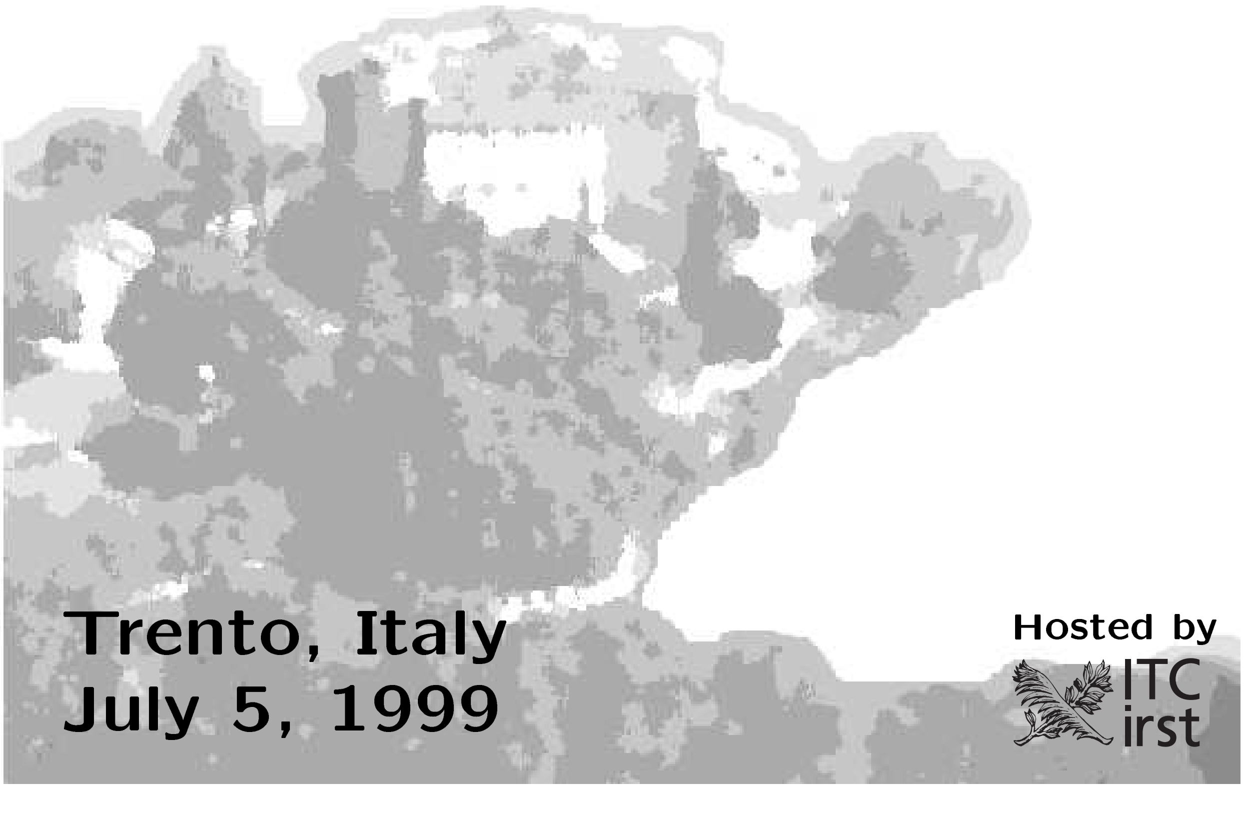
The 1999
Federated Logic Conference

Proceedings of the FLoC Workshop

The Second International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs



Trento, Italy
July 5, 1999

Hosted by
ITC
irst

Preface

Recently, there has been growing interest in explicit substitutions as a means of bridging the gap between theory and practice in the implementation of programming languages, theorem provers and proof checkers. Theoretically, these typically rely on calculi that employ substitution operations implicitly at the meta-level; implementations are then framed in terms of these meta-level operations. Explicit substitutions bring the meta-level operations down into the object-level calculus, where they are represented explicitly. This allows formal and robust models to be given for the techniques actually used in implementations, and at the same time allows more flexibility and control on unfinished evaluations. Explicit substitutions, and more generally environment machines, have recently proved to be very useful in building more efficient programming languages and proof assistants.

The aim of this workshop is to bring together researchers working on both theoretical and applied aspects of explicit substitutions, to present recent work (possibly still in progress), and to discuss new ideas and emerging trends in topics including the following:

- Use of explicit substitution in implementation of programming languages, theorem provers and proof checkers.
- Explicit substitutions calculi used for the design of abstract machines.
- New concepts in substitution calculi Generalised techniques to show properties of substitution calculi.
- Relating explicit substitutions with logic formalisms such as sequent calculi, linear logic, game semantics, etc.
- Accommodating different reduction strategies and control operators.
- Higher order types and explicit substitutions. item Extensions of explicit substitution calculi to higher order formalisms.
- Different criteria useful to compare calculi with explicit substitutions.
- Applications of explicit substitutions to solve problems in other fields (e.g. higher order unification, set constraints etc.). item Different notions of substitutions, relations with environments and records. item Proof synthesis and proof search via explicit substitutions.

Program Committee

Luca Cardelli	Microsoft Research,	UK
Therese Hardin	University Paris VI,	France
Fairouz Kamareddine	Heriot-Watt University,	UK
Delia Kesner (Chair)	University Paris-Sud,	France
Shin-ya Nishizaki	Tokyo Institute of Technology,	Japan
Valeria de Paiva	University of Birmingham,	UK

Local Organizing Committee

Andrea Asperti	University of Bologne,	Italy
Fausto Giunchiglia	University of Trento,	Italy

Contents

On the relation between substitution calculus and Tarski semantics

Per Martin-Löf
Department of Mathematics
University of Stockholm
Sweden

Abstract

There is a strong formal similarity between the rules of Tarski's semantic valuation procedure and the rules of the calculus of substitutions. For example, the clause

$$[\![(\lambda x)b]\!]\rho(a) = [\![b]\!](\rho, x = a)$$

in Henkin's version of Tarski semantics for simple type theory, dating back to 1950, may be considered as the first manifestation of the law of beta conversion with an intervening substitution

$$((\lambda x)b)\sigma(a) = b(\sigma, a/x)$$

It will be explained exactly how the two, Tarski semantics and substitution calculus, are related.

The λ_l-calculus
(extended abstract)

René David, Bruno Guillaume

Laboratoire de Mathémathiques,
Université de Savoie
F-73376 Le Bourget du Lac Cedex.
Tel: +33-4-7975-8717. Fax: +33-4-7975-8742
{david,guillaume}@univ-savoie.fr

Abstract. Since Melliès has shown that $\lambda\sigma$ (a calculus of explicit substitutions) does not preserve the strong normalization of the β-reduction, it became a challenge to find a calculus satisfying the following properties : step by step simulation of the β-reduction, confluence on terms with meta-variables, strong normalization of the calculus of substitutions and preservation of the strong normalization of the λ-calculus. We present here such a calculus. The main novelty of the calculus (given with de Bruijn indices) is the use of *labels* to represent the updating functions. A typed version is also presented.

Keywords : λ-calculus, explicit substitutions, rewriting.

1 Introduction

Since the introduction of $\lambda\sigma$ ([1]), many calculi of explicit substitutions have been given. It has been thought for a long time that these calculi preserved the strong normalization of the λ-calculus. Melliès showed in [10] that this fails for $\lambda\sigma$ and the second author recently showed ([4]) that it also fails in λs_e. Finding a calculus satisfying the following properties became thus a challenge.

(β) Step by step simulation of β,
(MC) Confluence on terms with meta-variables,
$(SN(\sigma))$ Strong normalization of the calculus of substitutions,
(PSN) Preservation of the strong normalization of the λ-calculus.

The existing calculi may be divided into two groups.

- Those without interaction between substitutions. For example : $\lambda\nu$ ([9]), λs ([7]), λx ([12]). For these calculi, PSN is not a major problem but they do not satisfy MC.
- Those with interaction between substitutions. For example : $\lambda\sigma_{\Uparrow}$ ([2]), λs_e ([7]), $\lambda\chi_B$ci ([8]). $\lambda\sigma_{\Uparrow}$ ([10]) and λs_e ([4]) do not satisfy PSN. The PSN of $\lambda\chi_B$ci is an open question (despite the title of [8]). Moreover the reduction rules of this calculus are not local.

Other calculi give intermediate solutions. λd ([3]) has a "weak" composition for substitutions so that PSN is preserved but it is not enough to recover MC. $\lambda\zeta$ ([11]) satisfies both PSN and MC but the simulation of β is weak, i.e. it simulates only particular strategies.

Recently, Ritter proposed a characterization for systems who have the PSN property. He gave also a calculus (named $\lambda\sigma_n$) with named variables which is confluent on meta-variables and preserves strong normalization.

We propose here a calculus satisfying all the desired properties. As far as we know this is one of the first examples (with Ritter's one) of such a calculus. We also give a simply typed version of the calculus. The proofs of β, MC and $SN(\sigma)$ are standard. We give a sketch of the proof of PSN. The complete proofs are given in [5].

2 The λ_l-calculus

The counter-example to PSN in the λs_e-calculus relies on the fact that updating functions can be propagated without any control. The key point of our calculus is the limitation of this propagation and thus it may be seen as the λs_e-calculus where the rules on substitutions are more constrained.

The λ_l-calculus is obtained from de Bruijn calculus (denoted as λ_{db}) by adding two constructors : labels and explicit substitutions. The set of terms of the λ_l-calculus is defined by :

$$t ::= \underline{n} \mid \lambda t \mid (t\, t) \mid \langle k \rangle t \mid [i/t, j]t \quad \text{with} \quad n, i, j, k \in \mathbb{N}$$

For technical reasons, de Bruijn indices begin with $\underline{0}$ instead of $\underline{1}$ (e.g. $I = \lambda\underline{0}$).

The rules of λ_l are presented in Fig. 1. The l-calculus is the λ_l-calculus without the rules b_1 and b_2.

Let $\sigma^i(t, u)$ and $\tau_i^j(t)$ be the syntax used in the λs_e-calculus to represent the substitution and the updating. The relation of our calculus with the λs_e-calculus is the following :

- $\langle k \rangle t$ corresponds to $\tau_0^k(t)$ and thus represents the term t where all the free indices have been lifted by k.
- $[i/u, j]t$ is $\sigma^i(\tau_{i+1}^{j-1}(t), u)$ and thus represents the term t in which the variable i is substituted by u and the indices $k > i$ of t are increased by $j - 1$.

The fact that there is no rule to move the label $\langle k \rangle$ down into a term implies that l-normal forms are not usual λ_{db}-terms : an l-normal form is a λ_{db}-term with labels.

We believe that this representation of λ-terms, though unusual, is better than the usual one for the following reasons :

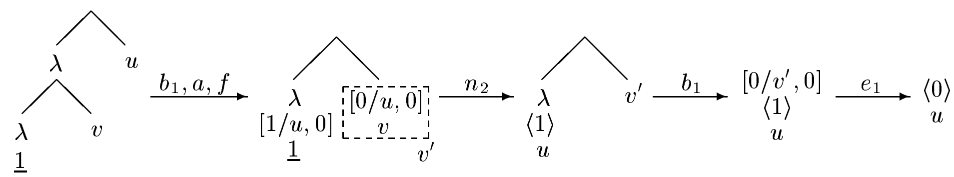
- Informations about bound variables are kept higher in the term. This can be used to get a better implementation : The lift are not propagated and many substitutions are erased.
- All the rules are local.

$$
\begin{array}{|llll|}
\hline
b_1 & (\lambda t\ u) & \longrightarrow [0/u,0]t & \\
b_2 & (\langle k\rangle \lambda t\ u) & \longrightarrow [0/u,k]t & \\
f & [i/u,j]\lambda t & \longrightarrow \lambda[i+1/u,j]t & \\
a & [i/u,j](t\ v) & \longrightarrow (([i/u,j]t)\ ([i/u,j]v)) & \\
e_1 & [i/u,j]\langle k\rangle t & \longrightarrow \langle j+k-1\rangle t & \text{if } i<k \\
e_2 & [i/u,j]\langle k\rangle t & \longrightarrow \langle k\rangle[i-k/u,j]t & \text{if } k\le i \\
n_1 & [i/u,j]\underline{k} & \longrightarrow \underline{k} & \text{if } k<i \\
n_2 & [i/u,j]\underline{i} & \longrightarrow \langle i\rangle u & \\
n_3 & [i/u,j]\underline{k} & \longrightarrow \underline{j+k-1} & \text{if } i<k \\
c_1 & [i/u,j][k/v,l]t & \longrightarrow [k/[i-k/u,j]v,j+l-1]t & \text{if } k\le i<k+l \\
c_2 & [i/u,j][k/v,l]t & \longrightarrow [k/[i-k/u,j]v,l][i-l+1/u,j]t & \text{if } k+l\le i \\
d & \langle i\rangle\langle j\rangle t & \longrightarrow \langle i+j\rangle t & \\
\hline
\end{array}
$$

Fig. 1. Rules of the λ_l-calculus

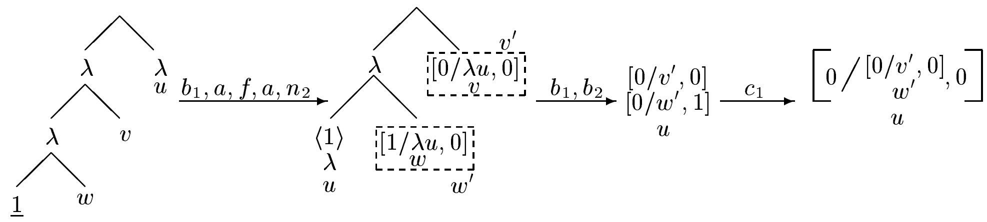
- Translating a labelled term to a "named" term does not cost more work than translating a de Bruijn term. In the same way, to compare the equality of two labelled terms can be done in linear time.
- The function E (cf. Sect. 5) that computes the λ_{db}-term represented by a term with labels is only used for stating the (β) property of the calculus. In an implementation of this calculus, this function would be useless because labels are intrinsic : No extra informations about the labels of t and t' is necessary to compute $(t\ t')$.
- From a logical point of view (cf. Sect. 7), a substitution corresponds to a cut in the proof tree and a label corresponds to a weakening. Whereas it is natural to move a cut up to the axioms in the proof tree, it is not interesting to move a weakening.

Example 1.



The previous reduction shows how labels avoid useless propagation of substitutions : in the e_1-reduction, the substitution $[0/v',0]$ is erased in one step (whatever the term u is).

Example 2.



In the last step (c_1-reduction), the "1" of $[0/w', 1]$ means that v' may be substituted in w' but not in u, and thus, one substitution is enough in the result.

Notations. *1. Let R be a reduction. $t \longrightarrow_R u$ (resp. $t \longrightarrow_R^* u$, $t \longrightarrow_R^+ u$) means that t reduces to u by one step (resp. some steps, at least one step) of R-reduction. $SN(R)$ denotes the set of terms which are strongly normalizable for the R-reduction.*

2. The complexity of a λ_l-term is defined by :
- $|\underline{n}| = 1$
- $|\lambda t| = |t| + 1$
- $|(t\, u)| = |t| + |u| + 1$
- $|\langle k \rangle t| = |t| + 1$
- $|[i/u, j]t| = |t| + |u| + 1$

3 Strong normalization of the l-calculus

Theorem 1. *The l-calculus is strongly normalizing.*

Proof. The rules of the l-calculus are divided in three groups :

- the rules that decrease the complexity of terms : $R_1 = e_1 \cup n_1 \cup n_2 \cup n_3 \cup d$,
- the rules that leave the complexity unchanged : $R_2 = f \cup e_2 \cup c_1$,
- the rules that increase the complexity : $R_3 = a \cup c_2$.

The first step of the proof is to show that l is SN iff $R_2 \cup R_3$ is SN. This is done by showing that the R_1-reductions can be postponed.

The following lemma (Zantema) is used to prove that $R_2 \cup R_3$ is SN.

Lemma 1. *If the reductions R_2 and R_3 satisfy :*

- *if $t \longrightarrow_{R_2} u$ then $|t| = |u|$,*
- *if $t \longrightarrow_{R_3} u$ then $|t| < |u|$,*
- *R_2 is SN,*
- *$R_2 \cup R_3$ is locally confluent,*
- *$R_2 \cup R_3$ if weakly normalizable,*

then $R_2 \cup R_3$ is SN.

The first two points are immediate. We show that :

- R_2 is SN : by using a decreasing measure on terms.
- $R_2 \cup R_3$ is locally confluent : by analyzing the critical pairs.
- $R_2 \cup R_3$ is weakly normalizable : by using the innermost strategy. $\square$

4 Confluence with meta-variables

We consider here λ_{l_o}, the λ_l-calculus on terms with meta-variables (also called open terms). The grammar of λ_{l_o} is : $t = \underline{n} \mid a \mid \lambda t \mid (t\,t) \mid \langle k \rangle t \mid [i/t, j]t$ where $n, i, j, k \in \mathbb{N}$ and a is a meta-variable. The rules of λ_{l_o} are those of λ_l. We call l_o the calculus with the rules of the l-calculus on the λ_{l_o}-terms.

Theorem 2. *The following calculi are confluent :* λ_l, λ_{l_o}, l *and* l_o.

Proof. It is easy to see (by analyzing the critical pairs) that l_o and λ_{l_o} are locally confluent. The strong normalization of the l-calculus implies the strong normalization of the l_o-calculus, so the l_o-calculus is confluent and every λ_{l_o}-term t has a unique l_o-normal form (denoted by $l_o(t)$).

For the confluence of λ_{l_o}, we define a reduction β' on the set $NF(l_o)$ of l_o-normal forms by : $t \longrightarrow_{\beta'} u$ iff $t \longrightarrow_{\{b_1, b_2\}} t'$ and $u = l_o(t')$. Using the interpretation method (cf. [6]), the confluence of the λ_{l_o}-calculus is equivalent to the confluence of β' on $NF(l_o)$. The confluence of β' is proved by the standard technique of parallel reductions.

The confluence of λ_l and l are direct consequences of those of λ_{l_o} and l_o. $\square$

5 The λ_l-calculus simulates β

Denote by $l(t)$ the l-normal form of t and by $NF(l)$ the set of l-normal forms.

Terms of $NF(l)$ are given by the following grammar :

$$t ::= u \mid \langle k \rangle u \quad \text{and} \quad u ::= \underline{n} \mid \lambda t \mid (t\,t) \quad \text{with} \quad n, k \in \mathbb{N}$$

Due to the fact that l-normal forms are not exactly λ_{db}-terms we introduce a reduction λ_{lab} on $NF(l)$ which is the usual β-reduction adapted to terms with labels.

Definition 1. *Let* λ_{lab} *be the calculus on* $NF(l)$ *with the rules :*

$$(\lambda t\, u) \longrightarrow l([0/u, 0]t)$$

$$(\langle k \rangle \lambda t\, u) \longrightarrow l([0/u, k]t)$$

The λ_{lab} is the λ_l-calculus where substitutions are "implicit". The link between λ_{lab} and λ_{db} is given via an erasing function E which moves labels down to compute the corresponding λ_{db}-term.

$$E(\underline{n}) = \underline{n} \qquad E(\lambda t) = \lambda E(t) \qquad E((t\,u)) = (E(t)\; E(u)) \qquad E(\langle k \rangle t) = \tau_0^k(E(t))$$

$$\tau_i^j(\lambda t) = \lambda \tau_{i+1}^j(t) \qquad \tau_i^j((t\,u)) = (\tau_i^j(t)\; \tau_i^j(u)) \qquad \tau_i^j(\underline{n}) = \begin{cases} \underline{n} & \text{if } n < i \\ \underline{n+j} & \text{if } n \geq i \end{cases}$$

Theorem 3.

1. *Let $t, u \in NF(l)$. If $t \longrightarrow_{\lambda_{lab}} u$ then $E(t) \longrightarrow_{\lambda_{db}} E(u)$.*
2. *Let $t \in NF(l)$ and u' be a λ_{db}-term. If $E(t) \longrightarrow_{\lambda_{db}} u'$, then $t \longrightarrow_{\lambda_{lab}} u$ for some $u \in NF(l)$ such that $u' = E(u)$.*

Theorem 4.

1. *Let t, u be λ_l-terms. If $t \longrightarrow_{\lambda_l} u$ then $l(t) \longrightarrow^*_{\lambda_{lab}} l(u)$.*
2. *Let $t, u \in NF(l)$. If $t \longrightarrow_{\lambda_{lab}} u$ then $t \longrightarrow^*_{\lambda_l} u$.*

6 Preservation of strong normalization

We have to show that $SN(\lambda_{db}) \subset SN(\lambda_l)$. It is easy to see that :

- If t is a λ_{db}-term then $t \in NF(l)$.
- For $t \in NF(l)$: $t \in SN(\lambda_{lab})$ iff $E(t) \in SN(\lambda_{db})$.

Thus, since the rigth notion of pure term for our calculus is λ-term with labels (cf. Sect.2), the PSN is given by :

Theorem 5. *Let $t \in NF(l)$ be such that $t \in SN(\lambda_{lab})$. Then $t \in SN(\lambda_l)$.*

6.1 The idea of the proof

The main problem is the following : let $t = [\sigma](\lambda x u\, v)$ where $[\sigma]$ is a substitution. In any calculus with explicit substitutions t can be reduced in $t' = (\lambda x[\sigma']u\,[\sigma]v)$. To avoid infinite reduction, we have to keep somewhere the information that x can not appear free in the image of $[\sigma']$. Both in $\lambda\sigma$ and in λs_e, this critical information can go down in the sub-term $[\sigma']u$, and thus, future interaction between $[\sigma']$ and the potential substitution $[x := [\sigma]v]$ is not prohibited.

In our calculus, the idea "keep information as high as possible" (by putting a label) avoids all future "potentially dangerous" interactions.

Recall that $t \in SN(R)$ means that t is strongly normalizable for the reduction R and that $t \in NF(l)$ means that t is in l-normal form.

PSN is a consequence of the following :

Fact. *If $t \in NF(l) - SN(\lambda_l)$ then, there is a term $u \in NF(l) - SN(\lambda_l)$ such that $t \longrightarrow_{\lambda_{lab}} u$.*

This implies the theorem because, using repeatedly this fact, we get an infinite λ_{lab}-reduction of a term in $NF(l) - SN(\lambda_l)$.

The contraction rule (d) makes the previous fact hard to prove. To keep a better control on the use of the contraction rule, we define the p-calculus which is the l-calculus without the rule d.

The p-calculus is confluent and strongly normalizable. The p-normal forms are given by the grammar :

$$t ::= \underline{n} \mid \lambda t \mid (t\ t) \mid \langle k \rangle t$$

Note that l-normal forms are p-normal forms but, for instance, $\langle 1\rangle\langle 2\rangle\underline{0}$ is a p-normal form and not an l-normal form.

The l-normal form can be recovered from the p-normal form by taking the d-normal form (the contraction rule is trivially confluent and strongly normalizable). Formally, $l(t) = d(p(t))$.

Instead of the previous fact, we thus use the following lemma to prove the theorem :

Lemma 2. *Let $t \in NF(p) - SN(\lambda_l)$. There is a term $\mathcal{T}(t) \in NF(p) - SN(\lambda_l)$ such that $d(t) \longrightarrow_{\lambda_{lab}} d(\mathcal{T}(t))$.*

This lemma also implies the theorem. Let $t_0 \in NF(l) - NF(\lambda_l)$, we can build an infinite sequence $t_0, t_1, \ldots$ of p-normal forms such that

$$t_0 = d(t_0) \longrightarrow_{\lambda_{lab}} d(t_1) \longrightarrow_{\lambda_{lab}} \cdots$$

and thus get a contradiction.

Let $t \in NF(l) - SN(\lambda_l)$.

1. (cf. Sect. 6.2) The term $\mathcal{T}(t)$ is given as follows (note that it is not necessarily unique) : Forget, to simplify, labels (for the general case we need the notion of "foot", "body" and "head" of the definition 2). Then t can be written as $t = \overrightarrow{\lambda}(H\ \overrightarrow{u})$ where H is either $(\lambda w\ v)$ or $\underline{n}$. $\mathcal{T}(t)$ is defined as follows :
   – if $H = \underline{n}$ (and thus some element of $\overrightarrow{u}$ is not in $SN(\lambda_l)$) or $H = (\lambda w\ v)$ and one of the sub-terms $\overrightarrow{u}, v, w$ is not in $SN(\lambda_l)$ then apply $\mathcal{T}$ to this term.
   – otherwise $\mathcal{T}(t)$ is the p-normal form of $\overrightarrow{\lambda}(H'\ \overrightarrow{u})$ where H' is the reduct of H.
2. (cf. Sect. 6.3) The fact that $\mathcal{T}(t) \notin SN(\lambda_l)$ is proved as follows : The reduction from t to $\mathcal{T}(t)$ can be done in two steps :
   – $t \longrightarrow t_0$ by reduction of the head redex (rule $b_1$or b_2).
   – $t_0 \longrightarrow^* u_0$ by propagation of the substitution created in t_0 (rules f, a, e_i and n_i). Note that, since t_0 has a unique substitution, the rules c_i are not used.

   Proving that $t \notin SN(\lambda_l)$ implies $t_0 \notin SN(\lambda_l)$ is easy. The hard part is to show that $u_0 \notin SN(\lambda_l)$. This is done by "translating" an infinite derivation of t_0 into an infinite derivation of u_0. We use a "tagged" calculus (cf. definition 5) to keep the "trace" of the substitution created in t_0. The translation consists in moving down all the "tagged" substitutions (cf. the definition 6).

In the next section, we give only the crucial points of the proof. For a complete proof see [5].

6.2 Definitions

The following notions are used in the definition of $\mathcal{T}$ as well as in the proof of Lem. 6.

Definition 2. *A context is a term with one hole denoted by $\{\!|\cdot|\!\}$. We define particular contexts (called* Foot *(F) and* Body *(B)) and terms (called* Head *(H)) as follows :*

$F = \{\!|\cdot|\!\} \mid F'$ *and* $F' = \lambda\{\!|\cdot|\!\} \mid \lambda F' \mid \langle k\rangle F'$

$B = \{\!|\cdot|\!\} \mid (B\; t) \mid \langle k\rangle B$ *where* $t \in NF(p)$

$H = \underline{n} \mid (\lambda u\; v) \mid (\langle k_1\rangle \ldots \langle k_n\rangle \lambda u\; v)$ *where* $u, v \in NF(p)$

Lemma 3. *Every term $t \in NF(p)$ can be uniquely written $t = F\{\!|B\{\!|H|\!\}|\!\}$.*

Example 3. Let $t = \lambda\langle 2\rangle\langle 3\rangle\lambda\langle 1\rangle(\langle 1\rangle\langle 3\rangle(\langle 2\rangle(\langle 1\rangle\langle 4\rangle\lambda v u)t_1)t_2)$. The decomposition of $t = F\{\!|B\{\!|H|\!\}|\!\}$ is given in Fig. 2.

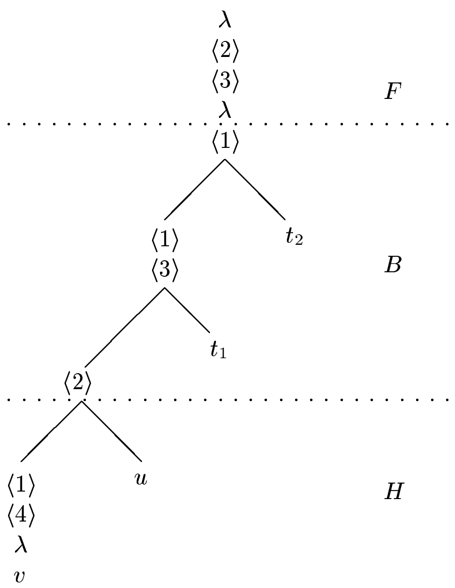


Fig. 2. Decomposition of the term t

Definition 3.

1. *Let B be a body, $Arg(B)$ (the set of* arguments *of B) is defined by :*
   – *if* $B = \{\!|\cdot|\!\}$ *then* $Arg(B) = \emptyset$,
   – *if* $B = (B'\; t)$ *then* $Arg(B) = Arg(B') \cup \{t\}$,
   – *if* $B = \langle k\rangle B'$ *then* $Arg(B) = Arg(B')$.
2. *Let $t = F\{\!|B\{\!|H|\!\}|\!\} \in NF(p)$. $Dst(t)$ (the set of* direct sub-terms *of t) is defined by :*
   – *if* $H = \underline{n}$ *then* $Dst(t) = Arg(B)$,

– *if* $H = (\lambda u\, v)$ *or* $H = (\langle k_1\rangle \ldots \langle k_n\rangle \lambda u\, v)$ *then* $Dst(t) = Arg(B) \cup \{u, v\}$.

Let t be as in example 3, then $Arg(B) = \{t_1, t_2\}$ and $Dst(t) = \{t_1, t_2, u, v\}$.

Definition 4. *Let* $t = F\{\!|B\{\!|H|\!\}|\!\} \in NF(p) - SN(\lambda_l)$. *We define* $\mathcal{T}(t) \in NF(p)$ *by :*

- *if* $Dst(t) \subset SN(\lambda_l)$
  - *if* $H = (\lambda v\; u)$ *then* $\mathcal{T}(t) = p(F\{\!|B\{\!|[0/u, 0]v|\!\}|\!\})$
  - *if* $H = (\langle k_1\rangle \ldots \langle k_n\rangle \lambda v\; u)$ *then* $\mathcal{T}(t) = p(F\{\!|B\{\!|[0/u, k]v|\!\}|\!\})$ *where* $k = \sum_{i=1}^{n} k_i$
- *otherwise, let* u *be (any) element in* $Dst(t) - SN(\lambda_l)$ *and* D *be the context such that* $t = D\{\!|u|\!\}$, *then* $\mathcal{T}(t) = D\{\!|\mathcal{T}(u)|\!\}$.

Let t be as in example 3. If $t_1, t_2, u, v \in SN(\lambda_l)$ and $t \notin SN(\lambda_l)$, then $\mathcal{T}(t) = \lambda\langle 2\rangle\langle 3\rangle\lambda\langle 1\rangle(\langle 1\rangle\langle 3\rangle(\langle 2\rangle p([0/u, 5]v)\; t_1)\; t_2)$.

6.3 Proof of PSN

Lemma 4. *Let* $t \in NF(p) - SN(\lambda_l)$. *Then,* $d(t) \longrightarrow_{\lambda_{lab}} d(\mathcal{T}(t))$

Proof. By induction on t. This comes from the decomposition $F\{\!|B\{\!|H|\!\}|\!\}$. (see [5] for the detailed proof). □

The key point in Lem. 2 is to show that $\mathcal{T}(t) \notin SN(\lambda_l)$.

We may assume that $F = \{\!| \cdot |\!\}$ and $Dst(t) \subset SN(\lambda_l)$. The first point comes from the fact that $F\{\!|B\{\!|H|\!\}|\!\} \in SN(\lambda_l)$ iff $B\{\!|H|\!\} \in SN(\lambda_l)$ and that $\mathcal{T}(F\{\!|B\{\!|H|\!\}|\!\}) = F\{\!|\mathcal{T}(B\{\!|H|\!\})|\!\}$. The second point comes from an easy induction on t.

Let $t = B\{\!|H|\!\} \in NF(p) - SN(\lambda_l)$ with $Dst(t) \subset SN(\lambda_l)$. We show that $\mathcal{T}(t) \notin SN(\lambda_l)$.

We have $H = (\langle k_1\rangle \ldots \langle k_n\rangle \lambda v\; u)$ (the case $H = (\lambda v\; u)$ is a particular case with $n = 0$). Let $k = \sum_{i=1}^{n} k_i$.

Lemma 5. *Let* $t_0 = B\{\!|[0/u, k]v|\!\}$. *Then* $t_0 \notin SN(\lambda_l)$

Proof. Since $Dst(t) \subset SN(\lambda_l)$, any infinite reduction must reduce the head redex. It is easy to see that the reduction of the head redex can be done before the other ones, and thus t_0 also has an infinite λ_l-reduction. □

We define a "tagged" calculus to be able to follow the residues of the substitution in t_0.

Definition 5.

1. *The set of* $\lambda_l^{\diamond}$*-terms is defined by :*

$$t = \underline{n} \mid \lambda t \mid (t\, t) \mid \langle k\rangle t \mid [i/t, j]t \mid [\![i/u, j]\!]v$$

   where $n, i, j, k \in \mathbb{N}$ *and* u, v *are* λ_l*-terms. A substitution of the form* $[\![i/u, j]\!]$ *is called a "tagged" substitution.*

2. *We add rules to move down tagged substitutions, these rules are the λ_l-rules where $[\![\ ]\!]$ replaces $[\ ]$: $f_\diamond$, $a_\diamond$, $e_{1\diamond}$, $e_{2\diamond}$, $n_{1\diamond}$, $n_{2\diamond}$, $n_{3\diamond}$. We also add rules to move down tagged substitutions over usual substitutions :*

$$c_{1\diamond}\ [\![i/u,j]\!][k/v,l]t \longrightarrow \Big[k/[\![i-k/u,j]\!]v, j+l-1\Big]t \qquad \textit{if } k \le i < k+l$$

$$c_{2\diamond}\ [\![i/u,j]\!][k/v,l]t \longrightarrow \Big[k/[\![i-k/u,j]\!]v, l\Big][\![i-l+1/u,j]\!]t \quad \textit{if } k+l \le i$$

These new rules will be called "tagged" rules.

3. *The $\lambda_l^\diamond$-calculus is the calculus with the λ_l rules and the tagged rules. The $\diamond$-calculus is the restriction of this calculus to the tagged rules.*

Note that it follows from the syntax that there is at most one tagged substitution in each branch of a term. Also note that there are no rules to move untagged substitutions over tagged ones.

Definition 6. *For a $\lambda_l^\diamond$-term t, let $\diamond(t)$ be the $\diamond$-normal form of t.*

It is easy to see (by using a measure on terms that decreases by reduction) that the $\diamond$-calculus is strongly normalizable. Since it has no critical pair the $\diamond$-normal forms are well-defined.

Recall that $t_0 = B\{\![[0/u,k]]\!\}$. Let $t_0' = B\{\![[\![0/u,k]\!]]\!\}$.

Lemma 6. *Let v be a $\lambda_l^\diamond$-term such that $t_0' \longrightarrow^*_{\lambda_l} v$. Then*

1. *Assume $[j/w,k][\![l/w',m]\!]w''$ is a sub-term of v. Then $j < l$.*
2. *Assume $[\![j/w,k]\!][l/w',m]w''$ is a sub-term of v. Then $j \ge l$ and one of the rules $c_{\diamond 1}$ or $c_{\diamond 2}$ can be applied.*
3. *Assume $[\![j/w,k]\!]w'$. Then $w \in SN(\lambda_l)$.*

Proof. This lemma requires a long proof. This is done by defining a subset of the $\lambda_l^\diamond$-terms called *well-tagged* terms. Formally, we define a binary relation $\mathcal{H}$ on $\lambda_l^\diamond \times \mathbb{N}$. $\mathcal{H}(t,m)$ means that all tagged substitutions in t is at depth m in t and any other substitution is at depth less than m. A term t is well-tagged if there is some m such that $\mathcal{H}(t,m)$. We prove that the relation $\mathcal{H}$ is preserved by reduction. The lemma is an immediate consequence of the fact that v is a well-tagged term. $\square$

Note that this lemma has no equivalent in λs_e because there is no way to forbid a composition of $[\ ]$ over $[\![\ ]\!]$. In the $\lambda\sigma$ or λs_e, if one would try to defined a tagged calculus, the Lem. 6 would fail. For instance, in λs_e, it is possible to have a term $[\sigma_1][\sigma_2]t$ such that "$[\sigma_1]$ and $[\sigma_2]$ can not interact" and, after some reduction steps, the term becomes $[\sigma_1][\sigma_2']t'$ such that "$[\sigma_1]$ and $[\sigma_2']$ can interact".

