y]X} \\ & \mapsto_{\sigma} \underline{[f\bullet X \ll f\bullet a]}X \\ & \mapsto_{\sigma} a\end{array}$$

$$\begin{array}{rl} & \overbrace{(\lambda Y.\Big(\overbrace{(\underline{\lambda y}.(y x_\perp(\lambda X.X)))}^{[\![f\bullet X \twoheadrightarrow X]\!]}\underline{Y}\Big))\Big(\overbrace{(\lambda x_1.\lambda z_1\lambda z_2.(z_2 x_1))}^{[\![f]\!]}\overbrace{(\lambda u_1\lambda u_2.u_1)}^{[\![a]\!]}\Big)}^{[\![Y \twoheadrightarrow (f\bullet X \twoheadrightarrow X)\bullet Y]\!]} \\ \mapsto_\beta & (\lambda Y.(Y x_\perp(\lambda X.X)))\Big((\underline{\lambda x_1}.\lambda z_1\lambda z_2.(z_2 x_1))\underline{(\lambda u_1\lambda u_2.u_1)}\Big) \\ \mapsto_\beta & \big(\underline{\lambda Y}.(Y x_\perp(\lambda X.X))\big)\underline{\big(\lambda z_1\lambda z_2.(z_2(\lambda u_1\lambda u_2.u_1))\big)} \\ \mapsto_\beta & \big(\underline{\lambda z_1}\lambda z_2.(z_2(\lambda u_1\lambda u_2.u_1))\big)\underline{x_\perp}(\lambda X.X) \\ \mapsto_\beta & \big(\underline{\lambda z_2}.(z_2(\lambda u_1\lambda u_2.u_1))\big)\underline{(\lambda X.X)} \\ \mapsto_\beta & (\underline{\lambda X}.X)\underline{(\lambda u_1\lambda u_2.u_1)} \\ \mapsto_\beta & (\lambda u_1\lambda u_2.u_1) \\ = & [\![a]\!]\end{array}$$

Fig. 3. Translation of a successful delayed matching

One can notice that the untyped translation also has a an interest in the translation of term rewriting systems into the λ-calculus. Indeed, some classes of TRS together with a given strategy can be automatically translated into the ρ-calculus (see for instance [6] for convergent TRS). Thus, we can design a two-step translation from (a subclass of) TRS into the λ-calculus via the ρ-calculus. We will not treat this matter further in this paper, since we are mainly interested in proving strong normalization for the ρ-calculus.

4 Recovering types in the translation

Here, instead of translating a term into a term, we translate a typed term into a (typable) term. By lack of space, we do not give the full typed translation, but we give examples for two key constructs appearing in the typed translation, explaining why System $\mathsf{F}\omega$ is needed.

4.1 Typing the translation of a constant

Let us start from the untyped translation in order to explain the modifications we have to make. We suppose $S = 2$ and $\Sigma = \{f : X_1 \twoheadrightarrow_{X_1:\sigma_1} \Xi\ ,\quad g : \Xi\}$. We have then:

$$\begin{array}{rl} \vdash [\![f]\!] & = \lambda X_1.\ (\lambda z_1\lambda z_2.(z_1\ X_1)) : \sigma_1 \to (\sigma_1 \to \beta) \to (\sigma_1 \to \beta) \to \beta \\ x_\perp : \tau \vdash [\![f\bullet X_1 \twoheadrightarrow A]\!] & = \lambda y.(y\ (\lambda X_1.[\![A]\!])\ x_\perp)\ : ((\sigma_1 \to \tau) \to (\sigma_1 \to \tau) \to \gamma) \to \gamma \end{array}$$

where τ is the type of $[\![A]\!]$.

Therefore, the λ-term $[\![f \bullet X_1 \twoheadrightarrow A]\!]\ ([\![f]\!][\![B_1]\!])$ has a valid type only if $\tau = \beta = \gamma$. The type variables β and γ should be changed to the return type of the function which will be applied to $[\![f \bullet B_1]\!]$. Since it is impossible to guess what function will be applied to a given term, we introduce the polymorphism of Girard's System F in the target language:

$$\begin{array}{rl} \vdash [\![f]\!] & = \lambda X_1.\lambda\beta.(\lambda z_1\lambda z_2.(z_1 X_1)) : \sigma_1 \to \Pi(\beta : *).\big((\sigma_1 \to \beta) \to (\sigma_1 \to \beta) \to \beta\big) \\ x_\perp : \Pi(\iota : *).\iota \vdash [\![f \bullet X_1 \twoheadrightarrow A]\!] & = \lambda y.(y\ \tau\ (\lambda X_1.[\![A]\!])\ x_\perp)\ : \big(\Pi(\beta : *).((\sigma_1 \to \beta) \to (\sigma_1 \to \beta) \to \beta)\big) \to \tau \end{array}$$

Another interest of polymorphism is that the variable $x_\perp$ with type $\Pi(\iota : *).\iota$ can be used whenever we want a placeholder with an arbitrary type: if $[\![\Gamma]\!] \vdash_{\mathsf{F}\omega} \sigma : *$, then $[\![\Gamma]\!] \vdash_{\mathsf{F}\omega} x_\perp \sigma : \sigma$. With this simple manipulation, the use of $x_\perp$ we had made in the untyped translation becomes compatible with typing.

4.2 Typing a variable

The translation of some terms can have can have many different types, even if they have the same type in the ρ-calculus. Consider the following examples:

$$\begin{array}{rll} (\Xi : *), (X : Y \twoheadrightarrow_{Y:\Xi} \Xi) & \vdash_\rho X & : Y \twoheadrightarrow_{Y:\Xi} \Xi \\ (\Xi : *) & \vdash_\rho Y \twoheadrightarrow_{Y:\Xi} Y & : Y \twoheadrightarrow_{Y:\Xi} \Xi \\ (\Xi : *), (f : Y \twoheadrightarrow_{Y:\Xi} \Xi) & \vdash_\rho f & : Y \twoheadrightarrow_{Y:\Xi} \Xi \end{array}$$

The two ρ-terms $Y \twoheadrightarrow Y$ and f can instantiate X since they have the same type. However, the typed translation gives:

$$\begin{array}{ll} \vdash_{\mathsf{F}\omega} \lambda(\beta_Y : *).\lambda(Y : \beta_Y).Y : \beta_Y \to \beta_Y & \\ \vdash_{\mathsf{F}\omega} [\![f]\!] & : \beta_Y \to \Pi(\beta : *).\big((\beta_Y \to \beta) \to \beta\big) \end{array}$$

We see that the return type can differ, but always uses β_Y. This leads us into using types depending on types: we add a type variable β_X (with kind $* \to *$) corresponding to the term variable X, and we give $[\![X]\!]$ the type $\beta_Y \to \beta_X \beta_Y$. The types of the instantiated variable and of the term can then be made equal by suitably instantiating β_X; in our examples, for $Y \twoheadrightarrow Y$ we take $\beta_X := \lambda(\gamma : *).\gamma$ and for f we take $\beta_X := \lambda(\gamma : *).\Pi(\beta : *).\big((\gamma \to \beta) \to \beta\big)$.

5 Strong normalization

Strong normalization is shown by Theorem 4 and by the well-typedness of the encoding:

Theorem 5 (Well-typed translation) *If $\Sigma, \Gamma \vdash_\rho A : \Phi$, then $\exists \tau,\ [\![\Gamma]\!] \vdash_{\mathsf{F}\omega} [\![A]\!] : \tau$.*

Theorem 6 (Strong normalization) *If $\Sigma, \Gamma \vdash_\rho A : \Phi$ then A is strongly normalizing.*

6 Conclusion and perspectives

We have proved strong normalization of the ρ-calculus in the simply-typed system of the ρ-cube. The proof relies on a translation from the ρ-calculus into System $\mathsf{F}\omega$. First, we have encoded untyped pattern matching in the λ-calculus, ensuring that every $\rho\sigma\delta$-reduction is translated into (at least) one β-reduction. Then we have shown that the typing mechanisms of the ρ-calculus can be reproduced only with the expressive power of System $\mathsf{F}\omega$, which is rather surprising since we only deal with the simply-typed system of the ρ-cube. It hints that the expressive power of the ρ-calculus could be greater than the one of the λ-calculus.

The property of strong normalization is finally proved by contradiction: a typed ρ-term translates into a typed λ-term, which can have only finite reductions.

One development of this work would be to extend the proof to the other type systems of the ρ-cube. In the long term, we could use the ρ-calculus as the basis for a proof assistant combining the λ-calculus (for deduction) and the rewriting (for computation). Strong normalization is a stepstone for this work, since logical soundness is related to termination.

Long version: `http://www.loria.fr/~wack/papers/rhoSN.ps.gz`

References

1. H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*. Clarendon Press, 1992.
2. G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Patterns Type Systems. In *Principles of Programming Languages - POPL2003, New Orleans, USA*. ACM, January 2003.
3. H. Cirstea and C. Kirchner. The typed rewriting calculus. In *Third International Workshop on Rewriting Logic and Application*, Kanazawa (Japan), September 2000.
4. H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In F. Honsell, editor, *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 166–180, Genova, Italy, April 2001.
5. H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. The rewriting calculus : some results, some problems. In D. Kesner, F. van Raamsdonk, and T. Nipkow, editors, *The first international workshop on Higher-Order Rewriting*, Copenhagen, Denmark, July 2002. FLoC'02. LORIA Internal research report A02-R-470.
6. H. Cirstea, L. Liquori, and B. Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. In *TYPES'03*, volume 3085 of *Lecture Notes in Computer Science*, Torino, 2004.

Termination of Simply-Typed Applicative TRSs

Takahito Aoto[1] and Toshiyuki Yamada[2]

[1] Research Institute of Electrical Communication, Tohoku University, Japan
aoto@nue.riec.tohoku.ac.jp
[2] Faculty of Engineering, Mie University, Japan
toshi@cs.info.mie-u.ac.jp

Abstract. Applicative term rewriting system handles terms having a tree structure with a function symbol or a variable at every leaf node and a unique application symbol at every internal node. Because of this term structure, direct application of standard termination proof methods, such as recursive path orderings, are often ineffective for applicative term rewriting systems. In this paper, we introduce a simply-typed version of applicative term rewriting systems and present a termination proof method for simply-typed applicative term rewriting systems.

1 Introduction

Applicative term rewriting system (TRS) is a popular encoding in term rewriting which can deal with higher-order functions. It is a first-order TRS where terms have a tree structure consisting of a function symbol or a variable at every leaf node and a unique application symbol at every internal node. For example, the higher-order function map, is expressed as the following applicative TRS:

$$\begin{aligned}(\mathsf{map}\text{'}F)\text{'}\mathsf{nil} &\to \mathsf{nil}\\ (\mathsf{map}\text{'}F)\text{'}((\mathsf{cons}\text{'}x)\text{'}xs) &\to (\mathsf{cons}\text{'}(F\text{'}x))\text{'}((\mathsf{map}\text{'}F)\text{'}xs)\end{aligned}$$

where " ' " denotes the infix application symbol.

In an *applicative* term formulation, such as above, all subterms except constants and variables have the same leading symbol. This makes a contrast with the usual *functional* term formulation, such as $\mathsf{map}(F, \mathsf{cons}(x, xs))$. In applicative TRS, direct application of standard termination proof methods, such as recursive path orderings [4] and the dependency pair technique [3], is ineffective. This is because these techniques make use of the difference of leading function symbols.

In this paper, we introduce a simply-typed version of applicative TRSs and present a proof method for their termination. We give a transformation from applicative TRSs to TRSs in functional form for which standard termination proof methods are effectively applied.

2 Simply-Typed Applicative TRSs

For a set B of *basic types*, the set of *simple types* is the smallest set $\mathrm{ST}(B)$ such that (1) $B \subseteq \mathrm{ST}(B)$, and (2) $\tau \to \rho \in \mathrm{ST}(B)$ whenever $\tau, \rho \in \mathrm{ST}(B)$. The set $\mathrm{ST}(B)$ is abbreviated to ST. As usual we assume $\to$ is right-associative and omit unnecessary parentheses. Based on this abbreviation, each simple type τ is uniquely represented by $\tau_1 \to \tau_2 \to \cdots \to \tau_k \to \tau_0$ with $\tau_1, \ldots, \tau_k \in \mathrm{ST}(B)$ and $\tau_0 \in B$, where k is called the *arity* of τ and denoted by $\mathrm{arity}(\tau)$.

Each *function symbol* or *variable* is associated with its type. The sets of function symbols and variables of type τ are denoted by Σ^τ and V^τ. The sets of all function symbols and variables are denoted by Σ and V. The arity of a function symbol $f \in \Sigma^\tau$ is defined as

arity(f) = arity(τ). Apart from members of Σ, we assume that there is a binary infix function symbol " ' " called an *application symbol*. The set $\mathrm{AT}_{\mathrm{ST}}(\Sigma, V)^\tau$ of *simply-typed applicative terms* of type τ is defined as follows: (1) $\Sigma^\tau \cup V^\tau \subseteq \mathrm{AT}_{\mathrm{ST}}(\Sigma, V)^\tau$, and (2) if $s \in \mathrm{AT}_{\mathrm{ST}}(\Sigma, V)^{\tau\to\rho}$ and $t \in \mathrm{AT}_{\mathrm{ST}}(\Sigma, V)^\tau$ then $(s't) \in \mathrm{AT}_{\mathrm{ST}}(\Sigma, V)^\rho$. Each simply-typed applicative term has a unique type and thus τ is also referred to as the type of s (denoted by type(s)). The arity of a simply-typed applicative term s is defined by its type. The set of all simply-typed applicative terms is denoted by $\mathrm{AT}_{\mathrm{ST}}(\Sigma, V)$ or $\mathrm{AT}_{\mathrm{ST}}$. The set of variables occurring in a term t is written as $\mathrm{V}(t)$. A term without variables is said to be *ground*. We assume the application symbol " ' " is left-associative and usually omit unnecessary parentheses as well as the outermost one. Based on this abbreviation, each term s is uniquely represented as $a's_1'\cdots's_n$ for some $a \in \Sigma \cup V$ and $s_1, \ldots, s_n \in \mathrm{AT}_{\mathrm{ST}}$, where a is called the *head* of the term s and denoted by head(s). A *substitution* is a mapping $\sigma : V \to \mathrm{AT}_{\mathrm{ST}}(\Sigma, V)$ that satisfies the following conditions: (1) $\mathrm{Dom}(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite, and (2) for each $x \in \mathrm{Dom}(\sigma)$, x and $\sigma(x)$ have the same type. The homomorphic extension of σ to $\mathrm{AT}_{\mathrm{ST}}(\Sigma, V)$ is also denoted by σ. As usual, $\sigma(t)$ is written as $t\sigma$. A substitution is *ground* if $\sigma(x)$ is ground for any $x \in \mathrm{Dom}(\sigma)$.

A pair of simply-typed applicative terms written as $l \to r$ is a *simply-typed applicative rewrite rule* when (1) l and r have the same type, (2) head(l) $\in \Sigma$, and (3) $V(r) \subseteq V(l)$. A triple $\langle B, \Sigma, R \rangle$ consisting of a set B of basic types, a set Σ of function symbols, and a set R of rewrite rules is called a *simply-typed applicative TRS* (SATRS, for short). The *rewrite relation* $\to_{\mathcal{R}}$ induced by an SATRS $\mathcal{R} = \langle B, \Sigma, R \rangle$ is the smallest relation over $\mathrm{AT}_{\mathrm{ST}}(\Sigma, V)$ satisfying the following conditions: (1) $l\sigma \to_{\mathcal{R}} r\sigma$ for all $l \to r \in R$ and for all substitutions σ, and (2) if $s \to_{\mathcal{R}} t$ then $(u_1's) \to_{\mathcal{R}} (u_1't)$ and $(s'u_2) \to_{\mathcal{R}} (t'u_2)$ for any u_1 and u_2. An SATRS $\mathcal{R}$ is *terminating* if there is no infinite reduction sequences $s_0 \to_{\mathcal{R}} s_1 \to_{\mathcal{R}} \cdots$.

Example 1 (simply-typed applicative rewriting). Let $\mathcal{R} = \langle B, \Sigma, R \rangle$ be an SATRS where $B = \{\mathrm{N}, \mathrm{L}\}$, $\Sigma = \{0^{\mathrm{N}}, \mathsf{s}^{\mathrm{N}\to\mathrm{N}}, \mathsf{id}^{\mathrm{N}\to\mathrm{N}}, \mathsf{add}^{\mathrm{N}\to\mathrm{N}\to\mathrm{N}}, \mathsf{nil}^{\mathrm{L}}, \mathsf{cons}^{\mathrm{N}\to\mathrm{L}\to\mathrm{L}}, \mathsf{map}^{(\mathrm{N}\to\mathrm{N})\to\mathrm{L}\to\mathrm{L}}\}$, and

$$R = \begin{cases} \mathsf{id}'x & \to x \\ \mathsf{add}'0 & \to \mathsf{id} \\ \mathsf{map}'F'\mathsf{nil} & \to \mathsf{nil} \end{cases} \qquad \begin{cases} \mathsf{add}'(\mathsf{s}'x)'y & \to \mathsf{s}'(\mathsf{add}'x'y) \\ \mathsf{map}'F'(\mathsf{cons}'x'xs) & \to \mathsf{cons}'(F'x)'(\mathsf{map}'F'xs) \end{cases}$$

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

$$\begin{aligned} \mathsf{map}'(\mathsf{add}'0)'(\mathsf{cons}'(\mathsf{s}'0)'\mathsf{nil}) &\to_{\mathcal{R}} \mathsf{map}'\mathsf{id}'(\mathsf{cons}'(\mathsf{s}'0)'\mathsf{nil}) \to_{\mathcal{R}} \mathsf{cons}'(\mathsf{id}'(\mathsf{s}'0))'(\mathsf{map}'\mathsf{id}'\mathsf{nil}) \\ &\to_{\mathcal{R}} \mathsf{cons}'(\mathsf{s}'0)'(\mathsf{map}'\mathsf{id}'\mathsf{nil}) \to_{\mathcal{R}} \mathsf{cons}'(\mathsf{s}'0)'\mathsf{nil} \end{aligned}$$

3 Removing Head Variables by Cover Instantiation

An applicative term can be translated into the corresponding term in functional form if there is no variable at any head position. In this section, we introduce a transformation on SATRSs for removing head variables.

Each rewrite rule can be transformed into an equivalent set of instantiated rules if every variable is instantiated by appropriate terms which cover all instances. Such instantiation can be used to remove head variables in an SATRS.

Definition 1 (cover instantiation). Let $l \to r$ be a rewrite rule and x^τ a variable in l. A set S of terms of type τ is a *coverset* with respect to $l \to r$ and x if (1) every ground term t^τ can be expressed as the form $t = s\sigma$ for some $s \in S$ and substitution σ, and

(2) $\mathrm{V}(s) \cap (\mathrm{V}(l) \setminus \{x\}) = \emptyset$ for every $s \in S$. A mapping $\psi : V \to \mathcal{P}(\mathrm{AT}_{\mathrm{ST}})$ is a *covermap* with respect to $l \to r$ if (1) its domain includes $\mathrm{V}(l)$, (2) $\psi(x)$ is a coverset with respect to $l \to r$ and x for every $x \in \mathrm{V}(l)$, and (3) $\psi(x)$ and $\psi(y)$ do not share the same variable for different variables x and y. Let ψ be a covermap with respect to $l \to r$. This covermap is used to transform the rewrite rule into a set of its instances as follows:

$$\mathrm{Inst}(l \to r, \psi) = \{l\sigma \to r\sigma \mid \forall x \in \mathrm{V}(l)\ \ \sigma(x) \in \psi(x)\}$$

Let $\mathcal{R} = \langle B, \Sigma, R \rangle$ be an SATRS. An indexed set of mappings $\Psi = \{\psi_{l\to r} \mid l \to r \in R\}$ is a *covermap system* with respect to R if each $\psi_{l\to r}$ is a covermap with respect to $l \to r$. A covermap system is used to transform R into a set of instantiated rewrite rules as follows:

$$\mathrm{Inst}(R, \Psi) = \bigcup_{l\to r\in R} \mathrm{Inst}(l \to r, \psi_{l\to r})$$

Definition 2 (head variable). The set of *head variables* in a term t is defined as follows:

$$\begin{array}{lll} \mathrm{HV}(a) & = \emptyset & \text{if } a \in \Sigma \cup V \\ \mathrm{HV}(f\text{'}t) & = \mathrm{HV}(t) & \text{if } f \in \Sigma \\ \mathrm{HV}(x\text{'}t) & = \{x\} \cup \mathrm{HV}(t) & \text{if } x \in V \\ \mathrm{HV}(s\text{'}t) & = \mathrm{HV}(s) \cup \mathrm{HV}(t) & \text{otherwise} \end{array}$$

Lemma 1 (removing head variables). *Let $\mathcal{R} = \langle B, \Sigma, R \rangle$ be an SATRS. Suppose a covermap system $\Psi = \{\psi_{l\to r} \mid l \to r \in R\}$ satisfies the properties that, for all rewrite rules $l \to r \in R$, (1) $\mathrm{HV}(\psi_{l\to r}(x)) = \emptyset$ for any $x \in \mathrm{V}(l)$, and (2) $V \cap \psi_{l\to r}(x) = \emptyset$ for any $x \in \mathrm{HV}(l) \cup \mathrm{HV}(r)$. Define $R' = \mathrm{Inst}(R, \Psi)$ and $\mathcal{R}' = \langle B, \Sigma, R' \rangle$. Then, $s \to_{\mathcal{R}} t$ if and only if $s \to_{\mathcal{R}'} t$ for all ground terms s and t. Moreover, none of rewrite rules in R' contain a head variable.*

Because instantiated rewrite rules can simulate only ground rewrite steps, existence of a ground term of types in concern is required.

Definition 3 (ground term existence condition). Let $\mathcal{R} = \langle B, \Sigma, R \rangle$ be an SATRS. We say $\mathcal{R}$ satisfies the *ground term existence condition* (GTEC) if the following conditions are satisfied: (1) for every variable x appearing in R, there exists a ground term with the same type as x, and (2) for every function symbol $f \in \Sigma$ of type $\tau_1 \to \cdots \to \tau_n \to \rho$ ($\rho \in B$), there exists ground terms u_i of type τ_i for any $i = 1, \ldots, n$.

Lemma 2 (termination by cover instantiation). *Let $\mathcal{R} = \langle B, \Sigma, R \rangle$ be an SATRS satisfying GTEC and Ψ be a coverset system with respect to R. Define $\mathcal{R}' = \langle B, \Sigma, \mathrm{Inst}(R, \Psi) \rangle$. If $\mathcal{R}'$ is terminating then $\mathcal{R}$ is terminating.*

Based on the observations so far, we define a transformation on SATRSs for removing head variables which preserves non-termination property.

Definition 4 (transformation HVI). Let $\mathcal{R} = \langle B, \Sigma, R \rangle$ be an SATRS. The covermap system for *head variable instantiation* $\Psi_{\mathrm{HVI}} = \{\psi_{l\to r}\}_{l\to r\in R}$ is defined as

$$\psi_{l\to r}(x) = \begin{cases} \{f\text{'}x_1\text{'}\cdots\text{'}x_n \mid f \in \Sigma,\ \mathrm{type}(f) = \tau_1 \to \cdots \to \tau_n \to \mathrm{type}(x)\} & \text{if } x \in \mathrm{HV}(l) \cup \mathrm{HV}(r) \\ \{x\} & \text{otherwise} \end{cases}$$

for all $x \in \mathrm{V}(l)$ where $x_1, \cdots, x_n$ are pairwise distinct fresh variables. The transformation HVI is defined as:

$$\mathrm{HVI}(\mathcal{R}) = \langle B, \Sigma, \mathrm{Inst}(R, \Psi_{\mathrm{HVI}}) \rangle$$

Example 2 (transformation HVI). Let $\mathcal{R} = \langle B, \Sigma, R \rangle$ be the SATRS given in Example 1. Then the transformed set of rewrite rules of $\mathrm{HVI}(\mathcal{R})$ are

$$\begin{cases} \mathsf{id}'x & \to x \\ \mathsf{add}'0 & \to \mathsf{id} \\ \mathsf{add}'(\mathsf{s}'x)'y & \to \mathsf{s}'(\mathsf{add}'x'y) \\ \mathsf{map}'F'\mathsf{nil} & \to \mathsf{nil} \\ \mathsf{map}'\mathsf{s}'(\mathsf{cons}'x'xs) & \to \mathsf{cons}'(\mathsf{s}'x)'(\mathsf{map}'\mathsf{s}'xs) \\ \mathsf{map}'\mathsf{id}'(\mathsf{cons}'x'xs) & \to \mathsf{cons}'(\mathsf{id}'x)'(\mathsf{map}'\mathsf{id}'xs) \\ \mathsf{map}'(\mathsf{add}'y)'(\mathsf{cons}'x'xs) & \to \mathsf{cons}'(\mathsf{add}'y'x)'(\mathsf{map}'(\mathsf{add}'y)'xs) \end{cases}$$

Lemma 3 (properties of transformation HVI). *Let $\mathcal{R} = \langle B, \Sigma, R \rangle$ be an SATRS satisfying GTEC. If* $\mathrm{HVI}(\mathcal{R})$ *is terminating then $\mathcal{R}$ is terminating. Moreover, all rewrite rules in* $\mathrm{HVI}(\mathcal{R})$ *do not contain any head variables in both hand sides.*

4 Translation to Functional Form

In this section, we introduce a translation from an SATRS to a TRS.

Definition 5 (translation FF). Let $\mathcal{R} = \langle B, \Sigma, R \rangle$ be an SATRS where all rules contain no head variables. The set of function symbols in functional form is defined as

$$\mathrm{FF}(\Sigma) = \{ f_k \mid f \in \Sigma, 0 \le k \le \mathrm{arity}(f) \}$$

where the arity of f_k is k. For each applicative term $s \in \mathrm{AT}_{\mathrm{ST}}(\Sigma, V)$ without head variables, its functional form $\mathrm{FF}(s) \in \mathrm{T}(\mathrm{FF}(\Sigma), V)$ is defined as follows:

$$\mathrm{FF}(a't_1'\cdots't_n) = \begin{cases} a_n(\mathrm{FF}(t_1), \ldots, \mathrm{FF}(t_n)) & \text{if } a \in \Sigma \\ a & \text{if } a \in V \end{cases}$$

Note here that $a \in V$ implies $n = 0$ by the absence of head variable. The set of rules and the rewrite system are translated as follows:

$$\begin{aligned} \mathrm{FF}(R) &= \{\ \mathrm{FF}(l'x_1'\cdots'x_k) \to \mathrm{FF}(r'x_1'\cdots'x_k) \mid l \to r \in R,\ 0 \le k \le \mathrm{arity}(l)\ \} \\ \mathrm{FF}(\mathcal{R}) &= \langle \mathrm{FF}(\Sigma), \mathrm{FF}(R) \rangle \end{aligned}$$

where $x_1, \cdots, x_n$ are pairwise distinct fresh variables.

Example 3 (translation FF). Let B, Σ, and R be the ones in Example 1. Then, we have $\mathrm{FF}(\mathsf{map}'(\mathsf{add}'0)'(\mathsf{cons}'(\mathsf{s}'0)'\mathsf{nil})) = \mathsf{map}_2(\mathsf{add}_1(0_0), (\mathsf{cons}_2(\mathsf{s}_1(0_0), \mathsf{nil}_0)))$. The set $\mathrm{FF}(\mathrm{HVI}(R))$ is

$$\begin{cases} \mathsf{id}_1(x) & \to x \\ \mathsf{add}_1(0_0) & \to \mathsf{id}_0 \\ \mathsf{add}_2(0_0, x) & \to \mathsf{id}_1(x) \\ \mathsf{add}_2(\mathsf{s}_1(x), y) & \to \mathsf{s}_1(\mathsf{add}_2(x, y)) \\ \mathsf{map}_2(F, \mathsf{nil}_0) & \to \mathsf{nil}_0 \\ \mathsf{map}_2(\mathsf{s}_0, \mathsf{cons}_2(x, xs)) & \to \mathsf{cons}_2(\mathsf{s}_1(x), \mathsf{map}_2(\mathsf{s}_0, xs)) \\ \mathsf{map}_2(\mathsf{id}_0, \mathsf{cons}_2(x, xs)) & \to \mathsf{cons}_2(\mathsf{id}_1(x), \mathsf{map}_2(\mathsf{id}_1, xs)) \\ \mathsf{map}_2(\mathsf{add}_1(y), \mathsf{cons}_2(x, xs)) & \to \mathsf{cons}_2(\mathsf{add}_2(y, x), \mathsf{map}_2(\mathsf{add}_1(y), xs)) \end{cases}$$

Lemma 4 (simulation of rewriting). *Let $\mathcal{R} = \langle B,\ \Sigma, R\rangle$ be an SATRS where every rewrite rule contains no head variables. Let s and t be ground simply-typed applicative terms. Then $s \to_{\mathcal{R}} t$ implies* $\mathrm{FF}(s) \to_{\mathrm{FF}(\mathcal{R})} \mathrm{FF}(t)$.

Theorem 1 (soundness of $\mathrm{FF} \circ \mathrm{HVI}$**).** *Let $\mathcal{R} = \langle B,\ \Sigma, R\rangle$ be an SATRS satisfying GTEC. If* $\mathrm{FF}(\mathrm{HVI}(\mathcal{R}))$ *is terminating then $\mathcal{R}$ is terminating.*

Example 4 (termination proof I). Let $\mathcal{R}$ be the SATRS in Example 1. Then $\mathrm{FF}(\mathrm{HVI}(\mathcal{R}))$ is the one in Example 3. Its termination is shown by the recursive path ordering with the precedence $\mathsf{add}_1 > \mathsf{id}_0$; $\mathsf{map}_2 > \mathsf{add}_2, \mathsf{cons}_2 > \mathsf{id}_1, \mathsf{s}_1$. Therefore, by Theorem 1, $\mathcal{R}$ is terminating.

Example 5 (termination proof II). Let $\mathcal{R} = \langle B,\ \Sigma, R\rangle$ be an SATRS where $B = \{\mathrm{N}\}$, $\Sigma = \{\mathsf{0}^{\mathrm{N}}, \mathsf{s}^{\mathrm{N}\to\mathrm{N}}, \mathsf{add}^{\mathrm{N}\to\mathrm{N}\to\mathrm{N}}, \mathsf{mult}^{\mathrm{N}\to\mathrm{N}\to\mathrm{N}}, \mathsf{rec}^{(\mathrm{N}\to\mathrm{N}\to\mathrm{N})\to\mathrm{N}\to\mathrm{N}\to\mathrm{N}}, \mathsf{fact}^{\mathrm{N}\to\mathrm{N}}\}$ and

$$R = \begin{cases} \mathsf{add}\text{'}\mathsf{0}\text{'}y \to y & \mathsf{add}\text{'}(\mathsf{s}\text{'}x)\text{'}y \to \mathsf{s}\text{'}(\mathsf{add}\text{'}x\text{'}y) \\ \mathsf{mult}\text{'}\mathsf{0}\text{'}y \to \mathsf{0} & \mathsf{mult}\text{'}(\mathsf{s}\text{'}x)\text{'}y \to \mathsf{add}\text{'}(\mathsf{mult}\text{'}x\text{'}y)\text{'}y \\ \mathsf{rec}\text{'}F\text{'}x\text{'}\mathsf{0} \to x & \mathsf{rec}\text{'}F\text{'}x\text{'}(\mathsf{s}\text{'}y) \to F\text{'}(\mathsf{s}\text{'}y)\text{'}(\mathsf{rec}\text{'}F\text{'}x\text{'}y) \\ \mathsf{fact} \to \mathsf{rec}\text{'}\mathsf{mult}\text{'}(\mathsf{s}\text{'}\mathsf{0}) & \end{cases}$$

Termination of $\mathrm{FF}(\mathrm{HVI}(\mathcal{R}))$ is shown by using the lexicographic path ordering [5]. We note that termination of this SATRS can not be shown by other termination proof methods for lambda-free higher-order TRSs such as [6], [7] or (the product-free fragments of) [1], [2], [8], [9].

Acknowledgments

The authors thank Mario Rodríguez-Artalejo for his comments, by which our translation FF is inspired.

References

1. T. Aoto and T. Yamada. Proving termination of simply typed term rewriting systems automatically. *IPSJ Transactions on Programming*, 44(SIG 4 PRO 17):67–77, 2003. In Japanese.
2. T. Aoto and T. Yamada. Termination of simply typed term rewriting systems by translation and labelling. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 380–394, 2003.
3. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
4. N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.
5. S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering. Unpublished manuscript, University of Illinois, 1980.
6. K. Kusakari. On proving termination of term rewriting systems with higher-order variables. *IPSJ Transactions on Programming*, 42(SIG 7 PRO 11):35–45, 2001.
7. K. Kusakari. Higher-order path orders based on computability. *IEICE Transactions on Information and Systems*, E87–D(2):352–359, 2003.
8. M. Lifantsev and L. Bachmair. An lpo-based termination ordering for higher-order terms without λ-abstraction. In *Proceedings of the 11th International Conference Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *Lecture Notes in Computer Science*, pages 277–293. Springer-Verlag, 1998.
9. Y. Toyama. Termination of S-expression rewriting systems: Lexicographic path ordering for higher-order terms. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, to appear.

Unification and Matching Modulo Type Isomorphism

Dan Dougherty[1] and Carlos C. Martínez[2]

[1] Worcester Polytechnic Institute,
Department of Computer Science,
Worcester, MA 01609 USA
dd@cs.wpi.edu
[2] Wesleyan University,
Department of Mathematics and Computer Science,
Middletown, CT 06459 USA
cmartinez@wesleyan.edu

Abstract. We present some initial results in an investigation of higher-order unification and matching in the presence of type isomorphism.

1 Introduction

Two simple types S and T are isomorphic if their interpretations are isomorphic in every model of the simply-typed λ-calculus, or equivalently, if there exist terms $f : S \to T$ and $g : T \to S$, such that $g \circ f$ and $f \circ g$ are each $\beta\eta$-convertible with the identity.