The first point of the lemma ensures that missing rules of the $\lambda_l^\diamond$-calculus (rules moving untagged substitutions over tagged ones) are not necessary in any reduction of t_0'.

Thus, we may define an infinite $\lambda_l^\diamond$-reduction $t'_0 \longrightarrow t'_1 \longrightarrow \ldots$ from the infinite reduction of t_0 (t'_i is the term t_i with some tagged substitutions).

Let u_0 be the p-normal form of t_0. It is easy to see that $u_0 = \diamond(t'_0)$. For $i > 0$, let $u_i = \diamond(t'_i)$.

The second point of Lem. 6 implies that for each $i \in \mathbb{N}$, $\diamond(t'_i)$ is a λ_l-term, i.e. contains no tagged substitutions.

The various steps in the reduction of t'_0 can be divided in :

- S_1-reduction : propagation of a tagged substitution (i.e. a tagged rule is used) or reduction "inside" a tagged substitution (i.e., $[\![i/u, j]\!]v \longrightarrow [\![i/u', j]\!]v$).
- S_2-reduction : any other reduction (i.e. a λ_l-rule outside a tagged substitution).

Lemma 7. *Let v a term of the tagged calculus such that $t_0 \longrightarrow_{\lambda_l^\diamond} v$. The term v is strongly normalizable for the S_1-reduction.*

Proof. Because $\diamond$ is strongly normalizing and terms inside tagged substitutions come from a term which is in $SN(\lambda_l)$. $\square$

Lemma 8.

- *If $t'_i \longrightarrow_{S_1} t'_{i+1}$ then $u_i \longrightarrow^*_{\lambda_l} u_{i+1}$*
- *If $t'_i \longrightarrow_{S_2} t'_{i+1}$ then $u_i \longrightarrow^+_{\lambda_l} u_{i+1}$*

Proof. By induction on t'_i. The only difficult cases are the ones where the reduction occurs at the root of t. We get the result by case analysis of the reduction of t'_i to t'_{i+1}. The proof is essentially the same as the proof of the local confluence. $\square$

Lemma 9. $u_0 \notin SN(\lambda_l)$

Proof. By the following diagram.

$$\begin{tikzcd}
t'_0 \arrow[r, "S_1^*"] \arrow[d, mapsto, "\diamond"'] & t'_{i_1} \arrow[r, "S_2"] \arrow[d, mapsto, "\diamond"'] & t'_{i_1+1} \arrow[r, "S_1^*"] \arrow[d, mapsto, "\diamond"'] & t'_{i_2} \arrow[r, "S_2"] \arrow[d, mapsto, "\diamond"'] & t'_{i_2+1} \arrow[d, mapsto, "\diamond"'] & \ldots \\
u_0 \arrow[r, dotted, "\lambda_l^*"] & u_{i_1} \arrow[r, dotted, "\lambda_l^+"] & u_{i_1+1} \arrow[r, dotted, "\lambda_l^*"] & u_{i_2} \arrow[r, dotted, "\lambda_l^+"] & u_{i_2+1} & \ldots
\end{tikzcd}$$

$\square$

7 The typed λ_l-calculus

We give here a simply typed version of the calculus. Types are, as usual, given by $T = V \mid T \longrightarrow T$ and contexts are (ordered) lists of types. The typing rules for the λ_l-calculus are given below (where $i = length(\Gamma) \quad j = length(\Delta)$).

$$\frac{}{\Gamma, A, \Delta \vdash \underline{i} : A}\,(axiom)$$

$$\frac{A, \Gamma \vdash t : B}{\Gamma \vdash \lambda t : A \to B}\,(\to_i) \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash (t\,u) : B}\,(\to_e)$$

$$\frac{\Delta \vdash t : A}{\Gamma, \Delta \vdash \langle i \rangle t : A}\,(weakening) \quad \frac{\Delta, \Pi \vdash u : A \qquad \Gamma, A, \Pi \vdash t : B}{\Gamma, \Delta, \Pi \vdash [i/u, j]t : B}\,(cut)$$

Theorem 6. *If $t \longrightarrow_{\lambda_l} u$ and $\Gamma \vdash t : A$ then $\Gamma \vdash u : A$*

Problem 1. It is most likely that typed terms are strongly normalizing but we do not know how to extend the usual proofs of strong normalization to λ_l. It seems that this could be done by using the same kind of techniques as in the proof of PSN.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
2. P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitution. *Journal of the ACM*, 43(2):362–397, 1996.
3. M. Ferreira, D. Kesner, and L. Puel. λ-calculi with explicit substitutions and composition which preserve β-strong normalisation. *Lectures Notes in Computer Science*, 1139:284–298, September 1996.
4. B. Guillaume. The λs_e-calculus does not preserve strong normalisation. *to appear in Journal of Functional Programming*, 1998.
5. B. Guillaume. *Un calcul de substitutions avec étiquettes.* PhD thesis, Université de Savoie, 1999. (URL : `http://lama.www.univ-savoie.fr/~Bruno.Guillaume`).
6. T. Hardin. Confluence results for the pure strong categorical logic CCL : λ-calculus as subsystems of CCL. *Theorical computer science*, 65(2):291–342, 1989.
7. F. Kamareddine and A. Ríos. Extending a λ-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *Journal of Functional Programming*, 7(4):395–420, July 1997.
8. F. Lang and K.-H. Rose. Two Calculi of Explicit Substitution with Confluence on Metaterms and Preservation of Strong Normalization. *Proceedings of WESTAPP*, 1998.
9. P. Lescanne. From $\lambda\sigma$ to $\lambda\nu$, a journey through calculi of explicit substitutions. *Proceedings of the 21st conference on Principles of Programming Languages*, pages 60–69, 1994.
10. P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. *Lecture Notes in Computer Science*, 902:328–334, 1995.
11. C. Muñoz. Confluence and preservation of strong normalisation in an explicit substitutions calculus. *Proceedings of LICS'96*, pages 440–447, 1996.
12. K.-H. Rose. Explicit substitution – tutorial & survey. Lecture Series LS–96–3, BRICS, 1996.

Cut Rules and Explicit Substitutions

René Vestergaard
Heriot-Watt University
http://www.cee.hw.ac.uk/~jrvest/

Joe Wells*
Heriot-Watt University
http://www.cee.hw.ac.uk/~jbw/

May 20, 1999

Abstract

We introduce a method to associate calculi of proof terms and rewrite rules with cut elimination procedures for logical deduction systems (i.e., Gentzen-style sequent calculi) in the case of intuitionistic logic. We illustrate this method using two different variations of the cut rule for a variant of the intuitionistic fragment of Kleene's logical deduction system G3. Our systems are in fact calculi of explicit substitution, where the cut rule introduces an explicit substitution and the left-$\rightarrow$ rule introduces a binding of the result of a function application. Cut propagation steps of cut elimination correspond to propagation of explicit substitutions and propagation of weakening (to eliminate it) corresponds to propagation of index-updating operations. We prove various subject reduction, termination, and confluence properties for our calculi.

Our calculi improve on some earlier calculi for logical deduction systems in a number of ways. By using de Bruijn indices, our calculi qualify as first-order term rewriting systems (TRS's), allowing us to correctly use certain results for TRS's about termination. Unlike in some other calculi, each of our calculi has only one cut rule and we do not need unusual features of sequents.

We show that the substitution and index-updating mechanisms of our calculi work the same way as the substitution and index-updating mechanisms of Kamareddine and Ríos' λs and λt, two well known systems of explicit substitution for the standard λ-calculus. By a change in the format of sequents, we obtain a similar result for a known λ-calculus with variables and explicit substitutions, Rose's $\lambda bxgc$.

1 Introduction

1.1 Background and Motivation

Natural and Logical Deduction. We assume logic systems are presented in sequent style, i.e., they allow derivations of sequents $\Gamma \vdash \tau$, where Γ is the *antecedent* or *assumptions*, generally a collection of formulae, and τ is the *succedent* or *consequence*. In many logics (e.g., in some presentations of classical logic), succedents are also collections of formulae. Because this paper deals exclusively with intuitionistic logic (in fact, only the implicative fragment), we require succedents to be a single consequent formula.

Natural deduction systems, which we choose to call N-systems, are symbolic logics generally given via introduction and elimination rules for the logical connectives which operate on the right, i.e., they manipulate the succedent formula. Examples are Gentzen's NJ and NK [21]. Logical deduction systems are given via left-introduction and right-introduction rules for the logical connectives. Although others have called these systems "sequent calculi", we call them L-systems to avoid confusion with other systems given in sequent style. Examples are Gentzen's LK and LJ [21]. In this paper we are primarily interested in L-systems. The advantage of N-systems is that they seem closer to actual reasoning, while L-systems on the other hand seem to have an easier proof theory. L-systems are often extended

*Supported by EPSRC grant GR/L 36963 and NSF grants CCR–9417382 and CCR–9806747.

with a "cut" rule as part of showing that for a given L-system and N-system, the derivations of each system can be encoded in the other. For example, NK proves the same as LK + cut [21].

Proof Normalization. A system is *consistent* when it is impossible to prove false, i.e., derive absurdity from zero assumptions. A system is *analytic* (has the *analycity* property) when there is an effective method to decompose any conclusion sequent into simpler premise sequents from which the conclusion can be obtained by some rule in the system such that the conclusion is derivable iff the premises are derivable.

To achieve the goals of consistency and analycity, it has been customary to consider *normal-form* fragments of logics, which ideally only include derivations satisfying the *subformula property*. This property can be simply stated as "every formula appearing in the derivation also appears as a subformula in the final sequent". Having a set of normal derivations with the subformula property has clear benefits: (1) the space of possible derivations of a sequent is easier to search, and (2) it can be shown that the sequent having absurdity as a consequence of nothing is not the result of a normal derivation using the simple reason that the only subformula of absurdity is itself. Normalization is the process of taking any derivation and finding a derivation in the normal-form fragment with the same final sequent. A logic with an effective normalization procedure will thus ideally be consistent and analytic.

For N-systems, a normal derivation is one where an introduction rule is not followed by an elimination rule that eliminates the connective just introduced. For L-systems, the notion of normal is simpler: a derivation is normal if it is cut-free. Normalization for an L-system shows the cut rule is admissible, i.e., unnecessary for provability of propositions. Gentzen's Hauptsatz [21] ("main statement") is cut elimination for LK, i.e., LK + cut = LK, and, hence, in turn showing the consistency of LK + cut and therefore also of NK.

Curry-Howard Correspondence. The process of normalization can be used to obtain a correspondence between a logic and a calculus. The correspondence was observed by Howard [27] in the case of the simply typed λ-calculus [12] and the N-system NJ and by Curry [16] in the similar case of combinators and a particular Hilbert-style logic, Combinatory Logic. (A Hilbert-style logic generally only has axioms and *modus ponens* for rules and dispenses with assumptions.) The correspondence maps formulae into types and proofs into terms. Derived formulae are associated with proof terms and assumptions are associated with variables. Rewriting (a.k.a. reduction) of terms is proof transformation and rewriting normal forms are normal proofs. The correspondence is sometimes an isomorphism where each proof corresponds with exactly one term, after proofs are identified up to α-conversion and excessive context. An isomorphism is easy to obtain by making the appropriate term constructor for each proof rule. In the case of an isomorphism, typed terms form a succinct notation for proofs.

Explicit Substitution Calculi. A calculus with rewrite rules which involve substitution, e.g., the λ-calculus, can be simulated by a system with additional rules and term constructors which carries out the substitutions in a stepwise and incremental manner [1]. Among other benefits, explicit substitution makes rewrite steps closer to actual machine operations and allows more laziness in evaluation. Explicit-substitution calculi have been used for, among other things, work on proof-search [44], unification [17], representation of incomplete proofs [29, 35], proof theory [26], and functional compiler back-ends [24].

Curry-Howard Correspondence for L-Systems. While the Curry-Howard Correspondence is straightforward and, hence, well developed for N-systems, it has also been applied to L-systems. Although right-introduction rules are generally treated the same way as for N-systems, there has been considerable variation in the treatment of the cut rule and left-introduction rules. An older approach associates cut and left-introduction rules with the results of meta-level substitutions:

$$\text{(cut)} \quad \frac{\Gamma \vdash e_1 : \sigma \quad x : \sigma, \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_2[x{:=}e_1] : \tau}$$

$$(\rightarrow\text{-left}) \quad \frac{\Gamma \vdash e_1 : \sigma \quad x : \tau, \Gamma \vdash e_2 : \rho}{z : \sigma \rightarrow \tau, \Gamma \vdash e_2[x{:=}(z e_1)] : \rho}$$

Here the expression $e_2[x{:=}e_1]$ denotes the *result* of a substitution rather than a new kind of syntax. This approach allows a certain kind of comparison with N-systems, as the set of proof terms can be the same. Examples of this approach include work on L-systems for intuitionistic linear logic with proof terms in N-system style [2, 4], work which discusses the relationship between LJ and NJ [54, 41, 42], and other work [37].

A more recent approach gives explicit syntax for cut and the left-introduction rules which can be viewed as an explicit substitution or alternatively as a variable binding "let" construct. So where the earlier approach used a meta-level substitution, this approach might use rules like this:

$$\text{(cut)} \quad \frac{\Gamma \vdash e_1 : \sigma \quad x : \sigma, \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_2\{\!\{x := e_1\}\!\} : \tau}$$

$$(\rightarrow\text{-left}) \quad \frac{\Gamma \vdash e_1 : \sigma \quad x : \tau, \Gamma \vdash e_2 : \rho}{z : \sigma \rightarrow \tau, \Gamma \vdash \text{let } x = (z\ e_1) \text{ in } e_2 : \rho}$$

Here, the expressions $e_2\{\!\{x := e_1\}\!\}$ and (let $x = (z\ e_1)$ in e_2) denote new syntactic constructs rather than the results of meta-level substitutions. Generally in this approach, rewriting corresponds to the cut propagation of the cut elimination procedure. The results we present in this paper follow this approach. One example of this approach is by Gallier [20], who gives a term assignment for a system similar to LJ. Some example rewrite rules are given, but there is no detailed study and no comparison with the explicit substitution literature. In another example, Pfenning [40, 39] represents a cut elimination proof in Elf.

A second example of this approach is the work of Wadler [53], who gives a proof term assignment for an L-system as well as rewrite rules. The largest differences between Wadler's approach and ours are that he (1) uses names for variables instead of de Bruijn indices and (2) does not give explicit syntax to the structural rules, which in his calculus are contraction and weakening. In Wadler's calculus, a let-binding which is syntactically evident can be spontaneously duplicated due to the presence of a use of contraction the position of which is *not* syntactically evident. Similarly, a let-binding can be spontaneously annihilated due to the presence of a use of weakening the position of which is not syntactically evident.

A third example of this approach is the work of Herbelin [25]. Herbelin presents an L-system LJT, a restriction of LJ, where one of the assumptions in each sequent may be designated as being in the "stoup". The axiom, $\rightarrow$-left, and contraction rules must use the stoup assumption. There are two different cut rules, C_H and C_M, one of which uses the stoup assumption. There is no weakening rule. The calculus $\bar{\lambda}$ which Herbelin defines to be in correspondence with LJT has two sorts, argument lists and terms, with explicit substitutions for both sorts. The proof rules are then divided into two categories, those for applicative contexts (i.e., argument lists) and those for terms. Each of the cut rules must actually be specialized into two versions, one for each sort, so there end up being four cut rules total. Herbelin shows that the normal forms of his system are isomorphic to normal N-system derivations, although the non-normal forms differ quite substantially.

A final example of this approach is in the very recent work of Cerrito and Kesner [11]. They develop a typed calculus in which the pattern matching of ML-like languages is given meaning via rewrite rules which are essentially performing cut elimination. When one erases the term annotations from their typing rules, the logic that results is an L-system with implication ($\rightarrow$), conjunction ($\times$), and disjunction ($+$). Because of choices they make in formulating their system, it appears that both cut and contraction are not admissible rules of their system. They specifically note that their rewrite rules do not eliminate all uses of cut. In the application of their system for programming languages this seems to be beneficial.

There are other approaches to assigning terms to the cut rule and the left-introduction rules which do not neatly fit into the categories given above. Breazu-Tannen et al. associate left-introduction rules with pattern matching constructs [8], but in a manner which does not correspond to what goes on in cut elimination proofs. Dychoff and Pinto [19] as well as Schwichtenberg [47] give syntactic constructs to left rules in L-systems without cut. They show the equivalence classes of L-system proofs induced by an epimorphic mapping between L-systems and N-system [41] are axiomatised by *permutative conversions*. They are concerned with completeness (SN and CR) of permutative conversions.

L-System Formulations The proof theory and the behavior of normalization of an L-system depends on subtle details of its formulation. Gentzen [21] designed the first L-system, called G1 by many later researchers [49]. The intuitionistic fragment of G1, called LJ by Gentzen, has a $\rightarrow$-left rule somewhat like this one:[1]

$$(\rightarrow\text{-left}) \quad \frac{\Gamma \vdash \sigma \quad \tau, \Gamma \vdash \rho}{\sigma \rightarrow \tau, \Gamma \vdash \rho}$$

Unfortunately, G1 also needs a contraction rule:

$$(\text{contraction}) \quad \frac{\Gamma, \tau, \tau \vdash \sigma}{\Gamma, \tau \vdash \sigma}$$

The impossibility of eliminating uses of contraction from a G1 proof means that proof search must consider contraction, introducing non-determinism and making it more difficult to bound the size of the search space. Also, handling contraction as part of a Curry-Howard isomorphism is painful. In particular, the straightforward term annotation for contraction is an explicit duplication of a variable, e.g., (duplicate x as y, z in e) which can require intricate propagation rules. Also, in the presence of any such explicit duplication, all other rules which allow sharing assumptions between multiple premises may need term annotations that record similar variable duplicating information. Getting this right can be quite tricky and not doing it may prevent contraction or cut elimination from succeeding. Another issue for G1 is that cut elimination by propagation may need, in intermediate stages, a more powerful *multicut* rule.

To eliminate the need for contraction, Kleene [34] devised his G3 system,[2] which we call GK3, the intuitionistic fragment of which has a $\rightarrow$-left rule somewhat like this one:

$$(\rightarrow\text{-left}) \quad \frac{\sigma \rightarrow \tau, \Gamma \vdash \sigma \quad \tau, \sigma \rightarrow \tau, \Gamma \vdash \rho}{\sigma \rightarrow \tau, \Gamma \vdash \rho}$$

The difference from G1 is that this rule requires the formula constructed by this rule in the antecedent of the derived sequent to already be present in the antecedents of both premise sequents. This paper will use the implicative, intuitionistic fragment of GK3 with only one change — we will treat the antecedent as a sequence rather than as a multiset.

Later, Dragalin [18] invented another system, called G3 by Troelstra and Schwichtenberg [49], the intuitionistic fragment of which has a $\rightarrow$-left rule somewhat like this one:

$$(\rightarrow\text{-left}) \quad \frac{\sigma \rightarrow \tau, \Gamma \vdash \sigma \quad \tau, \Gamma \vdash \rho}{\sigma \rightarrow \tau, \Gamma \vdash \rho}$$

The difference from GK3 is that this rule requires the constructed formula to already be present in the antecedent of the left premise, but not in the antecedent of the right premise. With Dragalin's G3, both cut and contraction can now be eliminated, but the intermediate stages of the contraction elimination procedure need an inversion principle or rule. Unfortunately, as shown by one of us (Vestergaard) [50], inversion leads to a computational anomoly, because it implies that the functions it is used on are constant functions. Thus, Dragalin's G3 is a poor candidate for attempting to form a Curry-Howard isomorphism.

de Bruijn Indices. Traditionally, the λ-calculus and similar calculi have represented variables with names. De Bruijn introduced the idea of representing variables with indices instead of names [10]. Using indices avoids the need to work with α-equivalence classes. When carrying out substitution and also using names, renaming by α-conversion is needed to correctly preserve bindings during substitution. When using indices instead of names, this job is performed by index-updating operators. The essential feature of indices is that index-updating operators can introduce the equivalent of fresh "names" by shifting all other indices.

[1] In Gentzen's system, the premises of the $\rightarrow$-left rule do not share assumptions.

[2] Unfortunately, many systems are known by this name.

de Bruijn Indices and Explicit Substitution. In explicit substitution, the index-updating operators become term-level constructs with their own rewrite rules. Among explicit-substitution calculi simulating the λ-calculus, many use de Bruijn indices, e.g., $C\lambda\xi\phi$ [9]; the calculi of categorical combinators [14]; $\lambda\sigma, \lambda\sigma_{\Uparrow}, \lambda\sigma_{SP}$ [1, 15, 43] (the $\lambda\sigma$-family); $\lambda\nu$ [3], $\lambda\zeta$ [28] (the $\lambda\sigma$-descendants); $\lambda s, \lambda s_e, \lambda t$ [30, 31, 32]. Explicit-substitution calculi for the λ-calculus with names include calculi such as $\lambda bxgc$ and its variants [6, 7, 46, 45].

1.2 Our Contribution

We present a fragment $GK3i_{dB}$ of the intuitionistic portion of Kleene's G3 with ordered assumptions. We define two variations on the cut rule, ($\mathrm{Cut}_{\mathrm{ns}}$) and ($\mathrm{Cut}_{\mathrm{cs}}$), which differ in how they handle sharing of assumptions between their premises, as well as a weakening rule (Weak) and an assumption permuting rule (Reloc). We then consider the logics $GK3i_{dB}$ + (Weak) + (Reloc) + ($\mathrm{Cut}_{\mathrm{ns}}$) and $GK3i_{dB}$ + $(\mathrm{Weak})_{k,1}$ + ($\mathrm{Cut}_{\mathrm{cs}}$). For both of these logics, we define proof term annotations and rewrite rules, yielding the calculi λ^{ns} and λ^{cs}. The two variations allow showing correspondences with the substitution mechanisms of two existing systems of explicit substitution, λs and λt.

Both variations share several properties. The proof term annotations use de Bruijn indices to manage assumptions. Both variations on the cut rule introduce an explicit substitution. The $\to$-left rule binds the result of a function application. In the rewrite rules, cut propagation becomes explicit substitution propagation and weakening propagation becomes index-updating propagation. For both variations, we are able to prove the following:

- Each calculus (either λ^{ns} or λ^{cs}) has the subject reduction (SR) property. Thus, the proof transformations represented by the rewrite rules are correct.

- Each calculus (either λ^{ns} or λ^{cs}) satisfies the weak innermost normalization (WIN) property on well typed terms. Furthermore, normal forms are cut-free. Thus, there is a cut elimination procedure for the logic ($GK3i_{dB}$ + (Weak) + (Reloc) + ($\mathrm{Cut}_{\mathrm{ns}}$) or $GK3i_{dB}$ + $(\mathrm{Weak})_{k,1}$ + ($\mathrm{Cut}_{\mathrm{cs}}$)).

- A slight restriction (either $\lambda^{\mathrm{ns}}_{-}$ or $\lambda^{\mathrm{cs}}_{-}$) of each calculus (either λ^{ns} or λ^{cs}) which removes critical pairs of the rewrite rules satisfies (1) SR, (2) the confluence or Church-Rosser (CR) property, and (3) the strong normalizing (SN) property. Thus, for the restricted calculus, procedures for cut elimination by cut propagation can use any strategy of choosing which cut to propagate next. This is often called "strong" cut elimination.

- The way each calculus (either λ^{ns} or λ^{cs}) propagates explicit substitions corresponds closely to the way a well known system of explicit substitution for the λ-calculus (either λs or λt) propagates substitutions.

The advantages of our approach over some other L-systems with explicit syntax for cut and the left-introduction rules are the following:

- In comparison with Herbelin's LJT logic and its associated $\bar{\lambda}$-calculus, we have only one syntactic sort and one rule for cut. We do not need any term constructors corresponding to argument list prefixing or concatenation. Also, as our sequents do not have a stoup, we do not have the restriction that the axiom and $\to$-left rules must use only the stoup formula, so we avoid the need to use Herbelin's somewhat awkward encoding of LJ into LJT which requires the introduction of applicative contexts.

- Unlike some previous work, we treat all structural rules explicitly. This aids in understanding what the computational meaning of these rules might be. Also, by having term constructors for every rule, our terms can be seen as a succinct syntax for derivations.

- Our calculi are the first for L-systems to represent variables by de Bruijn indices instead of by names. The structural rules of our logics are associated with term-level index-updating operators. Explicit index updating allows our calculi to be first-order term rewriting systems (TRS's). If we relied on implicit α-conversion to do this renaming, then our systems would be higher-order rewrite systems, but would not qualify as first-order. The fact that our calculi are first-order TRS's allows us to safely and correctly use results in the literature for first-order TRS's which give conditions when WIN implies SN [38, 22]. This gives a solid foundation for our strong cut-elimination result, in contrast to some other work in the literature which incorrectly attempts to use theorems about first-order systems for higher-order systems.

- The two calculi λ^{ns} and λ^{cs} help to illustrate our contention that *the structure of inference rules matters.* The difference in the assumption maintenance mechanism of the logics $GK3i_{dB}$ + (Weak) + (Reloc) + $(\mathrm{Cut}_{\mathrm{ns}})$ and $GK3i_{dB}$ + $(\mathrm{Weak})_{k,1}$ + $(\mathrm{Cut}_{\mathrm{cs}})$ is reflected in the way that λ^{ns} and λ^{cs} update indices and introduce the weakening and assumption reordering operators.

- We show a weak correspondence with the existing explicit substitution calculi λs and λt. This weak correspondence shows that the mechanisms for propagating explicit substitutions and updating indices in λ^{ns} and λ^{cs} are nearly the same as the mechanisms of well known and established explicit substitution calculi. We also discuss a very similar weak correspondence with Rose's $\lambda bxgc$. These correspondences with the substitution mechanisms of established explicit substitution calculi are the first shown for a calculus for an L-system.

1.3 Additional Related Work

Other related works, somewhat orthogonal to the results here, are as follows. Maraist et al. [36] present a typed let-calculus. The type system is an N-system using a cut rule to type the let-construct. The proof theory of the type system is not considered (apart from the statement of a SR-result) and no comparative results as to the explicit-substitution nature of the let construct are given. In fact, meta-level substitution is used in reducing the let construct. Di Cosmo and Kesner [13] encode calculi with explicit substitution into proof nets in order to show the calculi SN. Pagano develops a logic corresponding to the $\lambda\sigma_{\Uparrow}$ explicit substitution calculus. This is the only other work we are aware of which combines de Bruijn indices with a presentation of cut elimination. However, the logic is an N-system and hence orthogonal to our goals.

2 Logical Deduction and Computation

We define $GK3i_{dB}$, a proof-term-annotated L-system for the implicative fragment of intuitionistic logic. The inference rules of $GK3i_{dB}$ are given in Figure 2 using the operations and conventions for formulae, contexts, and metavariables given in Figure 1. When the rules of a logic are annotated with terms and a fixed order is imposed on the assumptions in a sequent, the antecedent (material to the left of "$\vdash$") is called a *context.* The system $GK3i_{dB}$ is more than a logic because it has proof terms. It is not exactly an inductive definition of a set of (explicitly typed) terms of a calculus, because it is context-sensitive due to the assumptions prescribing a set of permissible indices. However, both a logic and a set of terms are easily extracted from the definition of $GK3i_{dB}$ and we will sometimes take $GK3i_{dB}$ to refer to the logic or the set of terms. We define the *end sequent* of a derivation to be its final sequent with the proof term removed.

The system $GK3i_{dB}$ is a refinement of the implicative fragment of Kleene's G3 [34], which we call GK3. Kleene's original definition effectively took contexts to be multisets of formulae, but we take contexts to be lists of formulae. (We will briefly consider contexts as multisets in Section 3.3).

For the $(\to\mathrm{L})$ rule, notice that the (left) introduced formula is required to already be present in the assumptions of both premises. This allows proving the admissibility proofs of the structural and cut rules (in Figures 3 and

Indices, formulae (τ)

$$\begin{array}{rcl} k, m, l, v, n, i & ::= & 0 \mid 1 \mid \ldots \\ \tau, \sigma, \nu, \rho & ::= & \alpha \mid \tau \to \tau \end{array}$$

Contexts (Γ), context-smash ($\oplus$)

$$\begin{array}{rcl} \Gamma & ::= & \tau_1, \ldots, \tau_k \\ |\tau_1, \ldots, \tau_k| & = & k \\ \Gamma^1 \oplus \Gamma^2 & = & \begin{cases} \Gamma^2 & \text{if } \Gamma^1 \text{ is a prefix of } \Gamma^2 \\ \Gamma^1 & \text{if } \Gamma^2 \text{ is a proper prefix of } \Gamma^1 \\ \text{undefined} & \text{otherwise} \end{cases} \end{array}$$

Figure 1: Contexts (Γ) as finite but unbounded lists of formulae.

$$\frac{}{\Gamma^1, \tau, \Gamma^2 \vdash n : \tau}\,(\mathrm{Ax})_n \quad \text{if } |\Gamma^1| = n-1 \qquad\qquad \frac{\tau, \Gamma \vdash e : \sigma}{\Gamma \vdash \lambda e : \tau \to \sigma}\,(\to\mathrm{R})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \sigma, \Gamma \vdash e_2 : \nu}{\Gamma \vdash \mathrm{bind}_1\ (k\, e_1)\ \mathrm{in}\ e_2 : \nu}\,(\to\mathrm{L})_k \quad \text{if } \Gamma = \Gamma^1, \tau \to \sigma, \Gamma^2 \text{ and } |\Gamma^1| = k-1$$

Figure 2: The logical rules, together called $GK3i_{dB}$.

4) without needing a contraction rule (cf. [49] and the discussion on L-System Formulations in Section 1). The subscript 1 on "bind_1" in the term annotation serves merely as a reminder that this form binds index 1 in the body e_2 to the result of the function call $(k\, e_1)$.

We will independently consider the two different cut rules for $GK3i_{dB}$ given in Figure 3. Both are indexed by the level (position in the context) of the assumed formula of the right premise they "cut out". They differ only in the way they manage the two contexts in the premises. The ($\mathrm{Cut_{cs}}$) rule is context-sharing, like ($\to$L). In contrast, the ($\mathrm{Cut_{ns}}$) rule can express completely non-context-sharing cuts in the case when Γ' is empty. We avoid restricting ($\mathrm{Cut_{ns}}$) to always be non-context-sharing because then we would need a contraction rule (see the discussion on L-System Formulations in Section 1 for why we avoid having a contraction rule). Instead, ($\mathrm{Cut_{ns}}$) has some built-in ability to do the equivalent of contraction, exactly the amount needed for correct cut propagation (see the (β_{ns}) rule in Figure 7 which introduces new cuts needing the contraction ability). We consider both the context-sharing and non-context-sharing versions of cut because they are both commonly used and also because their natural propogation rules turn out to behave similarly to the rules of some existing calculi of explicit substitution (see Section 3).

The structural rules, (Weak) and (Reloc) given in Figure 4, are necessary in intermediate steps during the cut elimination procedure. Eliminating the ($\mathrm{Cut_{ns}}$) rule requires both structural rules while eliminating ($\mathrm{Cut_{cs}}$) only requires a restriction of (Weak). The structural rules are themselves admissible for $GK3i_{dB}$ (with elimination procedures given in Figures 5 and 6). A rule name without its indices, e.g., (Weak), stands for the collection of the rule at all possible values of the indices.

Theorem 1 (Admissibility) *Cuts + Struct are admissible for $GK3i_{dB}$.*

Proof Sketch: From the proof theory viewpoint, for the uses of rules in *Cuts + Struct* in a derivation, we show how to propagate at least one non-leaf use towards the leaves and how to eliminate leaf uses while preserving the derivation's end sequent. By the Curry-Howard isomorphism, we can specify this via rewrite rules on proof terms, which we give

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma^1, \tau, \Gamma' \vdash e_2 : \sigma}{\Gamma^1, (\Gamma \oplus \Gamma') \vdash e_2\{\!\{k := e_1\}\!\} : \sigma} \ (\mathrm{Cut}_{\mathrm{ns}})_k \quad \text{if } \Gamma \oplus \Gamma' \text{ is defined and } |\Gamma^1| = k-1$$

$$\frac{\Gamma^1, \Gamma^2 \vdash e_1 : \tau \qquad \Gamma^1, \tau, \Gamma^2 \vdash e_2 : \sigma}{\Gamma^1, \Gamma^2 \vdash e_2[\![k := e_1]\!] : \sigma} \ (\mathrm{Cut}_{\mathrm{cs}})_k \quad \text{if } |\Gamma^1| = k-1$$

Figure 3: The cut rules, together called *Cuts*.

$$\frac{\Gamma^1, \Gamma^2 \vdash e : \sigma}{\Gamma^1, \Gamma, \Gamma^2 \vdash \mathcal{W}_k^m(e) : \sigma} \ (\mathrm{Weak})_{k,m} \quad \text{if } |\Gamma^1| = k \text{ and } |\Gamma| = m$$

$$\frac{\Gamma^1, \Gamma^2, \tau, \Gamma^3 \vdash e : \sigma}{\Gamma^1, \tau, \Gamma^2, \Gamma^3 \vdash \mathcal{R}_l^k(e) : \sigma} \ (\mathrm{Reloc})_{k,l} \quad \text{if } |\Gamma^1, \Gamma^2| = k-1 \text{ and } |\Gamma^1| = l-1$$

Figure 4: The structural rules, together called *Struct*.

in Figures 5, 6, 7, and 8. The proof is by a (standard) case analysis on the last rule applied in the premise(s). For *Cuts* there are two cases: (1) at least one premise has a *non-principal cut-formula* ((Realign$_{_}$), (Abstr$_{_}$), (Bind$^1_{_}$), (Bind$^2_{_}$) of Figures 7, and 8) and (2) both have principal cut-formulae ((Subst$_{_}$), (Reindex$_{_}$), ($\beta_{_}$)). Then we prove (by standard argument) the innermost strategy for the propagation rules terminates using the sizes of cut-formulae and derivations for innermost cut occurrences. Details are available in [51]. □

Corollary 2 (Hauptsätze) *Every end sequent derived from GK3i$_{dB}$ + Struct + Cuts can be derived from GK3i$_{dB}$.*

The above results are presented from the proof theory viewpoint. Now we present them again from the computation viewpoint.

Definition 3 (Proof Terms) *Let Λ^G be the set of valid proof terms for $G \subseteq$ GK3i$_{dB}$ + Cuts + Struct.*

Definition 4 (Calculi of Proof Terms) *Let*

λ^{ns} *be the terms*
$\Lambda^{\mathrm{ns}} = \Lambda^{GK3i_{dB} + (\mathrm{Weak}) + (\mathrm{Reloc}) + (\mathrm{Cut_{ns}})}$ *with rewriting relation*
$\to_{\mathrm{ns}} = \to_w \cup \to_r \cup \to_{c_{ns}}$*, and*

λ^{cs} *be the terms*
$\Lambda^{\mathrm{cs}} = \Lambda^{GK3i_{dB} + (\mathrm{Weak})_{k,1} + (\mathrm{Cut_{cs}})}$ *with rewriting relation* $\to_{\mathrm{cs}} = \to_w \cup \to_{c_{cs}}$.

Observe that λ^{cs} does not need all of the possible instances of (Weak).

Theorem 5 *The calculi λ^{ns}, λ^{cs} enjoy SR and WIN. Their normal-form fragment is $\Lambda^{GK3i_{dB}}$ (which is "Cuts + Struct"-free).*

2.1 Strengthening the Results

We will now strengthen the already obtained results in a non-proof-theoretical direction using the following result [38].

Theorem 6 (O'Donnell) *For an orthogonal (left-linear and non-overlapping rules) TRS,* WIN $\Leftrightarrow$ SN.

The calculi λ^{ns} and λ^{cs} qualify as first-order TRS's. Unfortunately, they contain critical pairs, specifically, (Bind$^1_{\mathrm{X}}$) with any of (Realign$_{\mathrm{X}}$), (Abstr$_{\mathrm{X}}$), (Bind$^2_{\mathrm{X}}$), and (Subst$_{\mathrm{X}}$) of Figures 7 and 8, where X ranges over ns and cs. Correspondingly, it is well known in the proof theory community that "logical deduction proofs do not have unique normal forms". However, this is easy to fix using the rules (Bind$^3_{\mathrm{ns}}$) and (Bind$^3_{\mathrm{cs}}$) presented in Figure 9 which are restrictions of (Bind$^1_{\mathrm{ns}}$) and (Bind$^1_{\mathrm{cs}}$). Let $\lambda^{\mathrm{ns}}_{-}$ and $\lambda^{\mathrm{cs}}_{-}$ be the result of replacing (Bind$^1_{\mathrm{ns}}$) and (Bind$^1_{\mathrm{cs}}$) in λ^{ns} and λ^{cs}

$$
\begin{array}{lll}
\mathcal{W}_k^m(n) & \to_w & \begin{cases} n & \text{if } n \leq k \\ n+m & \text{if } n > k \end{cases} \\
\mathcal{W}_k^m(\lambda e) & \to_w & \lambda \mathcal{W}_{k+1}^m(e) \\
\mathcal{W}_k^m(\text{bind}_1\ (n\, e)\ \text{in}\ e') & \to_w & \begin{cases} \text{bind}_1\ (n\, \mathcal{W}_k^m(e))\ \text{in}\ \mathcal{W}_{k+1}^m(e') & \text{if } n \leq k \\ \text{bind}_1\ ((n+m)\, \mathcal{W}_k^m(e))\ \text{in}\ \mathcal{W}_{k+1}^m(e') & \text{if } n > k \end{cases}
\end{array}
$$

Figure 5: Rewrite rules for $\mathcal{W}_k^m(\,\cdot\,)$ (propagation of (Weak)).

$$
\begin{array}{lll}
\mathcal{R}_l^k(n) & \to_r & \begin{cases} n & \text{if } n < l \vee n > k \\ l & \text{if } n = k \\ n+1 & \text{if } l \leq n < k \end{cases} \\
\mathcal{R}_l^k((\lambda e)) & \to_r & \lambda \mathcal{R}_{l+1}^{k+1}(e) \\
\mathcal{R}_l^k((\text{bind}_1\ (n\, e)\ \text{in}\ e')) & \to_r & \begin{cases} \text{bind}_1\ (n\, (\mathcal{R}_l^k(e)))\ \text{in}\ \mathcal{R}_{l+1}^{k+1}(e') & \text{if } n < l \vee n > k \\ \text{bind}_1\ (l\, (\mathcal{R}_l^k(e)))\ \text{in}\ \mathcal{R}_{l+1}^{k+1}(e') & \text{if } n = k \\ \text{bind}_1\ ((n+1)\, (\mathcal{R}_l^k(e)))\ \text{in}\ \mathcal{R}_{l+1}^{k+1}(e') & \text{if } l \leq n < k \end{cases}
\end{array}
$$

Figure 6: Rewrite rules for $\mathcal{R}_l^k(\,\cdot\,)$ (propagation of (Reloc)).

respectively by ($\text{Bind}^3_{\text{ns}}$) and ($\text{Bind}^3_{\text{cs}}$). These new calculi retain the nice properties proven for λ^{ns} and λ^{cs} but are also confluent and have unique normal forms. Furthermore, cut elimination terminates regardless of the cut propagation strategy.

Theorem 7 *λ^{ns}_{-} and λ^{cs}_{-} are orthogonal TRS's and have the SR, SN, and CR properties. Their normal-form fragment is $\Lambda^{GK3i_{dB}}$.*

Proof Sketch: Orthogonality is easily checked, although it requires a syntactic translation into the TRS framework which uses a countably infinite number of symbols and which translates each rule of λ^{ns}_{-} and λ^{cs}_{-} into a countably infinite number of TRS rules. CR is obtained by orthogonality of the rules while the specific details for obtaining SN via Theorem 6 are given in [51]. □

General SN results for cut-elimination procedures obtained through general TRS results have also been studied by Tahhan Bittar [5] and Selhab [48]. Dragalin [18] presents a strong cut-elimination procedure in its own right (that is, without using TRS results). A closer comparison with these results, although interesting, falls outside the scope of the present paper.

3 Relations to Known Calculi

This section compares the explicit substitution machinery of λ^{ns} and λ^{cs} with similarly machinery previously developed by others, the λs and λt calculi of Kamareddine and Ríos. The key to this comparison is the following observation. Instead of using an explicit binding construct for ($\to$L), we could have relied on ordinary λ-binding and application syntax. Hence, instead of writing: "$\text{bind}_1\ (n\, e)$ in e'", we could have written: "$(\lambda e')\ (n\, e)$".

3.1 Translating to λs and λt.

The λs and λt calculi, defined in Figure 10, are explicit substitution versions of the untyped λ-calculus with de Bruijn indices developed by Kamareddine and Ríos [30, 32]. Both the λs and λt calculi are confluent on closed terms,[3] preserve strong normalisation (PSN) w.r.t. the λ-calculus, and have an SN simply-typed version. In addition, for λs there is a confluent extension to open terms [31] which, unfortunately, is not PSN w.r.t. the λ-calculus [23] but is WN the simply-typed case. The λt-calculus is slightly closer than λs to the style of λv and $\lambda\sigma$ in that it does index updating in smaller increments.

[3] The closed terms are the usual terms.

$$
\begin{array}{rcll}
n\{\!\{k := e\}\!\} & \to_{c_{ns}} & \begin{cases} n & \text{if } n < k \\ n-1 & \text{if } n > k \end{cases} & (\text{Realign}_{\text{ns}}) \\
(\lambda e')\{\!\{k := e\}\!\} & \to_{c_{ns}} & \lambda(e'\{\!\{k+1 := e\}\!\}) & (\text{Abstr}_{\text{ns}}) \\
e\{\!\{k := (\text{bind}_1\ (n\, e_1) \text{ in } e_2)\}\!\} & \to_{c_{ns}} & \text{bind}_1\ ((n+k-1)\,(\mathcal{W}_0^{k-1}(e_1))) \text{ in } \mathcal{R}_1^k((\mathcal{W}_k^1(e))\{\!\{k := e_2\}\!\}) & (\text{Bind}^1_{\text{ns}}) \\
(\text{bind}_1\ (n\, e_1) \text{ in } e_2)\{\!\{k := e\}\!\} & \to_{c_{ns}} & \begin{cases} \text{bind}_1\ (n\, e_1\{\!\{k := e\}\!\}) \text{ in } (e_2\{\!\{k+1 := e\}\!\}) & \text{if } n < k \\ \text{bind}_1\ ((n-1)\, e_1\{\!\{k := e\}\!\}) \text{ in } (e_2\{\!\{k+1 := e\}\!\}) & \text{if } n > k \end{cases} & (\text{Bind}^2_{\text{ns}}) \\
k\{\!\{k := e\}\!\} & \to_{c_{ns}} & \mathcal{W}_0^{k-1}(e) & (\text{Subst}_{\text{ns}}) \\
(\text{bind}_1\ (k\, e_1) \text{ in } e_2)\{\!\{k := m\}\!\} & \to_{c_{ns}} & \text{bind}_1\ ((m+k-1)\, e_1\{\!\{k := m\}\!\}) \text{ in } (e_2\{\!\{k+1 := m\}\!\}) & (\text{Reindex}_{\text{ns}}) \\
(\text{bind}_1\ (k\, e_1) \text{ in } e_2)\{\!\{k := \lambda e\}\!\} & \to_{c_{ns}} & e_2\{\!\{k+1 := \lambda e\}\!\}\{\!\{1 := (\mathcal{W}_1^{k-1}(e))\{\!\{1 := e_1\{\!\{k := \lambda e\}\!\}\}\!\}\}\!\} & (\beta_{\text{ns}})
\end{array}
$$

Figure 7: Rewrite rules for $\cdot\{\!\{k := \cdot\}\!\}$ (propagation of (Cut_{ns})).

$$
\begin{array}{rcll}
n\llbracket k := e\rrbracket & \to_{c_{cs}} & \begin{cases} n & \text{if } n < k \\ n-1 & \text{if } n > k \end{cases} & (\text{Realign}_{\text{cs}}) \\
(\lambda e')\llbracket k := e\rrbracket & \to_{c_{cs}} & \lambda(e'\llbracket k+1 := \mathcal{W}_0^1(e)\rrbracket) & (\text{Abstr}_{\text{cs}}) \\
e\llbracket k := (\text{bind}_1\ (n\, e_1) \text{ in } e_2)\rrbracket & \to_{c_{cs}} & \text{bind}_1\ (n\, e_1) \text{ in } \mathcal{W}_0^1(e)\llbracket k+1 := e_2\rrbracket & (\text{Bind}^1_{\text{cs}}) \\
(\text{bind}_1\ (n\, e_1) \text{ in } e_2)\llbracket k := e\rrbracket & \to_{c_{cs}} & \begin{cases} \text{bind}_1\ (n\, e_1\llbracket k := e\rrbracket) \text{ in } (e_2\llbracket k+1 := \mathcal{W}_0^1(e)\rrbracket) & \text{if } n < k \\ \text{bind}_1\ ((n-1)\, e_1\llbracket k := e\rrbracket) \text{ in } (e_2\llbracket k+1 := \mathcal{W}_0^1(e)\rrbracket) & \text{if } n > k \end{cases} & (\text{Bind}^2_{\text{cs}}) \\
k\llbracket k := e\rrbracket & \to_{c_{cs}} & e & (\text{Subst}_{\text{cs}}) \\
(\text{bind}_1\ (k\, e_1) \text{ in } e_2)\llbracket k := m\rrbracket & \to_{c_{cs}} & \text{bind}_1\ (m\, e_1\llbracket k := m\rrbracket) \text{ in } (e_2\llbracket k+1 := m+1\rrbracket) & (\text{Reindex}_{\text{cs}}) \\
(\text{bind}_1\ (k\, e_1) \text{ in } e_2)\llbracket k := \lambda e\rrbracket & \to_{c_{cs}} & e_2\llbracket k+1 := \mathcal{W}_0^1(\lambda e)\rrbracket\llbracket 1 := e\llbracket 1 := e_1\llbracket k := \lambda e\rrbracket\rrbracket\rrbracket & (\beta_{\text{cs}})
\end{array}
$$

Figure 8: Rewrite rules for $\cdot\llbracket k := \cdot\rrbracket$ (propagation of (Cut_{cs})).

Both λs and λt are untyped calculi while λ^{ns} and λ^{cs}, as presented, are simply-typed. However, we will be treating the type-free versions of λ^{ns} and λ^{cs} in the following. We define the following translations from the terms of λ^{ns} and λ^{cs} into the terms of λs and λt, respectively.

Definition 8 $\mathcal{S} : \Lambda^{GK3i_{dB}+(\text{Cut}_{\text{ns}})+(\text{Weak})} \longrightarrow \Lambda s$ *is given as follows*

$$
\begin{array}{rcl}
\mathcal{S}(n) & = & n \\
\mathcal{S}(\lambda e) & = & \lambda \mathcal{S}(e) \\
\mathcal{S}(\text{bind}_1\ (n\, e) \text{ in } e') & = & (\lambda \mathcal{S}(e'))\,(n\, \mathcal{S}(e)) \\
\mathcal{S}(e\{\!\{k := e'\}\!\}) & = & \mathcal{S}(e)\sigma^k \mathcal{S}(e') \\
\mathcal{S}(\mathcal{W}_k^m(e)) & = & \varphi_k^{m+1}\mathcal{S}(e)
\end{array}
$$

Definition 9 $\mathcal{T} : \Lambda^{\text{cs}} \longrightarrow \Lambda t$ *is given as follows*

$$
\begin{array}{rcl}
\mathcal{T}(n) & = & n \\
\mathcal{T}(\lambda e) & = & \lambda \mathcal{T}(e) \\
\mathcal{T}(\text{bind}_1\ (n\, e) \text{ in } e') & = & (\lambda \mathcal{T}(e'))\,(n\, \mathcal{T}(e)) \\
\mathcal{T}(e\llbracket k := e'\rrbracket) & = & \mathcal{T}(e)\varsigma^k \mathcal{T}(e') \\
\mathcal{T}(\mathcal{W}_k^1(e)) & = & \theta_k \mathcal{T}(e)
\end{array}
$$

Observe that $\mathcal{S}$ does not translate λ^{ns} terms of the form $\mathcal{R}_l^k(e)$. The λs calculus does not have a construct with the correct index-updating behavior to be the target of the translation. The calculi of explicit substitution which we considered as targets for the translation of λ^{ns} and λ^{cs} do not support either the simple exchange rule or the slightly more powerful relocation rule (equivalent to multiple uses of exchange) that appears as (Reloc) in λ^{ns}. Thus, the statements that follow will say less about the relationship between λ^{ns} and λs than they will about the relationship between λ^{cs} and λt. As can be seen from Figure 7, uses of the (Reloc) rule are only introduced in one case. The need for (Reloc) in that case could be avoided altogether by using a more general rule than ($\to$L) which allowed any formula in the right premise to be discharged, rather than just the first. However, this would merely be moving the difficulty, as such a more general rule also does not seem to have a translation.

We now define restrictions and slight modifications of λs and λt to operate on the images of the terms of λ^{ns} and λ^{cs} under $\mathcal{S}$ and $\mathcal{T}$. Let $\Lambda s^- = \mathcal{S}(\Lambda^{\text{ns}})$ and $\Lambda t^- = \mathcal{T}(\Lambda^{\text{cs}})$. Define $a \to_{\lambda s^-} a'$ whenever for $a \in \Lambda s^-$ there is some a'' such that $a \to_{\lambda s} a''$ and $a'' \to^*_{\lambda s} a'$ is the shortest rewrite sequence such that $a' \in \Lambda s^-$. Define $b \to_{\lambda t^-} b'$ similarly. The restricted cal-

$$
\begin{array}{lll}
e\{\!\{k := (\mathrm{bind}_1\ (n\, e_1)\ \mathrm{in}\ e_2)\}\!\} & \rightarrow_{c_{ns}} & \mathrm{bind}_1\ ((n+k-1)\,(\mathcal{W}_0^{k-1}(e_1)))\ \mathrm{in}\ \mathcal{R}_1^k((\mathcal{W}_k^1(e))\{\!\{k := e_2\}\!\}) \quad (\mathrm{Bind}^3_{\mathrm{ns}}) \\
\quad \text{where } e = (\mathrm{bind}_1\ (k\, e_1')\ \mathrm{in}\ e_2') & & \\
e[\![k := (\mathrm{bind}_1\ (n\, e_1)\ \mathrm{in}\ e_2)]\!] & \rightarrow_{c_{cs}} & \mathrm{bind}_1\ (n\, e_1)\ \mathrm{in}\ ((\mathcal{W}_0^1(e))[\![k+1 := e_2]\!]) \quad (\mathrm{Bind}^3_{\mathrm{cs}}) \\
\quad \text{where } e = (\mathrm{bind}_1\ (k\, e_1')\ \mathrm{in}\ e_2') & &
\end{array}
$$

Figure 9: Replacements for $(\mathrm{Bind}^1_{\mathrm{ns}})$ and $(\mathrm{Bind}^1_{\mathrm{cs}})$ to remove critical pairs.

λs

$$
\begin{array}{rcl}
a \in \Lambda s & ::= & n \mid a\,a \mid \lambda a \mid a\sigma^i a \mid \varphi_k^i a \\
 & & \text{where } i, n \geq 1, k \geq 0 \\
(\lambda a_1)\, a_2 & \rightarrow_{\lambda s} & a_1 \sigma^1 a_2 \\
(\lambda a_1)\sigma^i a_2 & \rightarrow_{\lambda s} & \lambda(a_1 \sigma^{i+1} a_2) \\
(a_1\, a_2)\sigma^i a_3 & \rightarrow_{\lambda s} & (a_1 \sigma^i a_3)\,(a_2 \sigma^i a_3) \\
n \sigma^i a & \rightarrow_{\lambda s} & \begin{cases} n-1 & \text{if } n > i \\ \varphi_0^i a & \text{if } n = i \\ n & \text{if } n < i \end{cases} \\
\varphi_k^i n & \rightarrow_{\lambda s} & \begin{cases} n & \text{if } n \leq k \\ n+i-1 & \text{if } n > k \end{cases} \\
\varphi_k^i(\lambda a) & \rightarrow_{\lambda s} & \lambda(\varphi_{k+1}^i a) \\
\varphi_k^i(a_1\, a_2) & \rightarrow_{\lambda s} & (\varphi_k^i a_1)\,(\varphi_k^i a_2)
\end{array}
$$

λt

$$
\begin{array}{rcl}
b \in \Lambda t & ::= & n \mid b\,b \mid \lambda b \mid b\varsigma^i b \mid \theta_k b \\
 & & \text{where } i, n \geq 1, k \geq 0 \\
(\lambda b_1)\, b_2 & \rightarrow_{\lambda t} & b_1 \varsigma^1 b_2 \\
(\lambda b_1)\varsigma^i b_2 & \rightarrow_{\lambda t} & \lambda(b_1 \varsigma^{i+1} (\theta_0 b_2)) \\
(b_1\, b_2)\varsigma^i b_3 & \rightarrow_{\lambda t} & (b_1 \varsigma^i b_3)\,(b_2 \varsigma^i b_3) \\
n \varsigma^i b & \rightarrow_{\lambda t} & \begin{cases} n-1 & \text{if } n > i \\ b & \text{if } n = i \\ n & \text{if } n < i \end{cases} \\
\theta_k n & \rightarrow_{\lambda t} & \begin{cases} n & \text{if } n \leq k \\ n+1 & \text{if } n > k \end{cases} \\
\theta_k(\lambda b) & \rightarrow_{\lambda t} & \lambda(\theta_{k+1} b) \\
\theta_k(b_1\, b_2) & \rightarrow_{\lambda t} & (\theta_k b_1)\,(\theta_k b_2)
\end{array}
$$

Figure 10: The λs and λt calculi.

culi λs^- and λt^- are then $\lambda s^- = (\Lambda s^-, \rightarrow_{\lambda s^-})$ and $\lambda t^- = (\Lambda t^-, \rightarrow_{\lambda t^-})$. Thus, we have restricted all applications to be of the form $(\lambda\cdot)\,(n\,\cdot)$ and we treat such constructs as atomic with respect to the rewrite rules. Let $\rightarrow_{\varphi^-}$ be the rewrite relation using only rules of λs^- pertaining to the φ operator and let $\rightarrow_{\theta^-}$ similarly use only rules of λt^- pertaining to the θ operator.

Proposition 10 *For* $e, e' \in \Lambda^{GK3i_{dB}+(\mathrm{Cut_{ns}})+(\mathrm{Weak})}$ *we have*

$$e \rightarrow_w e' \quad \Leftrightarrow \quad \mathcal{S}(e) \rightarrow_{\varphi^-} \mathcal{S}(e')$$

Proof: Straightforward inspection. □

That is, the index-updating mechanisms of λs and λ^{ns} coincide, except for the lack of (Reloc).

Proposition 11 *For* $e, e' \in \Lambda^{\mathrm{cs}}$ *we have*

$$e \rightarrow_w e' \quad \Leftrightarrow \quad \mathcal{T}(e) \rightarrow_{\theta^-} \mathcal{T}(e')$$

Although the above propositions show the index-updating mechanisms correspond, the rules (Bind^1_{-}) of Figure 7 and 8 (and any specialisation of them) are neither supported by λs nor by λt (but see Section 3.2 below). Hence we are led to consider the calculi without these rules. Define the restricted calculus $\lambda^{\mathrm{ns}-}$ to be λ^{ns} without $(\mathrm{Bind}^1_{\mathrm{ns}})$ and the restricted calculus $\lambda^{\mathrm{cs}-}$ to be λ^{cs} without $(\mathrm{Bind}^1_{\mathrm{cs}})$.

Theorem 12 λs *simulates* $\lambda^{\mathrm{ns}-}$*, i.e., for* $e, e' \in \Lambda^{GK3i_{dB}+(\mathrm{Cut_{ns}})+(\mathrm{Weak})}$*:*

$$e \rightarrow_{\mathrm{ns}-} e' \quad \Rightarrow \quad \mathcal{S}(e) \rightarrow_{\lambda s^-} \mathcal{S}(e')$$

Theorem 13 λt *simulates* $\lambda^{\mathrm{cs}-}$*, i.e., for* $e, e' \in \Lambda^{\mathrm{cs}}$*:*

$$e \rightarrow_{\mathrm{cs}-} e' \quad \Rightarrow \quad \mathcal{T}(e) \rightarrow_{\lambda t^-} \mathcal{T}(e')$$

Proofs are available in [51].

3.2 Relating Full Cut Elimination

So we have seen that most of the syntax and rules of λ^{ns} map onto syntax and rules of λs and all of the syntax and most of the rules of λ^{cs} map onto syntax and rules of λt. However, this leaves out the rewrite rule (Bind^1_{-}).

Translating this rule into λt as suggested by $\mathcal{T}$ we obtain:

$$e\varsigma^k((\lambda e_2)\,(n\,e_1)) \quad\to\quad (\lambda(\theta_0 e)\varsigma^{k+1}e_2)\,(n\,e_1)$$

The essential core of this rule in λ-calculus notation is

$$(\lambda x.M)((\lambda y.N)(z\,P)) \to (\lambda y.(\lambda x.M)N)(z\,P)$$

Note that this rewrite rule appears (subject to variations of syntax and various restrictions) in a wide variety of places in the literature. (See [33] for a discussion of some of these occurrences.) This suggests that we should consider adding the rule to λt to obtain a better correspondence. It remains to verify the properties of the modified λt.

Alternatively, we could map the bind-construct to λs and λt via a meta-level substitution. This would lead to "λs and λt as homomorphic images of cut propagation" thus showing that the Zucker-Pottinger epimorphism [54, 41] factors through calculi with explicit substitution [52].

We are presently working on both these issues where especially the latter is found to be interesting from both the perspectives of proof theory and explicit substitution calculi.

3.3 The Format of Contexts (Reloc); $\lambda bxgc$

The two cut rules we have considered are also admissible when we consider contexts as multisets of formulae instead of as lists of formulae. We can, unfortunately, not present the formal results relating to these systems here due to space constraints. We would like to say the following, though: (i) by taking contexts as multisets of assumptions we naturally arrive at calculi of proof terms with variable names — the contexts are represented as sets of "assumption classes" identified by variable names, (ii) both cut-elimination calculi *à la* de Bruijn collapse to the same such calculus with variable names, which (iii) is formally related to Rose's $\lambda bxgc$ exactly as stated in Theorems 12 & 13. Furthermore, the garbage collection rule of $\lambda bxgc$ corresponds to the propagation of a cut on a newly weakened assumption, cf. [51] in the case of garbage collection rules for λs and λt.

4 Conclusion and Future Work

We have studied Curry-Howard Isomorphisms between L-systems and λ-like calculi with explicit substitutions. We constructed such isomorphisms by inducing rewrite rules on calculi of proof terms from admissibility proofs of cut rules based on cut propagation. Many aspects of these calculi are interesting in their own right, but we have focused on cuts-as-explicit-substitutions.

As a first, we have shown an interesting correspondence between the dynamics of the cut and structural rules of L-systems on one hand and known explicit substitution calculi *à la* de Bruijn on the other. Also, we have also used known results on first-order TRS's [38] in a correct way to obtain strong cut elimination results.

In carrying this work out, some issues have arisen which we have left to future work. One issue is the study of cut elimination in L-systems with higher-order type constructors. Typed calculi with explicit substitutions for such constructs have proven difficult in N-system settings. Another issue is to further investigate the correspondence between the proof theory of L-systems and N-systems based on calculi with explicit substitutions. And yet another issue is further study of the use of the proof theory of L-systems to interrelate known calculi with explicit substitutions.

4.1 Acknowledgements

The present work owes a lot to the kind attention of Fairouz Kamareddine. Thanks are also due to Roy Dyckhoff and various anonymous referees for comments on earlier versions of this paper. For invitations to talk about the material presented here, thanks are due to Roy Dyckhoff, the ML-club at the University of Edinburgh, and the Logik Oberseminar run by Helmut Schwichtenberg at the LMU, Munich.

References

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

[2] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[3] Zine-El-Abidine Benaissa, Daniel Briaud, Pierre Lescanne, and Jocelyne Rouyer-Degli. $\lambda\nu$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, September 1996.

[4] Nick Benton, Gavin Bierman, Valeria de Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 75–90, Utrecht, The Netherlands, March 1993.

[5] Elias Tahhan Bittar. Strong normalisation proofs for cut elimination in Gentzen's sequent calculi. Prépublication du Laboratoire de Logique Algorithmique et Informatique de Clermont, Numéro 56, Université D'Auvergne, October 1996.

[6] Roel Bloo. *Preservation of Termination for Explicit Substitution*. PhD thesis, Technical University of Eindhoven, 1997.

[7] Roel Bloo and Kristoffer Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *CSN-95: Computer Science in the Netherlands*, November 1995.

[8] Val Breazu-Tannen, Delia Kesner, and Laurence Puel. A typed pattern calculus. In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 262–274, 1993.

[9] N.G. de Bruijn. A namefree lambda calculus with facilities for internal definition of expressions and segments. Technical Report TH-Report 78-WSK-03, Department of Mathematics, Technical University of Eindhoven, 1978.

[10] N.G. de Bruijn. Lambda calculus notation with namefree formulas involving symbols that represent reference transforming mappings. *Indag. Math.*, 40(3):348–356, 1979.

[11] Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. In *Proc. 14th Ann. IEEE Symp. Logic in Computer Sci.*, July 1999.

[12] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[13] Roberto Di Cosmo and Delia Kesner. Strong normalization of explicit substitutions via cut elimination in proof nets. In *12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, 1997.

[14] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Progress in Theoretical Computer Science. Birkhaüser, 1993.

[15] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, March 1996.

[16] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[17] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. In *Symposium on Logic in Computer Science (LICS '95)*, pages 366–507, Los Alamitos, Ca., USA, June 1995. IEEE Computer Society Press.

[18] A. G. Dragalin. *Mathematical Intuitionism: Introduction to Proof Theory*, volume 67 of *Translations of Mathematical Monographs*. American Mathematical Society, Providence, Rhode Island, 1988.

[19] Roy Dyckhoff and Luis Pinto. Permutability of proofs in intuitionistic sequent calculi. Draft, November 1996.

[20] Jean Gallier. Constructive logics, Part I: A tutorial on proof systems and typed λ-calculi. *Theoretical Computer Science*, 110:249–339, 1993. Elsevier Science Publishers.

[21] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.

[22] Bernhard Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticae*, September 1995. Preliminary version appeared as Relating Innermost, Weak, Uniform and Modular Termination of Term Rewriting Systems, LPAR'92.

[23] Bruno Guillome. The λs_e does not have PSN. Manuscript, November 1997.

[24] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional back-ends within the lambda-sigma calculus. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 25–33, 24–26 May 1996.

[25] H. Herbelin. A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Annual Conference of the European Association for Computer Science Logic, CSL'94, Kazimierz (Poland), Selected Papers*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 1995.

[26] Hugo Herbelin. *Séquents qu'on calcule*. PhD thesis, Université Paris VII, January 1995.

[27] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 470–490. Academic Press, 1980.

[28] César Muñoz Hurtado. Confluence and preservation of strong normalisation in an explicit substitutions calculus. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 440–447, 1996.

[29] César Muñoz Hurtado. Proof representation in type theory: State of the art. In *XXII Latin-American Conference of Informatics CLEI Panel*, 1996.

[30] Fairouz Kamareddine and Alejandro Ríos. A λ-calculus à la de Bruijn with explicit substitutions. In *Programming Languages, Implementations, Logics, and Programs*, volume 982, pages 45–62, 1995.

[31] Fairouz Kamareddine and Alejandro Ríos. Extending a λ-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *Journal of Functional Programming*, 7(4):395–420, July 1997.

[32] Fairouz Kamareddine and Alejandro Ríos. Bridging de Bruijn indices and variable names in explicit substitutions calculi. Submitted, February 1998.

[33] Assaf J. Kfoury and J. B. Wells. Addendum to "New notions of reduction and non-semantic proofs of β-strong normalization in typed λ-calculi". Tech. Rep. 95-007, Comp. Sci. Dept., Boston Univ., 1995.

[34] Stephen C. Kleene. *Introduction to Metamathematics*. North-Holland Publishing Co., 1952.

[35] Lena Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.

[36] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. Technical Report 5-95, Fakultät für Informatik, University of Karlsruhe, 1995.

[37] Gregory Mints. Normal forms for sequent derivations. In Piergiorgio Odifreddi, editor, *Kreiseliana; about and around Georg Kreisel*, pages 469–492. Wellesley, Massachusetts, A.K. Peters, 1996.

[38] Michael O'Donnell. *Computing in Systems Described by Equations*. Number 58 in Lecture Notes in Computer Science. Springer-Verlag, 1977.

[39] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[40] Frank Pfenning. A proof of the church-rosser theorem and its representation in a logical framework. Technical Report CMU-CS-92-186, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 1992. To appear in Journal of Automated Reasoning.

[41] Garrel Pottinger. Normalization as a homomorphic image of cut-elimination. *Annals of Mathematical Logic*, 12:323–357, 1977. North-Holland Publishing Company.

[42] Dag Prawitz. *Natural Deduction*. Almquist and Wiksell, Uppsala, 1965.

[43] Alejandro Ríos. *Contribution à l'étude des λ-calculs avec substitutions explicites*. PhD thesis, Université de Paris 7, 1993.

[44] E. Ritter, D. Pym, and L. A. Wallen. On the intuitionistic force of classical search. In P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors, *Proc of the Fifth Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, volume 1071 of *Lecture Notes in Artificial Intelligence*, pages 295–311, Berlin, Heidelberg, New York, 1996. Springer Verlag.

[45] Kristoffer Rose. Explicit substitutions – tutorial & survey. Technical Report LS-96-13, BRICS, Aarhus University, Denmark, September 1996.

[46] Kristoffer Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, University of Copenhagen, 1996.

[47] Helmut Schwichtenberg. Termination of permutative conversions in intuitionistic Gentzen calculi. Available on the authors homepage: http://www.mathematik.uni-muenchen.de/~schwicht, April 1997.

[48] Sohame Selhab. *Logiques et Réécriture*. PhD thesis, Université Henri Poincaré — Nancy 1 (Spécialité Informatique), Nancy, France, May 1998.

[49] Anne Sjerp Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1996.

[50] René Vestergaard. A computational anomaly in the Troelstra-Schwichtenberg G3i(m) system. Available from the author's homepage: http://www.cee.hw.ac.uk/~jrvest/, October 1998.

[51] René Vestergaard. The cut rule and explicit substitutions (author's cut). Available from the author's homepage: http://www.cee.hw.ac.uk/~jrvest/. Also available as TR-1998-9, Dept. of CS, Uni. of Glasgow, April 1998.

[52] René Vestergaard. The correspondence between cut-propagation, β-reduction, and explicit substitutions. Forthcoming, March 1999.

[53] Philip Wadler. A Curry-Howard isomorphism for sequent calculus. Manuscript, December 1993.

[54] J. Zucker. The correspondence between cut-elimination and normalization, Part I & II. *Annals of Mathematical Logic*, 7:1–112,113–155, 1974. North-Holland Publishing Company.

Handling Substitutions Explicitely in the π–Calculus

Daniel Hirschkoff

dh@cermics.enpc.fr, CERMICS - ENPC/INRIA and Université Paris 6

Abstract. We present two calculi aimed at describing the mechanism of name manipulation in the π–calculus. Following the ideas underlying the introduction of the λs and $\lambda\sigma$ calculi, we define πs and $\pi\sigma$. The former calculus includes operators on De Bruijn indices, and is first introduced to give an intuitive description of the De Bruijn representation of π–calculus terms. The latter is a calculus of explicit substitutions, where the computations pertaining to name manipulation are described by a Term Rewrite System. We introduce the two corresponding notions of bisimulation, and show that they can be put in correspondence; in doing this, we establish a relation with what could be considered "usual" (early) bisimulation on π–calculus terms in De Bruijn notation. These results shed light on the mechanism of name–passing, which can be of interest both for the implementation and the formal treatment (e.g. in a logical framework) of related calculi.

Introduction

Calculi of explicit substitutions originate in Categorical Combinatory Logic, and have been designed for many variants of λ–calculus and related formalisms (e.g. type systems, natural deduction, or higher–order logics). Besides the treatment of (functional) sequential computation, such an approach has also been recently adopted to study the object–oriented paradigm [KML98,LLL99]. Our goal in this paper is to provide a similar account of *concurrency*, through the analysis of one of his most popular algebraic models, namely π–calculus. We concentrate on the mechanism of name–passing, which lies at the heart of the expressiveness of this formalism, and analyse it by describing its effect on π–calculus terms. In this attempt, our work differs considerably from [FMQ96], where a calculus of explicit substitutions (called $\pi\xi$) is introduced by focusing on the *environment* of a π–calculus process during its execution, rather than on the effect of the computation on the term. We shall return on this point later on.

In *name–passing calculi*, the means of communications (typically *channels*, or *names*) can be used as the object of the communication; within such an approach, one can describe systems whose topology is changing along the computation, which can be source of great expressiveness. In the π–calculus, the only information being exchanged between processes is names; this is however enough to encode many paradigms of computer science, essentially through the mechanism of *name extrusion*. Name extrusion can be illustrated on the following example: consider the π–calculus transition

$$a(x).P \mid (\nu c)\,\overline{a}c.Q \;\xrightarrow{\tau}\; (\nu c)\,(P_{\{x:=c\}} \mid Q)\,.$$

The first subterm is liable to *receive* a name x along some channel a, and then to behave like some process P: this term is written $a(x).P$. In the latter expression, P depends in general on x (in the same way a λ–term M depends on x in $\lambda x.\,M$). In parallel with $a(x).P$, we find a process willing to *send* some value (i.e. name) c on a, and then to proceed according to Q. The information being sent on a is not known to the receiver before the communication: this is represented using the restriction operator ν, that makes the usage of c private to $\overline{a}c.Q$. We see *en passant* that the π–calculus has two binders, namely *abstraction* (embedded in the receiving process, and sometimes written λ) and *restriction* (as we shall see, this binder is not directly related to the notion of substitution). As both processes synchronise on a, they perform a τ–transition, which has two effects: in P, the formal parameter x is instanciated by the value being received, namely c; at the same time, the name c is now known by both actors of the synchronisation, hence the scope of the restriction on c has grown: we observe the extrusion of c.

Let us now move to the framework of De Bruijn indices [dB72], and see how these mechanisms are expressed. Recall that we are working with two binders, λ and ν ($a(x).P$ and $(\nu c)\,\overline{a}c.Q$ should thus roughly look like "$a\lambda.P$" and "$\nu\overline{a}0.Q$" respectively – notice that we adopt the convention that De Bruijn indices start at 0). From the emitter's point of view, no update on De Bruijn indices needs to be performed, since the execution environment has not been modified; quite remarkably, this holds also for the receiver, because in P the name bound by the abstraction on x becomes bound by the restriction on c, which means that nothing changes in the De Bruijn representation of P. So the situation seems simpler than in λ–calculus, where β–reduction involves in general some updates in the free names of the term. We recover anyway such a phenomenon in the case of the communication of a free name (i.e. a name not bound by restriction), where a "true" instanciation takes place. Note however that in the π–calculus, we always substitute a name for a name, while a whole term can be provided in the λ–calculus. We therefore need to be able to perform at least two basic manipulations on names, that is replacement of a name with another name, and *name lifting*, to preserve the "meaning" of De Bruijn indices as we cross a binder; actually, in the π–calculus, we need an extra primitive manipulation, which is related to the behaviour of the restriction operator ν. Indeed, in the transition

$$T \stackrel{def}{=} (\nu b)(\nu c)\,\overline{a}c.Q \;\xrightarrow{(\nu c)\,\overline{a}[c]}\; (\nu b)\,Q\,,$$

when process T performs the emission of private name c along a, it "sends" the restriction on c, while the restriction on b remains above Q. As the restriction on c somehow "crosses" the restriction on b, the internal De Bruijn representation of Q has to evolve in order to remain coherent, by exchanging indices 0 and 1. This phenomenon, which is closely related to the evolution of ν scopes, induces us to treat the ability of permuting two consecutive indices in a term as a primitive component of name manipulation.