The study of type isomorphism is currently an active area of research, well-represented in Roberto di Cosmo's book [DC95]. There are connections with logic (cf Tarski's "high school algebra problem"), with category theory [FCB02], and with information retrieval in software libraries [Rit90,Rit91,RT91,ZW93].

Some of the most interesting work concerns polymorphic type disciplines but our focus here will be restricted to simple types. Indeed, in this preliminary report *we consider arrow-types only*, in particular we exclude product and unit types. Much of complexity of type isomorphism per se is avoided in this setting, but the novel issues surrounding unification and matching still arise, as we will see. We also work only with pure terms (ie, with no constants).

It is natural to want be sensitive to type isomorphism when one is doing higher-order rewriting, in particular if we are interested in code querying or transformation. For example, suppose one wants to perform a code transformation with a certain function-pattern of type $(A \to B \to C)$. The use of standard higher-order matching allows us to ignore the *names* of the arguments in a code fragment potentially matching the pattern. But the *order* in which these parameters appear in the code is significant, since it determines the code's type. Since the type $(B \to A \to C)$ is isomorphic to the original $(A \to B \to C)$, a code fragment of the this type may very well be a candidate that we want to consider. So it seems that a more refined tool than standard higher-order matching would be useful.

Indeed, what we require is a richer notion of matching which accepts a match as long as the term being matched is *the same as the target term modulo a type isomorphism*. That is, type isomorphism induces a notion of equality on terms, more lenient than equality modulo $\beta\eta$, and it is this equality that we will want to guide our matching. To our knowledge this relation on terms — which we call "term isomorphism" for want of a better phrase — has not been explored in the published literature.

Definition 1. Given $s : S$ and $t : T$, we say that s and t are *term isomorphic*, written $s \simeq t$, if there is a type isomorphism $p : S \to T$ with $ps = t$.

Here and below, the equality symbol "=" denotes convertibility modulo β and η.

In addition to higher-order matching it is also natural to consider higher-order *unification* modulo type isomorphism, for example in the context of higher-order logic programming, when λ-terms are ubiquitous as data structures, and type isomorphism induces a natural equivalence on this data.

In fact the problems we face and the solutions we propose arise equally in unification and in matching. So for simplicity in this abstract we focus on unification: our goal is to explore algorithms to solve the following problem.

Unification modulo Type Isomorphism

INPUT: Two terms s and t, of isomorphic types

OUTPUT: A substitution θ such that $\theta s \simeq \theta t$

2 A naive algorithm

We first note that the consideration of type isomorphism does not have any consequences as to decidability.

Dezani [DC76] defined the notion of *finite hereditary permutation*, which is a term of the following form:

$$\lambda z x_1 \ldots x_n . z(p_1 x_{\pi(1)}) \ldots (p_n x_{\pi(n)})$$

where π is a permutation of $[1..n]$ and each p_i is a finite hereditary permutation, and proved that a normalizable untyped term p is invertible iff it is a finite hereditary permutation.

Any finite hereditary permutation is typable. So, as observed in [BCL92], a term $p : A \to B$ is a type isomorphism iff its type-erasure is a finite hereditary permutation.

It is not hard to see that at any type $A \to B$ there are only finitely many finite terms whose type-erasures are hereditary permutations. So we have the following

Proposition 2. *For each pair of types S and T there are finitely many type isomorphisms $p : S \to T$, and these can be effectively generated given S and T.*

In this way we can reduce any unification/matching problem modulo type isomorphism to finitely many similar standard problems.

Definition 3 (Naive algorithm). Given $s : S$ and $t : T$ as an instance of the problem of unification or matching modulo type isomorphism, the naive algorithm for the problem is:

For each type isomorphism p from S to T, generate the standard problem $ps = t$. If any of these problems is solvable, return that solution; otherwise return failure.

Proposition 4. *The algorithm of Definition 3 is sound and complete relative to the soundness and completeness of the standard algorithms used.*

In particular, at any type where the classical higher-order matching is decidable, the problem of matching modulo type isomorphism is decidable. Also, unification of higher-order patterns [Mil91] modulo type isomorphism is decidable.

This is reassuring, but since there can be exponentially many finite hereditary permutations at a given pair of types, such a generate-and-test algorithm is clearly not practical. We want to "build-in" (in the sense of [Plo72]) type isomorphism to the matching algorithm.

3 Building in type isomorphism

Consider a set of standard transformations for pure higher-order unification, such as described in [GS89]: Imitation, Projection, Variable Elimination, and Decomposition. The idea here is to enhance this set of transformations so that in addition to gradually building up an answer substitution as a problem is solved, we also build up a finite hereditary permutation that serves as part of the witness to the problem's solution.

The main work in unification transformations is the "guessing" component, where substitutions are generated and propagated. These are represented in higher-order unification by the Imitation, Projection, and Variable Elimination transformations. There is another transformation — Decomposition — which breaks a problem into subproblems or recognizes that the current problem is not solvable and reports failure.

(Traditional) Decomposition

$$\frac{E\ ;\ (\lambda\overline{x}.x_e\,\overrightarrow{t}\ \doteq\ \lambda\overline{y}\overline{x}.x_d\,\overrightarrow{u})}{E\ ;\ (\lambda\overline{x}.t_1\ \doteq\ \lambda\overline{x}.u_1)\ ;\ \ldots\ ;\ (\lambda\overline{x}.t_k\ \doteq\ \lambda\overline{x}.u_k)}\quad \text{if } e=d\text{, else fail}$$

Perhaps surprisingly, with a little work one can see that in defining transformations for higher-order unification modulo type isomorphism Imitation, Projection, and Variable Elimination need not be changed at all. That is, the complexity of considering type isomorphism is completely reflected in the need for a refinement of the Decomposition transformation.

This is because the soundness of the Decomposition transformation is the embodiment of a basic fact about *equality* between terms: by the Church-Rosser theorem $\lambda\overline{x}.x_e\,\overrightarrow{t}$ and $\lambda\overline{y}\overline{x}.x_d\,\overrightarrow{u}$ are $\beta\eta$-equal if and only if $e=d$ and corresponding arguments are equal. This is false when the equality in question is $\simeq$.

So we see that the essence of building in type isomorphisms to the unification algorithm is to have a good characterization of when an equation is *valid* (as opposed to unifiable) modulo type isomorphism. That is, we are led to seek an incremental analysis of term isomorphism.

4 Characterizing term isomorphism

4.1 Labelled trees

In order to analyze the combinatorics of term isomorphism it is convenient to abstract away from variable binding and work with ordinary labelled trees. In fact we work with partially labelled trees and partial functions between trees in anticipation of the facts that terms with free variables will correspond to trees which are not fully labelled.

Definition 5. Let $\mathscr{L}$ be a set of *labels* together with an arity function $\mathit{arity}:\mathscr{L}\to\mathbb{N}$. A *labelled tree* $\mathcal{T}$ is a tree domain T and a partial map $\mathit{label}:T\to\mathscr{L}$ such that for $\alpha\in\mathrm{domain}(\mathit{label})$ the number of children of α is $\mathit{arity}(\mathit{label}(\alpha))$. Furthermore, a labelled tree comes with an arity-consistent equivalence relation $\approx$ defined on its set of labels.

The definition of labelled tree mapping below is somewhat tedious; the informal idea is as follows. Labelled trees are isomorphic if they are isomorphic modulo permuting the order of the children of a node — except that nodes with equivalent labels must have their children permuted in the same way. See Example 7 below.

Definition 6. Let T and U be tree domains. A *partial tree mapping* $\Phi:T\to U$ is a pair (Φ^a,Φ^p) such that

- Φ^a is a partial function from T to U whose domain is a subtree of T,
- $\Phi^p(\alpha)$ is a permutation of $[1..\mathit{arity}(\alpha)]$ for each $\alpha\in\mathrm{domain}(\Phi^a)$, and

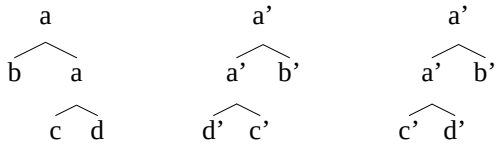
– Φ^a maps the i-th child of α to the $\Phi^p(\alpha)(i)$-th child of $\Phi^a(\alpha)$. That is, $\Phi^a(\alpha \cdot i) = \Phi^a(\alpha) \cdot \Phi^p(\alpha)(i)$.

If $\mathcal{T}$ and $\mathcal{U}$ are labelled trees, a *partial labelled tree mapping* from $\mathcal{T}$ to $\mathcal{U}$ is a partial tree mapping Φ defined on labelled addresses of $\mathcal{T}$ satisfying

– $label(\alpha) = label(\beta)$ implies $label(\Phi^a(\alpha)) = label(\Phi^a(\beta))$, and
– $label(\alpha) \approx label(\beta)$ implies $\Phi^p(\alpha) = \Phi^p(\beta)$.

The labelled trees $\mathcal{T}$ and $\mathcal{U}$ are *isomorphic* as labelled trees if there is a labelled tree homomorphism Φ from $\mathcal{T}$ to $\mathcal{U}$ with Φ^a a bijection between the nodes of $\mathcal{T}$ and $\mathcal{U}$.

Example 7.



The first and second labelled trees above are isomorphic; the first and third labelled trees are *not* isomorphic.

Remark 8. If $\mathcal{T}$ and $\mathcal{U}$ are labelled trees with disjoint label sets $\mathscr{L}$ and $\mathscr{L}'$, a partial tree mapping $\Phi : \mathcal{T} \to \mathcal{U}$ induces a unique partial mapping from $\mathscr{L}$ to $\mathscr{L}'$ in an obvious way. We say that this mapping on labels is *induced by* Φ.

4.2 Labelled trees for terms

We first need some notation allowing us to track relationships among bound-variable occurrences in a Böhm tree.

Definition 9. Let $t : T$ be a pure term. We let $BT(t)$ denote the Böhm tree for t. In $BT(t)$ we make the following definitions.

If the node at address α has binder $\lambda x_1 \ldots \lambda x_n$ and head variable y then we say that y has an *occurrence* at address α and for each i we say that bound variable x_i is *introduced* at address α, at *index* i.

We assume that whenever one or more Böhm trees are under consideration no name is used for both a free and bound variable, and no bound variable name is introduced in more than one place.

Note the distinction between the introduction of a bound variable and an occurrence of a variable (in particular an introduction is not an occurrence). Note also that a variable may have any number of occurrences, but it is only introduced at one address and index.

Now, given a term t, we define $LT(t)$, the *labelled tree associated to* t, essentially by forgetting the binders in the Böhm tree for t (so that $LT(t)$ will fail to have a label at those tree addresses with free variables). The precise definition is Definition 10 below. By using bound-variable names from $BT(t)$ as our labels we are of course not determining the labels of $LT(t)$ uniquely. But since we have adopted a convention that Böhm trees always obey our strong variable conventions about not reusing variable names, the differences in labelled trees we obtain for a term are identical up to renaming of labels. Indeed it will often be convenient to be able to assume that the labels of a given pair of trees are disjoint.

Definition 10. The underlying tree of $LT(t)$ is the underlying tree of $BT(t)$. If there is a bound-variable occurrence at α in $BT(t)$ the label in $LT(t)$ at address α is that bound variable name.

The relation $\approx$ is the smallest equivalence relation satisfying: $x \approx y$ if

– x is introduced in $BT(t)$ at address αa, index i

– y is introduced $BT(t)$ at address $\alpha' a$, index i, and
– $label(\alpha) \approx label(\alpha')$

Theorem 11. *Let $t : T$ and $u : U$ be terms of isomorphic type. Then $t \simeq u$ if and only if $LT(t)$ and $LT(u)$ are isomorphic labelled trees.*

5 The transformations

Theorem 11 allows us to define transformations for higher-order unification under type isomorphism; as described earlier the key is defining a sound Decomposition transformation.

A *system* $E \ \rhd \ \Lambda \mid \sigma$ is given by a set E of equations, a partial mapping Λ on labels, and a substitution σ. Λ and σ denote the label mapping and answer substitutions computed "so far". Transformations are presented as inference rules for deriving systems. If e is an equation, the notation $E;e$ is shorthand for $E \cup e$.

In this short abstract we only present the transformations which are different from the standard ones.

Fail

$$\frac{E \ ; \ (\lambda \bar{x}.x_e \overrightarrow{t} \doteq \lambda \bar{y}.y_d \overrightarrow{u}) \ \rhd \ \Lambda \mid \sigma}{\textsf{Fail}}$$

if the type of x_e is not isomorphic to the type of y_d.

We note that using the techniques of [ZGC03], failure can be detected in constant time, after a linear-time preprocessing step on the types of the original terms.

Decomposition

$$\frac{E \ ; \ (\lambda \bar{x}.x_e \overrightarrow{t} \doteq \lambda \bar{y}.y_d \overrightarrow{u}) \ \rhd \ \Lambda \mid \sigma}{E \ ; \ (\lambda \bar{x}.t_1 \doteq \lambda \bar{y}.u_{\pi(1)}) \ ; \ \ldots \ ; \ (\lambda \bar{x}.t_k \doteq \lambda \bar{y}.u_{\pi(k)}) \ \rhd \ \Lambda_+ \mid \sigma} \quad \text{if } \Lambda_+ \text{ is a consistent label mapping}$$

where Λ_+ is $\Lambda \cup \{x_e \mapsto y_d\}$. The condition "$\Lambda_+$ is a consistent label mapping" means that Λ_+ is induced by a set of partial labelled tree mappings between (the labelled trees for) the terms on the left-hand and right-hand sides of the equations in the system.

It is possible that no permutation π allows Λ to be extended to a consistent mapping by mapping x_e to y_d. In this case the procedure fails at this point. The decomposition can succeed either because Λ itself determines the decomposition π, or because some label $\approx$ with x_e or with y_d was already bound by Λ, or that no label was $\approx$ with either x_e or y_d and so we had freedom to extend Λ by $\{x_e \mapsto y_d\}$.

Theorem 12. *Replacing the traditional Decomposition transformation by the Decomposition and Fail transformations above yields a sound and complete set of transformations for higher-order unification and matching modulo type isomorphism. In fact the result of a successful sequence of transformations also yields the finite hereditary permutation witnessing the term-isomorphism between the instantiated terms.*

6 Ongoing work

This is a preliminary report, most of the interesting work remains to be done.

Of course we need to incorporate product types into our setting. We do not anticipate any conceptual challenges here, but the algorithmic complexity of working with the types is known to increase due to the fact that products allow more succinct representations of types.

The most important task before us is to derive efficient algorithms to guide the Decomposition transformation defined above. We need to explore the problem of determining when two labelled trees are isomorphic and in particular we require a top-down algorithm (an "online algorithm" in algorithms parlance) in order to be applicable to trees that are being generated during the unification process. Inspired by the results of Zibin, Gil and Considine in [ZGC03] we hope to find algorithms which will ultimately lead to unification and matching procedures which incur only a modest performance penalty for treating the more flexible notion of equality modulo type isomorphism.

It will be important to derive complexity results and to do empirical studies of the performance of our algorithms in cases known to be decidable, such as matching at low orders and unification of patterns.

References

[BCL92] Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

[DC76] Mariangiola Dezani-Ciancaglini. Characterization of normal forms possesing an inverse in the $\lambda_{\beta\eta}$-calculus. *Theoretical Computer Science*, 2:323–337, 1976.

[DC95] Roberto Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhauser, 1995. ISBN-0-8176-3763-X.

[FCB02] Marcelo Fiore, Roberto Di Cosmo, and Vincent Balat. Remarks on isomorphisms in typed lambda calculi with empty and sum types. In *Logic in Computer Science*, pages 147–156, Los Alamitos, CA, USA, July 22–25 2002. IEEE Computer Society.

[GS89] J. H. Gallier and W. Snyder. Higher-order unification revisited: complete sets of transformations. *Journal of Symbolic Computation*, 8:101–140, 1989.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Plo72] Gordon Plotkin. Building in equational theories. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence*, volume 7, pages 73–90. Edinburgh University Press, 1972.

[Rit90] M. Rittri. Retrieving library identifiers via equational matching of types. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 603–617. Springer, Berlin, Heidelberg, 1990.

[Rit91] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.

[RT91] C. Runciman and I. Toyne. Retrieving re-usable software components by polymorphic type. *jfp*, 1(2):191–211, 1991.

[ZGC03] Yoav Zibin, Yossi Gil, and Jeffrey Considine. Efficient algorithms for isomorphisms of simple types. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages (POPL 2003)*, pages 160–171. ACM Press, 2003.

[ZW93] Amy Moormann Zaremsky and Jeannette M. Wing. Signature matching: a key to reuse. In *First ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 182–190, 1993. ISBN:0-89791-625-5.