In the paper, the syntax of π–calculus we shall work with is the following:

$$P \;=\; \mathbf{0} \mid \overline{a}b \mid a(b).P \mid (P_1|P_2) \mid (\nu x)\, P\,,$$

where a, b range over a basic sort of *names*. This is a monadic, sum free, finite, asynchronous calculus. Polyadicity will be treated in Sect. 4, while the other features are not relevant for our study (in particular, the absence of a continuation for emitting processes implies that the examples seen above cannot be directly treated in our calculus; this is anyway harmless, as – for our purposes only – a term of the form $\overline{a}[x].P$ can be replaced by $\overline{a}[x] \mid P$ without changing the managment of names). The inactive process, $\mathbf{0}$, is written in bold font to distinguish it from the De Bruijn's index 0. We present below the operational semantics of our calculus. Actions, ranged over with μ, can be of three kinds, namely receptions (written $a(b)$), emissions (written $\overline{a}(b)$ or $\overline{a}b$, depending whether the emitted name is private to the sender or not), and synchronisations (written τ). $n(\mu)$, $\mathrm{fn}(\mu)$ and $\mathrm{bn}(\mu)$ denote respectively the sets of names, free and bound names of an action μ ($\mathrm{bn}(\mu)$ is defined by saying that b is bound in $\overline{a}(b)$). Symmetrical versions of rules $\textsc{par}_l$, $\textsc{comm}_1$ and $\textsc{close}_1$ are omitted.

$$(\textsc{inp})\ a(b).P \xrightarrow{a(c)} P_{\{b:=c\}} \qquad (\textsc{out})\ \overline{a}b \xrightarrow{\overline{a}b} \mathbf{0}$$

$$(\textsc{open})\ \frac{P \xrightarrow{\overline{a}b} P'}{(\nu b)\, P \xrightarrow{\overline{a}(b)} P'}\ b \neq a \qquad (\textsc{res})\ \frac{P \xrightarrow{\mu} P'}{(\nu x)\, P \xrightarrow{\mu} (\nu x)\, P'}\ x \notin n(\mu)$$

$$(\textsc{comm}_1)\ \frac{P \xrightarrow{a(b)} P' \quad Q \xrightarrow{\overline{a}b} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad (\textsc{par}_l)\ \frac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P'|Q}\ \ \mathrm{bn}(\mu) \cap \mathrm{fn}(P) = \emptyset$$

$$(\textsc{close}_1)\ \frac{P \xrightarrow{a(b)} P' \quad Q \xrightarrow{\overline{a}(b)} Q'}{P|Q \xrightarrow{\tau} (\nu b)\, (P'|Q')}\ \ b \notin \mathrm{fn}(Q)$$

Following the approach of the λs–calculus [KR95], we introduce πs, the calculus of mobile processes written in the De Bruijn notation. W.r.t. λs, we have, in addition to the possibility of performing substitution and name lifting, an operator to exchange two consecutive indices, written ψ. In λs and $\lambda\sigma$, the process of β reduction is decomposed into a β–rule and some rules to compute the propagation of a substitution in the term; likewise, we introduce here the operational semantics via two relations, written $\xrightarrow{\mu}_s$ and $\leadsto_s$. $\xrightarrow{\mu}_s$ corresponds to the firing of a transition, that can generate one or several computations on names, described by $\leadsto_s$. As usual in the De Bruijn framework, some side conditions in the definition of the transition relation can be embedded in the implementation of $\xrightarrow{\mu}_s$, due to the representation of bound names. This presentation naturally gives rise to a notion of bisimilarity, written $\sim_s$, that incorporates both relations.

One can then refine the description of the calculus, by adopting the point of view of the $\lambda\sigma$–calculus, and define $\pi\sigma$. Within such an approach, we develop a calculus of processes and substitutions, where the counterpart of operator ψ, written $\updownarrow$, enriches the calculus σ of $\lambda\sigma$. A consequence of this presentation is that the corresponding relation $\rightsquigarrow_\sigma$ is defined as a "plain" Term Rewrite System (without control structure). One is then interested in relating both formalisms, and we shall see that $\pi\sigma$ can be considered as giving a finer account on the terms described by πs, without introducing "extra behaviours".

As said above, the main goal of this work is to focus on the technical details that are hidden in the traditional formal definition of π–calculus terms. Such an approach is in particular of interest for a theorem prover formalisation of π–calculus, where one may want to stay close to the "mathematical" definition rather than design a specific implementation of processes[1]. At larger scale, this attempt may also be useful for the analysis of a naming operator like ν, that is not specific to the π–calculus.

The plan of the paper is as follows. Sections 1 and 2 introduce the πs- and $\pi\sigma$-calculi respectively, together with the corresponding notions of bisimilarity. Section 3 is devoted to the comparison of both formalisms. We establish the correspondence between the two descriptions (Theorem 4), using in particular an up–to technique for bisimulation [San95]. In Sect. 4, we discuss variants of our approach as well as related work (including the $\pi\xi$ calculus of [FMQ96]), and conclude.

1 The Monadic πs–calculus

We present here a calculus of π–terms in the De Bruijn notation, and try to give an intuition of its design choices. While being related to [Amb91], where a translation from π–calculus terms with names into the De Bruijn setting is presented, this part of our work remains closer to the tradition of substitutions calculi (à la λs), by treating the operators on names as part of the calculus, and not at meta level. Such an approach naturally fits to the introduction of $\pi\sigma$ in the next section.

1.1 Syntax

Terms of the π–calculus are built upon a basic notion of *names*; in the De Bruijn notation, names (sometimes referred to as *channels*) are represented by natural numbers, interpreted as indices. The definition of the πs–calculus follows the approach of the λs calculus [KR95]; accordingly, the syntactic sorts of terms (and of names) include *operators*, that explicitely handle the manipulations to be made on De Bruijn indices.

Definition 1 (πs–calculus – syntax). *The terms of the πs–calculus are defined as follows:*

$$a = \mathbb{N} \quad \textit{name values} \qquad na = a \mid \boldsymbol{op}\, na \quad \textit{names}$$
$$\boldsymbol{op} = \langle na \rangle^i \mid \varphi^i \mid \psi^i\,, \quad i \in \mathbb{N} \quad \textit{operators}$$
$$P = \mathbf{0} \mid \overline{a}b \mid a\lambda.P \mid (P_1|P_2) \mid \nu P \mid \boldsymbol{op}\, P \quad \textit{processes}$$

[1] This work actually originates in [Hir97], where such an effort is conducted.

Let us comment on the above definitions. The notion of *name value* shall be useful in the definition of the semantics, and corresponds to the sort of names where no occurrence of operators is allowed. We find here two operators from λs, namely substitution and lifting (written φ), while operator ψ is specific to the π–calculus; we shall describe and justify the behaviour of these constructs below. Let us remark that we use here a form of overloading, as operators can be applied both to names and to processes. Note that we do not have a formal definition of *"π–calculus terms in the De Bruijn notation"*, and instead directly work within the πs–calculus; however, we shall more or less retrieve such a notion in Sect. 3, where we shall reason on πs processes without occurrences of the operators (i.e. terms that are in some way "fully evaluated" w.r.t. the calculus of operators).

1.2 Early Operational Semantics

Actions. We now turn to the presentation of the behaviour of πs–calculus terms, by defining an *early* operational semantics for processes. The choice of an early version of the semantics strongly influences our description of the operational semantics, since in such a framework one is compelled to know the shape of the environment where the continuation of a comitting term shall be executed. Accordingly, we are led to separate receptions into *bound* and *free* input actions, in a way that is symmetrical to what is usually done with output actions. We shall see in the next subsection that a synchronisation involving a name which is private to the sender does not change the context of the communicating agents: the sender still "sees" the restriction after the synchronisation, while from the point of view of the receiver, an abstraction has been replaced by a restriction, still yielding the same meaning for the De Bruijn indices. On the contrary, a communication of a free name involves a "true" substitution.

The syntax for actions is as follows (we describe the reception and the emission of a restricted name using a somewhat peculiar use of the restriction operator ν):

Definition 2 (Actions). Actions*, ranged over by μ, are defined as follows:*

$$\mu \;=\; a(b) \mid a(\nu) \mid \overline{a}b \mid \overline{a}(\nu) \mid \tau\,.$$

$\mathrm{bn}(\mu)$ *is the number of* bound names *of an action, and is defined by saying that* $\mathrm{bn}(\mu) = 1$ *for actions of the shape* $a(\nu)$ *and* $\overline{a}(\nu)$*,* $\mathrm{bn}(\mu) = 0$ *otherwise.*

Note that operators do not arise in the definition of actions: this has to do with the introduction of name values in Definition 1, and will be explained below.

Representation of the Transition Rules. The behaviour of πs terms is described by two relations; the transition relation $P \xrightarrow{\mu}_s P'$ (meaning that process P is liable to perform action μ and become process P') describes the communicating behaviour of terms, and $\rightsquigarrow_s$ corresponds to the calculus of operators, as they are introduced in the syntax. We now comment on the shape of the transition rules as they are implemented in the De Bruijn notation; these rules are summed up on Fig. 1. The computation of operators, as defined on Fig. 2, shall be justified along the explanations below.

The rules for prefixed processes are INP, INP_ν and OUT; they all involve a premise saying that the components of the action are name values. This means that the prefixes involved in a transition should be fully evaluated. Such a condition is indeed needed for the definition of both the operational semantics (where we have to match two symmetric actions in order to infer a synchronisation) and the behavioural equivalence (the notion of simulation also involves a matching between two actions, namely those performed by the processes to be compared). This requirement is reminiscent of the notion of *weak head normal form* for the λ–calculus, where the head of the term is compelled to be a variable[2].

The definition of rule OUT is straightforward. As said above, the reception of a restricted (new) name consists, from the receiver point of view, in replacing a λ by a ν; this is (partially) embedded in rule INP_ν, the ν being provided as rule CLOSE is applied (see below). Rule INP can be seen as a rule for σ–generation, the substitution of b for 0 in the receiving term being the result of the transition: it is reminiscent of the β–rule in the λs calculus.

Similarly, PAR and RES can be seen as the rules for φ and ψ generation respectively. They are both written using a form of abuse of notation, in order to shorten the presentation; the subscript $\text{bn}(\mu)$ in the occurrence of the operators is interpreted as follows: if $\text{bn}(\mu) = 0$, then the operator reduces to the identity (i.e. we apply no operator), while $\text{bn}(\mu) = 1$ implies that the operator should be taken as it is, without the subscript. Such a notation allows us to avoid describing the various instances of rules RES and PAR, for each kind of action. In order to describe the meaning of operators φ and ψ, we shall now suppose that $\text{bn}(\mu) = 1$. Rule PAR is used to infer a transition for a parallel composition in the case where one process is performing the action (here P) and the other one (Q) is watching. $\text{bn}(\mu) = 1$ means that when the synchronisation will take place, and extra restriction will be brought in between Q and its "environment"; therefore, in order to keep the same meaning for the indices of free variables in Q, we have to *lift* all indices by 1: this is achieved by operator φ (the superscript 0 corresponds to the depth where the modification is applied in the term – see below). This operator is thus akin to operator φ of the λs calculus; its behaviour is given on Fig. 2.

Similarly, rule RES describes the behaviour of a restricted process in the case where the topmost restriction is not involved in the transition. Here again, we make use of a notation to abbreviate our presentation: $\uparrow\mu$ stands for an action where all free names involved in the action are of the form $k+1$, $k \in \mathbb{N}$ or for the action τ, the action μ thus representing the same action where every $k+1$ is replaced by k (or τ again). Intuitively, whenever a process P is liable to perform action $\uparrow\mu$, then νP can perform the same action "hidden" by a restriction: we

[2] Moreover, this analogy can be related to the encodings of the lazy λ–calculus into the π–calculus [Mil92], where the strategy in "attacking" λ–terms by evaluation of their topmost redex is closely mirrored on the evolution of the encoding processes.

$$
\begin{array}{llll}
(\textsc{inp}) & \dfrac{a, b \text{ values}}{a\lambda.P \xrightarrow{a(b)} \langle b\rangle^0 P} & (\textsc{inp}_\nu) & \dfrac{a \text{ value}}{a\lambda.P \xrightarrow{a(\nu)} P} \\[2ex]
(\textsc{out}) & \dfrac{a, b \text{ values}}{\overline{a}b \xrightarrow{\overline{a}b} \mathbf{0}} & (\textsc{open}) & \dfrac{P \xrightarrow{\overline{a+1}\,0} P'}{\nu\, P \xrightarrow{\overline{a}(\nu)} P'} \\[2ex]
(\textsc{par}) & \dfrac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P' \mid \varphi^0_{\mathrm{bn}(\mu)} Q} & (\textsc{res}) & \dfrac{P \xrightarrow{\uparrow\mu} P'}{\nu\, P \xrightarrow{\mu} \nu\, \psi^0_{\mathrm{bn}(\mu)} P'} \\[2ex]
(\textsc{comm}) & \dfrac{P \xrightarrow{a(b)} P' \quad Q \xrightarrow{\overline{a}b} Q'}{P|Q \xrightarrow{\tau} P'|Q'} & (\textsc{close}) & \dfrac{P \xrightarrow{a(\nu)} P' \quad Q \xrightarrow{\overline{a}(\nu)} Q'}{P|Q \xrightarrow{\tau} \nu\,(P'|Q')}
\end{array}
$$

Fig. 1. πs–calculus Operational Semantics: relation $\xrightarrow{\mu}_s$

thus have to decrease the free variables of μ by one. Accordingly, the execution context of P' has to be modified, as the migrating bound name (recall that we suppose $\mathrm{bn}(\mu) = 1$) crosses the "immobile" restriction: we therefore need an operator ψ, to exchange indices 0 and 1 in P' (the evaluation rules of ψ are given on Fig. 2).

In the case where the topmost restriction corresponds to the name that is emitted (that thus has to be 0), rule OPEN applies (while RES does not, for the action is not of the shape $\uparrow\mu$): as before, the channel carrying the emitted name evolves from $a+1$ into a, while the restriction gets "picked up" by the emission. Symmetrically with respect to the $\textsc{inp}_\nu$ rule, process P' does not need to be modified, as its execution environment does not change (it already "knows" the restriction that takes part in the communication).

We are left with the rules for synchronisation. Since we work with an early semantics, their formulation is easy, because all the work has been done *before* the communication takes place. COMM and CLOSE are straightforwardly introduced to describe communication in the case of free and bound outputs respectively.

Figure 1 presents the rules we have just exposed; as said above, rules (INP), (RES), and (PAR) can be seen as generation rules for operators σ, ψ and φ respectively. One can remark that a transition involves at most one σ–generation and zero or several φ and ψ–generations. Relation $\rightsquigarrow_s$ is the smallest relation satisfying the rules of Fig. 2. Each operator is indexed by an integer representing the depth at which it is applied, and which gets incremented each time we cross a binding construct (λ or ν).

Example. Consider the transition

$$(\nu c)(\nu x)\,\overline{a}[x].P \mid a(y).Q \mid R \;\xrightarrow{\tau}\; (\nu x)\,(\,(\nu c)\,P \mid Q_{\{y:=x\}}\,) \mid R\,;$$

it becomes in πs:

$$\nu\nu\,\overline{k}[0].P \mid k(\lambda).Q \mid R \xrightarrow{\tau}_s \nu\,(\nu\,\psi^0 P \mid Q) \mid \varphi^0 R\,.$$

σ–destruction	$\langle b\rangle^i\, a \rightsquigarrow_s$	$\begin{cases} a & if\ a < i \\ b+i & if\ a = i \\ a-1 & if\ a > i \end{cases}$
σ–inp transition	$\langle b\rangle^i\, a\lambda.P \rightsquigarrow_s$	$((\langle b\rangle^i\, a)\lambda.(\langle b\rangle^{i+1}\, P)$
σ–res transition	$\langle b\rangle^i\, \nu P \rightsquigarrow_s$	$\nu\, (\langle b\rangle^{i+1} P)$
σ–out transition	$\langle b\rangle^i \overline{a}c \rightsquigarrow_s$	$\overline{\langle b\rangle^i a}(\langle b\rangle^i c)$
σ–— transition	$\langle b\rangle^i (P_1 \vert P_2) \rightsquigarrow_s$	$\langle b\rangle^i P_1 \mid \langle b\rangle^i P_2$
σ–$\mathbf{0}$ transition	$\langle b\rangle^i \mathbf{0} \rightsquigarrow_s$	$\mathbf{0}$
φ–destruction	$\varphi^i\, a \rightsquigarrow_s$	$\begin{cases} a & if\ a < i \\ a+1 & if\ a \geq i \end{cases}$
φ–inp transition	$\varphi^i\, a\lambda.P \rightsquigarrow_s$	$(\varphi^i\, a)\lambda.(\varphi^{i+1}\, P)$
φ–res transition	$\varphi^i\, \nu P \rightsquigarrow_s$	$\nu\, (\varphi^{i+1}\, P)$
φ–out transition	$\varphi^i\, \overline{a}b \rightsquigarrow_s$	$\overline{\varphi^i a}(\varphi^i b)$
φ–— transition	$\varphi^i\, (P_1 \vert P_2) \rightsquigarrow_s$	$\varphi^i\, P_1 \mid \varphi^i\, P_2$
φ–$\mathbf{0}$ transition	$\varphi^i\, \mathbf{0} \rightsquigarrow_s$	$\mathbf{0}$
ψ–destruction	$\psi^i\, a \rightsquigarrow_s$	$\begin{cases} a & if\ a < i \\ i+1 & if\ a = i \\ i & if\ a = i+1 \\ a & if\ a > i+1 \end{cases}$
ψ–inp transition	$\psi^i\, a\lambda.P \rightsquigarrow_s$	$(\psi^i a)\lambda.(\psi^{i+1} P)$
ψ–res transition	$\psi^i\, \nu P \rightsquigarrow_s$	$\nu\, (\psi^{i+1} P)$
ψ–out transition	$\psi^i\, \overline{a}b \rightsquigarrow_s$	$\overline{\psi^i a}(\psi^i b)$
ψ–— transition	$\psi^i\, (P_1 \vert P_2) \rightsquigarrow_s$	$\psi^i P_1 \mid \psi^i P_2$
ψ–$\mathbf{0}$ transition	$\psi^i \mathbf{0} \rightsquigarrow_s$	$\mathbf{0}$

Fig. 2. Monadic πs–calculus operators: relation $\rightsquigarrow_s$

Behavioural Equivalence. Given the definition of relations $\xrightarrow{\mu}_s$ and $\rightsquigarrow_s$, we write $\overset{*}{\rightsquigarrow}_s$ for the reflexive transitive closure of $\rightsquigarrow_s$, and introduce the following notion of equivalence:

Definition 3 **($\sim_s$).** *A relation $\mathcal{R}$ between terms of the πs–calculus is an s–bisimulation iff whenever $P\mathcal{R}Q$ and $P\overset{*}{\rightsquigarrow}_s\xrightarrow{\mu}_s\overset{*}{\rightsquigarrow}_s P'$, there exists Q' such that $Q\overset{*}{\rightsquigarrow}_s\xrightarrow{\mu}_s\overset{*}{\rightsquigarrow}_s Q'$ and $P'\mathcal{R}Q'$, and the symmetrical condition on transitions of Q. s–bisimilarity, written $\sim_s$, is the greatest s–bisimulation.*

Let us comment on this definition. An observable transition of a process corresponds here to the composition of $\overset{*}{\rightsquigarrow}_s$, $\xrightarrow{\mu}_s$, and $\overset{*}{\rightsquigarrow}_s$ again. One could wonder if we can get rid of one of the two $\overset{*}{\rightsquigarrow}_s$. The first one (before the μ transition) is needed to trigger all possible evolvings of P and Q, in order to respect the branching structure captured by the notion of bisimulation: indeed, as no $\xrightarrow{\mu}_s$ transition can occur under an operator, we do not want to miss some transitions because some prefixes are not "evaluated" in a process. One solution to get rid of the first $\overset{*}{\rightsquigarrow}_s$ could thus be to consider relations on terms where all topmost prefixes are evaluated (which would correspond to a notion of *weak head normal form*). Alternatively, one may want to avoid the $\overset{*}{\rightsquigarrow}_s$ *after* the μ step: this would amount to adopt a policy in writing our relations, so that terms involved in the relation are reached right after the μ transition. We have kept the definition above in order to allow any kind of definition for the relations between processes, and preserve symmetry; the properties we prove in Sect. 3 shall help in giving a more precise meaning to the considerations we have just made[3].

2 The Monadic $\pi\sigma$–calculus

We introduce here the $\pi\sigma$–calculus, which is a calculus of names, processes and substitutions. It provides a definition of the names–handling mechanism as a "plain" Term Rewrite System, while control constructs are used in πs to describe the manipulation of names using operators.

2.1 Syntax

The syntax of $\pi\sigma$–calculus terms is given below; we simultaneously define *names* (ranged over by a, b), *processes* (ranged over by P) and *substitutions* (ranged over by s). According to the presentation of $\lambda\sigma$, integers are not primitive anymore, and are instead represented using constants 0 and $\uparrow$:

Definition 4 ($\pi\sigma$–calculus terms).

$$\begin{array}{rll} a = & 0 \mid a[s] & \textit{names} \\ P = & \mathbf{0} \mid \overline{a}b \mid a\lambda.P \mid (P_1|P_2) \mid \nu P \mid P[s] & \textit{processes} \\ s = & id \mid a.s \mid \uparrow \mid \updownarrow \mid s \circ s & \textit{substitutions} \end{array}$$

With respect to the calculus of substitutions of [ACCL91], we remark that we have an extra constant substitution, written $\updownarrow$, which is the counterpart of operator ψ in πs[4]. As will be seen later on, this constant can be encoded in σ, the calculus of substitutions without $\updownarrow$. We decide to keep it for the seek of clarity, and because it corresponds to a primitive operation in the definition of the operational semantics. We shall use some conventions to allievate the notation of explicit substitutions:

Notations. (i) Substitutions composition involving $\uparrow$ and $\updownarrow$ will sometimes be noted without the $\circ$ symbol, e.g. $\uparrow\uparrow\updownarrow$.

(ii) De Bruijn representants are noted using underlined natural numbers, by writing $\underline{1}, \underline{2}, \ldots$. More generally, $\underline{k}$ stands for $0[\underbrace{\uparrow\ldots\uparrow}_{k \text{ times}}]$ (also written $0[\uparrow^k]$).

[3] Note as well that the choices we examined are reminiscent of the various notions of *weak* equivalences or preorders on π–calculus processes, where τ–transitions play the rôle of $\rightsquigarrow_s$ moves, i.e. in some way of "unobservable" actions.

[4] P. Lescanne pointed to us that a calculus introduced by De Bruijn in 1978 already had an operator to swap two consecutive indices, like our $\updownarrow$. The presentation of this calculus given in [BBLRD95] actually uses the same symbol.

2.2 Semantics

The transition relation associated to the terms of the $\pi\sigma$–calculus, written $\rightarrow_\sigma$, is directly adapted from $\rightarrow_s$. *Values* for names correspond to De Bruijn indices representants (written as underlined integers), and actions are introduced accordingly, exactly like in πs. $\rightarrow_\sigma$ is defined on Fig. 3.

$$
\begin{array}{llll}
(\textsc{inp}) & \dfrac{a, b \text{ values}}{a\lambda.P \xrightarrow{a(b)} P[b.id]} & (\textsc{inp}_\nu) & \dfrac{a \text{ value}}{a\lambda.P \xrightarrow{a(\nu)} P} \\[2ex]
(\textsc{out}) & \dfrac{a, b \text{ values}}{\overline{a}b \xrightarrow{\overline{a}b} \mathbf{0}} & (\textsc{close}) & \dfrac{P \xrightarrow{a(\nu)} P' \quad Q \xrightarrow{\overline{a}(\nu)} Q'}{P|Q \xrightarrow{\tau} \nu\,(P'|Q')} \\[2ex]
(\textsc{par}) & \dfrac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P' \mid \uparrow^{\mathrm{bn}(\mu)} Q} & (\textsc{res}) & \dfrac{P \xrightarrow{\uparrow\mu} P'}{\nu\, P \xrightarrow{\mu} \nu \updownarrow^{\mathrm{bn}(\mu)} P'} \\[2ex]
(\textsc{comm}) & \dfrac{P \xrightarrow{a(b)} P' \quad Q \xrightarrow{\overline{a}b} Q'}{P|Q \xrightarrow{\tau} P'|Q'} & (\textsc{open}) & \dfrac{P \xrightarrow{\overline{\uparrow a}0} P'}{\nu\, P \xrightarrow{\overline{a}(\nu)} P'}
\end{array}
$$

Fig. 3. $\pi\sigma$–calculus transition relation

The corresponding calculus of substitutions, given by relation $\rightsquigarrow_\sigma$, is defined on Fig. 4. The rewrite rules can be decomposed into three sets; a first set of rules, involving all the substitutions constructors except $\updownarrow$, comes from the calculus σ of [ACCL91]. Rules 0–$\updownarrow$ to $\uparrow$–$\uparrow$–$\updownarrow$–s deal with constant $\updownarrow$, while the rules at the bottom of the figure propagate substitutions inside terms.

Definition 5 **($\sim_\sigma$).** *σ–bisimulation and $\sim_\sigma$ are introduced as in Def. 3, where $\xrightarrow{\mu}_\sigma$ and $\rightsquigarrow_\sigma$ replace $\xrightarrow{\mu}_s$ and $\rightsquigarrow_s$ respectively.*

3 Properties of πs and $\pi\sigma$

We now turn to the comparison between πs and $\pi\sigma$, and the associated notions of bisimilarity. As will be seen, it turns out that these calculi basically describe the same behaviours, and differ only in the granularity of the description ($\pi\sigma$ being finer than πs). A third notion of processes, corresponding to π–calculus terms in De Bruijn notation, will arise along our study. For lack of space, we omit most proofs; they can be found in [Hir99].

3.1 Confluence Properties of the Substitutions Calculi

We first focus on the calculi of substitutions, given by relations $\rightsquigarrow_s$ and $\rightsquigarrow_\sigma$; we prove uniqueness of normal forms for relation $\rightsquigarrow_\sigma$, and then define a translation from πs to $\pi\sigma$ that will allow us to establish the same result for $\rightsquigarrow_s$.

0–id	$0[id] \rightsquigarrow_\sigma 0$	0–$\updownarrow$	$0[\updownarrow] \rightsquigarrow_\sigma \underline{1}$
0–cons	$0[a.s] \rightsquigarrow_\sigma a$	1–$\updownarrow$	$\underline{1}[\updownarrow] \rightsquigarrow_\sigma 0$
[]–[]	$a[s][t] \rightsquigarrow_\sigma a[s \circ t]$	0–$\updownarrow$–s	$0[\updownarrow \circ s] \rightsquigarrow_\sigma \underline{1}[s]$
id–s	$id \circ s \rightsquigarrow_\sigma s$	1–$\updownarrow$–s	$\underline{1}[\updownarrow \circ s] \rightsquigarrow_\sigma 0[s]$
$\uparrow$–id	$\uparrow \circ\, id \rightsquigarrow_\sigma \uparrow$	$\uparrow$–$\uparrow$–$\updownarrow$	$\uparrow \circ (\uparrow \circ \updownarrow) \rightsquigarrow_\sigma \uparrow \circ \uparrow$
$\uparrow$–cons	$\uparrow \circ (a.s) \rightsquigarrow_\sigma s$	$\uparrow$–$\uparrow$–$\updownarrow$–s	$\uparrow \circ (\uparrow \circ (\updownarrow \circ s)) \rightsquigarrow_\sigma \uparrow \circ (\uparrow \circ s)$
cons–$\circ$	$(a.s) \circ t \rightsquigarrow_\sigma a[t].(s \circ t)$		
$\circ$–$\circ$	$(s_1 \circ s_2) \circ s_3 \rightsquigarrow_\sigma s_1 \circ (s_2 \circ s_3)$		
s–inp	$(a.\lambda P)[s] \rightsquigarrow_\sigma a[s].\lambda(P[0.s \circ \uparrow])$	s–ν	$(\nu P)[s] \rightsquigarrow_\sigma \nu(P[s])$
s–out	$(\overline{a}b)s \rightsquigarrow_\sigma \overline{a[s]}b[s]$	s–par	$(P_1 \mid P_2)[s] \rightsquigarrow_\sigma P_1[s] \mid P_2[s]$
s–$\mathbf{0}$	$\mathbf{0}[s] \rightsquigarrow_\sigma \mathbf{0}$		

Fig. 4. Monadic $\pi\sigma$–calculus: calculus of substitutions

Proposition 1. *$\rightsquigarrow_\sigma$ is strongly normalising and locally confluent.*

Proof. (Sketch – SN part) The idea is to exploit the strong normalisation of σ, the subpart of $\lambda\sigma$ [ACCL91] that handles substitutions, by encoding the calculus of substitutions of $\pi\sigma$ into a $\updownarrow$–free calculus, and encoding the resulting calculus of processes into $\lambda\sigma$.

Let us now turn to $\rightsquigarrow_s$; we exploit the results we just proved using a translation from πs into $\pi\sigma$. We first introduce some notation.

Abbreviations. We define the following substitutions:

$$id^i(s) \stackrel{def}{=} 0.\underline{1}.\ldots.\underline{i-1}.s \qquad a^i \stackrel{def}{=} id^i(a[\uparrow^i].\uparrow^{i+1})$$
$$\uparrow_i \stackrel{def}{=} id^i(\uparrow^{i+1}) \qquad \updownarrow_i \stackrel{def}{=} id^i(\updownarrow \circ \uparrow^i)$$

Definition 6 (Translation from πs–calculus to $\pi\sigma$–calculus).
The translation function $\mathcal{T}$ from πs to $\pi\sigma$ is defined on Fig. 5. Note that we use an overloaded notation, so that $\mathcal{T}$ acts on names, processes and operators). Moreover, the translation of names and actions being straightforward, we shall not mention applications of $\mathcal{T}$ on such constructs, and write names and actions in the same way in πs and $\pi\sigma$.

Lemma 1. *(i) $P \rightsquigarrow_s P'$ implies $\mathcal{T}(P) \stackrel{+}{\rightsquigarrow}_\sigma \mathcal{T}(P')$; (ii) $P \xrightarrow{\mu}_s P'$ iff $\mathcal{T}(P) \xrightarrow{\mu}_\sigma \mathcal{T}(P')$.*

This Lemma indicates that the evolution of a πs–process can be simulated by its translation in $\pi\sigma$; note that *(ii)* basically boils down to say that the translation of an evaluated prefix is again an evaluated prefix.

$$
\begin{array}{llll}
k \in \mathbb{N}, \mathcal{T}(k) = \underline{k} & \text{names} & \mathcal{T}(\mathbf{0}) = \mathbf{0} & \text{processes} \\
\mathcal{T}(\mathbf{op}\, na) = \mathcal{T}(na)[\mathcal{T}(\mathbf{op})] & & \mathcal{T}(a\lambda.P) = \mathcal{T}(a)\lambda.\mathcal{T}(P) & \\
 & & \mathcal{T}(\overline{a}b) = \overline{\mathcal{T}(a)}\mathcal{T}(b) & \\
\mathcal{T}(\phi^i) = \uparrow_i & \text{operators} & \mathcal{T}(P_1|P_2) = \mathcal{T}(P_1) \mid \mathcal{T}(P_2) & \\
\mathcal{T}(\psi^i) = \updownarrow_i & & \mathcal{T}(\nu P) = \nu\mathcal{T}(P) & \\
\mathcal{T}(\langle a\rangle^i) = a^i.id & & \mathcal{T}(\mathbf{op}\, P) = \mathcal{T}(P)[\mathcal{T}(\mathbf{op})] &
\end{array}
$$

Fig. 5. Function $\mathcal{T}$

Proposition 2. *$\rightsquigarrow_s$ is strongly normalising and locally confluent.*

Notation. We write $P\rightsquigarrow_\sigma$ to mean that there exists P' s.t. $P\rightsquigarrow_\sigma P'$; when this does not hold, we write $P \not\rightsquigarrow_\sigma$ (we extend this notation to $\not\rightsquigarrow_s$). For any process P of the $\pi\sigma$–calculus, we define $P\!\downarrow_\sigma$ as the *unique* (by confluence) process P_0 such that $P \overset{*}{\rightsquigarrow}_\sigma P_0$ and $P_0 \not\rightsquigarrow_\sigma$. Accordingly, we write $P \overset{*}{\rightsquigarrow}_\sigma P'\!\downarrow_\sigma$ to denote the computation of a normal form for $\overset{*}{\rightsquigarrow}_s$.

Lemma 2. *$\mathcal{T}(P\!\downarrow_s) = \mathcal{T}(P)\!\downarrow_\sigma$; reciprocally, $P \not\rightsquigarrow_\sigma$ implies $\exists P_0 . P = \mathcal{T}(P_0) \wedge P_0 \not\rightsquigarrow_s$.*

3.2 Relating Behavioural Equivalences

We now turn to the comparison between $\sim_s$ and $\sim_\sigma$. For this task, we first establish some results that allow us to reason on terms which cannot be rewritten using $\rightsquigarrow_\sigma$. Similar properties for $\rightsquigarrow_s$ will then make it possible to establish a close correspondence between $\sim_s$ and $\sim_\sigma$. We first introduce an up–to technique [San95], which will be used in the proof of Prop. 3.

Theorem 1 (σ–bisimulation up to σ proof technique). *We say that a relation $\mathcal{R}$* progresses to *a relation $\mathcal{S}$, written $\mathcal{R} \rightarrow \mathcal{S}$, iff whenever $P\,\mathcal{R}\,Q$ and $P \overset{*}{\rightsquigarrow}_\sigma \xrightarrow{\mu}_\sigma \overset{*}{\rightsquigarrow}_\sigma P'$, there exists Q' s.t. $Q \overset{*}{\rightsquigarrow}_\sigma \xrightarrow{\mu}_\sigma \overset{*}{\rightsquigarrow}_\sigma Q'$ and $P'\,\mathcal{S}\,Q'$, and the symmetrical condition on transitions of Q.*

Given a relation $\mathcal{R}$, we define $\mathcal{F}_\sigma(\mathcal{R})$ as follows:

$$\mathcal{F}_\sigma(\mathcal{R}) \;=\; \mathcal{R} \cup \{(P, Q\!\downarrow_\sigma).\, P\,\mathcal{R}\,Q\} \cup \{(P\!\downarrow_\sigma, Q).\, P\,\mathcal{R}\,Q\}\,.$$

Then, for any relation $\mathcal{R}$, $\mathcal{R} \rightarrow \mathcal{F}_\sigma(\mathcal{R})$ (pronounced "$\mathcal{R}$ is a bisimulation up to $\mathcal{F}_\sigma$") implies $\mathcal{R} \subseteq \sim_\sigma$.