Pure Type Systems, Cut and Explicit Substitutions

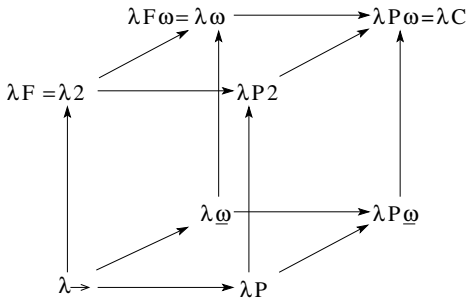
Romain Kervarc and Pierre Lescanne

École Normale Supérieure de Lyon – Laboratoire de l'Informatique du Parallélisme
46 allée d'Italie, 69364 Lyon cedex 07, France
romain.kervarc@ens-lyon.fr, pierre.lescanne@ens-lyon.fr

Abstract Pure type systems are a general formalism allowing to represent many type systems – in particular, Barendregt's λ-cube, including Girard's system $\mathcal{F}$, dependent types, and the calculus of constructions. We built a variant of pure type systems by adding a cut rule associated to an explicit substitution in the syntax, according to the Curry-Howard-de Bruijn correspondence. The addition of the cut requires the addition of a new rule for substitutions, with which we are able to show type correctness and subject reduction for all explicit systems. Moreover, we proved that the explicit λ-cube obtained this way is strongly normalizing.

1 Introduction

The calculus of constructions was designed by Coquand and Huet [7] as an extension of Girard's system $\mathcal{F}$ [11] in order to provide a general typed language for proof assistants based on λ-calculus. It was decomposed according to the different dependency kinds (type on type, term on type, type on term) in the formalism of pure type systems (PTS in short) into the so-called λ-cube by Barendregt in [3] (to which the reader may refer for a general presentation), starting from simple types and adding a kind of dependency (i.e. an axiom between sorts) on each edge to culminate at the calculus of constructions (cf. figure below).



Been based on the λ-calculus, the cube misses two key points. On the computational side, the λ-calculus does not give a complete account of the substitution since it is described in the meta-theory. Making it a first-class citizen yields calculi known as *calculi of explicit substitution* forking into two main families, namely with de Bruijn indices [1,17] and with explicit names [6,10]. On the logical side, it misses the important *cut rule*, which comes naturally as the typing rule for explicit substitution. Unlike Verstergaard and Wells [19] who advocate for de Bruijn indices when dealing with cut, we have chosen to consider the calculus with explicit names λx due to Bloo and Rose [6].

Following Bloo [5], we designed a variant of PTS that we call *explicit pure type systems* (EPTS in short) as he does, although our system is slightly different from his: our *cut* rule uses explicit substitution in both subject and predicate, and we have a new rule called x*pand*, which was needed

for subject reduction and is an avatar of the *drop* rule of Lengrand *et al.* in [16]. So with all these rules, we are able to prove correctness and subject reduction for *explicit pure type systems*. With axioms on sorts we build a cube similar to Barendregt's cube for λ-calculus; thus the main result of this paper is a proof of its strong normalization.

Related works. Other approaches considering cuts are Di Cosmo and Kesner [8] and Di Cosmo, Kesner and Polonovski [9], Verstergaard and Wells [19] and Herbelin [13]. In [18], Muñoz studies dependent types and explicit substitutions. Those approaches do not consider the whole cube and they are all but [13] and part of [9] in the framework of de Bruijn indices. Note also that Lengrand *et al.* [16], speak about cut. Anyway the closest work related to ours is this of Bloo [4,5], but he considers only explicit substitution in terms, not in types.

2 Explicit pure type systems

2.1 Syntax, reduction and typing

The basic definition of pure type systems remains unchanged: a pure type system is a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ of sorts, axioms (pairs of sorts) and rules (triples of sorts).

Definition 2.1: $\lambda\mathfrak{T}\mathtt{x}$*-calculus*

Let $\mathfrak{T} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ be a PTS. For $\sigma \in \mathcal{S}$, let $\mathcal{U}_\sigma$ be an infinite countable set of variables of *nature* σ. The set $\mathcal{E}\mathtt{x}(\mathfrak{T})$ of $\mathfrak{T}$*-expressions (with explicit substitutions)* is defined by:

$$E ::=^{\sigma} x \mid \sigma \mid E\,E \mid \lambda^{\sigma} x{:}E.E \mid \Pi^{\sigma} x{:}E.E \mid E\langle {}^{\sigma}x{:=}E\rangle \quad ; \quad {}^{\sigma}x \in \mathcal{U}_\sigma \quad ; \quad \sigma \in \mathcal{S}$$

Knowing that in $M\langle x{:=}N\rangle$, x is bound in M but not in N, the notion of $\alpha\mathtt{x}$*-equivalence* is defined as usual, and the quotient set $\Lambda\mathfrak{T}\mathtt{x} = \mathcal{E}\mathtt{x}(\mathfrak{T})/\overset{\alpha\mathtt{x}}{\equiv}$ is the set of the terms of the $\lambda\mathfrak{T}\mathtt{x}$-calcul, to which the operations of $\mathcal{E}\mathtt{x}(\mathfrak{T})$ can be canonically extended. In the rest of this section, we will forget about sort decoration for variables.

The sets $bv(M)$ and $fv(M)$ of the *bound* and *free* variables of M are defined as usual. The set $av(M)$ of the *available* variables of M is defined as in [16]; this notion is more relevant than free variables for terms with substitution. In what follows we shall as usual consider terms up to α-conversion and use Barendregt's convention.

The notion of reduction can now be defined:

Definition 2.2: $\beta\mathtt{x}$*-reduction in explicit PTS*

One considers the following reduction rules:

(B)	$(\lambda x{:}A.B)C \xrightarrow{B} B\langle x{:=}C\rangle$				
(quant)	$(\Pi y{:}A.B)\langle x{:=}C\rangle \xrightarrow{X} \Pi y{:}A\langle x{:=}C\rangle.B\langle x{:=}C\rangle$	(var)	$y\langle x{:=}N\rangle \xrightarrow{X} y$	if $x \neq y$	
(abs)	$(\lambda y{:}A.B)\langle x{:=}C\rangle \xrightarrow{X} \lambda y{:}A\langle x{:=}C\rangle.B\langle x{:=}C\rangle$	(gc)	$M\langle x{:=}N\rangle \xrightarrow{X} M$	if $x \notin av(M)$	
(app)	$(AB)\langle x{:=}C\rangle \xrightarrow{X} A\langle x{:=}C\rangle\, B\langle x{:=}C\rangle$	(subst)	$x\langle x{:=}N\rangle \xrightarrow{X} N$		

The $\mathtt{x}$*-reduction,* denoted $\xrightarrow{\mathtt{x}}$, is defined as the reduction relation induced by $\xrightarrow{X}$, which is called *head-*$\mathtt{x}$*-reduction*. The $\beta\mathtt{x}$*-reduction,* $\xrightarrow{\beta\mathtt{x}}$, is defined as the reduction relation induced by $\xrightarrow{X}$ and $\xrightarrow{B}$. Here we make use of the (gc) rule instead of the (var) rule on its own. In fact this does not make much difference.

Definition 2.3: *Typing in explicit PTS*

A *type assertion* is a couple denoted $M : N$ where M, the *subject*, belongs to $\Lambda\mathfrak{T}\mathtt{x}$ and N, the *predicate*, belongs to $\overline{\Lambda\mathfrak{T}\mathtt{x}} = \Lambda\mathfrak{T}\mathtt{x} \uplus \{\int\}$. The additional $\int$ element, named *pseudo-sort*, is a special

type introduced for reasons explained afterwards. $\overline{\mathcal{S}}$ is the disjoint union $\mathcal{S} \uplus \{\int\}$ and $\overline{\mathcal{A}}$ is the disjoint union $\mathcal{A} \uplus \{(\sigma, \int) / \sigma \in \mathcal{S}\}$.
Typing context are finite ordered sequences (just as in PTS) with predicates in $\Lambda\mathfrak{T}\mathtt{x}$.
Type judgements, of the form $\Gamma \vdash_{\mathfrak{T}} M : N$, are derived from the inference rules enounced in table 1 below. Index $\mathfrak{T}$ will be omitted in non-ambiguous cases.

$$\frac{(\overline{\sigma}, \overline{\tau}) \in \overline{\mathcal{A}}}{\vdash \overline{\sigma} : \overline{\tau}}\ \text{(axiom)} \qquad \frac{\Gamma \vdash A{:}\rho \quad \Gamma, x{:}A \vdash B{:}\sigma \quad (\rho,\sigma,\tau) \in \mathcal{R}; x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \Pi x{:}A.B : \tau}\ \text{(rule)}$$

$$\frac{\Gamma \vdash A : \sigma \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x{:}A \vdash x{:}A}\ \text{(hypothesis)} \qquad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : \sigma \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : C \vdash A : B}\ \text{(weakening)}$$

$$\frac{\Gamma \vdash (\Pi x{:}A.B){:}\sigma \quad \Gamma, x{:}A \vdash M{:}B \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda x{:}A.M : (\Pi x{:}A.B)}\ \text{(}\Pi\text{ - I)} \qquad \frac{\Gamma \vdash M : (\Pi x{:}A.B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B\langle x{:=}N\rangle}\ \text{(}\Pi\text{ - E)}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\langle x{:=}N\rangle : B\langle x{:=}N\rangle}\ \text{(cut)} \qquad \frac{\Gamma \vdash M{:}B \quad \Delta \vdash N{:}A \quad P\langle x{:=}N\rangle \xrightarrow[head]{X} M}{\Gamma \vdash P\langle x{:=}N\rangle : B}\ \text{(xpand)}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : \overline{\sigma} \quad A \overset{\beta\mathtt{x}}{\equiv} B}{\Gamma \vdash M : B}\ \text{(conversion)}$$

Table 1. Typing rules for EPTS

Explicit PTS contain two new rules with respect to PTS. (cut) corresponds to Bloo's (substitution) rule in [4] with substitution in both subject and predicate (i.e. term and type expression). It is the version with explicit names of Muñoz's (Clos_Π) rule in [18]. (xpand) is a not so straightforward generalization of the (drop) rule introduced in [16].

Well-formedness and reduction and equivalence between contexts are defined as usual.

2.2 The "pseudo-sort" type

The need for the extra pseudo-type $\int$ comes from the fact that in $\lambda\mathfrak{T}\mathtt{x}$, we apply substitutions not only to term expressions, but also to type expressions – whereas for instance Bloo, in [4], only applies explicit substitution to assertion subjects (i.e. term expressions), and not to assertion predicates (i.e. type expressions). This choice implies the possible apparition of types of the form $\sigma\langle x_1{:=}N_1\rangle \ldots \langle x_k{:=}N_k\rangle$, which in fact is the same as sort σ.
We therefore wished to extend the PTS (conversion) rule: $\frac{\Gamma \vdash M{:}A \qquad \Gamma \vdash B{:}\sigma \qquad A \overset{\beta\mathtt{x}}{\equiv} B}{\Gamma \vdash M{:}B}$ in order to be able to swap types A and B in case they both represent (i.e. are $\beta\mathtt{x}$-equivalent to) a same sort σ.

The premise $\Gamma \vdash B : \sigma$ of the (conversion) rule ensures that only correct types are used. The problem is how to extend it enough to allow to swap A and B and nevertheless preserve the correctness. It turned out that the most appropriate solution was to accept that B might be of the form $\tau\langle x_1{:=}N_1\rangle \ldots \langle x_k{:=}N_k\rangle$ provided that all N_i should be typable. As a solution we introduce the pseudo-sort $\int$, which intuitively means "disguised sort", but should not be included in the set of sorts, because it is well-known that type systems with the sort of all sorts are not consistent. We will manipulate $\int$ like a sort using (axiom) and (xpand), but we insist that **$\int$ is no sort**, since it appears

nowhere in the set of rules nor in a context type assertion. This solution is in fact very appropriate from the point of view of type correction.

2.3 The (xpand) rule

The introduction of this rule answers a specific identification problem linked with explicit substitutions. On the one hand, this rule generalizes the (drop) rule (cf. [16]), which enables to type some terms like $yz\langle x{:=}zy\rangle$. On the other hand, and this is the main reason of introduction of this rule, it solves the following problem: it is sometimes needed to invert the discarding order of two hypotheses, e.g. $x : A$ and $y : B$, but then x may well occur free in B. In the case of implicit substitutions, a *subsitution lemma* solves the problem by establishing that if $\Gamma, y : D, \Delta \vdash M : N$ and $\Gamma \vdash C : D$, then $\Gamma, \Delta[C/y] \vdash M[C/y] : N[C/y]$. But this lemma does not hold for explicit substitutions, two terms equal up to $\mathtt{x}$-conversion are then not *syntactically* equal. It is therefore necessary to lift the substitutions, and the ($\mathtt{x}$pand) rule does it. In particular, ($\mathtt{x}$pand) allows us to prove the subject reduction for general pure type systems with explicit substitution, unlike Bloo [4,5], who has a counter-example:

$$(\lambda x{:}a.(\lambda z{:}a.z)x)\langle a{:=}b\rangle \xrightarrow{\beta\mathtt{x}}\!\!\!\!\rightarrow_{head} \lambda x{:}b.((\lambda z{:}a.z)x)\langle a{:=}b\rangle.$$

One could object that the following rule is not satisfactory from the point of view of type inference, because it performs a kind of subject expansion. But in fact this is not the case, as for type inference rules must be read upward: this rule simply allows to "push" the explicit substitution inward enough to be able to type the term – which solves Bloo's problem. Moreover, this corresponds to the intuitive perception of explicit systems as lazy systems, where substitutions are not performed when not needed.

In the conclusion of [18], Muñoz writes about problems related to a rule that he calls (Clos_Π), which is in the framework of $\lambda\sigma$-like calculi of explicit substitution our cut rule. The problems which he mentions are solved by ($\mathtt{x}$pand).

3 Properties of explicit PTS

3.1 Type derivation lemmas

Lemma 3.1: *Initialization*
For any Γ well-formed: if $(\overline{\sigma} : \overline{\tau}) \in \overline{\mathcal{A}}$, then $\Gamma \vdash \overline{\sigma} : \overline{\tau}$ and if $(x : A) \in \Gamma$, then $\Gamma \vdash x : A$.

Lemma 3.2: *Weakening*
Let Γ and Δ be contexts, A, B be terms of $\overline{\Lambda\mathfrak{T}\mathtt{x}}$ such that $\Gamma \subseteq \Delta$, Δ be well-formed and $\Gamma \vdash A : B$. Then $\Delta \vdash A : B$.

Lemma 3.3: *Generation*
Let Γ be a context, M, T be terms of $\overline{\Lambda\mathfrak{T}\mathtt{x}}$. If $\Gamma \vdash M{:}T$, then:

$$\begin{array}{lll}
(i) & M = \sigma{\in}\overline{\mathcal{S}} & \Rightarrow \exists\tau{\in}\overline{\mathcal{S}}, T \overset{\beta\mathtt{x}}{\equiv} \tau \wedge (\sigma : \tau){\in}\mathcal{A} \\
(ii) & M = x{\in}\mathcal{U} & \Rightarrow \exists\tau{\in}\mathcal{S}, \exists U{\in}\Lambda\mathfrak{T}\mathtt{x}, \Gamma \vdash U : \tau \wedge (x : U){\in}\Gamma \wedge T \overset{\beta\mathtt{x}}{\equiv} U \\
(iii) & M = \Pi x{:}A.B & \Rightarrow \exists(\rho,\sigma,\tau){\in}\mathcal{R}, \Gamma \vdash A : \rho \wedge \Gamma, x : A \vdash B : \sigma \wedge T \overset{\beta\mathtt{x}}{\equiv} \tau \\
(iv) & M = \lambda x{:}A.B & \Rightarrow \exists\sigma{\in}\mathcal{S}, \exists C{\in}\Lambda\mathfrak{T}\mathtt{x}, \Gamma \vdash (\Pi x{:}A.C){:}\sigma \wedge \Gamma, x{:}A \vdash B{:}C \wedge T \overset{\beta\mathtt{x}}{\equiv} \Pi x{:}A.C \\
(v) & M = AB & \Rightarrow \exists C, D{\in}\Lambda\mathfrak{T}\mathtt{x}, \Gamma \vdash A : (\Pi x{:}C.D) \wedge \Gamma \vdash B : C \wedge T \overset{\beta\mathtt{x}}{\equiv} D\langle x{:=}B\rangle \\
(vi) & M = A\langle x{:=}B\rangle & \Rightarrow \exists C{\in}\Lambda\mathfrak{T}\mathtt{x}, D{\in}\overline{\Lambda\mathfrak{T}\mathtt{x}}, \Gamma, x : C \vdash A : D \wedge \Gamma \vdash B : C \wedge T \overset{\beta\mathtt{x}}{\equiv} D\langle x{:=}B\rangle \\
 & & \vee\ \exists\Delta, \exists C, D, E{\in}\Lambda\mathfrak{T}\mathtt{x}, \Gamma \vdash E{:}D \wedge \Delta \vdash B{:}C \wedge A\langle x{:=}B\rangle \xrightarrow{X} E \wedge T \overset{\beta\mathtt{x}}{\equiv} D
\end{array}$$

3.2 Type correctness and subject reduction

Theorem 3.4: *Type correctness*
Let $k \in \mathbb{N}$ be an integer, Γ be a context, A, B be two terms of $\overline{\Lambda\mathfrak{T}\mathtt{x}}$ such that $\Gamma \vdash A : B$. Then $\exists \overline{\sigma} \in \overline{\mathcal{S}}, \Gamma \vdash B : \overline{\sigma}$.

Theorem 3.5: *Subject reduction*
Let Γ, Γ' be contexts and A, A', B be three terms of $\Lambda\mathfrak{T}\mathtt{x}$ such that $\Gamma \vdash A : B$, $\Gamma \xrightarrow{\beta\mathtt{x}}\!\!\!\!\rightarrow \Gamma'$ and $A \xrightarrow{\beta\mathtt{x}}\!\!\!\!\rightarrow A'$. Then $\Gamma' \vdash A' : B$.

4 Strong normalization for the explicit λ-cube

In this part we will only consider EPTS from the λ-cube, that is, with the following sorts and axioms: $\mathcal{S} = \{*, \Box\}$ and $\mathcal{A} = \{* : \Box\}$ and rules among $\mathcal{R}^{e}_{\mathcal{S}} = \{[*,*], [*,\Box], [\Box,*], [\Box,\Box]\}$. To show that thess systems are strongly normalizing, it is enough to show that $\lambda\mathtt{Cx}$ is, as:

Lemma 4.1:
Let $\mathfrak{T}$ be an EPTS of the λ-cube. Let Γ, A, B be such that $\Gamma \vdash_{\mathfrak{T}} A : B$. Then $\Gamma \vdash_{\lambda\mathtt{Cx}} A : B$.

The proof of strong normalization in $\lambda\mathtt{Cx}$ is achieved through a two-step reduction:

(*i*) show that $\lambda\mathtt{Cx}$ is strongly normalizing if $\lambda\omega\mathtt{x}$ is;
(*ii*) show that $\lambda\omega\mathtt{x}$ is strongly normalizing if $\mathcal{F}\,\mathrm{x}$ is.

$\mathcal{F}\,\mathrm{x}$ is an adaptation to $\lambda\mathrm{x}$ of Girard's system $\mathcal{F}$ (cf. [12]), which we defined in a former work [14] and proved to be strongly normalizing.And hence the theorem holds:

Theorem 4.2: *Strong normalization for the* $\lambda\mathrm{x}$*-cube*
All pure type systems of the explicit λ-cube are strongly normalizing.

5 Conclusion

We have studied explicit pure type systems (EPTS) which are an extension of pure type systems (PTS) where λ-calculus is replaced by the calculus of explicit substitution $\lambda\mathrm{x}$.

We have defined in these EPTS an equivalent of Barendregt's λ-cube and we have proven that the explicit pure type systems of this cube fulfill *subject reduction* and are *strongly normalizing*. We claim that our EPTS are the simplest and the most complete extensions of PTS with mechanisms for internalized substitution.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, J.-J. Lévy. Explicit substitutions. In *Proc. of 17th ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, California, U.S.A., 1990. ACM.
2. H. Barendregt. *Lambda-calculus, its Syntax and its Semantics.* Elsevier Science Publishers B.V. (North Holland), Amsterdam, 1984.
3. H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2 (*Background: Computational Structures*), pages 117–309. Oxford University Press, 1992.
4. R. Bloo. *Preservation of Termination for Explicit Substitution*. Proefschrift ter verkrijging van de graad van Doctor, Technische Universiteit Eindhoven, Oct. 1997.

5. R. Bloo. Pure type systems with explicit substitution. *Mathematical Structures in Computer Science*, 11(1):3–19, Feb. 2001.
6. R. Bloo, K. H. Rose. Preservation of strong normalisation in named lambda-calculi with explicit substitution and garbage collection. In *CSN '95*, pages 62–72, 1995.
7. T. Coquand, G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2):95–120, 1988.
8. R. Di Cosmo, D. Kesner. Strong normalization of explicit substitutions via cut elimination in proof nets. In *Proc. of 12th IEEE Symposium on Logic in Computer Science*, pages 35–46, Warsaw, Poland, 1997. Warsaw University, IEEEC Society Press.
9. R. Di Cosmo, D. Kesner, E. Polonovski. Proof nets and explicit substitutions. *Mathematical Structures in Computer Science*, 13(3):409–450, June 2003.
10. D. Dougherty, P. Lescanne. Reduction, intersection types and explicit substitutions. *Mathematical Structures in Computer Science*, 13(1):55–85, 2003.
11. J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. thèse de doctorat, Université Paris VII, June 1972.
12. J.-Y. Girard, Y. Lafont, P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.
13. H. Herbelin. Explicit substitutions and reducibility. *Journal of Logic and Computation*, 11(3):29–449, 2001.
14. R. Kervarc. Substitutions explicites dans le λ-cube. Rapport de DÉA, École normale supérieure de Lyon, July 2002. LIP DÉA Report Nr. 2002-04.
15. J.-L. Krivine. *Lambda-calcul, types et modèles*. Masson, Paris, 1990.
16. S. Lengrand, P. Lescanne, D. Dougherty, M. Dezani-Ciancaglini, S. van Bakel. Intersection types for explicit substitutions. *Information and Computation*, 189(1):17–42, 2004.
17. P. Lescanne. From λσ to λ*v*: a journey through calculi of explicit substitutions. In *Proc. of 21th ACM Symposium on Principles of Programming Languages*, pages 60–69, Portland, Oregon, U.S.A., 1994. ACM.
18. C. Muñoz. Dependent types, explicit substitutions. *Mathematical Structures in Computer Science*, 11(1):91–129, 2001.
19. R. Vestergaard, J. B. Wells. Cut rules and explicit substitutions. *Mathematical Structures in Computer Science*, 11(1):131–168, 2001.

PSN Implies SN

Emmanuel Polonovski

PPS, CNRS - Université Paris 7
Emmanuel.Polonovski@pps.jussieu.fr

Abstract. In the framework of explicit subsitutions there is two termination properties: preservation of strong normalization (PSN), and strong normalization (SN). Since there are not easily proved, only one of them is usually established (and sometimes none). We propose here a connection between them which helps to get SN when one already has PSN. For this purpose, we formalize a general proof technique of SN which consists in expanding substitutions into "pure" λ-terms and to inherit SN of the whole calculus by SN of the "pure" calculus and by PSN. We apply it successfully to a large set of calculi with explicit substitutions, allowing us to establish SN, or, at least, to trace back the failure of SN to that of PSN.

1 Introduction

Calculi with explicit substitutions were introduced [1] as a bridge between λ-calculus [7] and concrete implementations of functional programming languages. Those calculi intend to refine the evaluation process by proposing reduction rules to deal with the substitution mechanism – a *meta*-operation in the traditional λ-calculus. It appears that, with those new rules, it was much harder (and sometimes impossible) to get termination properties. The two main termination properties of calculi with explicit substitutions are:

- **Preservation of strong normalization** (PSN), which says that if a pure term (*i.e.* without explicit substitutions) is strongly normalizing (*i.e.* cannot be infinitely reduced) in the pure calculus (*i.e.* the calculus without explicit substitutions), then this term is also strongly normalizing with respect to the calculus with explicit substitutions.
- **Strong normalization** (SN), which says that, with respect to a typing system, every typed term is strongly normalizing in the calculus with explicit substitutions, *i.e.* every terms in the subset of typed terms cannot be infinitely reduced.

These two properties are not redundant, and Fig. 1 shows the differences between them. PSN says that the horizontally and diagonally hatched rectangle is included in the diagonally hatched rectangle. SN says that the vertically hatched rectangle is included in the diagonally hatched rectangle. Even if they work on a different set of terms, there is a common part: the vertically and horizontally hatched rectangle, wich represent the typed pure terms.

SN and PSN are both termination properties, although their proofs are not always clearly related: sometimes SN is shown independently of PSN (directly, by simulation, *etc.*, see for example [12,11]), sometimes SN proofs uses PSN (see for example [4]). We present here a general proof technique of SN via PSN, initially suggested by H. Herbelin, which uses that common part of typed pure terms.

In section 2, we formalize the technique and in section 3 we summarize the results we achieved by applying it to a set of calculi. This set has been choosen for the variety of

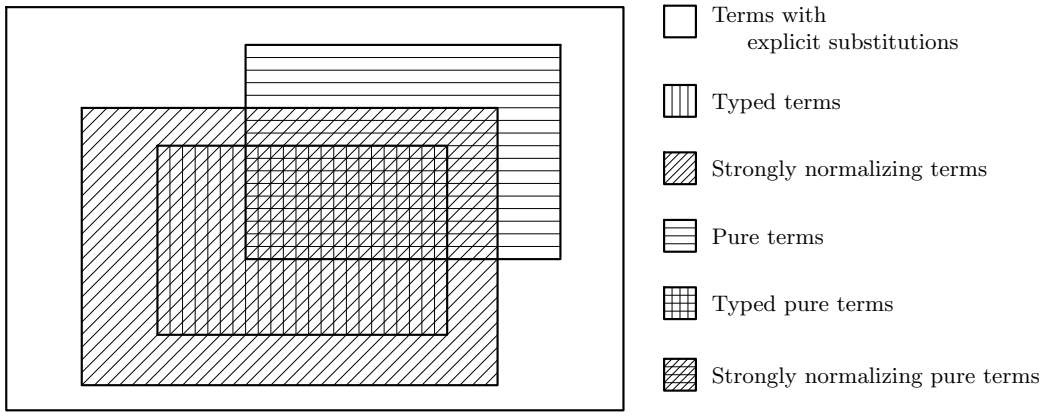


Fig. 1. Normalization properties of terms with and without explicit substitutions

their definitions: with or without De Bruijn indices, unary or multiple substitutions, with or without composition of substitutions, and even a symmetric non-deterministic calculus. In the last section, we briefly talk about perspectives in this framework.

2 Proof Technique

The idea of this technique is the following. Let t be a typed term with explicit substitutions for which we want to show termination. With the help of its typing judgment, we build a typed pure term t' which can be reduced to t. For that purpose, we expand the substitutions of t into redexes. We call this expansion *Ateb* (the opposite of *Beta* which is usually the name of the rule which creates explicit substitutions). Then, with SN of the pure calculus and PSN, we can export the strong normalization of t' (in the pure calculus) to t (in the calculus with explicit substitutions).

In practice, this sketch will only apply in some cases, and some others will require some adjustment to this technique. For our technique to work, we need that the *Ateb* expansion satisfies some properties. The first one is always easily checked.

Property 1 (Preservation of typability). If t is typable, with respect to a typing system T, in the calculus with explicit substitution, then $Ateb(t)$ is typable, with respect to a typing system T' (possibly $T' = T$) in the pure calculus.

Only some calculi can exhibit an *Ateb* function which satisfies the second one.

Property 2 (Initialization). $Ateb(t)$ reduces to t in zero or more steps in the calculus with explicit substitutions.

If we can get it, then we use the direct proof to be presented in section 2.1. Otherwise, we need to use the simulation proof to be presented in section 2.2. In the sequel, $\mathcal{SN}$ will be the set of strongly normalizing pure terms and $\mathcal{SN}_x$ will be the set of strongly normalizing terms of the calculus with explicit substitutions.

2.1 Direct proof

We can immediately establish the theorem.

Theorem 1. *For all typing systems T and T' such that, in the pure calculus, all typable terms with respect to T are strongly normalizing, if there exists a function Ateb from explicit substitution terms to pure terms satisfying properties 1 and 2 then PSN implies SN (with respect to T').*

Proof. For every typed term t of the calculus with explicit substitution, $Ateb(t)$ is a pure typed term (by property 1). By the strong normalization hypothesis of the typed pure calculus, we have $Ateb(t) \in \mathcal{SN}$. By hypothesis of PSN we obtain that $Ateb(t)$ is in $\mathcal{SN}_x$. By property 2, we get $Ateb(t) \to^* t$, which gives us directly $t \in \mathcal{SN}_x$.

2.2 Simulation proof

We must relax some constraints on $Ateb$. We will try to find an expansion of t to t' such that t' reduces to a term u and there exists a relation $\mathcal{R}$ with $u\mathcal{R}t$. The choosen relation must, in addition, enable a simulation of the reductions of t by the reduction of u. If it is possible, we can infer strong normalization of t from strong normalization of u.

To proceed with the simulation, we first split the reduction rules of the calculus with explicit substitutions into two disjoints sets. The set R_1 contains rules which are trivially terminating, and R_2 contains the others. Secondly, we build a relation $\mathcal{R}$ which satisfies the following properties.

Property 3 (Initialisation). For every typed term t, there exists a term $u\mathcal{R}t$ such that $Ateb(t)$ reduces in 0 or more steps to u in the calculus with explicit substitutions.

Property 4 (Simulation *). For every term t, if $t \to_{R_1} t'$ then, for every $u\mathcal{R}t$, there exists u' such that $u \to^* u'$ and $u'\mathcal{R}t'$.

Property 5 (Simulation $^+$). For every term t, if $t \to_{R_2} t'$ then, for every $u\mathcal{R}t$, there exists u' such that $u \to^+ u'$ and $u'\mathcal{R}t'$.

We display those properties as diagrams :

Initialisation	Simulation *	Simulation $^+$
$\begin{array}{rc} & t \\ \swarrow & \mathcal{R} \\ Ateb(t) \to^* & u \end{array}$	$\begin{array}{ccc} t & \to_{R_1} & t' \\ \mathcal{R} & & \mathcal{R} \\ u & \to^* & u' \end{array}$	$\begin{array}{ccc} t & \to_{R_2} & t' \\ \mathcal{R} & & \mathcal{R} \\ u & \to^+ & u' \end{array}$

With this material, we can establish the theorem.

Theorem 2. *For all typing systems T and T' such that, in the pure calculus, all typable terms with respect to T are strongly normalizing, if there exists a function Ateb from explicit substitution terms to pure terms and a relation $\mathcal{R}$ on explicit substitutions terms satisfying properties 1, 3, 4 and 5 then PSN implies SN (with respect to T').*

Proof. We prove it by contradiction. Let t be a typed term with explicit substitutions which can be infinitly reduced. By property 3 there exists a term u such that $Ateb(t) \rightarrow^* u$, and $Ateb(t)$ is a pure typed term (by property 1). By the strong normalization hypothesis of the typed pure calculus, we have $Ateb(t) \in \mathcal{SN}$. By hypothesis of PSN we obtain that $Ateb(t)$ is in $\mathcal{SN}_x$ and it follows that $u \in \mathcal{SN}_x$.

By property 3, we also have $u\mathcal{R}t$, and, with properties 4 and 5, we can build an infinite reduction from u, contradicting the strong normalization of u.

3 Results

3.1 λx-calculus

The λx-calculus [6,5] is probably the simplest calculus with explicit substitutions. It only makes the subtitution explicit. Since this calculus provides no rules to deal with substitutions composition, it preserves strong normalization. It is for this calculus that the technique has been originately used by Herbelin. Therefore, we can without surprises apply the direct proof to get strong normalization.

3.2 $\lambda\upsilon$-calculus

The $\lambda\upsilon$-calculus [16,3] is the De Bruijn counterpart of λx. As λx, it has no composition rules, and therefore satisfies PSN. For this calculus, we must use the simulation proof to deal with indices modification operators. We succeed to use it and it is, as far as we know, the first proof of SN for a simply typed version of $\lambda\upsilon$ (see [19]).

3.3 λ_{ws}-calculus

The λ_{ws}-calculus [13,9,10] introduces an explicit weakening operator, which allows to preserve strong normalization even with composition rules. It has already been shown to be SN [12]. We fail to apply the technique, due to the explicit weakening operator combined with the rigidity of the typing environment one usually has in calculi with De Bruijn indices.

3.4 λ_{wsn}-calculus

In [12] a named version of λ_{ws} was proposed. In current work, we developed a new version of this calculus : λ_{wsn}. We already have a SN proof for this calculus, almost similar to the original one, and this technique can be applied, using the direct proof. We cannot conclude to SN by this way, since PSN has not yet been shown (see [19]).

3.5 $\lambda\sigma$-calculus

The well known $\lambda\sigma$-calculus [1] does not have either PSN nor SN, as shown in [17]. However, we can successfully apply our technique, using the simulation proof. It does not gives us SN, but it reduces the SN problem to that of PSN. If someone proposes a strategy which preseves strong normalization, our work will give immediately a SN proof.

3.6 $\lambda\sigma_n$-calculus

Introduced in the same work [1], the named version of $\lambda\sigma$ suffers the same problem concerning PSN. We can also apply the simulation proof to it, and conclude similarly.

3.7 $\overline{\lambda}\mu\tilde{\mu}x$-calculus

The $\overline{\lambda}\mu\tilde{\mu}$-calculus [8,14] is a symmetric version of the $\lambda\mu$-calculus [18]. As for symmetric λ-calculus [2], the symmetry raises difficulties in normalization proofs. We can build an explicit substitutions version "à la" $\lambda\mathtt{x} : \overline{\lambda}\mu\tilde{\mu}\mathtt{x}$. In [20], we apply successfully the technique, by direct proof, to show its strong normalization.

4 Perspectives

It seems that this technique can be used for many calculi with explicit substitutions. Its application on named calculi is easy, in general, and leads to a simple direct proof. For some others, as for calculi with De Bruijn indices, we must use the simulation proof, which tend to be not so easy. Further work includes its application to the λ_{ws}-calculus and to the λlxr-calculus [15].