We can now establish the following property, that is also in some sense a proof technique, and that will be useful to prove Theorem 2:

Proposition 3. *Suppose we have two $\pi\sigma$–processes Q and Q' s.t. $Q \overset{*}{\rightsquigarrow}_\sigma Q'$; then $(P \sim_\sigma Q)$ iff $(P \sim_\sigma Q')$.*

Theorem 2. *$P{\sim_\sigma}Q$ iff $P{\downarrow_\sigma}{\sim_\sigma}Q{\downarrow_\sigma}$.*

Manipulating processes of the form $P{\downarrow_\sigma}$ suggests another notion of bisimulation, relating only such terms:

Definition 7 **($\sim_{\sigma\downarrow}$).** *A relation $\mathcal{R}$ between $\pi\sigma$–processes is a $\sim_{\sigma\downarrow}$–bisimulation iff for all $(P,Q) \in \mathcal{R}$, $P \not\rightsquigarrow_\sigma$, $Q \not\rightsquigarrow_\sigma$, and, whenever $P \xrightarrow{\mu}_\sigma \stackrel{*}{\rightsquigarrow}_\sigma P'{\downarrow_\sigma}$, there exists Q' s.t. $Q \xrightarrow{\mu}_\sigma \stackrel{*}{\rightsquigarrow}_\sigma Q'{\downarrow_\sigma}$ and $P'{\downarrow_\sigma}\, \mathcal{R}\, Q'{\downarrow_\sigma}$, and the symmetrical condition on transitions of Q. $\sim_{\sigma\downarrow}$ is the greatest $\sim_{\sigma\downarrow}$–bisimulation.*

Lemma 3. *$P{\downarrow_\sigma}{\sim_\sigma}Q{\downarrow_\sigma}$ iff $P{\downarrow_\sigma}{\sim_{\sigma\downarrow}}Q{\downarrow_\sigma}$.*

Lemma 3 is interesting because $\sim_{\sigma\downarrow}$–bisimulations are in general “much smaller” than full $\sim_\sigma$–bisimulations. We also establish similar results for $\sim_s$:

Theorem 3. *(i) $P{\sim_s}Q$ iff $P{\downarrow_s}{\sim_s}Q{\downarrow_s}$ and (ii) $P{\downarrow_s}{\sim_s}Q{\downarrow_s}$ iff $P{\downarrow_s}{\sim_{s\downarrow}}Q{\downarrow_s}$.*

Intuitively, $\sim_{s\downarrow}$– and $\sim_{\sigma\downarrow}$–bisimulations reflect the equivalence on “usual” π–calculus terms in the De Bruijn notation: a transition of such a term corresponds to a transition in one of these calculi followed by the full evaluation according to the corresponding calculus of substitutions. We finally establish:

Theorem 4. *$P{\sim_s}Q$ iff $\mathcal{T}(P){\sim_\sigma}\mathcal{T}(Q)$.*

The results we have established (Theorems 2, 3 and 4) provide great freedom in the strategy for computing manipulations on names, which can lead to the design of several equivalent notions of bisimulation. In particular, one could be interested in avoiding useless computations, that typically arise in presence of *dead code* (e.g. terms of the shape $(\nu a)\,\overline{a}x.P$) or in conjunction with the *bisimulation up to parallel composition* proof technique [San95], where it should be possible to discard some terms without propagating the substitutions. Indeed, the call–by–need flavour imposed by the definition of the transition relations suggests a “lazy” computation of relations $\rightsquigarrow$, that could perhaps be performed along the computation of $\xrightarrow{\mu}$ steps in an implementation (the relationship with weak λ–calculi of explicit substitutions [CHL96] could also be investigated).

4 Conclusion

We have presented two calculi that provide a description of the mechanism of name manipulation in the π–calculus, and shown that they basically describe the same behaviour, leaving great freedom to the user to design a strategy for computation on names. We now turn to a few observations that can be made regarding this work, discuss other presentations of the π–calculus where the issue of name managment is tackled, and propose possible extensions. These considerations are presented in more details in [Hir99], together with related technical definitions.

Discussion - Variants. Our transition systems for πs and $\pi\sigma$ correspond to an *early* version of the semantics, which comes from the will to stay close to the presentation of [Hir97]. Not much work has to be done to adapt our study to a *late* semantics; this basically amounts to delay the application of the substitutions by moving the corresponding operators from the (INP) to the (COMM) rule. Actually, the late presentation can be considered as closer to the spirit of our substitutions calculi, since it allows for a cleaner separation between the "operational" part of a transition and the corresponding name manipulations.

Polyadicity, on the contrary, requires some care, mostly for technical reasons. Without entering the details of the latter, let us try to sketch the problem. One of the so-called "structural congruence laws" in the (monadic or polyadic) π–calculus equates processes $(\nu x)(\nu y)\,P$ and $(\nu y)(\nu x)\,P$, for any x, y and P; in πs, this law is written $\nu\nu\,P{\sim_s}\nu\nu\,\psi^0 P$. In a polyadic setting, we can consider the situation where $P = \overline{a}[x,y].P'$; in this case, if we naively adapt the rules for $\xrightarrow{\mu}_s$ (and $\xrightarrow{\mu}_\sigma$), the aforementioned property does not hold, since the two processes, represented using De Bruijn's indices by $P = \overline{k}[0,1].P'$ and $P = \overline{k}[1,0]$ respectively, perform syntactically different actions. [Hir99] presents two tentative solutions to solve this problem.

Related and Future Work. [FMQ96] presents a π–calculus with explicit substitutions called $\pi\xi$. In $\pi\xi$, the processes come with an environment part, denoted by ξ, used to record the modifications brought by the previous computation steps of the process. Such a "configuration" is written $\xi :: P$. The fundamental transition rule in $\pi\xi$ is:

$$\frac{P \xrightarrow{\omega} P'}{\xi :: P \overset{\delta(\xi,\xi',\omega)}{\longmapsto} \xi' :: P'} \quad \xi' \in \eta(\xi,\omega)\,.$$

Basically, this rule makes a junction between a substitution-free transition system (written $\xrightarrow{\omega}$) and the manipulation of names, acting on ξ, and which is computed by functions δ and η. By succeeding in stripping off the name managment part of the operational semantics, Ferrari et al. obtain a transition system "à la CCS", which, together with the definition of various pairs (δ,η), makes it possible to give a uniform presentation of different notions of bisimulation (early, late, open), as well as to provide axiomatisations for these equivalences. Moreover, another interesting point about $\pi\xi$ is that it corresponds to the practice of π–calculus implementations, as in the abstract machine of PICT [Tur95].

It turns out that the work we have presented departs considerably, also in its motivations, from $\pi\xi$. It could actually be said that the two approaches are somehow symmetrical: while in $\pi\xi$ a transition generates some computation on names "outside the term", i.e. in ξ, we are interested on the contrary in a framework where a transition results in some transformations to be performed *inside* the term, via substitutions. Accordingly, the calculi πs and $\pi\sigma$ really are *calculi of substitutions*, which means that substitutions are handled within the calculus, while the question of name managment is not treated in a fully explicit manner in $\pi\xi$. Indeed, usual conventions involving silent α–conversion on names, the axiomatisation of part of the managment of names (postulating the existence of some functions to collect free names in a term, or to create a fresh variable name), as well as a side condition insuring freshness of names in the transition rule (BANG)[5] somehow "hide" implementation details that are not relevant for the purposes of the study made in [FMQ96].

Again, we believe that the differences we can point out between both approaches are related to their distinct motivations, and not to merits or drawbacks of the respective calculi. It would actually be interesting to mix both point of views; this way, one could either propagate a substitution inside the term or on the contrary store its effect in the environment. Perhaps adding a "ξ" component to our calculi, and relaxing the requirement on names to be values in the transition rules for prefixes, could be a way to achieve this.

Internal mobility, as embodied in the πI calculus [San96], is expressed by communications of private names; it is proved in [Bor98] that πI has the same expressiveness than π calculus. In πI, only restricted names can be emitted, which introduces a kind of symmetry between input and output actions, and simplifies the presentation of the semantics. Indeed, to quote [San96]:

> *"(...) the theory of πI remains very close to the theory of CCS: Alpha conversion is, essentially, the new ingredient."*

The framework of πs can provide a good illustration of this claim: indeed, to define $\pi \mathrm{I}s$, we just have to remove rules (INP) and (COMM) from Fig. 1, which results in getting rid of operator $\langle\,\rangle$. Thus, only φ and ψ are needed in derivations, in order to preserve the coherence of the representation of names (this is the counterpart of α–conversion in the De Bruijn setting). It would be interesting to further investigate how the properties of πI can be expressed within our framework, and to what extent such a presentation can shed light on internal mobility in the π–calculus.

In our calculi, relations $\xrightarrow{\mu}$ and $\rightsquigarrow$ are kept separate, and do not really interfer (as the β rule does with the σ calculus in $\lambda\sigma$). Interplay between the computation of substitutions and the evolving of processes should arise if we were to perform bisimilarity proofs in πs or $\pi\sigma$ (e.g. establishing the structural congruence laws, as in [Hir97]). Alternatively, we could also consider the integration of $\rightsquigarrow$ steps in the definition of *weak bisimilarity* for πs and $\pi\sigma$.

Acknowledgments. We would like to thank René Lalement for discussions about this work, as well as the anonymous referees for useful remarks.

References

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

[5] Note that we have no such rule in our finite calculus; however, as [Hir97] testifies, adding it should not complicate our presentation.

[Amb91] Simon J. Ambler. A de Bruijn notation for the π-calculus. Technical Report 569, Dept. of Computer Science, Queen Mary and Westfield College, London, May 1991.

[BBLRD95] Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. Lambda-upsillon, a Calculus of Explicit Substitutions which Preserves Strong Normalisation. unpublished, 1995.

[Bor98] M. Boreale. On the Expressiveness of Internal Mobility in Name-Passing Calculi. *TCS*, 195(2):205–226, 1998.

[CHL96] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.

[dB72] N.G. de Bruijn. Lambda Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation, with Application to the Curch-Rosser Theorem. In *Indagationes Mathematicae*, volume 34, pages 381–392. 1972.

[FMQ96] G. Ferrari, U. Montanari, and P. Quaglia. A π–calculus with Explicit Substitutions. *TCS*, 168(1):53–103, November 1996.

[Hir97] D. Hirschkoff. A full formalisation of π-calculus theory in the Calculus of Constructions. In *Proceedings of TPHOL'97*, volume 1275, pages 153–169. LNCS, Springer Verlag, 1997.

[Hir99] D. Hirschkoff. Handling Substitutions Explicitely in the π-calculus. Technical Report 163–99, CERMICS - ENPC, 1999.

[KML98] D. Kesner and P.E. Martínez-López. Explicit Subsitutions for Objects and Functions. In *Proceedings of PLILP/ALP '98*, volume 1490 of *LNCS*, pages 195–212. Springer Verlag, 1998.

[KR95] F. Kamareddine and A. Ríos. A λ–calculus à la De Bruijn with explicit substitutions. In *Proceedings of PLILP '95*, number 982 in LNCS, pages 45–62. Springer Verlag, 1995.

[LLL99] F. Lang, P. Lescanne, and L. Liquori. A Framework for Defining Object-Calculi. In *Proceedings of FM '99*, 1999. to appear.

[Mil92] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[San95] D. Sangiorgi. On the bisimulation proof method. Revised version of Technical Report ECS–LFCS–94–299, University of Edinburgh, 1994. An extended abstract can be found in Proc. of MFCS'95, LNCS 969, 1995.

[San96] D. Sangiorgi. Internal Mobility and Agent Passing Calculi. *Theoretical Computer Science*, 167(2), 1996.

[Tur95] D. N. Turner. *The Polymorphic Pi–calculus: Theory and Implementation*. PhD thesis, University of Edimburgh, 1995.

Explicit substitutions and Programming Languages

Jean-Jacques Lévy
INRIA-Rocquencourt
France

Abstract

We insist on the difference between weak and strong explicit substitutions. The weak form provides a useful setting for studying the operational semantics of programming languages, with application for describing intermediate code in compilers, dependency analysis for incremental computations, or the theory of sharing in the evaluation of functional languages. The theory of strong explicit substitutions is more related to modeling of the binding of variables in various logical frameworks. For instance, it is used to isolate higher-order matching from beta reduction. It is also related to the incomprehensible theory of sharing in the strong lambda calculus. But the theory of strong explicit substitutions suffers from strange properties, such as the non preservation of strong normalisation in the standard first-order typed case.

Pure Type Systems with Explicit Substitution

Roel Bloo*
E-mail: bloo@win.tue.nl

Abstract

We define an extension of pure type systems with explicit substitution. It is shown that the type systems with explicit substitution are strongly normalizing iff their ordinary counterparts are. Subject reduction is shown to fail in general but a weaker—still useful—subject reduction property is established. Two other extensions are studied (one by Muñoz) for which subject reduction *does* hold in general.

1 Introduction

As a generalization of simply typed λ-calculus, the notion of pure type system (abbreviated as PTS) incorporates many well-known type systems such as the calculus of constructions λC (cf. [1]) and system F of Girard (cf. [6]). In this paper we introduce the notion of explicit pure type system, abbreviated as EPTS, which is a pure type system extended with the notion of explicit substitution as present in λx.

λx is a named λ-calculus with explicit substitutions, it was introduced independently by Rose in [9] and the author in [2]. Details and properties can be found in [3].

In this paper we use an extension of λx (including the Π-operator and types of variables) which we call typed λx. This does not influence the properties of the $\xrightarrow[\lambda\mathrm{x}]{}$-reduction such as confluence, termination of $\xrightarrow[\mathrm{x}]{}$ and preservation of strong normalization of $\xrightarrow[\lambda\mathrm{x}]{}$ with respect to $\xrightarrow[\beta]{}$. Proofs as given in [3] or [4] are easily adapted to typed λx.

In an arbitrary PTS it is possible that bound variables occur in types of typable λ-terms. This has as a consequence that subject reduction is not guaranteed for arbitrary EPTSs. We give counterexamples in section 3 and show that this is not as bad as it seems since a weakened form of subject reduction still holds: reducing an untypable reduct to substitution normal form (i.e., eliminating all explicit substitutions) restores typability. In section 4 we briefly discuss an approach by Muñoz (cf. [7]) to type systems with explicit substitutions which uses 'type reminders' to ensure subject reduction. Then we present a different approach to regain subject reduction that admits substitutions in the context of judgements, and show that for these explicit type systems, subject reduction does hold.

*Faculty of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands. The research for this article was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO).

2 Explicit pure type systems

Pure type systems were introduced independently by Berardi and Terlouw. An overview can be found in the article [1] of Barendregt. Here we suffice by giving the necessary definitions for the sake of self-containedness and to fix notation.

Definition 2.1

- *A PTS $\mathcal{R}$ is a triple (S, A, R) where S is a set of constants called* sorts, *A is a set of pairs of sorts called* axioms *and R is a set of triples of sorts called* rules. *In the context of PTSs, we use the letter s to denote arbitrary sorts. An axiom (s_1, s_2) is usually written as $s_1 : s_2$. A sort s is called* topsort *if there is no axiom $s : s'$ for any s'.*

- *Let $\mathcal{R}$ be a PTS. Terms of $\mathcal{R}$ are defined inductively by*

$$t ::= x \mid s \mid tt \mid \lambda_{x:t}.t \mid \Pi_{x:t}.t$$

  where x ranges over a set of variables and s over sorts.

  The set of free variables in a term t is denoted by $FV(t)$. As usual for named λ-calculus, we assume the variable convention and do not distinguish α-equivalent terms. We sometimes write $t_1 \to t_2$ instead of $\Pi_{x:t_1}.t_2$ if $x \notin FV(t_2)$.

  We use the letters a, b, c, d, e, t to denote arbitrary terms and the letters $x, y, z, \alpha, \beta, \gamma$ to denote arbitrary variables.

  Ordinary β-reduction $\underset{\beta}{\to}$ in $\mathcal{R}$ is defined as the contextual closure of the rule

$$(\lambda_{x:a}.b)c \underset{\beta}{\to} b[c/x],$$

  where $[c/x]$ denotes the substitution of c for the free occurrences of x.

- *Let $\mathcal{R}$ be a PTS. A* context *in $\mathcal{R}$ is a series $x_1{:}t_1, \ldots, x_m{:}t_m$ of* statements *where the x_i are pairwise distinct variables and the t_i are terms. We use the letters Γ, Δ to denote arbitrary contexts, the empty context is denoted by ε.*

  A judgement *is written as $\Gamma \vdash_{\mathcal{R}} a : b$; by $\Gamma \vdash_{\mathcal{R}} a : b : c$ we denote that both $\Gamma \vdash_{\mathcal{R}} a : b$ and $\Gamma \vdash_{\mathcal{R}} b : c$ hold.*

  The derivation rules *for judgements and their names in $\mathcal{R}$ are*

$$\frac{(s_1, s_2) \in A}{\varepsilon \vdash_{\mathcal{R}} s_1 : s_2} \quad \text{(axiom)}$$

$$\frac{\Gamma \vdash_{\mathcal{R}} c : s \qquad x \textit{ fresh w.r.t. } \Gamma}{\Gamma, x{:}c \vdash_{\mathcal{R}} x : c} \quad \text{(start)}$$

$$\frac{\Gamma \vdash_{\mathcal{R}} a : b \qquad \Gamma \vdash_{\mathcal{R}} c : s \qquad x \textit{ fresh w.r.t. } \Gamma}{\Gamma, x{:}c \vdash_{\mathcal{R}} a : b} \quad \text{(weakening)}$$

$$\frac{\Gamma \vdash_{\mathcal{R}} a_1 : \Pi_{x:b}.c \qquad \Gamma \vdash_{\mathcal{R}} a_2 : b}{\Gamma \vdash_{\mathcal{R}} a_1 a_2 : c[a_2/x]} \quad \text{(application)}$$

$$\frac{\Gamma, x{:}a \vdash_{\mathcal{R}} b : c \qquad \Gamma \vdash_{\mathcal{R}} \Pi_{x:a}.c : s}{\Gamma \vdash_{\mathcal{R}} \lambda_{x:a}.b : \Pi_{x:a}.c} \quad \text{(abstraction)}$$

$$\frac{\Gamma \vdash_{\mathcal{R}} a : s_1 \qquad \Gamma, x{:}a \vdash_{\mathcal{R}} b : s_2 \qquad (s_1, s_2, s_3) \in R}{\Gamma \vdash_{\mathcal{R}} \Pi_{x:a}.b : s_3} \quad \text{(formation)}$$

$$\frac{\Gamma \vdash_{\mathcal{R}} a : b \qquad \Gamma \vdash_{\mathcal{R}} b' : s \qquad b =_{\beta} b'}{\Gamma \vdash_{\mathcal{R}} a : b'} \quad \text{(conversion)}$$

A context Γ is called legal *or $\mathcal{R}$-legal, if for some terms a, b, $\Gamma \vdash_{\mathcal{R}} a : b$.*

The notion of PTS is a generalization of simply typed λ-calculus $\lambda_{\rightarrow}$, namely $\lambda_{\rightarrow}$ is the PTS with sorts $*$ and $\Box$, the single rule $(*, *, *)$ and the axiom $* : \Box$.

Similar to simply typed λ-calculus, PTSs have some nice properties. Proofs can be found in [1].

Lemma 2.2 (Substitution)
Let $\mathcal{R}$ be a PTS. If $\Gamma, x{:}a, \Delta \vdash_{\mathcal{R}} b : d$ and $\Gamma \vdash_{\mathcal{R}} c : a$ then

$$\Gamma, \Delta[c/x] \vdash_{\mathcal{R}} b[c/x] : d[c/x]$$

where $(x_1{:}t_1, \ldots, x_m{:}t_m)[c/x] \equiv x_1{:}t_1[c/x], \ldots, x_m{:}t_m[c/x]$. □

Proposition 2.3 (Subject Reduction)
Let $\mathcal{R}$ be a PTS. If $\Gamma \vdash_{\mathcal{R}} a : b$ and $a \twoheadrightarrow_{\beta}^{*} a'$ then $\Gamma \vdash_{\mathcal{R}} a' : b$. □

Some PTSs type only strongly normalizing terms.

Definition 2.4 *A PTS $\mathcal{R}$ is called* strongly normalizing *(SN) if all $\mathcal{R}$-typable terms are strongly normalizing with respect to $\twoheadrightarrow_{\beta}$-reduction.*

Theorem 2.5 *Let $\mathcal{R} = (S, A, R)$ be a PTS such that $S = \{*, \Box\}$, $A = \{* : \Box\}$ and $R \subseteq \{(*, *, *), (*, \Box, \Box), (\Box, *, *), (\Box, \Box, \Box)\}$. Then $\mathcal{R}$ is strongly normalizing.* □

The eight systems with $(*, *, *) \in R$ described by Theorem 2.5 are known as the *systems of the λ-cube*. More details on the systems of the λ-cube and PTSs in general can be found in [1], e.g., the generation lemma, unicity of types lemma (not for arbitrary PTSs) and the type-correctness lemma.

The calculus underlying PTSs is ordinary typed λ-calculus. We now define the notion of explicit pure type system where the underlying calculus is a typed version of the named λ-calculus with explicit substitution λx introduced independently by Rose [9] and the author [3]. For a more elaborate discussion and properties of λx we refer to [3].

Definition 2.6

- *Terms of typed λx are defined inductively by*

$$t ::= x \mid s \mid tt \mid \lambda_{x:t}.t \mid \Pi_{x:t}.t \mid t\langle x := t\rangle,$$

where x ranges over a set of variables and s over sorts. A λx-term is called pure *if ti does not contain any substitution $\langle x := t\rangle$.*

- *An explicit pure type system (abbreviated as EPTS) $\mathcal{R}$x is a triple (S, A, R) where S is a set of sorts, A a set of axioms and R a set of rules.*

- *Reduction* $\xrightarrow[\lambda\mathrm{x}]{}$ *in* $\mathcal{R}\mathrm{x}$ *is defined as the union of* $\xrightarrow[\mathrm{Beta}]{}$ *and* $\xrightarrow[\mathrm{x}]{}$ *which are defined as the contextual closure of respectively*

$$\begin{array}{rcl} (\lambda_{x:a}.b)c & \xrightarrow[\mathrm{Beta}]{} & b\langle x := c\rangle \\ & \textit{and} & \\ (ab)\langle x := c\rangle & \xrightarrow[\mathrm{x}]{} & a\langle x := c\rangle b\langle x := c\rangle \\ (\lambda_{y:a}.b)\langle x := c\rangle & \xrightarrow[\mathrm{x}]{} & \lambda_{y:a\langle x:=c\rangle}.b\langle x := c\rangle \\ (\Pi_{y:a}.b)\langle x := c\rangle & \xrightarrow[\mathrm{x}]{} & \Pi_{y:a\langle x:=c\rangle}.b\langle x := c\rangle \\ x\langle x := c\rangle & \xrightarrow[\mathrm{x}]{} & c \\ a\langle x := c\rangle & \xrightarrow[\mathrm{x}]{} & a \qquad \textit{if } x \notin FV(a) \end{array}$$

- *Let* $\mathcal{R}\mathrm{x}$ *be an EPTS. The derivation rules for judgements in* $\mathcal{R}\mathrm{x}$ *are those for* $\mathcal{R}\mathrm{x}$ *considered as a PTS where (conversion) is replaced by*

$$\frac{\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b \qquad \Gamma \vdash_{\mathcal{R}\mathrm{x}} b' : s \qquad b =_{\lambda\mathrm{x}} b'}{\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b'} \quad \text{(conversion)}$$

extended with the rule for typing of substitutions

$$\frac{\Gamma, x{:}a \vdash_{\mathcal{R}\mathrm{x}} b : d \qquad \Gamma \vdash_{\mathcal{R}\mathrm{x}} c : a}{\Gamma \vdash_{\mathcal{R}\mathrm{x}} b\langle x := c\rangle : d[c/x]} \quad \text{(substitution)}$$

$\mathcal{R}\mathrm{x}$ *is called* strongly normalizing *if all typable terms are strongly normalizing for* $\xrightarrow[\lambda\mathrm{x}]{}$*-reduction.*

We mention some properties of $\xrightarrow[\lambda\mathrm{x}]{}$-reduction that we need in the sequel. Proofs are a straightforward extension of those in [3].

Proposition 2.7

- $\xrightarrow[\mathrm{x}]{}$ *is strongly normalizing; the* $\xrightarrow[\mathrm{x}]{}$*-normalform of a term* a *is denoted by* $\mathrm{x}(a)$.
- *If* $a \xrightarrow[\lambda\mathrm{x}]{*} a'$ *then* $\mathrm{x}(a) \xrightarrow[\beta]{*} \mathrm{x}(a')$. □

Remark 2.8 The (substitution) rule is rather strong. It allows typing of $\xrightarrow[\mathrm{Beta}]{}$-reducts of untypable terms. For instance, let $\lambda\mathrm{x}_{\rightarrow}$ be the EPTS $(\{*, \square\}, \{* : \square\}, \{(*, *, *)\})$. Then $\beta{:}* \vdash_{\lambda\mathrm{x}_{\rightarrow}} \alpha\langle\alpha := \beta\rangle : *$, but $(\lambda_{\alpha:*}.\alpha)\beta$ is not typable in $\lambda\mathrm{x}_{\rightarrow}$ since the abstraction in $\lambda_{\alpha:*}.\alpha$ is not allowed, it requires rule $(\square, \square, s)$ for some sort s. As we shall see in section 3, the strong (substitution) rule worsens the problem with subject reduction.

A weaker variant of the (substitution) rule which checks for the appropriate formation rule is

$$\frac{\Gamma \vdash_{\mathcal{R}\mathrm{x}} c : a : s_1 \qquad \Gamma, x{:}a \vdash_{\mathcal{R}\mathrm{x}} b : d : s_2 \qquad (s_1, s_2, s) \in R \text{ for some } s \in S}{\Gamma \vdash_{\mathcal{R}\mathrm{x}} b\langle x := c\rangle : d[c/x]}$$

We choose for the (substitution) rule because it is stronger and still permits to prove Proposition 2.12. Apart from the difference for subject reduction the theory for EPTSs seems independent of the choice made here.

The metasubstitution $[c/x]$ in the type of the (substitution) rule is chosen instead of an explicit substitution $\langle x := c\rangle$, since the latter complicates properties of EPTSs such as the type correctness lemma below considerably. This is due to explicit substitutions being applied to topsorts such as $\square$ in $\lambda\mathrm{x}_{\rightarrow}$.

In the sequel, $\mathcal{R}\mathrm{x}$ will be an EPTS. We mention some properties of EPTSs that also hold for ordinary PTSs (see [1]). Proofs are by straightforward induction on derivations, using the previous properties.

Lemma 2.9

1. **(Context)** *If* $\Gamma, x{:}a, \Delta$ *is an* $\mathcal{R}\mathrm{x}$*-legal context, then for some sort* s, $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : s$.
2. **(Substitution)** *If* $\Gamma, x{:}a, \Delta \vdash_{\mathcal{R}\mathrm{x}} b : c$ *and* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} d : a$ *then* $\Gamma, \Delta[d/x] \vdash_{\mathcal{R}\mathrm{x}} b[d/x] : c[d/x]$ *where* $(x_1{:}t_1, \ldots, x_n{:}t_n)[d/x] \equiv x_1{:}t_1[d/x] \ldots x_n{:}t_n[d/x]$.
3. **(Thinning)** *If* $\Gamma, \Delta \vdash_{\mathcal{R}\mathrm{x}} a : b$ *and* $\Gamma, x{:}c, \Delta$ *is an* $\mathcal{R}\mathrm{x}$*-legal context, then* $\Gamma, x{:}c, \Delta \vdash_{\mathcal{R}\mathrm{x}} a : b$.
4. **(Type Correctness)** *If* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$ *then* b *is a topsort or for some sort* s, $\Gamma \vdash_{\mathcal{R}\mathrm{x}} b : s$. □

Lemma 2.10 (Generation lemma for EPTSs)

1. *If* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} x : c$ *then* $\Gamma \equiv \Gamma_1, x{:}b, \Gamma_2$ *for some contexts* Γ_1, Γ_2 *and term* b, *and either* $c \equiv b$, *or* $c =_{\lambda\mathrm{x}} b$ *and* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} c : s$ *for some sort* s.
2. *If* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} \Pi_{x:a}.b : c$ *then* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : s_1$, $\Gamma, x{:}a \vdash_{\mathcal{R}\mathrm{x}} b : s_2$, $(s_1, s_2, s_3) \in R$ *and either* $c \equiv s_3$, *or* $c =_{\lambda\mathrm{x}} s_3$ *and* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} c : s$ *for some sort* s.
3. *If* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} \lambda_{x:a}.b : c$ *then* $\Gamma, x{:}a \vdash_{\mathcal{R}\mathrm{x}} b : d$ *and* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} \Pi_{x:a}.d : s$ *for some term* d *and sort* s, *and either* $c \equiv \Pi_{x:a}.d$, *or* $c =_{\lambda\mathrm{x}} \Pi_{x:a}.d$ *and* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} c : s'$ *for some sort* s'.
4. *If* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} ab : c$ *then* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : \Pi_{x:d}.e$ *and* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} b : d$ *for some terms* d, e *and either* $c \equiv e[b/x]$, *or* $c =_{\lambda\mathrm{x}} e[b/x]$ *and* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} c : s$ *for some sort* s.
5. *If* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a\langle x := b\rangle : c$ *then* $\Gamma, x{:}d \vdash_{\mathcal{R}\mathrm{x}} a : e$ *and* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} b : d$ *for some terms* d, e *and either* $c \equiv e[b/x]$, *or* $c =_{\lambda\mathrm{x}} e[b/x]$ *and* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} c : s$ *for some sort* s. □

Before we investigate subject reduction of EPTSs we show in two propositions that there is a nice correspondence between a PTS and the corresponding EPTS.

Proposition 2.11 *Let* $\mathcal{R} = (S, A, R)$ *be a PTS, let* $\mathcal{R}\mathrm{x}$ *be the corresponding EPTS* (S, A, R).
If $\Gamma \vdash_{\mathcal{R}} a : b$ *then* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$.

Proof: By induction on the derivation of $\Gamma \vdash_{\mathcal{R}} a : b$. Note that all rules of $\mathcal{R}$ are also rules of $\mathcal{R}\mathrm{x}$. □

Proposition 2.12 *Let* $\mathcal{R}\mathrm{x} = (S, A, R)$ *be an EPTS, let* $\mathcal{R}$ *be the corresponding PTS* (S, A, R).
If $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$ *then* $\mathrm{x}(\Gamma) \vdash_{\mathcal{R}} \mathrm{x}(a) : \mathrm{x}(b)$
where $\mathrm{x}(x_1{:}t_1, \ldots, x_m{:}t_m) \equiv x_1{:}\mathrm{x}(t_1), \ldots, x_m{:}\mathrm{x}(t_m)$. □

By the above relation between PTSs and EPTSs, strong normalization of typable terms in an EPTS can be related to strong normalization of PTSs. First we need a lemma.

Lemma 2.13 *If* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$ *and* a' *is a subterm of* a *then for some context* Δ *and term* b': $\Gamma, \Delta \vdash_{\mathcal{R}\mathrm{x}} a' : b'$. □

Theorem 2.14 *Let $\mathcal{R}$x be an EPTS, let $\mathcal{R}$ be the corresponding PTS. Then*

$$\mathcal{R} \textit{ is strongly normalizing} \iff \mathcal{R}\mathrm{x} \textit{ is strongly normalizing}$$

Proof: The proof relies on the set $\lambda\mathrm{x}^{<\infty}$ used in the proof of PSN for λx in [3] and [4]. It is detailed in the appendix. □

3 Failure of subject reduction

In the previous section we have seen that EPTSs have the same termination behaviour as their corresponding PTSs. However, in contrast to the subject reduction property of PTSs, not all λx-reducts of EPTS-typable terms are typable. We give a counterexample for the calculus $\lambda\mathrm{x}_{\to}$ which suggests that almost every EPTS fails subject reduction. However, this is mainly due to the choice for a strong (substitution) rule (cf. Remark 2.8), since pure $\lambda\mathrm{x}_{\to}$-typable terms all satisfy subject reduction. We then give pure counterexamples for the two calculi $\lambda 2$x and λPx of the λ-cube. This induces that the EPTSs $\lambda\omega$x, $\lambda P\underline{\omega}$x, $\lambda P2$x and λCx (see [1] for the non-explicit systems) do not have SR for pure terms either.

Proposition 3.1 *The EPTS $\lambda\mathrm{x}_{\to} = (\{*, \Box\}, \{* : \Box\}, \{(*, *, *)\})$ does not satisfy the subject reduction property for arbitrary terms.*

Proof: The term $(\lambda_{x:\alpha}.(\lambda_{z:\alpha}.z)x)\langle\alpha := \beta\rangle$ is $\lambda\mathrm{x}_{\to}$-typable, but its two-step $\xrightarrow[\lambda\mathrm{x}]{}$-reduct $\lambda_{x:\beta}.((\lambda_{z:\alpha}.z)x)\langle\alpha := \beta\rangle$ is not. □

A more detailed proof is given in the appendix. The counterexample in the above proposition is also a counterexample in the other systems of the explicit λ-cube.

Corollary 3.2 *None of the systems of the explicit λ-cube has the subject reduction property for arbitrary terms.* □

In section 4 we investigate a way to regain subject reduction.

Remark 3.3 The term failing SR in Proposition 3.1 is not pure. It is not a reduct of a pure, $\lambda\mathrm{x}_{\to}$-typable term either since such a term must contain a redex of the form $(\lambda_{\alpha:*}.t)\beta$, and the abstraction $\lambda_{\alpha:*}.-$ is not allowed in $\lambda\mathrm{x}_{\to}$ since it requires a rule $(\Box, s_1, s_2)$ for some sorts s_1, s_2.

This raises the question whether EPTSs satisfy subject reduction for *pure* terms. For the EPTSs corresponding to the PTSs $\lambda_{\to}$ and $\lambda\underline{\omega}$ of the λ-cube, this is unknown, but for EPTSs that allow more liberal abstraction, even subject reduction for pure terms does not hold. We give two counterexamples. For detailed proofs, we refer to [3].

Proposition 3.4 *The EPTS $\lambda 2\mathrm{x} = \big(\{*, \Box\}, \{* : \Box\}, \{(*, *, *), (\Box, *, *)\}\big)$ does not have the subject reduction property for pure terms.*

Proof: Let α and β be two distinct variables. Consider the pure term $t \equiv (\lambda_{\alpha:*}.\lambda_{f:\alpha\to\alpha}.\lambda_{x:\alpha}.fx)\beta$. and its $\xrightarrow[\lambda\mathrm{x}]{}$-reduct $t' \equiv \lambda_{f:\beta\to\beta}.(\lambda_{x:\alpha}.fx)\langle\alpha := \beta\rangle$. Then t is typable in $\lambda 2$x, but t' is not. □

The following counterexample to subject reduction for pure terms is due to Geuvers.

Proposition 3.5 *The EPTS* $\lambda P\mathrm{x} = \big(\{*,\Box\},\{*{:}\Box\},\{(*,*,*),(*,\Box,\Box)\}\big)$ *does not have the subject reduction property for pure terms.*

Proof: Consider the pure term $t \equiv (\lambda_{x:\alpha}.\lambda_{f:Px\to\alpha}.\lambda_{y:Px}.fy)z$ (P is a variable) and its reduct $t' \equiv \lambda_{f:Pz\to\alpha}.(\lambda_{y:Px}.fy)\langle x := z\rangle$. Then t is typable in $\lambda P\mathrm{x}$, but t' is not. $\Box$

As we already noted, the counterexamples in Propositions 3.4 and 3.5 imply that the systems $\lambda\omega\mathrm{x}$, $\lambda P\underline{\omega}\mathrm{x}$ and $\lambda C\mathrm{x}$ fail w.r.t. subject reduction for pure terms too.

The reason why subject reduction for pure terms fails for so many explicit PTSs is that substitutions are not performed globally at the moment of contracting a β-redex. Substitutions may be performed in one part of a term but may still be 'hanging around' in another part.

Consider the term $\lambda_{f:\beta\to\beta}.(\lambda_{x:\alpha}.fx)\langle\alpha := \beta\rangle$ of the proof of Proposition 3.4. The substitution $\langle\alpha := \beta\rangle$ has already changed the type of f, but not yet the type of x. Therefore, the application fx cannot be typed. If however the substitution $\langle\alpha := \beta\rangle$ is evaluated one step further, then the resulting term $\lambda_{f:\beta\to\beta}.\lambda_{x:\alpha\langle\alpha:=\beta\rangle}.(fx)\langle\alpha := \beta\rangle$, is typable once again.

The failure of subject reduction may lead one to believe that EPTSs are useless, however typable terms do not have *pure* reducts which are untypable. It is always possible to reduce further an untypable reduct and obtain a reduct that is typable. This weak subject reduction property is made precise below. First, we need a lemma on reduction in contexts.

We say that $x_1{:}t_1,\ldots,x_m{:}t_m \xrightarrow[\lambda\mathrm{x}]{*} x_1{:}t_1',\ldots,x_m{:}t_m'$ if for $i = 1,\ldots,m$, $t_i \xrightarrow[\lambda\mathrm{x}]{*} t_i'$.

Lemma 3.6 *Let* Γ *and* Δ *be contexts such that* Δ *is* $\mathcal{R}\mathrm{x}$*-legal and* $\Delta \xrightarrow[\lambda\mathrm{x}]{*} \Gamma$.
If $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$ *then* $\Delta \vdash_{\mathcal{R}\mathrm{x}} a : b$.

Proof: Induction on the derivation of $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$. We need the Lemmas 2.9.1 and 2.9.4. $\Box$

Proposition 3.7 (Weak Subject Reduction for EPTSs)
If $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$ *and* $a \xrightarrow[\lambda\mathrm{x}]{*} a'$*, then* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} \mathrm{x}(a') : b$.

Proof: Suppose $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$ and $a \xrightarrow[\lambda\mathrm{x}]{*} a'$. Then by Propositions 2.12, 2.7 and 2.3, $\mathrm{x}(\Gamma) \vdash_{\mathcal{R}} \mathrm{x}(a') : \mathrm{x}(b)$, so by Lemmas 2.11 and 3.6 $\Gamma \vdash_{\mathcal{R}\mathrm{x}} \mathrm{x}(a') : \mathrm{x}(b)$.

Applying Lemma 2.9.4 to $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$, we have $b \equiv s$ or $\Gamma \vdash_{\mathcal{R}\mathrm{x}} b : s$ for some sort s. In the first case, $\mathrm{x}(b) \equiv b$ so we are done; in the second case we can apply the (conversion) rule and conclude $\Gamma \vdash_{\mathcal{R}\mathrm{x}} \mathrm{x}(a') : b$. $\Box$

4 Two different ways to regain subject reduction

In this section we discuss two different extensions of EPTSs such that subject reduction holds once again. The first solution is due to Muñoz (cf. [7]), the second is inspired by joint work of the author with Kamareddine and Nederpelt on adding definitions to type systems (cf. [5]). We start by describing Muñoz' work.

Muñoz studies a De Bruijn indexed calculus with dependent types $\lambda\Pi\phi$. The underlying λ-calculus is $\lambda\phi$, a variant of $\lambda\sigma$, which was introduced in [8]. The type system of $\lambda\Pi\phi$ is—apart from some minor differences due to $\lambda\Pi\phi$ using indices—similar to that of the EPTS $\lambda P\mathrm{x}$.

Muñoz proves the following properties of $\lambda\Pi\phi$: confluence of reduction on typable terms (even for open terms), weak normalization of typable terms, uniqueness of types and subject reduction.

The last property is surprising since we have shown in Proposition 3.5 that λPx fails w.r.t. SR. Therefore, $\lambda\Pi\phi$ cannot simply be an indexed version of λPx. Indeed, reduction in $\lambda\Pi\phi$ (denoted by $\xrightarrow[\lambda\Pi\phi]{}$) differs from λx-reduction. We shall explain the difference by considering a named version of $\lambda\Pi\phi$. In a named setting, reduction in $\lambda\Pi\phi$ is similar to $\xrightarrow[\lambda\mathrm{x}]{}$-reduction except for the rules for distribution of a substitution over abstractions:

$$(\lambda_{y:a}.b)\langle x := c\rangle \xrightarrow[\lambda\Pi\phi]{} \lambda_{y:a\langle x:=c\rangle}.b\langle y{:}a, x := c\rangle$$

$$(\Pi_{y:a}.b)\langle x := c\rangle \xrightarrow[\lambda\Pi\phi]{} \Pi_{y:a\langle x:=c\rangle}.b\langle y{:}a, x := c\rangle$$

What happens in these rules is that the substitution in the body of the abstraction contains what we shall call a *type reminder* $y{:}a$. We do not want to go into all details of this reduction[1], but the result of the type reminder is that in the target b of $b\langle y{:}a, x := c\rangle$, y is considered to have type a instead of $a\langle x := c\rangle$. This is reflected by the typing rules where $\lambda\Pi\phi$ has the indexed variants of the following rules:

$$\frac{\Gamma, x{:}d \vdash_{\lambda\Pi\phi} y : a[c/x] \qquad \Gamma \vdash_{\lambda\Pi\phi} c : d}{\Gamma, x{:}d, \langle y{:}a\rangle \vdash_{\lambda\Pi\phi} y : a} \quad \text{(Cons)}$$

$$\frac{\Gamma, x{:}d, \langle y{:}t\rangle \vdash_{\lambda\Pi\phi} a : b \qquad \Gamma \vdash_{\lambda\Pi\phi} c : d}{\Gamma \vdash_{\lambda\Pi\phi} a\langle y{:}t, x := c\rangle : b[c/x]} \quad \text{(Clos)}$$

We finish our description of Muñoz' work by outlining how the λPx-untypable reduct of the term $(\lambda_{x:\alpha}.\lambda_{f:Px\to\alpha}.\lambda_{y:Px}.fy)z$ in Proposition 3.5 can be typed in $\lambda\Pi\phi$. Now the reduction is

$$\begin{aligned}
(\lambda_{x:\alpha}.\lambda_{f:Px\to\alpha}.\lambda_{y:Px}.fy)z &\xrightarrow[\lambda\Pi\phi]{} (\lambda_{f:Px\to\alpha}.\lambda_{y:Px}.fy)\langle x := z\rangle \\
&\xrightarrow[\lambda\Pi\phi]{} \lambda_{f:(Px\to\alpha)\langle x:=z\rangle}.(\lambda_{y:Px}.fy)\langle f{:}Px \to \alpha, x := z\rangle \\
&\xrightarrow[\lambda\Pi\phi]{} \lambda_{f:Pz\to\alpha}.(\lambda_{y:Px}.fy)\langle f{:}Px \to \alpha, x := z\rangle
\end{aligned}$$

The latter term is typable in context $\Gamma \equiv \alpha{:}*, P{:}\alpha \to *, \alpha{:}*, z{:}\alpha$ as follows:

$$\begin{array}{l}
\Gamma, f{:}Pz \to \alpha, x{:}\alpha \vdash_{\lambda\Pi\phi} z : \alpha, f : Pz \to \alpha \equiv (Px \to \alpha)[z/x] \\
\Gamma, f{:}Pz \to \alpha, x{:}\alpha, \langle f{:}Px \to \alpha\rangle \vdash_{\lambda\Pi\phi} f : Px \to \alpha \\
\Gamma, f{:}Pz \to \alpha, x{:}\alpha, \langle f{:}Px \to \alpha\rangle, y : Px \vdash_{\lambda\Pi\phi} y : Px, f : Px \to \alpha \\
\Gamma, f{:}Pz \to \alpha, x{:}\alpha, \langle f{:}Px \to \alpha\rangle \vdash_{\lambda\Pi\phi} \lambda_{y:Px}.fy : \Pi_{y:Px}.\alpha \\
\Gamma, f{:}Pz \to \alpha \vdash_{\lambda\Pi\phi} (\lambda_{y:Px}.fy)\langle f{:}Px \to \alpha, x := z\rangle : \Pi_{y:Pz}.\alpha \\
\Gamma \vdash_{\lambda\Pi\phi} \lambda_{f:Pz\to\alpha}.(\lambda_{y:Px}.fy)\langle f{:}Px \to \alpha, x := z\rangle : \Pi_{f:Pz\to\alpha}.\Pi_{y:Pz}.\alpha
\end{array}$$

Now we present a different approach to regain subject reduction for EPTSs. Where Muñoz adds type reminders to the context and to substitutions in terms, our solution is that, before a substitution is applied to a term, it is introduced into the context. Then substitutions in the context are allowed to influence conversion of types and to move from the context to the subject of a judgement.

This results in the following notion of explicit type system (abbreviated as ETS):

Definition 4.1

[1] For instance, multiple distribution steps create multiple type-reminders in a substitution.

- *An explicit type system* $\mathcal{S}\mathrm{x}$ *is a triple* (S, A, R) *as for EPTSs.*
- *A context in an ETS is a series* $\sigma_1, \ldots, \sigma_n$ *of statements where each* σ_i *is either of the form* $x_i : t_i$ *or* $\langle x := t'_i : t_i \rangle$*, where the* x_i *are pairwise distinct variables and the* t_i *and* t'_i *are* $\lambda\mathrm{x}$*-terms.*
- *Let* Γ *be an ETS-context. The equivalence* $=_\Gamma$ *on* $\lambda\mathrm{x}$*-terms (used in the (conversion) rule below) is defined by*
  1. *if* $b =_{\lambda\mathrm{x}} b'$ *then* $b =_\Gamma b'$*, and*
  2. *if* $\Gamma \equiv \Gamma_1, \langle x := c : d \rangle, \Gamma_2$ *and* b' *is the result of replacing one free occurrence of* x *in* b *by* c*, then* $b =_\Gamma b'$*.*
- *The derivation rules for judgements in an ETS are the rules (axiom), (start), (weakening), (application), (abstraction) and (formation) from Definition 2.1 and the following ones:*

$$\frac{\Gamma \vdash_{\mathcal{S}\mathrm{x}} a : s \qquad \Gamma \vdash_{\mathcal{S}\mathrm{x}} b : a \qquad x \textit{ fresh}}{\Gamma, \langle x := b : a \rangle \vdash_{\mathcal{S}\mathrm{x}} x : a} \quad \textit{(subst-start)}$$

$$\frac{\Gamma \vdash_{\mathcal{S}\mathrm{x}} a : s \qquad \Gamma \vdash_{\mathcal{S}\mathrm{x}} b : a \qquad \Gamma \vdash_{\mathcal{S}\mathrm{x}} c : d \qquad x \textit{ fresh}}{\Gamma, \langle x := b : a \rangle \vdash_{\mathcal{S}\mathrm{x}} c : d} \quad \textit{(subst-weakening)}$$

$$\frac{\Gamma \vdash_{\mathcal{S}\mathrm{x}} a : b \qquad \Gamma \vdash_{\mathcal{S}\mathrm{x}} b' : s \qquad b =_\Gamma b'}{\Gamma \vdash_{\mathcal{S}\mathrm{x}} a : b'} \quad \textit{(conversion)}$$

$$\frac{\Gamma, \langle x := c : d \rangle \vdash_{\mathcal{S}\mathrm{x}} a : b}{\Gamma \vdash_{\mathcal{S}\mathrm{x}} a\langle x := c \rangle : b[c/x]} \quad \textit{(substitution)}$$

This approach is very similar to and inspired by the type systems with definitions in [5] where $\langle x := c : d \rangle$ in a context is written as $(\lambda_{x:d}.-)c$, and where instead of a substitution in the subject of the conclusion $\Gamma \vdash a\langle x := c \rangle : b[c/x]$ of $\Gamma, \langle x := c : d \rangle \vdash a : b$ there is a redex: $\Gamma \vdash (\lambda_{x:t}.a)c : b[c/x]$. However, in [5], subject reduction is a simple consequence of the generation lemma due to that redex. Here we need to do a bit more work.

First a simple lemma about the new requirement in the (conversion) rule.

Lemma 4.2 *If* $a =_{\Gamma, \langle x := c : d \rangle} b$ *then* $a[c/x] =_\Gamma b[c/x]$*.* □

The properties of EPTSs that we presented in section 2 hold, when reworded in the obvious way, also for ETSs. In addition every ETS satisfies the subject reduction property. We briefly present some reworded properties for ETSs and outline a proof of subject reduction.

In the rest of this section, let $\mathcal{S}\mathrm{x}$ be an arbitrary ETS.

Lemma 4.3 (Context, type correctness and substitution for ETSs)

- *If* $\Gamma, x{:}a, \Delta$ *or* $\Gamma, \langle x := b : a \rangle, \Delta$ *is an* $\mathcal{S}\mathrm{x}$*-legal context, then for some sort* s*,* $\Gamma \vdash_{\mathcal{S}\mathrm{x}} a : s$*.*
- *If* $\Gamma \vdash_{\mathcal{S}\mathrm{x}} a : b$ *then* b *is a topsort or for some sort* s*,* $\Gamma \vdash_{\mathcal{S}\mathrm{x}} b : s$*.*
- *If* $\Gamma \vdash_{\mathcal{S}\mathrm{x}} d : a$ *and either* $\Gamma, x{:}a, \Delta \vdash_{\mathcal{S}\mathrm{x}} b : c$ *or* $\Gamma, \langle x := d' : a \rangle, \Delta \vdash_{\mathcal{S}\mathrm{x}} b : c$*, then* $\Gamma, \Delta[d/x] \vdash_{\mathcal{S}\mathrm{x}} b[d/x] : c[d/x]$ *where for the substitution in* Δ*,* $\langle y := e : f \rangle[d/x] \equiv \langle y := e[d/x] : f[d/x] \rangle$*.*

Proof: By straightforward induction on the derivation of judgements. $\square$

Due to the admission of substitutions to contexts, we have an extended thinning lemma.

Lemma 4.4 (Thinning for ETSs)

- *If* $\Gamma, \Delta \vdash_{\mathcal{S}\mathrm{x}} a : b$ *and* $\Gamma, x{:}c, \Delta$ *is an* $\mathcal{S}$x*-legal context, then* $\Gamma, x{:}c, \Delta \vdash_{\mathcal{S}\mathrm{x}} a : b$.
- *If* $\Gamma, x{:}c, \Delta \vdash_{\mathcal{S}\mathrm{x}} a : b$ *or* $\Gamma, \Delta \vdash_{\mathcal{S}\mathrm{x}} a : b$, *and* $\Gamma, \langle x := d : c\rangle, \Delta$ *is an* $\mathcal{S}$x*-legal context, then* $\Gamma, \langle x := d : c\rangle, \Delta \vdash_{\mathcal{S}\mathrm{x}} a : b$.

Proof: Induction on the derivation of judgements. $\square$

Using the thinning lemma we can prove that an ETS extends the corresponding EPTS, that is:

Lemma 4.5 *Let* $\mathcal{R}\mathrm{x} = (S, A, R)$ *be an EPTS, let* $\mathcal{S}\mathrm{x} = (S, A, R)$ *be the corresponding ETS. If* $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$ *then also* $\Gamma \vdash_{\mathcal{S}\mathrm{x}} a : b$.

Proof: Induction on the derivations. We treat the case where the last rule used in the $\mathcal{R}$x-derivation is the (substitution) rule. Then $\Gamma \vdash_{\mathcal{R}\mathrm{x}} b\langle x := c\rangle : d[c/x]$ as a conclusion from $\Gamma, x{:}a \vdash_{\mathcal{R}\mathrm{x}} b : d$ and $\Gamma \vdash_{\mathcal{R}\mathrm{x}} c : a$.

Now by the induction hypothesis, also $\Gamma, x{:}a \vdash_{\mathcal{S}\mathrm{x}} b : d$ and $\Gamma \vdash_{\mathcal{S}\mathrm{x}} c : a$. Then by the context lemma and the (subst-start) rule, $\Gamma, \langle x := c : a\rangle$ is an $\mathcal{S}$x-legal context, so by thinning, $\Gamma, \langle x := c : a\rangle \vdash_{\mathcal{S}\mathrm{x}} b : d$. Now by the (substitution) rule we can conclude $\Gamma \vdash_{\mathcal{S}\mathrm{x}} b\langle x := c\rangle : d[c/x]$. $\square$

The second, third and fourth cases of the generation lemma 2.10 for EPTSs also hold for ETSs when $=_{\lambda\mathrm{x}}$ is replaced by $=_\Gamma$. The rewording of cases 1. and 5. becomes:

Lemma 4.6 (Generation, variable and substitution case)

- *If* $\Gamma \vdash_{\mathcal{S}\mathrm{x}} x : c$ *then* $\Gamma \equiv \Gamma_1, x{:}b, \Gamma_2$ *or* $\Gamma \equiv \Gamma_1, \langle x := a : b\rangle, \Gamma_2$ *for some contexts* Γ_1, Γ_2, *terms* a, b, *and either* $c \equiv b$, *or* $c =_\Gamma b$ *and* $\Gamma \vdash_{\mathcal{S}\mathrm{x}} c : s$ *for some sort* s.
- *If* $\Gamma \vdash_{\mathcal{S}\mathrm{x}} a\langle x := b\rangle : c$ *then for some term* d, $\Gamma, \langle x := b : d\rangle \vdash_{\mathcal{S}\mathrm{x}} a : c$.

Proof: Induction on the derivation of judgements. The interesting case of the second item is the derivation of $\Gamma \vdash_{\mathcal{S}\mathrm{x}} a\langle x := c\rangle : b[c/x]$ from $\Gamma, \langle x := c : d\rangle \vdash_{\mathcal{S}\mathrm{x}} a : b$. We must show that also $\Gamma, \langle x := c : d\rangle \vdash_{\mathcal{S}\mathrm{x}} a : b[c/x]$.

By type correctness 4.3 we have $b \equiv s$ or $\Gamma, \langle x := c : d\rangle \vdash_{\mathcal{S}\mathrm{x}} b : s$ for some sort s. In the first case, $b \equiv b[c/x]$ and we are done. Otherwise, by the (substitution) rule we have $\Gamma \vdash_{\mathcal{S}\mathrm{x}} b[c/x] : s$ and by thinning 4.4, $\Gamma, \langle x := c : d\rangle \vdash_{\mathcal{S}\mathrm{x}} b[c/x] : s$. Since $b =_{\Gamma, \langle x := c : d\rangle} b[c/x]$ we are done by the (conversion) rule. $\square$

Now we can prove subject reduction.

Lemma 4.7 (Subject Reduction for ETSs)
Every ETS satisfies the subject reduction property.

Proof: Induction on the structure of terms of λx. We treat two cases.

- $\Gamma \vdash_{\mathcal{S}\mathrm{x}} (\lambda_{x:a}.b)c : d$ and $(\lambda_{x:a}.b)c \xrightarrow[\lambda\mathrm{x}]{} b\langle x := c\rangle$.

  By applying the generation lemma and (conversion) several times we get: $\Gamma \vdash_{\mathcal{S}\mathrm{x}} c : a$ and for some term e, $d =_\Gamma e[c/x]$ and $\Gamma, x{:}a \vdash_{\mathcal{S}\mathrm{x}} b : e$. Then by the Thinning Lemma 4.4 we get $\Gamma, \langle x := c : a\rangle \vdash_{\mathcal{S}\mathrm{x}} b : e$ and by the (substitution) rule, $\Gamma \vdash_{\mathcal{S}\mathrm{x}} b\langle x := c\rangle : e[c/x]$. Now by applying (conversion) once we are done.

- $\Gamma \vdash_{\mathcal{S}\mathrm{x}} (a_1 a_2)\langle x := c\rangle : b$ and $(a_1 a_2)\langle x := c\rangle \xrightarrow[\lambda\mathrm{x}]{} a_1\langle x := c\rangle a_2\langle x := c\rangle$.

  First note that $x \notin FV(b)$, and that b can not be a topsort. Therefore, by type correctness 4.3, $\Gamma \vdash_{\mathcal{S}\mathrm{x}} b : s$ for some sort s. By the generation lemma we get terms e, f such that $\Gamma, \langle x := c : d\rangle \vdash_{\mathcal{S}\mathrm{x}} a_1 : \Pi_{y:f}.e$, $\Gamma, \langle x := c : d\rangle \vdash_{\mathcal{S}\mathrm{x}} a_2 : f$ and either $b \equiv e[a_2/y]$ or $b =_{\Gamma,\langle x:=c:d\rangle} e[a_2/y]$.

  Now by application of the (substitution) rule (twice) and the (application) rule we get $\Gamma \vdash_{\mathcal{S}\mathrm{x}} a_1\langle x := c\rangle a_2\langle x := c\rangle : e[c/x][a_2\langle x := c\rangle/y]$, and we are done by the (conversion) rule if we show that $b =_\Gamma e[c/x][a_2\langle x := c\rangle/y]$.

  Since $x \notin FV(b)$, $b \equiv b[c/x]$. Now if $b \equiv e[a_2/y]$ then $b[c/x] \equiv e[a_2/y][c/x]$. Otherwise, if $b =_{\Gamma,\langle x:=c:d\rangle} e[a_2/y]$, then by Lemma 4.2 $b[c/x] =_\Gamma e[a_2/y][c/x]$.

  So in both cases $b \equiv e[a_2/y][c/x] \equiv e[c/x][a_2[c/x]/y] =_{\lambda\mathrm{x}} e[c/x][a_2\langle x := c\rangle/y]$. Hence $b =_\Gamma e[c/x][a_2\langle x := c\rangle/y]$. □

Finally, also strong normalization is preserved by ETSs. The proof is similar to that of Theorem 2.14.

Theorem 4.8
Let $\mathcal{S}$ be the PTS (S, A, R), let $\mathcal{S}\mathrm{x}$ be the corresponding ETS (S, A, R).
Then $\mathcal{S}$ is strongly normalizing $\iff$ $\mathcal{S}\mathrm{x}$ is strongly normalizing. □

5 Conclusion

We have introduced two notions of type systems for λ-calculus with explicit substitution based on the calculus λx and PTSs.

One of them, the class of explicit pure type systems, extends PTSs in an intuitive way. It preserves strong normalization, but does not satisfy the subject reduction property.

The other, the class of explicit type systems, needs a different notion of context but thenall ETSs preserve strong normalization and also satisfy the subject reduction property.

Since we consider type systems with explicit substitution based on λ-calculus with named variables, we could develop very general type systems. This is an advantage over calculi based on λ-calculus with de Bruijn indices since there only for some systems of the λ-cube explicit type systems are known.

We consider both the notions of EPTS as well as ETS interesting and hope that future research will point out which one is most useful.

5.1 Acknowledgements

The author is grateful to Fairouz Kamareddine and Rob Nederpelt for their encouragement to write this paper, and to the anonymous referees who pointed out numerous errors and imperfections in the first manuscript.

References

[1] H.P. Barendregt. Handbook of logic in computer science, volume 2: Background: Computational structures. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Lambda Calculi with Types*, pages 117–309. Oxford University Press, 1992.

[2] R. Bloo. Preservation of strong normalisation for explicit substitution. Technical Report 95-08, TUE Computing Science Reports, Eindhoven University of Technology, March 1995.

[3] R. Bloo. *Preservation of Termination for Explicit Substitutions*. PhD thesis, Eindhoven University of Technology, 1997.

[4] R. Bloo and H. Geuvers. Explicit substitution: on the edge of strong normalization. *Theoretical Computer Science*, 211:375–395, 1999.

[5] R. Bloo, F. Kamareddine, and R. Nederpelt. The Barendregt Cube with Definitions and Generalised Reduction. *Information and Computation*, 126(2):123–143, 1996.

[6] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[7] C. Muñoz. Dependent types with explicit substitutions: A meta-theoretical development. Technical report, Unité de recherche INRIA-Rocquencourt, 1997. Preprint electronically available at: `http://pauillac.inria.fr/~cesar/Papers/typ96.ps.gz`.

[8] C. Muñoz. Meta-theoretical properties of $\lambda\phi$: A left-linear variant of $\lambda\sigma$. Technical Report RR-3107, Unité de recherche INRIA-Rocquencourt, 1997.

[9] K.H. Rose. Explicit cyclic substitutions. In M. Rusinowitch and J.-L. Rémy, editors, *CTRS '92—3rd International Workshop on Conditional Term Rewriting Systems*, number 656 in LNCS, pages 36–50, Pont-a-Mousson, France, July 1992. Springer-Verlag.

A Appendix

Proof of Proposition 2.12

By induction on the derivation of $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$. Note that all rules of $\mathcal{R}$x but (conversion) and (substitution) are also rules of $\mathcal{R}$ when applied to pure terms. We only treat the cases (conversion) and (substitution).

- $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b'$ as a result of applying the (conversion) rule to $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$, $\Gamma \vdash_{\mathcal{R}\mathrm{x}} b' : s$ and $b =_{\lambda\mathrm{x}} b'$.

  Then by the induction hypothesis $\mathrm{x}(\Gamma) \vdash_{\mathcal{R}} \mathrm{x}(a) : \mathrm{x}(b)$ and $\mathrm{x}(\Gamma) \vdash_{\mathcal{R}} \mathrm{x}(b') : s$ (note that $\mathrm{x}(s) \equiv s$). Furthermore, since $b =_{\lambda\mathrm{x}} b'$, $\mathrm{x}(b) =_{\beta} \mathrm{x}(b')$, so by the (conversion) rule for PTSs, $\mathrm{x}(\Gamma) \vdash_{\mathcal{R}} \mathrm{x}(a) : \mathrm{x}(b')$.

- $\Gamma \vdash_{\mathcal{R}\mathrm{x}} b\langle x := c\rangle : d[c/x]$ as a result of applying the (substitution) rule to $\Gamma, x{:}a \vdash_{\mathcal{R}\mathrm{x}} b : d$ and $\Gamma \vdash_{\mathcal{R}\mathrm{x}} c : a$.

  Then by the induction hypothesis, $\mathrm{x}(\Gamma), x{:}\mathrm{x}(a) \vdash_{\mathcal{R}} \mathrm{x}(b) : \mathrm{x}(d)$ and $\mathrm{x}(\Gamma) \vdash_{\mathcal{R}} \mathrm{x}(c) : \mathrm{x}(a)$.

  Now by Lemma 2.2, $\mathrm{x}(\Gamma) \vdash_{\mathcal{R}} \mathrm{x}(b)[\mathrm{x}(c)/x] : \mathrm{x}(d)[\mathrm{x}(c)/x]$, and we are done since $\mathrm{x}(b\langle x := c\rangle) \equiv \mathrm{x}(b)[\mathrm{x}(c)/x]$ and $\mathrm{x}(d[c/x]) \equiv \mathrm{x}(d)[\mathrm{x}(c)/x]$. □

Proof of Lemma 2.13

We prove by induction on the derivation the following more general statement.

If $x_1{:}t_1, \ldots, x_{m-1}{:}t_{m-1} \vdash_{\mathcal{R}\mathrm{x}} t_m : b$ and for some i, $a' \subseteq t_i$ then

$$x_1{:}t_1, \ldots, x_{i-1}{:}t_{i-1}, \Delta \vdash_{\mathcal{R}\mathrm{x}} a' : b' \text{ for some context } \Delta \text{ and term } b'.$$

□

Proof of Theorem 2.14

For the proof we need to extend the set $\lambda\mathrm{x}^{<\infty}$ used for proving PSN of λx in [3, 4] to typed λx.

We define $\lambda\mathrm{x}^{<\infty}$ to be the set of terms of typed λx for which for every (non-strict) subterm s: $\mathrm{x}(s)$ is strongly normalizing with respect to $\xrightarrow[\beta]{}$-reduction.

Although $\lambda\mathrm{x}^{<\infty}$ differs from the set in [3], the proof of the following theorem doesn't change considerably.

Theorem A.1 (PSN for λx) *1. typed λx satisfies PSN*

2. $\lambda\mathrm{x}^{<\infty} = \{t \in \textit{typed } \lambda\mathrm{x} \mid t \textit{ is strongly normalizing for } \xrightarrow[\lambda\mathrm{x}]{}\textit{-reduction}\}$

□

We also need the following property of $\xrightarrow[\lambda\mathrm{x}]{}$-reduction.

Proposition A.2 *If $a \xrightarrow[\beta]{} b$ then $a \xrightarrow[\lambda\mathrm{x}]{+} b$.* □

Now we are ready for the proof.

If $\mathcal{R}$x is strongly normalizing then $\mathcal{R}$ is strongly normalizing by Proposition 2.11 and the above proposition.

On the other hand, if $\mathcal{R}$ is strongly normalizing, let $\Gamma \vdash_{\mathcal{R}\mathrm{x}} a : b$. By Lemma 2.13, any subterm a' of a is $\mathcal{R}$x-typable hence by Proposition 2.12, $\mathrm{x}(a')$ is $\mathcal{R}$-typable, so $\mathrm{x}(a') \in SN_{\beta}$. Therefore $a \in \lambda\mathrm{x}^{<\infty}$ and by the above theorem: $a \in SN_{\lambda\mathrm{x}}$. □

Proof of Proposition 3.1

The term $(\lambda_{x:\alpha}.(\lambda_{z:\alpha}.z)x)\langle\alpha := \beta\rangle$ is $\lambda\mathrm{x}_{\rightarrow}$-typable as follows (we do not give a full type derivation here):

$$\begin{array}{rll}
\beta{:}*, \alpha{:}*, x{:}\alpha, z{:}\alpha & \vdash_{\lambda\mathrm{x}_{\rightarrow}} & z : \alpha : *, \quad x : \alpha \\
\beta{:}*, \alpha{:}*, x{:}\alpha & \vdash_{\lambda\mathrm{x}_{\rightarrow}} & \Pi_{z:\alpha}.\alpha : * \qquad (\text{rule } (*,*,*)) \\
\beta{:}*, \alpha{:}*, x{:}\alpha & \vdash_{\lambda\mathrm{x}_{\rightarrow}} & \lambda_{z:\alpha}.z : \Pi_{z:\alpha}.\alpha \\
\beta{:}*, \alpha{:}*, x{:}\alpha & \vdash_{\lambda\mathrm{x}_{\rightarrow}} & (\lambda_{z:\alpha}.z)x : \alpha \\
\beta{:}*, \alpha{:}* & \vdash_{\lambda\mathrm{x}_{\rightarrow}} & \Pi_{x:\alpha}.\alpha : * \qquad (\text{rule } (*,*,*)) \\
\beta{:}*, \alpha{:}* & \vdash_{\lambda\mathrm{x}_{\rightarrow}} & \lambda_{x:\alpha}.(\lambda_{z:\alpha}.z)x : \Pi_{x:\alpha}.\alpha \\
\beta{:}* & \vdash_{\lambda\mathrm{x}_{\rightarrow}} & (\lambda_{x:\alpha}.(\lambda_{z:\alpha}.z)x)\langle\alpha := \beta\rangle : \Pi_{x:\beta}.\beta
\end{array}$$

We have the following λx-reduction:

$$\begin{array}{rl}
(\lambda_{x:\alpha}.(\lambda_{z:\alpha}.z)x)\langle\alpha := \beta\rangle & \underset{\mathrm{x}}{\twoheadrightarrow} \\
\lambda_{x:\alpha\langle\alpha:=\beta\rangle}.((\lambda_{z:\alpha}.z)x)\langle\alpha := \beta\rangle & \underset{\mathrm{x}}{\twoheadrightarrow} \\
\lambda_{x:\beta}.((\lambda_{z:\alpha}.z)x)\langle\alpha := \beta\rangle &
\end{array}$$

The latter term cannot be typed in $\lambda\mathrm{x}_{\rightarrow}$. This can be seen by applying the generation lemma 2.10 repeatedly:

$\lambda_{x:\beta}.((\lambda_{z:\alpha}.z)x)\langle\alpha := \beta\rangle$ is typable in context $\beta{:}*$ if (use the generation lemma) $((\lambda_{z:\alpha}.z)x)\langle\alpha := \beta\rangle$ is typable in context $\beta{:}*, x{:}\beta$.

Again by the generation lemma this term is typable only if for some term t, $(\lambda_{z:\alpha}.z)x$ is typable in context $\beta{:}*, x{:}\beta, \alpha{:}t$ and β has type t in context $\beta{:}*, x{:}\beta$.

It is clear that in $\lambda\mathrm{x}_{\rightarrow}$, the only type of β in context $\beta{:}*, x{:}\beta$ is $*$, hence $t \equiv *$. Now by the generation lemma $(\lambda_{z:\alpha}.z)x$ is typable if for some terms t', t'', $\lambda_{z:\alpha}.z$ has type $\Pi_{y:t'}.t''$ and x has type t' in context $\beta{:}*, x{:}\beta, \alpha{:}t$. But then for the last requirement $t' \equiv \beta$ and for the first requirement $t' \equiv \alpha$, a contradiction. $\square$

The polymorphic lambda calculus with explicit substitutions

Eduardo Bonelli

Laboratoire de Recherche en Informatique, Bât 490, Université de Paris-Sud, 91405, Orsay Cedex, France.

Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina.

email: bonelli@lri.fr

Abstract

We present F_{es}, a polymorphic lambda calculus with explicit substitutions. The usual metalevel notion of substitution is incorporated into the calculus in two flavours: term substitutions and type substitutions. The property of subject reduction is shown to hold by extending type assignments of the typing rules to allow non-pure types (types with possible occurrences of the type substitution operator). Also, we adapt Tait's version of the reducibility technique to prove strong normalization of F_{es}.

1 Introduction

Substitution constitutes a fundamental operation in the lambda calculus. From a computational cost point of view, in terms of atomic reductions steps implementing this operation, the picture is not so clear. The β-reduction axiom is expressed as $(\lambda x.M)N \rightarrow_\beta M\{x \leftarrow N\}$ and may be interpreted as: the result of executing function $\lambda x.M$ over argument N (the actual parameter) is obtained by substituting N for all (free) occurrences of x (the formal parameter). But any implementation of this axiom must include instructions for computing this process. Although from the metalevel an execution of the β-reduction axiom is atomic, the cost of computing $M\{x \leftarrow N\}$ depends highly on the form of M, moreso if unwanted variable capture conflicts must be avoided by renaming bound variables. Calculi of explicit substitution incorporate the substitution operation into the calculus and caters for its behaviour by incorporating additional reduction rules. Thus the study of the (non trivial) process of substitution may be developed in a uniform manner. In this way, calculi of explicit substitution are said to eliminate the gap between theory and practice in the implementation of functional programming languages.

The study of explicit substitution calculi was pioneered by the λ_σ-calculus [1], formulated in a de Bruijn indice setting. Problems such as confluence (on open terms) and preservation of strong normalization suffered by λ_σ motivated further studies in the area: other calculi formulated are $\lambda_{\sigma_\Uparrow}$ [10], $\lambda\upsilon$[13], λ_s [12], λ_d [11], λ_ζ [15], and λ_x [21]. All but the last of these calculi have been formulated with de Bruijn indices, λ_x uses variable names.

The F-calculus was introduced by J-Y.Girard [8] and independantly reinvented by J.C.Reynolds [18]. It consists in extending the simply typed lambda calculus with parametric polymorphism. Its interest from a programming point of view is that although all typable terms are terminating the computable functions which may be expressed is not restricted to the primitive recursive functions but also includes, for example, Ackermann's function. We shall study an explicit substitution formulation of F based on λ_x.

As regards related work in his thesis [5] Bloo defines Explicit Pure Type Systems (EPTS), a formalism which consists in extending the Pure Type Systems (PTS) [3] with explicit substitutions in a named variable setting. He proves that $\mathcal{R}$ is strongly normalizing if and only if $\mathcal{R}x$ is strongly normalizing, where $\mathcal{R}$ is a PTS and $\mathcal{R}x$ is its explicit substitution version in the EPTS formalism, each with its corresponding notion of reduction. On the other hand, failure of subject reduction for Fx (i.e. the explicit substitution version of the polymorphic lambda calculus F) is illustrated by providing counterexamples[1]. Our presentation differs substantially in that our proof of strong normalization relies on the reducibility technique and thus does not require the intermediate result of strong normalization of F. Also, our calculus uses two operators of explicit substitutions: one for terms and one for types. Since one of the primary goals intended by the introduction of explicit substitution operators is to "implement" the notion of metalevel substitution this has the benefit that unnecesary propagation of

[1] And other systems too.

substitutions is avoided in that substitutions for terms do not propagate to types. Furthermore, the subject reduction property is shown to hold for F_{es}.