References

1. Abadi, M., Cardelli, L., Curien, P.-L., Lévy, J.-J.: Explicit Substitutions. Journal of Functional Programming (1991).
2. Barbanera, F., Berardi, S.: A symmetric lambda-calculus for classical program extraction. Proceedings of TACS'94 (1994), Springer-Verlag LNCS **789**, 495–515.
3. Benaissa, Z.-E.-A., Briaud, D., Lescanne, P., Rouyer-Degli, J.: λv, a calculus of explicit substitutions which preserves strong normalisation. Journal of Functional Programming (1996).
4. Bloo, R.: Preservation of Termination for Explicit Substitutions. PhD thesis, Eindhoven University (1997).
5. Bloo, R., Geuvers, H.: Explicit Substitution: on the Edge of Strong Normalisation. Theoretical Computer Science (TCS 1999), **211**, 375–395.
6. Bloo, R., Rose, K.: *Preservation of strong normalization in named lambda calculi with explicit substitution and garbage collection.* In Computing Science in the Netherlands, pages 62-72. Netherlands Computer Science Research Foundation, 1995.
7. Church, A.: The Calculi of Lambda Conversion. Princeton Univ. Press (1941).
8. Curien, P.-L., Herbelin, H.: The duality of computation. Proceedings of ICFP'00 (2000), ACM Press, 233–243.
9. David, R., Guillaume, B.: The λ_l-calculus. In D. Kesner, editor, Proceedings of the 2nd Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs, pages 2-13, July 1999.
10. David, R., Guillaume, B.: A λ-calculus with explicit weakening and explicit substitution. Mathematical Structures in Computer Science, 11, 2001.
11. David, R., Guillaume, B.: Strong Normalisation of the Typed λ_{ws}-calculus. In Proceedings of the 17th International Worshop Computer Science Logic (CSL 2003), volume 2803 of Lecture Notes in Computer Science, pages 155-168. Springer, Vienna, 2003.
12. Di Cosmo, R., Kesner, D., Polonovski, E.: Proof nets and explicit substitutions. In J. Tiuryn, editor, Foundations of Software Science and Computation Structures (FOSSACS 2000), volume 1784 of Lecture Notes in Computer Science, pages 63-81. Springer-Verlag, Mar. 2000.
13. Guillaume, B.: Un calcul de substitution avec étiquettes. PhD thesis, Université de Savoie (1999).

14. Herbelin, H.: Explicit substitutions and reducibility. Journal of Logic and Computation (2001), **11**, 429–449.
15. Kesner, D., Lengrand, S.: Broadening the horizon of the explicit substitution paradigm via a logical model. Submitted paper (2004).
16. Lescanne, P.: From lambda-sigma to lambda-upsilon: a journey through calculi of explicit substitutions. In 21st ACM Symposium on Principles of Programming Languages (POPL'94), 16-19 Janvier 1994, Portland, Oregon, pp 60-69.
17. Mellies, P.-A.: Typed λ-calculi with explicit substitutions may not terminate. Proceedings of TLCA'95 (1995), Springer LNCS, **902**, 328–334.
18. Parigot, M.: $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. Proceedings of LICS'93 (1993), Computer Society Press, 39–46.
19. Polonovski, E.: Substitutions explicites, logique et normalisation. PhD thesis, Université Paris 7, (2004). In preparation.
20. Polonovski, E.: Strong normalization of $\overline{\lambda}\mu\tilde{\mu}$-calculus with explicit substitutions. In Proceedings of Foundations of Software Science and Computation Structures (FOSSACS 2004), LNCS 2987, Mar. 2004.

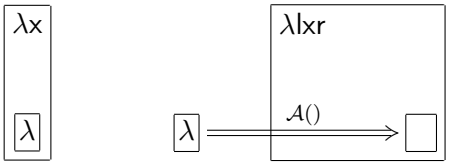
Deriving Strong Normalisation

Stéphane Lengrand

School of Computer Science, University of St Andrews, United Kingdom.
PPS, Université Paris 7, France. lengrand@pps.jussieu.fr
WWW home page: http://www.pps.jussieu.fr/lengrand/

Abstract We present a somewhat general technique to derive the strong normalisation of some specific terms of various calculi with explicit substitutions from the strong normalisation of some (well-chosen) λ-terms. One main application of the method is proving the *Preservation of Strong Normalisation* (PSN) of various explicit substitution calculi and the strong normalisation of the Cut-elimination in various sequent calculi.

- The notion of PSN concerns syntactic extensions of λ-calculus with their own reduction systems and states that if a λ-term is strongly normalising for the β-reduction, then it is still strongly normalising when considered as a term of the extended calculus undergoing its corresponding reduction system. Such extensions of λ-calculus are explicit reduction calculi like $\lambda\mathsf{x}$ [BR95], the reduction system of which reduces β-redexs and evaluate the explicit substitutions.
  The definition of the PSN property can be slightly generalised for calculi in which λ-calculus can be *embedded* (by a one-to-one translation, say $\mathcal{A}$), and the embedding need not be the identity/inclusion. In that case PSN states that if a λ-term is strongly normalising, then its encoding is also strongly normalising. This is the case for the explicit substitution calculus $\lambda\mathsf{lxr}$ introduced in [KL04] which requires terms to be linear and hence is not a syntactic extension of λ-calculus.



- Another application is deriving the strong normalisation of Cut-elimination in intuitionistic sequent calculus. When using a term-assignment system to denote proofs of sequent calculus, the Cut-rule is actually the typing rule of a term constructor that very much looks like an explicit substitution, so that Cut-elimination consists in fully propagating this "explicit substitution" operator.

1 The idea of simulation and its problems

The base idea to prove that a term M of a calculus with explicit substitutions is SN is to interpret it as a strongly normalising λ-term $\mathsf{L}(M)$ and then simulate its reductions by β-reductions of its interpretation. Hence, after a certain number of steps, only reductions

can take place that no longer modify the interpretation. Then it would suffice to show that the reductions that do not necessarily induce at least one β-reduction terminate, so that no infinite reduction sequence can start from M, as illustrated in Figure 1.

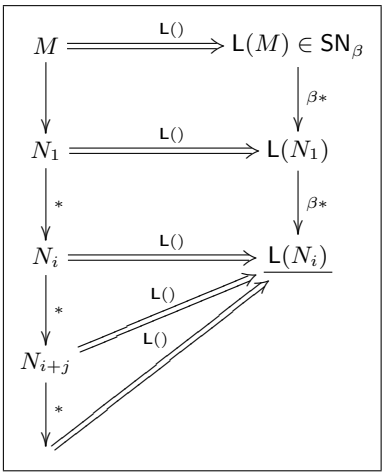


Figure 1. Deriving strong normalisation by simulation

For PSN, if $M = \mathcal{A}(t)$ where t is the λ-term supposed to be SN_β, then we would take $\mathsf{L}(M) = t$. For sequent calculus, it would be a typed (and hence strongly normalising) λ-term that denotes a proof in natural deduction of the same sequent (using Curry-Howard correspondence), so that the idea of simulating Cut-elimination by β-reductions is closely related to Prawitz's correspondence [Pra65].

There is one problem in doing so: an encoding into λ-calculus that allows the simulation needs to interpret explicit substitutions by implicit substitutions like $t\{x = u\}$. But should x not be free in t, all reduction steps taking place within the term of which u is the encoding would not induce any β-reduction in $t\{x = u\}$.

Therefore, the sub-system consisting of all the reductions that are not necessarily simulated by at least one β-reduction is too big to be proven terminating (and very often it is not).

2 Refining the simulation by using Church-Klop's λI-calculus

Our contribution is to overcome this problem by encoding a calculus with explicit substitutions in Church-Klop's λI-calculus [Klo80] instead of λ-calculus. On the one hand, λI extends the syntax of λ-calculus with a "memory operator" so that, instead of being thrown away, a term N can be retained and carried along in a construct $[-, N]$. With this operator, those bodies of substitutions are encoded that would otherwise disappear, as explained above. On the other hand, λI restricts λ-abstractions to variables that have at least one free occurrence, so that β-reduction never erases its argument.

Doing so requires the encoding in λI to be non-deterministic, i.e. we define a relation $\mathcal{H}$ between the calculus and λI, and the reason for this is that, since the reductions in λI are non-erasing reductions, we need to add this memory operator at random places in the encoding, using such a rule:

$$\frac{M \mathcal{H} T}{M \mathcal{H} [T, U]} U \in \lambda I$$

For instance, $\lambda x.x \mathcal{H} \lambda x.[x, x]$ but also $\lambda x.x \mathcal{H} [\lambda x.x, \lambda z.z]$, so that both $\lambda x.[x, x]$ and $[\lambda x.x, \lambda z.z]$ (and also $\lambda x.x$) are encodings of $\lambda x.x$.

Another problem that arises is that explicit substitutions are pushed down in the terms of the calculus, whereas the memory operator cannot be pushed down in λI, making the simulation difficult. Hence, we use a subtle side-condition for the encoding of an explicit substitution $\langle N/x \rangle M$:

$$\frac{M \mathcal{H} T \quad N \mathcal{H} U}{\langle N/x \rangle M \mathcal{H} T\{x = U\}} x \in FV(T) \text{ or } N \in \mathsf{SN}$$

1. Always requiring $x \in FV(T)$ would be too strong, because the substitution $\langle N/x \rangle$ can be propagated into the sub-terms of M whereas x has no reason to be free in the corresponding sub-terms of T, so the simulation theorem would not hold.
2. On the contrary, no side-condition would make the encoding too relaxed because U could disappear and the reduction steps that could take place within N would be simulated by zero reduction steps so we would have the same problem as with λ-calculus.
3. The side-condition $x \in FV(T)$ or $N \in \mathsf{SN}$ is the adequate balance:
   - First, when we reduce N or reduce T, the side-condition is still preserved (because reductions in λI are non-erasing).
   - If $N \notin \mathsf{SN}$, then $x \in FV(T)$ is required and for the same reason as in point 1 we cannot simulate the propagation of $\langle N/x \rangle$. So we need to prove that from a term which is not strongly normalising we can find an infinite reduction sequence that never propagates such a substitution, and this can be done because instead of propagating $\langle N/x \rangle$, we can leave it where it is and infinitely reduce N instead.
   - If on the contrary $N \in \mathsf{SN}$, then we can simulate its propagation because we never care whether $x \in FV(\ldots)$, but for the same reason as in point 2, reduction steps within N might be simulated by zero reduction steps, but because $N \in \mathsf{SN}$ this can only happen finitely many times (we call those reductions *safe reductions*).

Eventually, the reduction relation of the explicit substitution calculus is split into two parts Y and Z that satisfy the following simulation theorem:

Theorem 1 (Simulation). *Suppose $M \mathcal{H} T$.*

- *If $M \longrightarrow_Y N$, there is a U such that $N \mathcal{H} U$ and $T \longrightarrow^+ U$.*
- *If $M \longrightarrow_Z N$, there is a U such that $N \mathcal{H} U$ and $T \longrightarrow^* U$.*

Now it must be proven that every term M can be encoded into a strongly normalising term of λI. This depends on the calculus that is being treated, but the following method generally works:

- Encode the term M as a strongly normalising λ-term t, such that no sub-term is lost, i.e. *not* using implicit substitutions. For PSN, the original λ-term would do, because it is strongly normalising by hypothesis; for a proof-term of sequent calculus, t would be a λ-term typed in an appropriate typing system, the typing tree of which is derived from the proof-tree of the sequent (we would get $t \in \mathsf{SN}_\beta$ using a theorem stating that typed terms are SN_β).
- Then encode t in λI with the following encoding:

$$\boxed{\begin{array}{lll} \mathsf{i}(x) & = x & \\ \mathsf{i}(\lambda x.t) & = \lambda x.\mathsf{i}(t) & x \in FV(\mathsf{i}(t)) \\ \mathsf{i}(\lambda x.t) & = \lambda x.[\mathsf{i}(t), x] & x \notin FV(\mathsf{i}(t)) \\ \mathsf{i}(t\ u) & = \mathsf{i}(t)\ \mathsf{i}(u) & \end{array}}$$

  In [KL04] we prove that if a λ-term t is strongly normalising for β-reductions, then $\mathsf{i}(t)$ is weakly normalising in λI. The proof simply consists in simulating an adequate reduction sequence that starts from t and ends with a normal form, the encoding of which is a normal form of λI. What makes this simulation work is the fact that the reduction sequence is provided by a perpetual strategy (the definition of which involves side-conditions similar to that of the above inference rule). Also, weak normalisation implies strong normalisation in λI [Ned73], so $\mathsf{i}(t)$ is strongly normalising.
- Then prove that $\mathsf{i}(t)$ reduces to one of the non-deterministic encodings of M in λI, that is, that there is a term T such that $M \mathcal{H} T$ and $\mathsf{i}(t) \longrightarrow^* T$.

The technique is summarised in Figure 2.

Finally, it remains to be proven that the relation Z that consists of the reductions that are not simulated in at least one step is now small enough to be terminating.

3 Conclusion

This technique works for $\lambda\mathsf{x}$, giving a new proof of PSN (already shown in [BR95]), as well as for proving PSN of the explicit substitution calculus λ_{lxr} of [KL04], and for various sequent calculi that range from propositional logic to a logic as expressive as the Calculus of Constructions, and we believe that it can be applied to many other calculi.

References

[BR95] R. Bloo and K. H. Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *CSN '95 – Computer Science in the Netherlands*, pages 62–72, Koninklijke Jaarbeurs, Utrecht, November 1995.

[KL04] D. Kesner and Stéphane Lengrand. Broadening the horizon of the explicit substitution paradigm via a logical model. *submitted to CSL' 2004.*

[Klo80] Jan-Willem Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. CWI, Amsterdam, 1980. PhD Thesis.

[Ned73] Rob Nederpelt. *Strong Normalization in a Typed Lambda Calculus with Lambda Structured Types*. PhD thesis, Eindhoven University of Technology, 1973.

[Pra65] D. Prawitz. Natural deduction. a proof-theoretical study. In *Acta Universitatis Stockholmiensis*, volume 3. Almqvist & Wiksell, 1965.

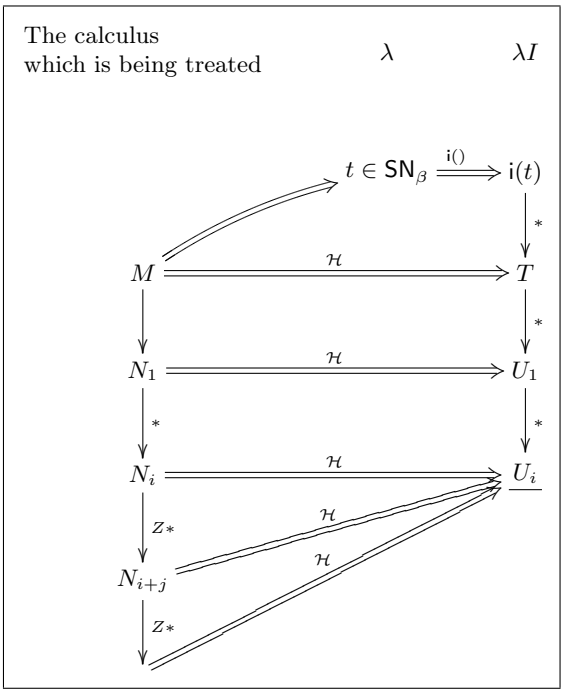


Figure 2. The general technique to prove that $M \in \mathsf{SN}$

Higher-Order Rewriting with Types and Arities

Jean-Pierre Jouannaud[1] and Femke van Raamsdonk[2] and Albert Rubio[3]

[1] LIX, École Polytechnique, 91400 Palaiseau, France
jouannaud@lix.polytechnique.fr
[2] Faculty of Sciences, Vrije Universiteit, Amsterdam, The Netherlands
femke@cs.vu.nl
[3] Technical University of Catalonia, Pau Gargallo 5, 08028 Barcelona, Spain
rubio@lsi.upc.es

Abstract. We introduce a new framework for higher-order rewriting where function symbols and variables have both a type and an arity. Rewriting is defined modulo $\beta\eta$. A novel feature is that rules of functional type are admitted. It is shown that rewriting and equality coincide, a critical pair lemma is given, and a termination method based on the higher-order recursive path ordering is given.

1 Introduction

Background. Higher-order rewrite rules are increasingly used in programming languages and logical systems, with three main goals: describing computations, rule-based decision procedures, and encoding other logical systems. As usual, the choice of the syntax in which the rules are expressed plays a crucial role, as well as the definition of rewriting itself. Several proposals for frameworks of higher-order rewriting have been made by Klop, Khasidashvili, Nipkow, and by Jouannaud and Okada.

The first proposal for a format of higher-order rewriting are the *Combinatory Reduction Systems* (CRSs) defined by Klop [8], inspired by the Contraction Schemes of Aczel [1]. Independently of Klop, Khasidashvili [7] introduced the similar framework of *Expression Reduction Systems* (ERSs). CRSs are second-order systems, and are untyped. A drawback is that the definition of rewriting makes use of a specific meta-language.

The second proposal are the *Higher-Order Rewrite Systems* (HRSs) introduced by Nipkow [11,9]. In HRSs we work with simply typed λ-calculus as a meta-language, and pattern-matching is modulo $\beta\eta$. This is called higher-order pattern matching. The left- and right-hand side of a rewrite rule of a HRS must be of *base type*. This rules out a rule of the form $\mathsf{diff}(\lambda x.\,\mathsf{sin}(x)) \to \lambda x.\,\mathsf{cos}(x)$. Further, rules must be written in η-long form, which may be cumbersome, as for instance in the following rule for map: $\mathsf{map}(\lambda x.\,F(x), \mathsf{cons}(h,t)) \to \mathsf{cons}(F(h), \mathsf{map}(\lambda x.\,F(x), t)$, where would be more natural to use F instead of $\lambda x.\,F(x)$.

The third proposal was made by Jouannaud and Okada [4,2]. In the *Algebraic-Functional Systems* (AFSs) the rewrite relation is induced by a set of algebraic rules and the β-reduction rule. In AFSs the definition of substitution is not internalized, so the right-hand side of the β-reduction rule is not a term but uses meta-notation. AFSs use plain pattern matching for firing rules, which is too weak for most encodings.

Problem. The question that is considered in this work is how to define typed higher-order rewriting modulo $\beta\eta$ (with higher-order pattern-matching) where function symbols and variables have types and arities such that higher-order rewrite rules can be allowed.

Contribution. We define a framework for higher-order rewriting that combines the advantages of the three earlier introduced frameworks. Rewriting is modulo $\beta\eta$ but uses $\beta\eta$-normal forms instead of long-$\beta\overline{\eta}$-normal forms as representatives of a $\beta\eta$-equivalence class. A novel feature of our framework is that rules of functional type are allowed.

2 The framework

In this section we discuss a few aspects of our framework.

Types. The set of (simple) *types* is inductively defined by the following grammar: $\sigma, \tau ::= s(\sigma_1, \ldots, \sigma_n) \,|\, \sigma \to \sigma'$, where s is a sort symbol of arity n. A type is *functional* if it is of the form $\sigma \to \sigma'$, and a *base type* otherwise.

Type declarations. In our framework, function symbols and variables have besides a *type* also an *arity* which is a natural number denoting the number of arguments it is supposed to have. That is, we combine a typed setting as in HRSs and AFSs with the presence of arities as in CRSs (all variables have arity 0 in AFSs). In order to express the combination of a type and an arity, we use type declarations as in AFSs. A *type declaration* is an expression of the form $\sigma_1 \times \ldots \times \sigma_n \Rightarrow \tau$ with $\sigma_1, \ldots, \sigma_n, \tau$ types. Such a type declaration is a notation for the type $\sigma_1 \to \ldots \to \sigma_n \to \tau$ in which it is also expressed that the *arity* is n.

Typing rules. We use the following *typing rules* to derive statements of the form $\Gamma \vdash t : \sigma$. Here Γ is an environment, containing type declarations for variables and function symbols, t is a term, and σ is a type.

$$\frac{X : \sigma_1 \times \ldots \times \sigma_n \Rightarrow \tau \in \Gamma \qquad \Gamma \vdash s_i : \sigma_i}{\Gamma \vdash X(s_1, \ldots, s_n) : \tau}\ var \qquad \frac{\Gamma, x :\Rightarrow \sigma \vdash t : \tau}{\Gamma \vdash \lambda x : \sigma.\, t : \sigma \to \tau}\ abs$$

$$\frac{f : \sigma_1 \times \ldots \times \sigma_n \Rightarrow \tau \in \Gamma \qquad \Gamma \vdash s_i : \sigma_i}{\Gamma \vdash f(s_1, \ldots, s_n) : \tau}\ fun \qquad \frac{\Gamma \vdash s : \sigma \to \tau \qquad \Gamma \vdash t : \sigma}{\Gamma \vdash @(s, t) : \tau}\ app$$

It is important that all bound variables have arity zero. Note that the variable x in the abstraction rule has arity zero. The variable X in the variable rule has arity n with $n \geq 0$. We use the notation convention that variables written as x, y, z have arity zero; the arity of variables written as $X, Y, Z, \ldots$ is zero or greater.

A second important point is that the typing system guarantees that symbols get exactly the number of arguments as prescribed by their arity. Therefore, we use the notations $f(s_1, \ldots, s_m)$ and $X(s_1, \ldots, s_m)$ if and only if f and X have arity m.

Modulo $\beta\eta$. In HRSs we work modulo $\beta\eta$ and use as representative of an equivalence class its (unique) long $\beta\overline{\eta}$-normal form. Here we take a different approach and use instead as a representative the (unique) $\beta\eta$-normal form. An expression is called a *term* if it is in $\beta\eta$-normal form, and a *preterm* if it not necessarily in $\beta\eta$-normal form. The η-reduction rule $\lambda x.\, @(s, x) \to_\eta s$ is subject to the usual side-condition, and in addition the arity of the symbols plays a role here. We explain this by means of an example. The symbol f in the term $\lambda x.\, @(f, x)$ has arity 0. Here we can apply the η-reduction rule which yields the term f. In contrast, if f is a symbol of arity 1, then $\lambda x.\, @(f, x)$ is not a term. Then $\lambda x.\, f(x)$ is a term, but here the η-reduction rule doesn't apply.

Rewrite relation. We adapt the definition of patterns to our framework. The left-hand side of a rewrite rule is required to be a term of the form $f(l_1, \ldots, l_n)$ with all l_i patterns. Rules of functional type are allowed, but the left-hand side is not allowed to be an abstraction. An example of a rewrite rule is $\mathsf{diff}(\lambda x.\, \mathsf{sin}(x)) \to \lambda x.\, \mathsf{cos}(x)$ using the declaration $\mathsf{diff} : (R \to R) \Rightarrow (R \to R)$. The rewrite relation is defined on terms (in $\beta\eta$-normal form).

3 Results

In this section we describe the results obtained for our framework; in some cases this is still work in progress.

Rewriting and equality. In HRSs, rules of functional type are not admitted. The reason is that they cause the equality relation induced by the rewrite rules (viewed as equations) to be different from the reflexive-transitive-symmetric closure of the rewrite relation.

In our framework, rules of functional type are admitted. We avoid the problem that occurs in the HRS setting by requiring that the left-hand side of a rewrite rule is of the form $f(l_1, \ldots, l_n)$. Because it is neither an abstraction nor an application, there is no overlap with the rules for β- and η-reduction that are applied on a meta-level. Using one technical lemma, we can show the following theorem.

Theorem 1. *We have $s =_R s'$ if and only if $s\downarrow \leftrightarrow s'\downarrow$.*

Here s and s' are preterms, and $t\downarrow$ denotes the $\beta\eta$-normal form of t. An alternative approach could be to add extensions rules as those by Stickel in order to ensure coherence [3].

Local confluence. We use the following adaptation of the definition of patterns due to Miller [10]: a *pattern* is an application-free term in $\beta\eta$-normal form where all free variables have only different bound variables as arguments. By slightly adapting the standard unification algorithm, it can be shown that unification of patterns (in our sense) is decidable.

The definitions of overlapping rules and critical pairs are almost the same as for HRSs. Also the proof of the following theorem is similar to the analogous result for HRSs [11,9].

Theorem 2. *The rewrite relation is locally confluent if and only if all critical pairs are joinable.*

Termination. Jouannaud and Rubio [5,6] introduce a higher-order version of the recursive path ordering (HORPO). We adapt HORPO to the present setting. This yields a termination method for our framework: the rewriting system is terminating if $l \succ r$ for every rewrite rule $l \to r$. Here $\succ$ is used to denote HORPO. As usual, a key step in the proof of correctness of this method is the following result.

Theorem 3. *The ordering $\succ$ is well-founded.*

The proof proceeds as follows: for every term s and computable substitution γ it is shown by induction on the typing derivation of s that $s\gamma$ is computable. Using HORPO, we can for instance show termination of the rewrite rule $h(g(X, F)) \to g(X, \lambda yz.\, h(@(F, y, z)))$ with the declarations $h : A \Rightarrow A$ and $g : A \times (A \to A \to A) \Rightarrow A$.

Further research. At the moment HORPO is not yet very useful for comparing an algebraic term of functional type with an abstraction. It would be interesting to make HORPO more powerful such that also those comparisons can be done smoothly.

Further, we aim to consider more general typing systems. Adding polymorphism as in [6] should not be too difficult. However, adding dependant types could be hard.

References

1. P. Aczel. A general Church-Rosser theorem. University of Manchester, July 1978.
2. F. Blanqui, J.-P. Jouannaud, and M. Okada. The calculus of algebraic constructions. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 9th International Conference on Rewriting Techniques and Applications (RTA '99)*, number 1631 in LNCS, pages 301–316, Trento, Italy, July 1999. Springer Verlag.
3. J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986.
4. J.-P. Jouannaud and M. Okada. A computation model for executable higher-order algebraic specification languages. In *Proceedings of the 6th annual IEEE Symposium on Logic in Computer Science (LICS '91)*, pages 350–361, Amsterdam, The Netherlands, July 1991.
5. J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proceedings of the 14th annual IEEE Symposium on Logic in Computer Science (LICS '99)*, pages 402–411, Trento, Italy, July 1999.
6. J.-P. Jouannaud and A. Rubio. Higher-order recursive path orderings à la carte. URL: http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud/biblio.html
7. Z.O. Khasidashvili. Expression Reduction Systems. In *Proceedings of I. Vekua Institute of Applied Mathematics*, volume 36, pages 200–220, Tblisi, Georgia, 1990.
8. J.W. Klop. *Combinatory Reduction Systems*. Number 127 in Mathematical Centre Tracts. CWI, Amsterdam, The Netherlands, 1980. PhD Thesis.
9. R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.
10. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
11. T. Nipkow. Higher-order critical pairs. In *Proceedings of the 6th annual IEEE Symposium on Logic in Computer Science (LICS '91)*, pages 342–349, Amsterdam, The Netherlands, July 1991.

Vincent van Oostrom and Kees-Jan van de Looij and Marijn Zwitserlood

Department of Philosophy, Universiteit Utrecht, The Netherlands
oostrom,vdlooij,marijn@phil.uu.nl

Abstract. An optimal implementation of $\lambda\beta$-calculus into interaction nets with one type of scope node $\sqcup$.

1 Introduction

We present an implementation of β-reduction on λ-terms. For any λ-term [1] (Section 2), translating it by $\square$ to an interaction net [5] (Section 3), then performing a number of interaction steps (Section 4), and finally unwinding the resulting interaction net by $\triangle$ to a tree-like net isomorphic to a λ-term again (Section 5), yields a λ-term which is reachable by a number of β-steps from the initial term.

Although optimality was the original motivation for our studies, we wil not high-light it here. Instead, we focus on just presenting the implementation and some intuitions.

2 λ-calculus

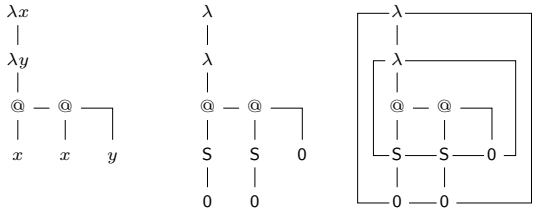
In this section we present a factorisation of β-reduction in the nameless λ-calculus [3] into *replication* and *scope extrusion*. The reason for presenting *this* factorisation is that the implementation of β-reduction into interaction nets in Section 4 is obtained by localising the global parts of this factorisation. We employ the following as a running example.

Example 1. The application $\underline{2}\,\underline{2}$ of the (Church) numeral $\underline{2} = \lambda\lambda(\mathsf{S}0)((\mathsf{S}0)0)$ to itself, reduces

$$\begin{aligned}
\underline{2}\,\underline{2} &\to_\beta \lambda\underline{2}(\underline{2}0)\\
&\to_\beta \lambda\underline{2}\lambda(\mathsf{S}0)((\mathsf{S}0)0)\\
&\to_\beta \lambda\lambda(\lambda(\mathsf{SS}0)((\mathsf{SS}0)0))((\lambda(\mathsf{SS}0)((\mathsf{SS}0)0))0)\\
&\to_\beta \lambda y\lambda z(\lambda x(\mathsf{SS}0)((\mathsf{SS}0)0))((\mathsf{S}0)((\mathsf{S}0)0))\\
&\to_\beta \lambda\lambda(\mathsf{S}0)((\mathsf{S}0)((\mathsf{S}0)((\mathsf{S}0)0)))
\end{aligned}$$

to $\underline{4}$ in the five steps displayed. (Application of Church numerals is exponentiation.)

We have employed unary notation for De Bruijn-indices in order to make the main point of this section which is that the set of (closed) De Bruijn terms is a *context free* set of terms [2]. Consider, from left to right, the syntax tree of $\underline{2}$ in the named λ-calculus [1]), the syntax tree of $\underline{2}$ itself, and that tree again with some boxes added to it



These boxes indicate how each S and $\mathsf{0}$ in the tree can be seen to *match* with a unique λ. This notion of matching can be easily formalised by means of a push down automaton (PDA): starting with an empty stack and walking from the root toward the leaves each λ corresponds to a push and each S or $\mathsf{0}$ to a pop (and @s have no effect). Then the closed De Bruijn terms are *exactly* captured by the requirement that each path is accepted by the PDA. Alternatively, paths may be mapped to strings of parentheses by setting $\lambda \mapsto ($ and $\mathsf{S}, \mathsf{0} \mapsto)$ and forgetting @. Then the closed De Bruijn terms are *exactly* captured by each path being mapped to a string of the form $({}^n)^m$ with $n \geq m$. The upshot is that one should *not* think of S as operating on the subterm *below* it, but instead of as matching with a (unique) λ *above* it. The rest of the abstract is a consequence of this shift of viewpoint.

The first observation is that the notion of a variable being bound by a λ-abstraction in the named λ-calculus corresponds in the De Bruijn representation to a $\mathsf{0}$ *matching* the λ-abstraction. For instance, the leftmost x being bound by the topmost λ in the named syntax tree in the figure, corresponds to the leftmost $\mathsf{0}$ *matching* the topmost λ in the unnamed syntax tree. An occurrence of $\mathsf{0}$ in t matching the λ of a β-redex $(\lambda t)s$ then has the operational meaning: put the argument s.

The second observation is that one may think of S as an end-of-scope operator [4]: if it matches some λ, then no symbol below the S can match with that same λ again; i.e. all these symbols are out-of-scope so to speak. That is, the boxes in the figure should be thought of as (explicit) representations of the notion of *scopes*. An occurrence of S in t matching the λ of a β-redex $(\lambda t)s$ has the operational meaning: throw the argument s away.

Combining these two observations yields the first *replication* phase of the β-reduction of $(\lambda t)s$ in which the argument s is put at all $\mathsf{0}$s in the body matching the λ, in which all Ss in the body matching the λ are elided, and in which finally the @ and λ of the redex are removed. Applied to the first β-step of our running example, this yields the first phase of its factorisation as (the first step in):

$$\underline{2}\,\underline{2} = (\lambda\lambda(\mathsf{S0})((\mathsf{S0})\mathsf{0}))\,\underline{2} \to \lambda(\mathsf{S}\underline{2})((\mathsf{S}\underline{2})\mathsf{0}) \to \lambda\underline{2}(\underline{2}\mathsf{0})$$

Note that the result $\lambda\mathsf{S}\underline{2}((\mathsf{S}\underline{2})\mathsf{0})$ of the first phase is not yet a De Bruijn term, since S is applied in it to a *term* instead of to a De Bruijn index. Rather it is a so-called *generalised* λ-term [2], where successors can be applied to any (generalised) term instead of just to De Bruijn indices. In order to turn the generalised λ-term into the ordinary λ-term $\lambda\underline{2}(\underline{2}\mathsf{0})$ again, the offending Ss are pushed to the leafs in a matching-preserving way, as in the second step above, a process which we call *scope extrusion* [4].