In [16] the $\lambda\Pi_{\mathcal{L}}$-calculus is presented as a theory of dependent types with explicit substitutions in a de Bruijn setting. The underlying explicit substitution calculus is a left-linear variant of $\lambda\sigma$ [17]. Subject reduction is achieved by using type reminders: when a substitution traverses an abstraction or a product, the type of the binder is recorded in the substitution. Weak normalization of the typed $\lambda\Pi_{\mathcal{L}}$-calculus is proved using the reducibility technique. The notion of reduction that is proved weakly normalizing is beta reduction followed by reduction to $\Pi_{\mathcal{L}}$-normal form, that is, to the explicit substitution subcalculus normal form. Although we deal with a calculus which does not involve dependent types we prove strong (in contrast to weak) normalization of our calculus. This means that all redexes must be considered since any reduction strategy must be shown to terminate. The difficulties that arise are dealt with by making use of a characterization of untyped strongly λ_x-normalizing terms introduced in [4].

The structure of this paper is as follows: Section 2 recalls very briefly some basic notions of λ-calculus and the λ_x-calculus. Section 3 introduces the F_{es}-calculus as an untyped calculus and shows some properties concerning the explicit substitution subcalculus. Section 4 presents de typing rules for typed F_{es} and then concentrates on the subject reduction property. In Section 5 and after introducing some basic definitions, we prove strong F_{es}-normalization of all typable terms. Finally we conclude and suggest future research directions.

The full version of this work is available by anonymous ftp at:
`ftp://ftp.lri.fr/LRI/articles/bonelli/westapp99full.ps.gz`.

2 Lambda calculus and explicit substitutions

In this section we briefly introduce the (untyped) lambda calculus [2] and the calculus of explicit substitutions λ_x [4] in order to fix notation. Let $\mathcal{X}$ be a denumerably infinite set of variables. Then the set of terms of the lambda-calculus, denoted $\mathcal{T}_\lambda$, is defined by the following grammar: $M ::= x \mid MM \mid \lambda x.M$ where $x \in \mathcal{X}$. We use letters $x, y, z, \dots$ for variables and $M, N, O, \dots$ for terms. The notion of free and bound variables are defined as usual and the following variable convention is adopted: names of bound variables are different from the names of free variables, and moreover, different occurrences of the abstraction operator have different binding variables. A more precise formulation of α-conversion shall be given in Section 3. The β-rule is stated as $(\lambda x.M)N \rightarrow_\beta M\{x \leftarrow N\}$. β-reduction is the contextual closure of the β-rule. The expression $M\{x \leftarrow N\}$ denotes the term resulting from substituting N for every free occurrence of the variable x in M. This operation takes place at the *metalevel*, thus it is external to the calculus.

A term M is said to be *strongly β-normalizing* if all β-reduction sequences terminate. The set of terms of the λ-calculus which are strongly β-normalizing shall be denoted SN_β. Also, we shall use the symbol $\equiv$ to denote syntactical equality modulo α-conversion (sometimes $\equiv_{L\ n}$ when the result follows by applying Lemma n) and $\stackrel{*}{\rightarrow}_\beta$ for the reflexive and transitive closure of $\rightarrow_\beta$. For the subterm relation we use $\subseteq$, i.e. $M \subseteq N$ is read as "M is a subterm of N".

Two well-known properties are:

Lemma 2.1 If $M \rightarrow_\beta M'$, then for every term N we have

1. $M\{x \leftarrow N\} \rightarrow_\beta M'\{x \leftarrow N\}$.
2. $N\{x \leftarrow M\} \stackrel{*}{\rightarrow}_\beta N\{x \leftarrow M'\}$.

Lemma 2.2 (Substitution lemma) $M\{x \leftarrow N\}\{y \leftarrow O\} \equiv M\{y \leftarrow O\}\{x \leftarrow N\{y \leftarrow O\}\}$ if $x \notin FV(O)$.

The λ_x calculus [21, 5] is a calculus of explicit substitutions for the λ-calculus formulated in a named variable setting. The operation of substitution is incorportated at the object level, thus the β reduction rule is transformed into the following *Beta* reduction rule:

$$(\lambda x.M)N \ \rightarrow_{Beta} \ M\langle x := N\rangle$$

where the operator $.\langle . := .\rangle$ is a new operator in the calculus. Thus rules describing its behaviour must be introduced.

Given a denumerably infinite set of variables $\mathcal{X}$, the set of terms of λ_x is defined by the grammar: $M ::= x \mid MM \mid \lambda x.M \mid M\langle x := M\rangle$ where $x \in \mathcal{X}$. The subset $\mathcal{T}_\lambda \subset \mathcal{T}_{\lambda_x}$ are called *pure terms.* The aforementioned variable convention is maintained with the additional observation that the variable x in the term $M_1\langle x := M_2\rangle$ binds the free occurrences of x in M_1. The rules of the λ_x calculus are

$$\begin{array}{llll}
(\lambda x.M)N & \rightarrow_{Beta} & M\langle x := N\rangle & \\
(MN)\langle x := P\rangle & \rightarrow_{xapp} & M\langle x := P\rangle N\langle x := P\rangle & \\
(\lambda y.M)\langle x := P\rangle & \rightarrow_{xlam} & \lambda y.M\langle x := P\rangle & \\
x\langle x := P\rangle & \rightarrow_{xvar} & P & \\
y\langle x := P\rangle & \rightarrow_{xgc} & y & x \not\equiv y
\end{array}$$

The λ_x calculus without the *Beta* rule is called the *substitution subcalculus of* λ_x and is denoted by x. This subcalculus is strongly normalizing and confluent, and its normal forms are pure terms. Thus if $M \in \mathcal{T}_{\lambda_x}$ then we use $x(M)$ to denote its unique x-normal form.

The following result shows that the substitution subcalculus x correctly implements metalevel substitution.

Lemma 2.3 $x(M\langle y := N\rangle) \equiv x(M)\{y \leftarrow x(N)\}$.

The set of terms which are strongly λ_x-normalizing shall be denoted SN_{λ_x}. The following definition and results are taken from [4].

Definition 2.1 $\lambda_x^{<\infty} =_{def} \{M \in \mathcal{T}_{\lambda_x} | \forall\ N \subseteq M,\ x(N) \in SN_\beta\}$

Lemma 2.4 below characterizes the set of strongly λ_x-normalizing terms. It was introduced in order to prove the property of Preservation of Strong Normalization (PSN). This property states that $SN_\beta \subset SN_{\lambda_x}$ (i.e. terms in $\mathcal{T}_\lambda$ which are strongly β-normalizing are strongly λ_x-normalizing). This is an essential property for calculi of explicit substitution to be well-behaved and is not trivial ([14]). The technique introduced in [4] allows to prove PSN for a variety of calculi of explicit substitution such as λ_s [12], λ_υ [13] and λ_d [11].

Lemma 2.4 $\lambda_x^{<\infty} = SN_{\lambda_x}$.

Proof. The inclusion $SN_{\lambda_x} \subseteq \lambda_x^{<\infty}$ follows from the fact that λ_x can simulate β-reductions. The other is proved using a recursive path ordering on labelled terms (see [4] for details).

∎

3 The untyped F-calculus with explicit substitutions

In this section we introduce the polymorphic lambda calculus [9] with explicit substitutions. This reduction system which we shall call F_{es} will first be introduced as an untyped calculus, section 4 shall deal with its typed version.

Let $\mathcal{V}$ be an infinite set of type variables $s, t, u, \ldots$, and $\mathcal{X}$ be an infinite set of term variables (referred to simply as variables) $x, y, z, \ldots$.

Definition 3.1 The set of *types* and *terms* (referred to, without distinction, as raw terms) of the F_{es} calculus is defined by the following two grammars, respectively:

$$\begin{array}{ll}
\textit{types} & \sigma ::= t \mid \sigma_1 \supset \sigma_2 \mid \forall t.\sigma \mid \sigma[t := \sigma] \\
\textit{terms} & M ::= x \mid \lambda x : \sigma.M \mid \Lambda t.M \mid MN \mid M\sigma \mid M\langle x := N\rangle \mid M[t := \sigma]
\end{array}$$

The operator $.\langle. := .\rangle$ is called the *term substitution operator* and the $.[. := .]$ the *type substitution operator*. A term which does not contain occurrences of a (term or type) substitution operator is called a *pure term*. Likewise, a type which does not contain occurrences of the type substitution operator is called a *pure type*.

We denote with $\mathcal{RT}$ the set of all raw terms, i.e. those generated by the *terms* and *types* grammars, and $\mathcal{T}$ the set of terms in $\mathcal{RT}$, i.e. those generated by the *terms* grammar.

Letters $\rho, \sigma, \tau, \ldots$ are used for types, letters $M, N, O, P, Q, \ldots$ are used for terms and letters $A, B, C, \ldots$ are used for raw terms.

The set of free variables of a term M is denoted $FV(M)$ and the set of free type variables of a raw term A is denoted $FTV(A)$. We also use $BV(M)$ and $BTV(A)$ to denote the set of bound variables of M and bound type variables of A, respectively. Also, we sometimes use $BTV(A, B)$ as a shorthand for $BTV(A) \cup BTV(B)$. These notions are defined as usual; we give below the definition of FTV as an example.

Definition 3.2 Let A be a raw term. The set of *free type variables of A* is defined as follows:

$$
\begin{array}{llllll}
FTV(t) & =_{def} & \{t\} & FTV(\Lambda t.P) & =_{def} & FTV(P) \setminus \{t\} \\
FTV(\sigma \supset \tau) & =_{def} & FTV(\sigma) \cup FTV(\tau) & FTV(PQ) & =_{def} & FTV(P) \cup FTV(Q) \\
FTV(\forall t.\sigma) & =_{def} & FTV(\sigma) \setminus \{t\} & FTV(P\sigma) & =_{def} & FTV(P) \cup FTV(\sigma) \\
FTV(\sigma[t := \tau]) & =_{def} & (FTV(\sigma) \setminus \{t\}) \cup FTV(\tau) & FTV(P\langle x := Q\rangle) & =_{def} & FTV(P) \cup FTV(Q) \\
FTV(x) & =_{def} & \emptyset & FTV(P[t := \sigma]) & =_{def} & (FTV(P) \setminus \{t\}) \cup FTV(\sigma) \\
FTV(\lambda x : \sigma.P) & =_{def} & FTV(\sigma) \cup FTV(P) & & &
\end{array}
$$

When dealing with calculi of explicit substitution, and in order to obtain first order rewriting systems, it is not uncommon to use de Bruijn indice notation (e.g. [1],[13],[11],[12]). Since our main objective is to concentrate on the properties of subject reduction and strong normalization we have chosen to use the variable name based λ_x in order to minimize the "noise" introduced by updating (i.e. indice updating in de Bruijn indice calculi) and thus provide the reader with a more intuitive setting.

We recall the familiar notions of substitution and α-conversion for terms (types are treated similarly).

$$
\begin{array}{lll}
x\{x \leftarrow N\} & =_{def} & N \\
y\{x \leftarrow N\} & =_{def} & y, \text{ if } x \not\equiv y \\
(M_1 M_2)\{x \leftarrow N\} & =_{def} & M_1\{x \leftarrow N\} M_2\{x \leftarrow N\} \\
(\lambda x.M)\{x \leftarrow N\} & =_{def} & \lambda x.M \\
(\lambda y.M)\{x \leftarrow N\} & =_{def} & \lambda y'.M\{y \leftarrow y'\}\{x \leftarrow N\}, \text{ if } y' \notin FV(N) \cup \{x\} \cup (FV(M) \setminus \{y\}) \\
(M_1\langle x := M_2\rangle)\{x \leftarrow N\} & =_{def} & M_1\langle x := M_2\rangle \\
(M_1\langle y := M_2\rangle)\{x \leftarrow N\} & =_{def} & M_1\{y \leftarrow y'\}\{x \leftarrow N\}\langle y' := M_2\{x \leftarrow N\}\rangle, \\
 & & \text{if } y' \notin FV(N) \cup \{x\} \cup (FV(M_1) \setminus \{y\})
\end{array}
$$

Syntactial equality modulo α-conversion is thus the smallest equivalence relation verifying:

$$
\begin{array}{lll}
\text{if } M \equiv P \text{ and } N \equiv Q & \text{then} & MN \equiv PQ \\
\text{if } M \equiv N \text{ and } y \notin FV(N) \setminus \{x\} & \text{then} & \lambda x.M \equiv \lambda y.N\{x \leftarrow y\} \\
\text{if } M \equiv P, N \equiv Q \text{ and } y \notin FV(N) \setminus \{x\} & \text{then} & M\langle x := N\rangle \equiv P\{x \leftarrow y\}\langle y := Q\rangle
\end{array}
$$

As in the untyped lambda calculus we shall adopt the following variable convention.

Convention 3.1 (Variable convention) We assume that the names of bound (term or type) variables shall always be chosen so that they differ from the free ones. Moreover, each occurrence of a (term or type) abstraction operator has a different binding variable.

Definition 3.3 (F_{es} and subsystems) The F_{es}-calculus is given by the rules in figure 1. The F_{es}-calculus without the rules *Beta*2 and τ*Beta* is referred to as the *ES* reduction system[2]. The F_{es}-calculus without the *Beta*2 rule is called the $ES^\bullet$ reduction system. The third group of rules is called the *ZT* reduction system.

We shall now prove that the $ES^\bullet$ reduction system is strongly normalizing (SN). Intuitively, one notes that the substitution calculus *ES* "pushes" substitution operators deeper and deeper into a term/type until it reaches a variable, in which case it is performed or eliminated. As for the rule τ*Beta* one notes that it eliminates an occurrence of the binder Λ. Thus we shall first prove that the *ES* reduction system is SN by interpreting it into a simpler calculus equipped with a well-founded reduction notion and showing that reduction is preserved. Then for the full system $ES^\bullet$ we use this result plus Lemmas A.1 (see appendix) and 3.5.

Definition 3.4 The set of terms constructed from the alphabet $\mathcal{A} = \{\star, \lambda, \Lambda, \supset, ., [\![]\!]\}$, denoted $\mathcal{S}$, is defined by the grammar $a ::= \star \mid \lambda(a, a) \mid \Lambda a \mid a.a \mid a[\![a]\!] \mid a \supset a$. Also, we define the following well-founded ordering $[\![]\!] \gg . \gg \star, \lambda, \Lambda, \supset$ on the alphabet $\mathcal{A}$.

Assuming all symbols of $\mathcal{A}$ have multiset status, this well-founded ordering induces a well-founded Recursive Path Ordering (RPO) on the full set of terms $\mathcal{S}$[6] (see appendix).

Definition 3.5 We define the translation $\mathcal{R}_1(.) : \mathcal{RT} \longrightarrow \mathcal{S}$ as

[2] A reduction system R is sometimes also referred to as a R-calculus

$$
\begin{array}{llll}
(\lambda x:\sigma.M)N & \rightarrow_{Beta2} & M\langle x := N\rangle & \\
(\Lambda t.M)\tau & \rightarrow_{\tau Beta} & M[t := \tau] & \\
 & & & \\
(MN)\langle x := P\rangle & \rightarrow_{zapp} & M\langle x := P\rangle N\langle x := P\rangle & \\
(\lambda y:\sigma.M)\langle x := P\rangle & \rightarrow_{zlam} & \lambda y:\sigma.M\langle x := P\rangle & \\
x\langle x := P\rangle & \rightarrow_{zvar} & P & \\
y\langle x := P\rangle & \rightarrow_{zgc} & y & x \not\equiv y \\
(M\sigma)\langle x := P\rangle & \rightarrow_{zappt} & M\langle x := P\rangle\sigma & \\
(\Lambda t.M)\langle x := P\rangle & \rightarrow_{zlamt} & \Lambda t.M\langle x := P\rangle & \\
 & & & \\
(MN)[t := \tau] & \rightarrow_{ztapp} & M[t := \tau]N[t := \tau] & \\
(\lambda y:\sigma.M)[t := \tau] & \rightarrow_{ztlam} & \lambda y:\sigma[t := \tau].M[t := \tau] & \\
t[t := \tau] & \rightarrow_{ztvar1} & \tau & \\
\sigma[t := \tau] & \rightarrow_{ztgc} & \sigma & t \notin FTV(\sigma) \\
x[t := \tau] & \rightarrow_{ztvar2} & x & \\
(M\sigma)[t := \tau] & \rightarrow_{ztappt} & M[t := \tau]\sigma[t := \tau] & \\
(\Lambda u.M)[t := \tau] & \rightarrow_{ztlamt} & \Lambda u.M[t := \tau] & \\
(\sigma_1 \supset \sigma_2)[t := \tau] & \rightarrow_{zt} & \sigma_1[t := \tau] \supset \sigma_2[t := \tau] & \\
(\forall u.\sigma)[t := \tau] & \rightarrow_{zt} & \forall u.\sigma[t := \tau] &
\end{array}
$$

Figure 1: The F_{es}-calculus

$$
\begin{array}{lll}
\mathcal{R}_1(x) = \mathcal{R}_1(t) & =_{def} & \star \\
\mathcal{R}_1(MN) & =_{def} & \mathcal{R}_1(M).\mathcal{R}_1(N) \\
\mathcal{R}_1(M\sigma) & =_{def} & \mathcal{R}_1(M).\mathcal{R}_1(\sigma) \\
\mathcal{R}_1(\lambda x:\sigma.M) & =_{def} & \lambda(\mathcal{R}_1(\sigma),\mathcal{R}_1(M)) \\
\mathcal{R}_1(\Lambda t.M) & =_{def} & \Lambda\mathcal{R}_1(M)
\end{array}
\qquad
\begin{array}{lll}
\mathcal{R}_1(M\langle x := N\rangle) & =_{def} & \mathcal{R}_1(M)[\![\mathcal{R}_1(N)]\!] \\
\mathcal{R}_1(M[t := \tau]) & =_{def} & \mathcal{R}_1(M)[\![\mathcal{R}_1(\tau)]\!] \\
\mathcal{R}_1(\sigma \supset \tau) & =_{def} & \mathcal{R}_1(\sigma) \supset \mathcal{R}_1(\tau) \\
\mathcal{R}_1(\forall t.\sigma) & =_{def} & \Lambda\mathcal{R}_1(\sigma)
\end{array}
$$

Lemma 3.2 The ES reduction system is strongly normalizing on $\mathcal{RT}$.

Proof.
We show that if A, B are raw terms in $\mathcal{RT}$ such that $A \rightarrow_{ES} B$ then we have $\mathcal{R}_1(A) \succ_{\mathcal{S}} \mathcal{R}_1(B)$. This is done by induction on the reduction $A \rightarrow_{ES} B$.

∎

Since the ES reduction system is also locally confluent, by Newman's Lemma we may conclude that it is confluent. Thus if $A \in \mathcal{RT}$ then we shall use $ES(A)$ to denote the unique ES-normal form of A. Since the ZT subcalculus is SN (it is included in ES) and locally confluent then by Newman's Lemma we may also conclude that it is confluent. Thus if σ is a type then we use $ZT(\sigma)$ to denote its unique ZT-normal form. Also, we shall use $=_{ZT}$ to denote the reflexive, symmetrical and transitive closure of one step reduction in ZT.

Remark 3.3 One may verify in the style of [19] that if σ is a type then $ZT(\sigma)$ is a pure type and if A is a raw term then $ES(A)$ is a pure term or type. Note that if $M \in \mathcal{T}$ then $ZT(M)$ not only may not be a pure term (for example if it has occurrences of the term substitution operator) but also may not have pure types since term substitutions may block the performing or discarding of type substitutions.

Before proceeding to the SN property of $ES^{\bullet}$, we mention some technical properties of ZT and ES which shall be used later.

Lemma 3.4 Let ρ, σ, τ be pure types, P and Q terms in $\mathcal{T}$, and τ_1, τ_2 any types.

1. $\rho\{t \leftarrow \tau\}\{u \leftarrow \sigma\{t \leftarrow \tau\}\} \equiv \rho\{u \leftarrow \sigma\}\{t \leftarrow \tau\}$ if $u \notin FTV(\tau)$.
2. $ZT(\tau_1[t := \tau_2]) \equiv ZT(\tau_1)\{t \leftarrow ZT(\tau_2)\}$.

3. $ES(P\langle x := Q\rangle) \equiv ES(P)\{x \leftarrow ES(Q)\}$.

4. $ES(P[t := \tau_1]) \equiv ES(P)\{t \leftarrow ZT(\tau_1)\}$.

Proof.

The first item is proved by induction on ρ. The second item is first proved for τ_1 and τ_2 pure types and then using remark 3.3. Item three is dealt with in a similar fashion. Item four is similar, but uses the observation that for a type σ, we have $ES(\sigma) \equiv ZT(\sigma)$ and also uses item (2).

∎

Next we extend the SN property of ES to the reduction system $ES^\bullet$, i.e. ES with the additional rule $\tau Beta$. For this we make use of Lemma A.1 (see appendix). We thus define a translation $\mathcal{R}_2(.)$ from terms in $\mathcal{RT}$ to $\mathcal{S}$ such that the conditions of Lemma A.1 are met.

Definition 3.6 We define the translation $\mathcal{R}_2(.) : \mathcal{RT} \longrightarrow \mathcal{S}$ as

$$\begin{array}{lll@{\qquad}lll}
\mathcal{R}_2(x) = \mathcal{R}_2(\sigma) & =_{def} & \star & \mathcal{R}_2(\Lambda t.M) & =_{def} & \Lambda \mathcal{R}_2(M) \\
\mathcal{R}_2(MN) & =_{def} & \mathcal{R}_2(M).\mathcal{R}_2(N) & \mathcal{R}_2(M\langle x := N\rangle) & =_{def} & \mathcal{R}_2(M)[\![\mathcal{R}_2(N)]\!] \\
\mathcal{R}_2(M\sigma) & =_{def} & \mathcal{R}_2(M) & \mathcal{R}_2(M[t := \tau]) & =_{def} & \mathcal{R}_2(M) \\
\mathcal{R}_2(\lambda x : \sigma.M) & =_{def} & \lambda(\mathcal{R}_2(\sigma), \mathcal{R}_2(M)) & & &
\end{array}$$

Lemma 3.5 Let M and N be terms in $\mathcal{T}$. Then

1. If $M \rightarrow_{\tau Beta} N$ then $\mathcal{R}_2(M) \succ_{\mathcal{S}} \mathcal{R}_2(N)$, and

2. If $M \rightarrow_{ES} N$ then $\mathcal{R}_2(M) \succeq_{\mathcal{S}} \mathcal{R}_2(N)$.

Proof.

In both items we use induction on the reduction $M \rightarrow_r N$ with $r \equiv \tau Beta$ or $r \in ES$. Let us consider the first item. Suppose the reduction is at the root. Then $M \equiv (\Lambda t.P)\tau$ and $N \equiv P[t := \tau]$ and $\mathcal{R}_2(M) \equiv \Lambda \mathcal{R}_2(P)$ and $\mathcal{R}_2(N) \equiv \mathcal{R}_2(P)$ and hence we obtain the desired result. If the reduction is internal then we apply the inductive hypothesis and the monotonicity of RPOs.

The second item is similar. Note that if $M \rightarrow_{ZT} N$ or $M \rightarrow_{zappt} N$ then $\mathcal{R}_2(M) \equiv \mathcal{R}_2(N)$. ∎

Corollary 3.6 The $ES^\bullet$ reduction system is strongly normalizing on the set of terms in $\mathcal{T}$.

Definition 3.7 (The z and λ_z reduction systems) Let us denote the reduction system obtained by taking ES and eliminating the subsystem ZT and rules $zappt$ and $zlamt$, z. We define λ_z as z and the rewrite rule $(Beta2)$. We recall these rules below.

$$\lambda_z \left\{ \begin{array}{l} (\lambda x : \sigma.M)N \quad \rightarrow_{Beta2} \quad M\langle x := N\rangle \\ z \left\{ \begin{array}{lll} (MN)\langle x := P\rangle & \rightarrow_{zapp} & M\langle x := P\rangle N\langle x := P\rangle \\ (\lambda y : \sigma.M)\langle x := P\rangle & \rightarrow_{zlam} & \lambda y : \sigma.M\langle x := P\rangle \\ x\langle x := P\rangle & \rightarrow_{zvar} & P \\ y\langle x := P\rangle & \rightarrow_{zgc} & y \end{array} \right. \end{array} \right. \qquad x \not\equiv y$$

The λ_z and z calculi are the explicit substitution calculus for the simply typed lambda calculus (typed λ_x) and its subcalculus for computing explicit substitutions (x in [5]), respectively. Note that from Lemma 3.2 we have that z is strongly normalizing.

4 The typed F-calculus with explicit substitutions

In this section we introduce the typing rules for F_{es}. As already mentioned, in [5] explicit substitutions were added to the Pure Type Systems formalism [3]. Although we deal only with an explicit substitution version of F, already for this calculus subject reduction was shown to fail. The counterexample exhibited in [5] is the pure term $M \equiv (\Lambda s.(\lambda f : s \supset s.\lambda x : s.fx))t$. Due to lack of space we refer the reader interested in Explicit Pure Type Systems to [5]. After proving subject reduction for F_{es}, we conclude by showing how the typing rules deal with M.

Definition 4.1 (Type assignment) A *type assignment* is a finite set of pairs $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ such that the variables are pairwise distinct and each σ_i is a (not necessarily pure) type. If the types σ_i with $i \in 1..n$ are pure then Γ is said to be a *pure type assignment*. Type assignments are denoted with capital greek letters $\Gamma, \Delta, \ldots$.

Definition 4.2 (Type judgement) A *type judgement* is an expression of the form $\Gamma \rhd M : \sigma$ where where Γ is a type assignment, M is a term in $\mathcal{T}$ and σ is a *pure* type. If, moreover, Γ is a pure type assignment and M is a pure term then $\Gamma \rhd M : \sigma$ is said to be a *pure type judgement*.

We shall sometimes use $\Gamma(x_i) = \sigma_i$ to denote the type assigned to the variable x_i by the type assignment $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$. Also we use the abbreviation $t \notin FTV(\Gamma)$ for $t \notin FTV(\Gamma(y))$ for every $y \in Dom(\Gamma)$.

Definition 4.3 Let $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ be a type assignment, t a type variable and τ a type. Then $\Gamma_{[t:=\tau]}$ is any member of the set of type assignments $\{\ \{x_1 : \sigma'_1, \ldots, x_n : \sigma'_n\}\ \mid\ \sigma'_i =_{ZT} \sigma_i[t := \tau]\ \textit{for all}\ i \in 1..n\}$.

Definition 4.4 (Derivable type judgements) The set of *derivable type judgements* in F_{es} is defined by the typing rules given in figure 2. The rule $(tabs)$ is subject to the following restriction: $t \notin FTV(\Gamma)$.

$$\frac{}{\Gamma \rhd x : ZT(\sigma)}\,(var) \qquad \textit{if}\ x : \sigma \in \Gamma$$

$$\frac{\Gamma \rhd M : \sigma \supset \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd MN : \tau}\,(app) \qquad \frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd (\lambda x : \sigma.M) : ZT(\sigma) \supset \tau}\,(abs)$$

$$\frac{\Gamma \rhd M : \forall t.\sigma}{\Gamma \rhd M\tau : \sigma\{t \leftarrow ZT(\tau)\}}\,(tapp) \qquad \frac{\Gamma \rhd M : \sigma}{\Gamma \rhd (\Lambda t.M) : \forall t.\sigma}\,(tabs)$$

$$\frac{\Gamma, x : \sigma \rhd M : \tau \quad \Gamma \rhd N : ZT(\sigma)}{\Gamma \rhd M\langle x := N\rangle : \tau}\,(subs) \qquad \frac{\Gamma \rhd M : \sigma}{\Gamma_{[t:=\tau]} \rhd M[t := \tau] : \sigma\{t \leftarrow ZT(\tau)\}}\,(tsubs)$$

Figure 2: Typing rules of the F_{es}-calculus

Definition 4.5 A term $M \in \mathcal{T}$ such that there exists a type assignment Γ and a pure type σ such that the judgement $\Gamma \rhd M : \sigma$ is derivable in the type system of figure 2 is called *typable*. The set of typable terms is denoted $\mathcal{T}^{\rightarrow}$.

In a typing judgement $\Gamma \rhd M : \sigma$ the type σ assigned to the term M is always a pure type.

Example 4.1 Two examples of type derivations follow.

$$\dfrac{\dfrac{\dfrac{\dfrac{f : s \supset s, x : s \rhd f : s \supset s \quad f : s \supset s, x : s \rhd x : s}{f : s \supset s, x : s \rhd fx : s}\,(app)}{f : s \supset s \rhd \lambda x : s.fx : s \supset s}\,(abs)}{f : t \supset t \rhd (\lambda x : s.fx)[s := t] : t \supset t}\,(tsubs)}{\rhd(\lambda f : t \supset t.(\lambda x : s.fx)[s := t]) : (t \supset t) \supset (t \supset t)}\,(abs)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{f : s \supset s, x : s \rhd f : s \supset s \quad f : s \supset s, x : s \rhd x : s}{f : s \supset s, x : s \rhd fx : s}\,(app)}{f : s \supset s \rhd \lambda x : s.fx : s \supset s}\,(abs)}{f : (s \supset s)[s := t] \rhd (\lambda x : s.fx)[s := t] : t \supset t}\,(tsubs)}{\rhd\lambda f : (s \supset s)[s := t].(\lambda x : s.fx)[s := t] : (t \supset t) \supset (t \supset t)}\,(abs)$$

Note that for the application of $(tsubs)$ in the left type derivation we use $(t \supset t) =_{ZT} (s \supset s)[s := t]$.

The typing rules of the usual (Church style presentation) of F is given by all rules in figure 2 without rules $(subs)$ and $(tsubs)$ and restricting all type judgements to pure type judgements. Thus when speaking of derivable type judgements in F we shall explicitly mention so, otherwise we shall assume that the type system referred to is F_{es}.

In devising a rule for typing type substitutions our first intuition is that it should resemble an application of the rule for type variable abstraction followed by that of type variable application. But the introduction of a type substitution in a term M presents some difficulties. Consider a derivable premise $\Gamma \rhd M : \sigma$ and suppose we add the type substitution $[t := \tau]$ to M in order to obtain $\Gamma \rhd M[t := \tau]$. Then we immediately note that the context Γ should not remain the same for if $x \in FV(M)$ then the type assigned to x by the type assignment Γ has now fallen under the scope of the type substitution. For example, consider the application of $(tsubs)$ in the left hand derivation of example 4.1. The variable f, which has type $s \supset s$ in the premise, has now fallen under the scope of type substitution $.[s := t]$, so its type must be affected accordingly. This motivates the use of $\Gamma_{[t:=\tau]}$ in the rule $(tsubs)$.

An alternative approach could be to include the application of the type substitutions in the types of the variables in the type assignment (Γ) from the start. This requires some mecanism of control in order to ensure that the type substitution applied to the term M is the same as the one which has been applied to the type of the variable in the type assignment (and in the same order if more than one has been applied).

We shall first consider the relation between typing in F and typing in F_{es}. As expected, all terms typable in F are typable in F_{es}(the typing rules for F_{es} include those of F). And if a term is typable in F_{es} then its ES normal form is typable in F (Lemma 4.4).

The following result is used when showing that F has the subject reduction property and shall be used later.

Lemma 4.2 Let $\Gamma, x : \tau \rhd M : \sigma$ and $\Gamma \rhd N : \tau$ be pure type judgements. If $\Gamma, x : \tau \rhd M : \sigma$ and $\Gamma \rhd N : \tau$ are derivable in F then $\Gamma \rhd M\{x \leftarrow N\} : \sigma$ is derivable in F.

Proof.
By induction on the size of the derivation of $\Gamma, x : \tau \rhd M : \sigma$.

∎

The following lemma expresses that derivability is closed under type substitution in F.

Lemma 4.3 Let $\Gamma \rhd M : \sigma$ be a pure type judgement and τ a pure type. If $\Gamma \rhd M : \sigma$ is derivable in F and $BTV(M, \sigma, \Gamma) \cap FTV(\tau) = \emptyset$ then $\Gamma\{t \leftarrow \tau\} \rhd M\{t \leftarrow \tau\} : \sigma\{t \leftarrow \tau\}$ is derivable in F.

Proof.
By induction on the size of the derivation of $\Gamma \rhd M : \sigma$.

∎

Note that the $(subs)$ rule internalises lemma 4.2 and rule $(tsubs)$ internalises lemma 4.3.

Lemma 4.4 Let $\Gamma \rhd M : \sigma$ be a derivable type judgement in F_{es}. Then $ZT(\Gamma) \rhd ES(M) : \sigma$ is derivable in F.

Proof.
By induction on the derivation of $\Gamma \rhd M : \sigma$ using lemmas 4.2 and 4.3 and lemma 3.4 (2) and (3).

∎

In order to prove the subject reduction property for F_{es} we need two auxiliary results, the weakening lemma and the context reduction lemma.

Lemma 4.5 (Weakening) If $\Gamma \rhd M : \sigma$ is a derivable type judgement, and $x : \tau$ is such that $x \notin Dom(\Gamma) \cup BV(M)$, and $BTV(M) \cap FTV(\tau) = \emptyset$ then $\Gamma, x : \tau \rhd M : \sigma$ is a derivable type judgement.

Proof.
By induction on the size of the derivation of $\Gamma \rhd M : \sigma$.

∎

We shall need the following lemma for the subject reduction property, it states that reduction in contexts conserves typability.

Lemma 4.6 (Context reduction) Let $\Gamma, x : \tau \rhd P : \sigma$ be any derivable type judgement and suppose $\tau \rightarrow_{ZT} \tau'$, then $\Gamma, x : \tau' \rhd P : \sigma$ is a derivable type judgement.

Proof.
By induction on the length of the derivation of $\Gamma, x : \tau \rhd P : \sigma$.

∎

Lemma 4.7 (Subject reduction) Let $\Gamma \rhd M : \sigma$ be a derivable type judgement and suppose $M \to_{F_{es}} N$. Then $\Gamma \rhd N : \sigma$ is a derivable type judgement.

Proof.
By induction on the reduction $M \to_{F_{es}} N$ using lemmas 4.5 and 4.6 (see appendix).

∎

We finish this section with the counterexample ([5], Prop. 7.27) mentioned in the introduction that states that the EPTS $\mathcal{F}x$ does not verify the subject reduction property and we show how this situation is remedied.

Example 4.8 Consider the pure term $M \equiv (\Lambda s.(\lambda f : s \supset s.\lambda x : s.fx))t$. It is typable in F_{es}

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f : s \supset s, x : s \rhd f : s \supset s \quad f : s \supset s, x : s \rhd x : s}{f : s \supset s, x : s \rhd fx : s}\,(app)}{f : s \supset s \rhd \lambda x : s.fx : s \supset s}\,(abs)}{\rhd \lambda f : s \supset s.\lambda x : s.fx : (s \supset s) \supset (s \supset s)}\,(abs)}{\rhd \Lambda s.(\lambda f : s \supset s.\lambda x : s.fx) : \forall s.(s \supset s) \supset (s \supset s)}\,(tabs)}{\rhd (\Lambda s.(\lambda f : s \supset s.\lambda x : s.fx))t : (t \supset t) \supset (t \supset t)}\,(tapp)$$

Now we have the following reduction sequence:

$$\begin{array}{lll} M & \to_{\tau Beta} & (\lambda f : s \supset s.\lambda x : s.fx)[s := t] \\ & \to_{ztlam} & \lambda f : (s \supset s)[s := t].(\lambda x : s.fx)[s := t] \\ & \stackrel{*}{\to}_{ZT} & \lambda f : t \supset t.(\lambda x : s.fx)[s := t] \end{array}$$

The latter term is also typable as illustrated by example 4.1 (as well as the intermediate term $\lambda f : (s \supset s)[s := t].(\lambda x : s.fx)[s := t]$).