Formally, the grammar for the set $G\Lambda$ of generalised λ-terms is

$$t \in G\Lambda ::= \mathsf{0} \mid \mathsf{S}t \mid \lambda t \mid tt$$

We employ t, s, u, ... to range over generalised λ-terms and i, j, k, ... to range over its subset of (De Bruijn) indices, i.e. repeated applications of S to $\mathsf{0}$. We abbreviate indices by natural numbers in sans-serif e.g. $\mathsf{SSS0}$ is abbreviated to $\mathsf{3}$. Ordinary nameless λ-terms are obtained by requiring successors to occur as part of indices. β-reduction on ordinary (nameless) λ-terms factorises then as follows. First, *replication* is performed according to

$$(\lambda t)s \to t[s]^{\mathsf{0}}$$

with *substitution* $t[s]^i$ of s in t at depth i defined $\mathsf{globally}$ in Table 1. Next, scopes are

$$
\begin{array}{rl}
\mathsf{0}[s]^0 &= s \\
\mathsf{0}[s]^{\mathsf{S}i} &= \mathsf{0} \\
(\mathsf{S}t)[s]^0 &= t \\
(\mathsf{S}t)[s]^{\mathsf{S}i} &= \mathsf{S}t[s]^i \\
(\lambda t)[s]^i &= \lambda t[s]^{\mathsf{S}i} \\
(t_1 t_2)[s]^i &= t_1[s]^i t_2[s]^i
\end{array}
\qquad\qquad
\begin{array}{rl}
t^0 &= \mathsf{S}t \\
\mathsf{0}^{\mathsf{S}i} &= \mathsf{0} \\
(\mathsf{S}t)^{\mathsf{S}i} &= \mathsf{S}t^i \\
(\lambda t)^{\mathsf{S}i} &= \lambda t^{\mathsf{S}\mathsf{S}i} \\
(t_1 t_2)^{\mathsf{S}i} &= t_1^{\mathsf{S}i} t_2^{\mathsf{S}i}
\end{array}
$$

Table 1. Global definitions of substitution (left) and minimal lifting (right)

extruded by reducing to normal form with respect to the *scope extrusion* rules

$$\mathsf{S}\lambda t \to_\lambda \lambda t^{\mathsf{S}\mathsf{0}}$$
$$\mathsf{S}(t_1 t_2) \to_@ \mathsf{S}t_1 \mathsf{S}t_2$$

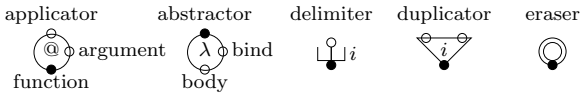
where, for index i, *minimal lifting* t^i is defined globally in Table 1. Note that using the extrusion rules, the first step of Example 1, indeed factorises in the way which was displayed above. In particular, note that $\mathsf{S}\underline{2} \to_\lambda \underline{2}$ holds since $\underline{2}$ is closed. Here a generalised λ-term t is closed if $\mathsf{0} \vdash t$ in the following inference system (read the rules top-down, like syntax-trees):

$$\frac{}{\mathsf{S}i \vdash \mathsf{0}}\,\mathsf{0} \qquad \frac{\mathsf{S}i \vdash \mathsf{S}t}{i \vdash t}\,\mathsf{S} \qquad \frac{i \vdash \lambda t}{\mathsf{S}i \vdash t}\,\lambda \qquad \frac{i \vdash t_1 t_2}{i \vdash t_1 \quad i \vdash t_2}\,@$$

An intuitive reading of the index i (the 'stack') in a judgment $i \vdash t$ (read: term t is well-formed under index i), is as the (number of) variables bound by λs above this subterm t. It is easy and instructive to verify that $\underline{2}$ is indeed closed.

3 From terms to nets

We present our translation of the nameless λ-terms to a class IN of graphs known as interaction nets [5], with which we assume familiarity. The signature of an interaction net consists of symbols each having a number of ports among which a designated *principal* port. The interaction net signature we employ is

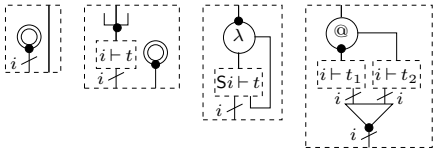


where $\circ$s indicate ports and $\bullet$s indicate principal ports, i.e. ports along which a symbol may interact (see below). Here i ranges over arbitrary indices, making the signature infinite.

Apart from @ and λ which will have the meaning one expects, the signature has symbols for explicitly representing the different operations of the factorisation of β-reduction, as presented in the previous section. In particular, the duplicator $\triangledown_i$ (share, fan) and the eraser $\circledcirc$ (garbage) will together serve to represent replication, as usual in graph implementations of first-order rewriting. The delimiter $\sqcup_i$ represents the higher-order aspect of scope. When an index is $\mathsf{0}$, it will be omitted.

Interaction nets are graphs the nodes of which are labelled by symbols of the signature, and the edges of which connect to the ports of the (symbol occurrences labelling the) nodes. To every port at most one edge may be connected. If no edge is connected to a port, then the port is called free. A net is closed if it does not have free ports.

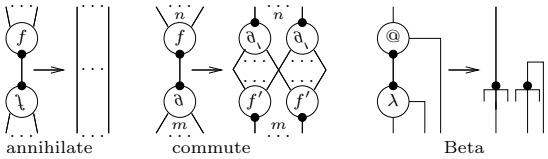
The function $\square : \Lambda \to \mathrm{IN}$ mapping closed terms to closed interaction nets is defined in two phases. First, a well-formed term $i \vdash t$ is mapped to a net having $i+1$ free ports, which is defined by induction and cases ($\mathsf{0}$, S, λ, and @) on the definition of well-formedness as:



After that a ◎-node (the *root*) is connected to the (unique by closedness) free port. Here a number i next to a slashed edge represents that in fact the edge is a 'bus' consisting of i edges connected to an appropriate number of copies of its connected node. Since the translation is uniform, it is on the one hand easy to prove properties about, but on the other hand very inefficient: it generates many duplicator-eraser combinations whose net-effect will be the same as that of an edge. To see this it is instructive to compute $\square(\underline{2})$. Below, we will work with the optimisation $\overline{2}$ instead where all redundant combinations have been removed.

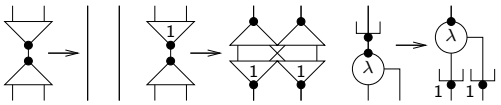
4 Interaction net reduction

The intuitive meaning of the symbols in our interaction net signature, as presented above, is operationalised by just two rule schemes, for f,g arbitrary but distinct, and a rule:



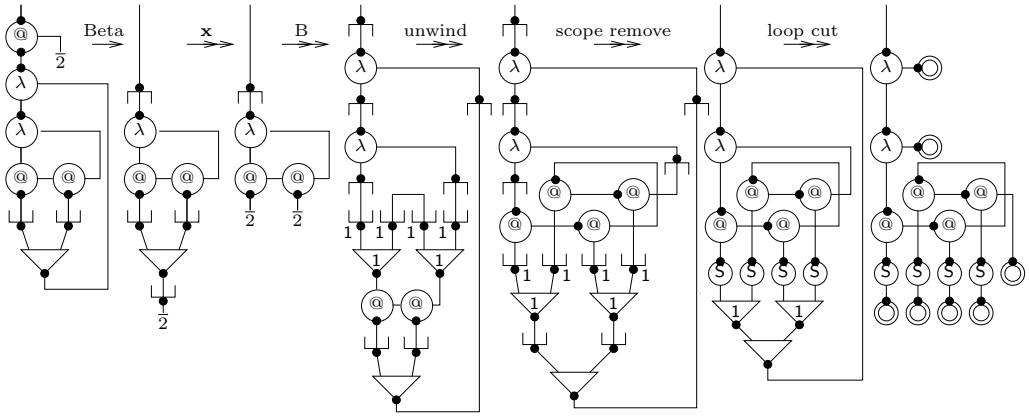
where f' and g' are either identical to or updates of the symbols f and g, respectively. An update is an increment of the index i (if any) of either symbol, which takes place iff the other symbol is either λ or $\sqcup_j$, with $i \geq j$. Instances of the two schemes are called **x**-rules.

Example 2. Annihilate, commute, and commute with update are exemplified by



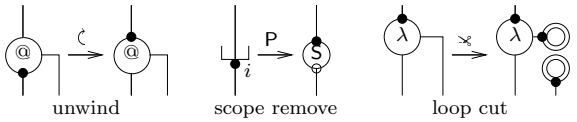
The set B of interaction rules which interest us is defined to be the union of **x** and Beta. Note that the effect of operators is indeed as expected: the eraser acts as a garbage collector erasing anything it encounters; the duplicator acts as a copier duplicating anything it encounters; the delimiter acts as an extruder putting anything it encounters into its scope. All act locally in the sense that they affect one node at the time. This restriction, which comes with the interaction net framework, explains our use of *indexed* delimiters: roughly speaking, the way in which the global definition of minimal lifting of the previous section is implemented, is to record the superscript there as an index to the delimiter. Similarly the superscript in the global definition of substitution corresponds to the index of a duplicator.

Example 3. We display only a few nets along the reduction of $\overline{2}$ applied to itself, to B-normal form. The net in the middle is the normal form which is a representation of $\underline{4}$. The remaining steps serve to retrieve this term from the net and are the next topic.



5 From nets to terms

The function $\triangle : \mathrm{IN} \to \Lambda$ mapping closed nets to closed terms is defined in three phases, each consisting of normalising w.r.t. an *action* first and the **x**-rules next. (Without touching the root-$\circledcirc$.) This yields the syntax tree of a unique (derivation of a) term, which is taken as the result of $\triangle(\nu)$. The three actions are the following graph rewrite rules:



Here the S is a new node type, the interaction of which is governed by the **x**-rules, i.e. S behaves as a non-indexed $\sqcup_i$. After the unwinding action, both abstractions *and* application have their north port as principal port. This makes that all replication and delimiter nodes are 'pushed toward the leafs/variables' (causing unsharing and extrusion) by the subsequent **x**-normalisation phase. Once extruded, it is safe to forget the indices of the remaining delimiters. By swapping their principal port as well, the scope removal phase causes the remaining upward directed delimiters (which all have index 0) to be pushed toward the leafs, and the replication nodes to become closer to the leafs than the delimiter nodes. Once this has been done, cutting the loops causes all replication nodes to vanish by having them interact with the eraser, yielding a tree proper as shown above.

6 Conclusion

The proof that the above is correct and is even optimal in the sense of Lévy is beyond the scope of this abstract. Here we just wanted to stress the correspondence between our implementation and the calculational approach to the λ-calculus of [3], and its simplicity; the calculus is completely reduction-based, fits on half a page, and is trivial to implement.

References

1. H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.
2. R.S. Bird and R.A. Paterson. De Bruijn notation as a nested datatype. *JFP*, 9(1):77–91, 1999.
3. N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.
4. D. Hendriks and V. van Oostrom. λ. In *CADE 19*, volume 2741 of *LNAI*, pages 136–150. Springer, 2003.
5. Y. Lafont. Interaction nets. In *POPL 17*, pages 95–108. ACM Press, 1990.

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult `http://aib.informatik.rwth-aachen.de/` or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

95-11 * M. Staudt / K. von Thadden: Subsumption Checking in Knowledge Bases

95-12 * G.V. Zemanek / H.W. Nissen / H. Hubert / M. Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology

95-13 * M. Staudt / M. Jarke: Incremental Maintenance of Externally Materialized Views

95-14 * P. Peters / P. Szczurko / M. Jeusfeld: Business Process Oriented Information Management: Conceptual Models at Work

95-15 * S. Rams / M. Jarke: Proceedings of the Fifth Annual Workshop on Information Technologies & Systems

95-16 * W. Hans / St. Winkler / F. Sáenz: Distributed Execution in Functional Logic Programming

96-1 * Jahresbericht 1995

96-2 M. Hanus / Chr. Prehofer: Higher-Order Narrowing with Definitional Trees

96-3 * W. Scheufele / G. Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates

96-4 K. Pohl: PRO-ART: Enabling Requirements Pre-Traceability

96-5 K. Pohl: Requirements Engineering: An Overview

96-6 * M. Jarke / W. Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools

96-7 O. Chitil: The ς-Semantics: A Comprehensive Semantics for Functional Programs

96-8 * S. Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics

96-9 M. Hanus (Ed.): Proceedings of the Poster Session of ALP'96 — Fifth International Conference on Algebraic and Logic Programming

96-10 R. Conradi / B. Westfechtel: Version Models for Software Configuration Management

96-11 * C. Weise / D. Lenzkes: A Fast Decision Algorithm for Timed Refinement

96-12 * R. Dömges / K. Pohl / M. Jarke / B. Lohmann / W. Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models

96-13 * K. Pohl / R. Klamma / K. Weidenhaupt / R. Dömges / P. Haumer / M. Jarke: A Framework for Process-Integrated Tools

96-14 * R. Gallersdörfer / K. Klabunde / A. Stolz / M. Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996

96-15 * H. Schimpe / M. Staudt: VAREX: An Environment for Validating and Refining Rule Bases

96-16 * M. Jarke / M. Gebhardt, S. Jacobs, H. Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization

96-17 M. Jeusfeld / T. X. Bui: Decision Support Components on the Internet

96-18 M. Jeusfeld / M. Papazoglou: Information Brokering: Design, Search and Transformation

96-19 * P. Peters / M. Jarke: Simulating the impact of information flows in networked organizations

96-20 M. Jarke / P. Peters / M. Jeusfeld: Model-driven planning and design of cooperative information systems

96-21 * G. de Michelis / E. Dubois / M. Jarke / F. Matthes / J. Mylopoulos / K. Pohl / J. Schmidt / C. Woo / E. Yu: Cooperative information systems: a manifesto

96-22 * S. Jacobs / M. Gebhardt, S. Kethers, W. Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

96-23 * M. Gebhardt / S. Jacobs: Conflict Management in Design

97-01 Jahresbericht 1996

97-02 J. Faassen: Using full parallel Boltzmann Machines for Optimization

97-03 A. Winter / A. Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

97-04 M. Mohnen / S. Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

97-05 * S. Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

97-06 M. Nicola / M. Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

97-07 P. Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

97-08 D. Blostein / A. Schürr: Computing with Graphs and Graph Rewriting

97-09 C.-A. Krapp / B. Westfechtel: Feedback Handling in Dynamic Task Nets

97-10 M. Nicola / M. Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

97-13 M. Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

97-14 R. Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

97-15 G. H. Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

98-01 * Jahresbericht 1997

98-02 S. Gruner/ M. Nagel / A. Schürr: Fine-grained and Structure-oriented Integration Tools are Needed for Product Development Processes

98-03 S. Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

98-04 * O. Kubitz: Mobile Robots in Dynamic Environments

98-05 M. Leucker / St. Tobies: Truth — A Verification Platform for Distributed Systems

98-07 M. Arnold / M. Erdmann / M. Glinz / P. Haumer / R. Knoll / B. Paech / K. Pohl / J. Ryser / R. Studer / K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects

98-08 * H. Aust: Sprachverstehen und Dialogmodellierung in natürlichsprachlichen Informationssystemen

98-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien

98-10 * M. Nicola / M. Jarke: Performance Modeling of Distributed and Replicated Databases

98-11 * A. Schleicher / B. Westfechtel / D. Jäger: Modeling Dynamic Software Processes in UML

98-12 * W. Appelt / M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web

98-13 K. Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation

99-01 * Jahresbericht 1998

99-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version

99-03 * R. Gallersdörfer / M. Jarke / M. Nicola: The ADR Replication Manager

99-04 M. Alpuente / M. Hanus / S. Lucas / G. Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing

99-07 Th. Wilke: CTL+ is exponentially more succinct than CTL

99-08 O. Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures

2000-01 * Jahresbericht 1999

2000-02 Jens Vöge / Marcin Jurdzinski: A Discrete Strategy Improvement Algorithm for Solving Parity Games

2000-04 Andreas Becks / Stefan Sklorz / Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach

2000-05 Mareike Schoop: Cooperative Document Management

2000-06 Mareike Schoop / Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling

2000-07 * Markus Mohnen / Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages

2000-08 Thomas Arts / Thomas Noll: Verifying Generic Erlang Client-Server Implementations

2001-01 * Jahresbericht 2000

2001-02 Benedikt Bollig / Martin Leucker: Deciding LTL over Mazurkiewicz Traces

2001-03 Thierry Cachat: The power of one-letter rational languages

2001-04 Benedikt Bollig / Martin Leucker / Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus

2001-05 Benedikt Bollig / Martin Leucker / Thomas Noll: Regular MSC Languages

2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic

2001-07 Martin Grohe / Stefan Wöhrle: An Existential Locality Theorem

2001-08 Mareike Schoop / James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling

2001-09 Thomas Arts / Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs

2001-10 Achim Blumensath: Axiomatising Tree-interpretable Structures

2001-11 Klaus Indermark / Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung

2002-01 * Jahresbericht 2001

2002-02 Jürgen Giesl / Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems

2002-03 Benedikt Bollig / Martin Leucker / Thomas Noll: Generalised Regular MSC Languages

2002-04 Jürgen Giesl / Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting

2002-05 Horst Lichter / Thomas von der Maßen / Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines

2002-06 Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata

2002-07 Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities

2002-08 Markus Mohnen: An Open Framework for Data-Flow Analysis in Java

2002-09 Markus Mohnen: Interfaces with Default Implementations in Java

2002-10 Martin Leucker: Logics for Mazurkiewicz traces

2002-11 Jürgen Giesl / Hans Zantema: Liveness in Rewriting

2003-01 * Jahresbericht 2002

2003-02 Jürgen Giesl / René Thiemann: Size-Change Termination for Term Rewriting

2003-03 Jürgen Giesl / Deepak Kapur: Deciding Inductive Validity of Equations

2003-04 Jürgen Giesl / René Thiemann / Peter Schneider-Kamp / Stephan Falke: Improving Dependency Pairs

2003-05 Christof Löding / Philipp Rohde: Solving the Sabotage Game is PSPACE-hard

2003-06 Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates

2003-07 Horst Lichter / Thomas von der Maßen / Alexander Nyßen / Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung

2003-08 Jürgen Giesl / René Thiemann / Peter Schneider-Kamp / Stephan Falke: Mechanizing Dependency Pairs

2004-02 Benedikt Bollig / Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic

2004-03 Delia Kesner / Femke van Raamsdonk / Joe Wells (eds.): Proceedings of the Second International Workshop on Higher-Order Rewriting (HOR 2004)

2004-04 Slim Abdennadher / Christophe Ringeissen (eds.): Proceedings of the Fifth International Workshop on Rule-Based Programming (RULE 2004)

2004-05 Herbert Kuchen (ed.): Proceedings of the 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004)

2004-06 Sergio Antoy / Yoshihito Toyama (eds.): Proceedings of the 4th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2004)

2004-07 Michael Codish / Aart Middeldorp (eds.): Proceedings of the 7th International Workshop on Termination (WST 2004)

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.