5 Strong Normalization

In this section we prove the strong normalization property for the typed F_{es} calculus. We follow the presentation given by Gallier in [7]. The idea is to define an erasing function that eliminates all typing information. We then show that if the term resulting from erasing all type information is strongly λ_x-normalizing then the original term is strongly F_{es}-normalizing. We conclude by showing that for every typable term in F_{es} the erased term obtained from it is strongly λ_x-normalizing.

5.1 Basic definitions

The following erasing functions removes all typing information from terms in $\mathcal{T}$.

Definition 5.1 (Erasing function) We define the following function $Erase(.) : \mathcal{T} \longrightarrow \mathcal{T}_{\lambda_x}$,

$$\begin{array}{lll@{\qquad}lll} Erase(x) & =_{def} & x & Erase(\Lambda t.M) & =_{def} & Erase(M) \\ Erase(MN) & =_{def} & Erase(M)Erase(N) & Erase(M\langle x := N\rangle) & =_{def} & Erase(M)\langle x := Erase(N)\rangle \\ Erase(\lambda x : \sigma.M) & =_{def} & \lambda x.Erase(M) & Erase(M[t := \sigma]) & =_{def} & Erase(M) \\ Erase(M\sigma) & =_{def} & Erase(M) & & & \end{array}$$

We recall that x is the substitution subcalculus of λ_x. It propagates the substitution operators until they reach the variables and then performs or eliminates them. The following lemma shows that a $\to_{F_{es}}$ reduction step may colapse to a λ_x-term via the erasing function or be simulated by a λ_x-reduction step.

Lemma 5.1 Let $M, N \in \mathcal{T}^{\to}$.

1. If $M \rightarrow_r N$ with $r \in (ES^\bullet \setminus z)$ then $Erase(M) \equiv Erase(N)$.

2. If $M \rightarrow_{\lambda_z} N$ then $Erase(M) \rightarrow_{\lambda_x} Erase(N)$.

Proof.
Both items are proved by induction on the reduction $M \rightarrow_r N$ with $r \in (ES^\bullet \setminus z)$ and $r \in \lambda_z$, respectively.

∎

Proposition 5.2 Let M be a term in $\mathcal{T}^{\rightarrow}$. If there is an infinite $\rightarrow_{F_{es}}$-reduction sequence starting from M, then there is an infinite $\rightarrow_{\lambda_x}$-reduction sequence starting from $Erase(M)$.

Proof.
Suppose we have an infinite $\rightarrow_{F_{es}}$-reduction sequence starting from M. Then since the reduction system $S = ES^\bullet \setminus z$ is strongly normalizing (as a consequence of corollary 3.6) the reduction sequence must have the form $M \equiv M_1 \overset{*}{\rightarrow}_S M_2 \rightarrow_{\lambda_z} M_3 \overset{*}{\rightarrow}_S M_4 \rightarrow_{\lambda_z} M_5 \ldots$ where the reductions $\rightarrow_{\lambda_z}$ occur infinitely many times. Then by lemma 5.1 we obtain an infinite λ_x-reduction sequence starting from $Erase(M)$.

∎

Our aim in the next subsection shall be to show that if M is a typable term in the polymorphic lambda calculus with explicit substitutions then the untyped λ_x term $Erase(M)$ is strongly $\rightarrow_{\lambda_x}$-normalizing, thus allowing us to conclude that M is strongly $\rightarrow_{F_{es}}$-normalizing. As the reader suspects, we shall use Tait's version of Girard's candidates of reducibility technique [22].

5.2 Strong normalization of F_{es}

In this section we apply Tait's version of the technique of candidates of reducibility in which the candidates are untyped λ_x-terms.

Definition 5.2 (Substitution) A *substitution* is a function $\phi : \mathcal{X} \longrightarrow \mathcal{T}_{\lambda_x}$ such that $\phi(x) \neq x$ for only finitely many $x \in \mathcal{X}$. The finite set $\{x \mid \phi(x) \neq x\}$ is called the *domain* of ϕ and is denoted $Dom(\phi)$. When ϕ has domain $Dom(\Gamma)$ for some type assignment Γ of interest we write $\phi : \Gamma \longrightarrow \mathcal{T}_{\lambda_x}$.

Definition 5.3 (P-substitution) A substitution ϕ is called a *p-substitution*[3] if for every $x_i \in Dom(\phi)$ we have $x_i \notin \bigcup_{x \in Dom(\phi)} FV(\phi(x))$.

Definition 5.4 (Extension of a substitution) Let ϕ be a (possibly p-) substitution and let $[i_1, \ldots, i_n]$ be any ordering of the variables in $Dom(\phi)$. Then an *extension* $\overline{\phi}$: $\mathcal{T}_{\lambda_x} \longrightarrow \mathcal{T}_{\lambda_x}$ of ϕ is defined as $\overline{\phi}(M) =_{def} M\langle x_{i_1} := \phi(x_{i_1})\rangle \ldots \langle x_{i_n} := \phi(x_{i_n})\rangle$.

Intuitively a p-substitution represents a parallel substitution. Usually when one defines a substitution ϕ and applies it to a term, the substitution process takes place "in parallel". Since λ_x incorporates sequential substitution into the object level and in order to apply the usual reducibility technique we give conditions (definition of p-substitutions) on the these sequential substitutions so that they may behave as desired.

Definition 5.5 Let C and D be sets of untyped terms in $\mathcal{T}_{\lambda_x}$. Then we define the *function space of C in D*, denoted $[C \rightarrow D]$, as $[C \rightarrow D] = \{M \in \mathcal{T}_{\lambda_x} \mid \forall N \in C,\ MN \in D\}$

We refer to $[\rightarrow]$ as the *function space constructor.*

Definition 5.6 A nonempty family of sets of (untyped) terms in $\mathcal{T}_{\lambda_x}$, $\mathcal{C}$, is called *type closed* if it verifies the following properties

1. every $C \in \mathcal{C}$ is nonempty,

2. $\mathcal{C}$ is closed under the function space constructor,

3. given any $\mathcal{C}$-indexed family $(A_C)_{C \in \mathcal{C}}$ of sets in $\mathcal{C}$, then $\bigcap_{C \in \mathcal{C}} A_C \in \mathcal{C}$, and

[3] P for "parallel"

4. every $C \in \mathcal{C}$ is closed under α-equivalence.

We shall use the notation $M \in \bigcup \mathcal{C}$ to say that M is an untyped λ_x-term in C for some member C of the family $\mathcal{C}$.

Definition 5.7 (Assignment) Let $\mathcal{C}$ be a type closed family. An *assignment* is a function $\eta : \mathcal{V} \longrightarrow \mathcal{C}$ such that $\eta(t) \not\equiv t$ for only finitely many $t \in \mathcal{V}$. Given a set $C \in \mathcal{C}$ and a type variable t we use $\eta[t := C]$ to denote the assignment such that for all $v \in \mathcal{V}$ we have $\eta[t := C](v) \equiv C$ if $v \equiv t$ and $\eta[t := C](v) \equiv \eta(v)$ otherwise.

Definition 5.8 Given an assignment $\eta : \mathcal{V} \longrightarrow \mathcal{C}$, for every pure type σ, the set $[\![\sigma]\!]\eta$ is defined as

$$\begin{array}{lll} [\![t]\!]\eta & =_{def} & \eta(t) \textit{ if } t \in \mathcal{V} \\ [\![(\sigma_1 \supset \sigma_2)]\!]\eta & =_{def} & [[\![\sigma_1]\!]\eta \rightarrow [\![\sigma_2]\!]\eta] \\ [\![\forall t.\sigma]\!]\eta & =_{def} & \bigcap_{C \in \mathcal{C}} [\![\sigma]\!]\eta[t := C] \end{array}$$

Lemma 5.3 Given two assignments $\eta_1 : \mathcal{V} \longrightarrow \mathcal{C}$ and $\eta_2 : \mathcal{V} \longrightarrow \mathcal{C}$, for every pure type σ, if η_1 and η_2 agree on $FTV(\sigma)$ then $[\![\sigma]\!]\eta_1 = [\![\sigma]\!]\eta_2$.

Proof.
By induction on σ ([7]). ∎

In the following lemma we assume by the variable convention that the bound type variables in σ do not occur free in τ.

Lemma 5.4 Let σ and τ be pure types. Then for every assignment $\eta : \mathcal{V} \longrightarrow \mathcal{C}$ we have $[\![\sigma\{t \leftarrow \tau\}]\!]\eta = [\![\sigma]\!]\eta[t := [\![\tau]\!]\eta]$.

Proof.
By induction on σ ([7]). ∎

The following lemma is referred to as "Girard's trick" in [7].

Lemma 5.5 Let $\mathcal{C}$ be a type closed family. Then for every assignment $\eta : \mathcal{V} \longrightarrow \mathcal{C}$, for every pure type σ, we have $[\![\sigma]\!]\eta \in \mathcal{C}$.

Proof.
By induction on the structure of the types (see [7]). ∎

In order to prove strong normalization on typable terms we impose some conditions on the members of a type closed family.

Before proceeding a comment on notation. If $M, N_1, \ldots, N_m \in \mathcal{T}_{\lambda_x}$ and $x_1, \ldots, x_m \in \mathcal{X}$ then we use $M^\star$ to denote the term $M\langle x_1 := N_1\rangle \ldots \langle x_m := N_m\rangle$.

Definition 5.9 We say that a family $\mathcal{C}$ of sets of untyped λ_x-terms is a *family of candidates of reducibility* iff it is type closed and satisfies the following. For every set $C \in \mathcal{C}$ we have,

R1. For every variable $x \in \mathcal{X}$, $x \in C$.

R2. For all $M, N \in \mathcal{T}_{\lambda_x}$, if $M\langle x := N\rangle \in C$ then $(\lambda x.M)\ N \in C$.

R3. For all $M \in \bigcup \mathcal{C}$ and for all $m \geq 0$, and $N_1, \ldots, N_m \in \bigcup \mathcal{C}$ if $M^\star \in C$ then $x\langle x := M\rangle^\star \in C$

R4. For all $M, N \in \bigcup \mathcal{C}$ and for every variable $x \in \mathcal{X}$ and for all $m \geq 0$ and $N_1, \ldots, N_m \in \bigcup \mathcal{C}$ if $M^\star \in C$ and $x \notin FV(M)$ then $M\langle x := N\rangle^\star \in C$.

R5. For all $M_1, M_2 \in \mathcal{T}_{\lambda_x}$, for every variable $x \in \mathcal{X}$ and for all $m \geq 0$, and $N, N_1, \ldots, N_m \in \bigcup \mathcal{C}$ if $(M_1\langle x := N\rangle M_2\langle x := N\rangle)^\star \in C$ then $(M_1 M_2)\langle x := N\rangle^\star \in C$.

R6. For all $M \in \mathcal{T}_{\lambda_x}$, for every variable $x \in \mathcal{X}$, and for all $m \geq 0$, and $N, N_1, \ldots, N_m \in \bigcup \mathcal{C}$, if $(\lambda y.M\langle x := N\rangle)^\star \in C$ and $y \notin FV(N)$ then $(\lambda y.M)\langle x := N\rangle^\star \in C$.

R7. For all $M, N \in \mathcal{T}_{\lambda_x}$, for all $m \geq 0$, for all $P, N_1, \ldots, N_m \in \bigcup \mathcal{C}$, if $y \notin FV(P)$ and $M\langle x := P\rangle\langle y := N\langle x := P\rangle\rangle^\star \in C$ then $M\langle y := N\rangle\langle x := P\rangle^\star \in C$.

Condition (R7) is the explicit version of the substitution lemma (lemma 2.2).

Now we are in conditions to prove (basically) that if a term in $\mathcal{T}$ is typable with type σ then its image via any extension of a p-substitution ϕ (satisfying the conditions of the lemma) is in the member set of the family $\mathcal{C}$ interpreting the type σ.

Lemma 5.6 Let $\mathcal{C}$ be a family of candidates of reducibility. For every derivation of $\Gamma \rhd M : \sigma$ for some term $M \in \mathcal{T}$, for every assignment $\eta : \mathcal{V} \longrightarrow \mathcal{C}$, for every p-substitution $\phi : \Gamma \longrightarrow \mathcal{T}_{\lambda_x}$, if $\phi(x) \in [\![ZT(\Gamma(x))]\!]\eta$ for every $x \in Dom(\Gamma)$, then for every extension $\overline{\phi}$ of ϕ we have $\overline{\phi}(Erase(M)) \in [\![\sigma]\!]\eta$.

Proof.
By induction on the length of the derivation of $\Gamma \rhd M : \sigma$ (see appendix for details).

∎

In order to obtain families of candidates of reducibility for proving properties of $Erase(M)$ where $M \in \mathcal{T}$ Girard has shown that conditions R1-R7 must be strengthened.

Definition 5.10 (Saturated set) Let S be a nonempty set of terms in $\mathcal{T}_{\lambda_x}$. We say that S is *closed* iff whenever $Mx \in S$ then $M \in S$ where $x \in \mathcal{X}$ and $M \in S$. A subset $C \subseteq S$ which is closed under α-equivalence is called *saturated* iff the following conditions hold

S1. For every variable $x \in \mathcal{X}$, for all $n \geq 0$ and all $P_1 \ldots P_n \in S$, $xP_1 \ldots P_n \in C$.

S2. For all $M, N \in \mathcal{T}_{\lambda_x}$, all $n \geq 0$ and all $P_1, \ldots, P_n \in S$, if $M\langle x := N\rangle P_1 \ldots P_n \in C$ then $(\lambda x.M)\, NP_1 \ldots P_n \in C$.

S3. For all $M \in S$ and for all $m \geq 0$,$n \geq 0$ and all $N_1, \ldots, N_m \in S$ and $P_1, \ldots, P_n \in S$ if $M^\star P_1 \ldots P_n \in C$ then $x\langle x := M\rangle^\star P_1 \ldots P_n \in C$.

S4. For all $M, N \in S$, all $m \geq 0$, $n \geq 0$, all $N_1, \ldots, N_m$, all $P_1, \ldots, P_n \in S$, and every variable x in $\mathcal{X}$ such that $x \notin FV(M)$ if $M^\star P_1 \ldots P_n \in C$ then $M\langle x := N\rangle^\star P_1 \ldots P_n \in C$.

S5. For all $M_1, M_2 \in \mathcal{T}_{\lambda_x}$, for every variable $x \in \mathcal{X}$ and for all $m \geq 0$, and $N, N_1, \ldots, N_m \in S$ and all $n \geq 0$ and $P_1, \ldots, P_n \in S$ if $(M_1\langle x := N\rangle M_2\langle x := N\rangle)^\star P_1 \ldots P_n \in C$ then $(M_1 M_2)\langle x := N\rangle^\star P_1 \ldots P_n \in C$.

S6. For all $M \in \mathcal{T}_{\lambda_x}$, for every variable $x \in \mathcal{X}$, and for all $m \geq 0$, and $N, N_1, \ldots, N_m \in S$, and all $n \geq 0$ and $P_1, \ldots, P_n \in S$ if $(\lambda y.M\langle x := N\rangle)^\star P_1 \ldots P_n \in C$ then $(\lambda y.M)\langle x := N\rangle^\star P_1 \ldots P_n \in C$.

S7. For all $M, N \in \mathcal{T}_{\lambda_x}$, for all $m \geq 0$, for all $P, N_1, \ldots, N_m \in S$, for all $n \geq 0$ and $P_1, \ldots, P_n \in S$ if $y \notin FV(P)$ and $M\langle x := P\rangle\langle y := N\langle x := P\rangle\rangle^\star P_1 \ldots P_n \in C$ then $M\langle y := N\rangle\langle x := P\rangle^\star P_1 \ldots P_n \in C$.

Lemma 5.7 Let S be a nonempty closed set of terms in $\mathcal{T}_{\lambda_x}$, let $\mathcal{C}$ be the family of saturated subsets of S, and assume that $S \in \mathcal{C}$. Then $\mathcal{C}$ is a family of candidates of reducibility.

Proof.
Note that conditions (S1)-(S7) imply conditions (R1)-(R7), respectively. Therefore we are left to verify that $\mathcal{C}$ is a type closed family. First of all note that the family $\mathcal{C}$ is nonempty since $S \in \mathcal{C}$. Also,

- By condition (S1) each saturated subset is nonempty since it contains all variables.
- Let C and D be saturated subsets of S. We must show that $[C \to D] = \{M \in \mathcal{T}_{\lambda_x} \mid \forall N \in C,\ MN \in D\}$ is a saturated subset of S.
  - $[C \to D]$ is a subset of S. Let $M \in [C \to D]$. Since there is some variable $x \in C$, we have $Mx \in D$. And since S is closed, $M \in S$.

- (S1). Since D is a saturated subset of S then by (S1) for every variable $x \in \mathcal{X}$ and for all $n \geq 0$ and $P_1, \ldots, P_n, P \in S$ we have $xP_1 \ldots P_n P \in D$. Since this is valid for every $P \in C$, we have $xP_1 \ldots P_n \in [C \to D]$.
- (S2). Let M and N be any terms in $\mathcal{T}_{\lambda_x}$. Let $P_1 \ldots P_n$ be terms in S. Suppose that $M\langle x := N\rangle P_1 \ldots P_n \in [C \to D]$. Then for every $P \in C$ we have $M\langle x := N\rangle P_1 \ldots P_n P \in D$. Since D is saturated, by (S2) we have $(\lambda x.M)\ NP_1 \ldots P_n P \in D$. Since this holds for every $P \in C$, $(\lambda x.M)\ NP_1 \ldots P_n \in [C \to D]$.
- (S3). Let M be a term in S. Let $N_1, \ldots, N_m$ and $P_1 \ldots P_n$ be terms in S. Suppose that $M\langle x_1 := N_1\rangle \ldots \langle x_m := N_m\rangle P_1 \ldots P_n \in [C \to D]$. Then for every $P \in C$ we have $M\langle x_1 := N_1\rangle \ldots \langle x_m := N_m\rangle P_1 \ldots P_n P \in D$ and since D is a saturated subset of S by (S3) $x\langle x := M\rangle\langle x_1 := N_1\rangle \ldots \langle x_m := N_m\rangle P_1 \ldots P_n P \in D$. Since this holds for every $P \in C$ we have $x\langle x := M\rangle\langle x_1 := N_1\rangle \ldots \langle x_m := N_m\rangle P_1 \ldots P_n \in [C \to D]$.

The remaining cases are dealt with likewise.

- Note that properties (S1)-(S7) of saturated subsets of S are preserved under arbitary intersections, and thus for the $\mathcal{C}$-indexed family of saturated subsets of S, $(A_C)_{C \in \mathcal{C}}$ we have $\bigcap_{C \in \mathcal{C}} A_C$ is a saturated subset of S.
- A saturated subset of S is closed under α-equivalence by definition.

∎

Proposition 5.8 Let S be a nonempty closed set of terms in $\mathcal{T}_{\lambda_x}$, let $\mathcal{C}$ be the family of all saturated subsets of S, and assume $S \in \mathcal{C}$. Let $M \in \mathcal{T}$ that type checks under the type assignment $\Gamma \equiv \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ and let $z_1, \ldots, z_n$ be fresh variables. Then $Erase(M)\langle x_{i_1} := z_1\rangle \ldots \langle x_{i_n} := z_n\rangle \in S$ where $i_1, \ldots, i_n$ is any ordering of the variables in $Dom(\Gamma)$.

Proof.
By Lemma 5.7 we have that $\mathcal{C}$ is a family of candidates of reducibility. Finally, we conclude by Lemma 5.6.

∎

Therefore in order to prove properties of $Erase(M)$ when M is a typable polymorphic term we need saturated sets satisfying these properties. In particular, since we are interested in strongly λ_x-normalizing terms (SN_{λ_x}) we shall prove that the set SN_{λ_x} is a closed saturated set. Its proof requires the following lemma.

Lemma 5.9 (Perpetuality lemma) For all $M, N_1, N_2, P, P_i, Q_j \in \mathcal{T}_{\lambda_x}$ with $i \in 0..n$ and $j \in 1..m$ we have

1. If $P_0\langle x := P_1\rangle P_2 \ldots P_n \in SN_{\lambda_x}$ then $(\lambda x.P_0)P_1 \ldots P_n \in SN_{\lambda_x}$.
2. If $Q_1\langle x_2 := Q_2\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n \in SN_{\lambda_x}$ then $x_1\langle x_1 := Q_1\rangle\langle x_2 := Q_2\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n \in SN_{\lambda_x}$
3. If $M\langle x_2 := Q_2\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n \in SN_{\lambda_x}$ and $y \notin FV(M)$ and $Q_1 \in SN_{\lambda_x}$ then $M\langle x_1 := Q_1\rangle\langle x_2 := Q_2\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n \in SN_{\lambda_x}$.
4. If $(N_1\langle x_1 := Q_1\rangle N_2\langle x_1 := Q_1\rangle)\langle x_2 := Q_2\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n \in SN_{\lambda_x}$ then $(N_1 N_2)\langle x_1 := Q_1\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n \in SN_{\lambda_x}$.
5. If $(\lambda y.M\langle x_1 := Q_1\rangle)\langle x_2 := Q_2\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n \in SN_{\lambda_x}$ and $y \notin FV(Q_1)$ then $(\lambda y.M)\langle x_1 := Q_1\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n \in SN_{\lambda_x}$.
6. If $M\langle x := P\rangle\langle y := N\langle x := P\rangle\rangle\langle x_1 := Q_1\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n \in SN_{\lambda_x}$ and $y \notin FV(P)$ then $M\langle y := N\rangle\langle x := P\rangle\langle x_1 := Q_1\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n \in SN_{\lambda_x}$.

Proof.
We shall consider items (4) and (6). The rest are dealt with similarly.

- Note that by lemma 2.4 it suffices to show that for every subterm N of the term $U \equiv (N_1N_2)\langle x_1 := Q_1\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n$ we have that $x(N)$ is β-strongly normalizing. By hypothesis we know that the term $T \equiv (N_1\langle x_1 := Q_1\rangle N_2\langle x_1 := Q_1\rangle)\langle x_2 := Q_2\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n$ is in SN_{λ_x}. Thus we now consider all possible subterms of U:
  - $N \subseteq N_1$ or $N \subseteq N_2$. Then by the hypothesis we are done.
  - $N \subseteq Q_i$ with $i \in 1..m$ or $N \subseteq P_i$ with $i \in 1..n$. Then by the hypothesis we are done.
  - $N \equiv (N_1N_2)$. Suppose $x(N) \notin SN_\beta$. Then there is an infinite β-reduction sequence starting from $x(N)$. Thus by lemma 2.1, there is an infinite β-reduction sequence starting from $x(N)\{x_1 \leftarrow x(Q_1)\}$. Now since,

$$\begin{array}{rcl} x(N)\{x_1 \leftarrow x(Q_1)\} & \equiv & x(N_1N_2)\{x_1 \leftarrow x(Q_1)\} \\ & \equiv & x(N_1)x(N_2)\{x_1 \leftarrow x(Q_1)\} \\ & \equiv & x(N_1)\{x_1 \leftarrow x(Q_1)\}x(N_2)\{x_1 \leftarrow x(Q_1)\} \\ & \equiv_{L\ 2.3} & x(N_1\langle x_1 := Q_1\rangle)x(N_2\langle x_1 := Q_1\rangle) \\ & \equiv & x(N_1\langle x_1 := Q_1\rangle N_2\langle x_1 := Q_1\rangle) \end{array}$$

    we contradict the hypothesis that U is in SN_{λ_x}.
  - $N \equiv (N_1N_2)\langle x_1 := Q_1\rangle$. Suppose that $x(N) \notin SN_\beta$. Then there is an infinite β-reduction sequence starting from $x(N)$. But note that

$$\begin{array}{rcl} x(N) & \equiv & x((N_1N_2)\langle x_1 := Q_1\rangle) \\ & \equiv & x(N_1\langle x_1 := Q_1\rangle N_2\langle x_1 := Q_1\rangle) \end{array}$$

    Thus we contradict the hypothesis as above.
  - The remaining cases are dealt with accordingly.

- Suppose $M\langle x := P\rangle\langle y := N\langle x := P\rangle\rangle\langle x_1 := Q_1\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n \in SN_{\lambda_x}$, that $P \in SN_{\lambda_x}$ and that $y \notin FV(P)$. We must show that $R \equiv M\langle y := N\rangle\langle x := P\rangle\langle x_1 := Q_1\rangle \ldots \langle x_m := Q_m\rangle P_1 \ldots P_n \in SN_{\lambda_x}$. We use lemma 2.4. Thus we must verify that for every subterm Q of R, $x(Q)$ is strongly β-normalizing
  - $Q \subseteq M$, $Q \subseteq P$, $Q \subseteq N$, $Q \subseteq Q_i$ with $i \in 1..m$, or $Q \subseteq P_j$ with $j \in 1..n$. Holds by hypothesis.
  - $Q \equiv M\langle y := N\rangle$. Suppose $x(Q) \equiv_{L\ 2.3} x(M)\{y \leftarrow x(N)\} \notin SN_\beta$. Then by lemma 2.1(1) we have $x(M)\{y \leftarrow x(N)\}\{x \leftarrow x(P)\} \notin SN_\beta$. And since

$$\begin{array}{rl} & x(M)\{y \leftarrow x(N)\}\{x \leftarrow x(P)\} \\ \equiv_{L\ 2.2} & x(M)\{x \leftarrow x(P)\}\{y \leftarrow x(N)\{x \leftarrow x(P)\}\} \\ \equiv & x(M\langle x := P\rangle\langle y := N\langle x := P\rangle\rangle) \end{array}$$

    We contradict the hypothesis.
  - The remaining cases are similarly dealt with.

∎

The perpetuality lemma is a key lemma in the proof of the following theorem.

Lemma 5.10 The set SN_{λ_x} is a closed saturated set.

Proof.
We must verify that SN_{λ_x} is closed and verifies (S1)-(S7). Note that if Mx is strongly λ_x-normalizable then M also, so SN_{λ_x} is closed. We are left to check that SN_{λ_x} verifies

- (S1). Suppose $P_1 \ldots P_n \in SN_{\lambda_x}$ then for every variable $x \in \mathcal{X}$, $xP_1 \ldots P_n$ is strongly λ_x-normalizing.
- (S2)-(S7). By the perpetuality lemma 5.9 (1)-(6).

∎

Therefore since SN_{λ_x} is a nonempty closed subset of $\mathcal{T}_{\lambda_x}$ and is itself saturated, if we construct the family of saturated subsets of SN_{λ_x} we obtain, by lemma 5.7, a family of candidates of reducibility. This allows us to prove the following corollary.

Corollary 5.11 Let M be a term in $\mathcal{T}^{\rightarrow}$. Then M is strongly $\rightarrow_{F_{es}}$-normalizing.

Proof.
Suppose $M \in \mathcal{T}^{\rightarrow}$. Then there is a type assignment $\Gamma \equiv \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ and a pure type σ such that $\Gamma \rhd M : \sigma$ is a derivable type judgement.

Let $z_1, \dots, z_n$ be fresh variables. Then by proposition 5.8 we have $Erase(M)\langle x_{i_1} := z_1\rangle \dots \langle x_{i_n} := z_n\rangle \in SN_{\lambda_x}$ where $i_1, \dots, i_n$ is any ordering of the variables in $Dom(\Gamma)$. But then we also have $Erase(M) \in SN_{\lambda_x}$ and finally by lemma 5.2 we may conclude that M is strongly F_{es}-normalizing.

■

6 Conclusions

We have introduced the polymorphic lambda-calculus with explicit substitutions F_{es} and have shown that it satisfies the subject reduction property and that all typable terms are strongly normalizing. In [5] Pure Type Systems with explicit substitutions (EPTS) are defined and already for Fx (the explicit substitution version of F) subject reduction is shown to fail. As regards strong normalization it is shown that Fx is SN if and only if F is SN[4], whereas our proof does not rely on the SN property of F.

Extending the reducibility technique to calculi of explicit substitution is an interesting goal. Recent work in this direction is E.Ritter's article [20]. There, the reducibility technique is used for showing PSN of explicit substitution calculi. As regards the present work the reducibility has been used to prove SN of a typed calculus and in a polymorphic setting. One could argue that the fact that the proof presented here requires the characterization lemma of strongly λ_x-normalizing untyped λ_x-terms (Lemma 2.4) yields it "nonpure". We believe that a detailed analysis of infinite reductions in λ_x should suffice for the proof to go through. This issue shall be studied further.

Lastly, although not the scope of the present work, the problem of type reconstruction would certainly be of interest.

Acknowledgements. I am grateful to Delia Kesner for valuable discussions and suggestions.

References

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

[2] H.P.Barendregt. *The Lambda Calculus: its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, revised edition, 1984.

[3] 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, Volume 2: Background: Computational Structures. Oxford University Press, 1992.

[4] R.Bloo and H.Geuvers. *Explicit Substitution: on the edge of strong normalization.* Theoretical Computer Science, 204, 1998.

[5] R.Bloo. *Preservation of Termination for Explicit Substitution.* PhD. Thesis, Eindhoven University, 1997.

[6] N. Dershowitz. *Orderings for term rewriting systems.* Theoretical Computer Science, 17(3):279-301, 1982.

[7] J.H.Gallier. *On Girard's "candidates de reducibilité".* In P.Odifreddi, ed. Logic and Computer Science. Number 31 in the APIC Series. pp.123-203.

[4] Actually, the result is stated for arbitrary PTS and EPTS $\mathcal{R}$ and $\mathcal{R}x$, respectively.

[8] J.-Y.Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur.* Thèse d'Etat, Université Paris VII, 1972.

[9] J.-Y.Girard, Y.Lafont, P.Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[10] T.Hardin and J-J.Lévy. *A confluent calculus of substitutions.* In France-Japan Artificial Intelligence and Computer Science Symposium, 1989.

[11] M.Ferreira, D.Kesner. and L.Puel. *Lambda-calculi with explicit substitutions and composition which preserve beta-strong normalization.* Proceedings of the 5th International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139, Springer Verlag, 1996.

[12] F. Kamareddine and A.Ríos. *A lambda calculus a la de Bruijn with Explicit Substitutions.* Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'95), LNCS 982, Springer Verlag, 1995.

[13] P. Lescanne. *From λ_σ to λ_υ, a journey through calculi of explicit substitutions.* In Ann. ACM Symp. on Principles of Programming Languages (POPL), pp. 60-69. ACM, 1994.

[14] P.A.Melliès. *Typed λ-calculi with explicit substitutions may not terminate.* In TLCA'95, LNCS 902, Springer Verlag, 1995.

[15] C.A.Muñoz. *Confluence and Preservation of Strong Normalisation in an Explicit Substitutions Calculus.* Proceedings of the Eleven Annual IEEE Symposium on Logic in Computer Science, 1996.

[16] C.Muñoz. *Dependent Types with Explicit Substitutions: A meta-theoretical development.* In Proceedings of the International Workshop TYPES'96, LNCS 1512, Springer Verlag, 1997.

[17] C.Muñoz. *A Left-Linear Variant of $\lambda\sigma$.* In Proceedings of the International Conference PLILP/ALP/HOA'97, Southampton (England), 1997.

[18] J.C.Reynolds. *Towards a Theory of Type Structure.* In Programming Symposium, Proceedings, Colloque sur la Programmation, ed. B.Robinet, LNCS 19, Springer Verlag, 1974.

[19] A. Ríos. *Contribution à l'étude des λ-calculs avec substitutions explicites.* Ph.D Thesis, Université de Paris VII, 1993.

[20] E. Ritter. *Characterising Explicit Subsitutions which Preserve Termination (Extended Abstract).* In Typed Lambda Calculi and Applications, 1999.

[21] K.Rose. *Explicit Cyclic Substitutions.* In CTRS'92, LNCS 656, Springer Verlag, 1992.

[22] W.W.Tait. *A realizability interpretation of the theory of species.* In Logic Colloquium, ed. R.Parikh, LNCS 453, Springer Verlag, 1975.

A Appendix

Definition A.1 (RPO) Let $s = f(s_1, .., s_m)$ and $t = g(t_1, .., t_n)$ be terms in $\mathcal{T}_l$. Then $s \succ_{\mathcal{S}} t$ if and only if one of the following holds:

1. (subterm) $s_i \succ_{\mathcal{S}} t$ or $s_i = t$ for some $i \in 1..m$
2. (decreasing heads) $f \gg g$ and $s \succ_{\mathcal{S}} t_1, .., s \succ_{\mathcal{S}} t_n$
3. (equal heads) $f = g$ and $\{\{s_1, .., s_m\}\} \succ'_{\mathcal{S}} \{\{t_1, .., t_n\}\}$

where $\succ'_{\mathcal{S}}$ is the extension of $\succ_{\mathcal{S}}$ to multisets.

Lemma A.1 Let $A = (S, R_1 \cup R_2)$ be an abstract reduction system and $>$ be a well-founded order on a set T such that

- R_2 is strongly normalizing,
- there exists a function $h : S \to T$ such that $a \to_{R_1} b$ implies $h(a) > h(b)$ and $a \to_{R_2} b$ implies $h(a) \geq h(b)$.

then $R_1 \cup R_2$ is strongly normalizing.

Lemma A.2 (Subject reduction) Let $\Gamma \rhd M : \sigma$ be a derivable type judgement and suppose $M \to_{F_{es}} N$. Then $\Gamma \rhd N : \sigma$ is a derivable type judgement.

Proof.
By induction on the reduction $M \to_r N$ with $r \in F_{es}$. Thus we consider the cases when the reduction takes place at the root and when the reduction is internal.
Suppose the reduction takes place at the root. The interesting cases are:

- $r \equiv \tau Beta$. Then $M \equiv (\Lambda t.P)\tau$ and $N \equiv P[t := \tau]$ and the derivation runs

$$\cfrac{\cfrac{\Gamma \rhd P : \sigma'}{\Gamma \rhd \Lambda t.P : \forall t.\sigma'}\,(tabs)}{\Gamma \rhd (\Lambda t.P)\tau : \sigma'\{t \leftarrow ZT(\tau)\}}\,(tapp)$$

where $t \notin FTV(\Gamma)$. We may obtain

$$\cfrac{\Gamma \rhd P : \sigma'}{\Gamma \rhd P[t := \tau] : \sigma'\{t \leftarrow ZT(\tau)\}}\,(tsubs)$$

Note that $\Gamma \in \{\ \{x_1 : \sigma'_1, \ldots, x_n : \sigma'_n\}\ \mid\ \sigma'_i =_{ZT} \sigma_i[t := \tau]\ \textit{for all}\ i \in 1..n\}$ where $\Gamma = \{\sigma_1, \ldots, \sigma_n\}$, since $t \notin FTV(\Gamma)$ and we may use the rule *ztgc*.

- $r \equiv zlam$. Then $M \equiv (\lambda y : \sigma_1.P)\langle x := Q\rangle$ and $N \equiv \lambda y : \sigma_1.P\langle x := Q\rangle$. The derivation runs

$$\cfrac{\cfrac{\Gamma, x : \tau, y : \sigma_1 \rhd P : \sigma_2}{\Gamma, x : \tau \rhd \lambda y : \sigma_1.P : ZT(\sigma_1) \supset \sigma_2}\,(abs) \qquad \Gamma \rhd Q : ZT(\tau)}{\Gamma \rhd (\lambda y : \sigma_1.P)\langle x := Q\rangle : ZT(\sigma_1) \supset \sigma_2}\,(subs)$$

Now by the variable convention we may assume that all bound type variables in Q do not occur free in σ_1 and that y does not occur bound in Q. Then by lemma 4.5 we have that $\Gamma, y : \sigma_1 \rhd Q : \tau$ is a derivable type judgement. This allows us to construct the following derivation.

$$\cfrac{\cfrac{\Gamma, x : \tau, y : \sigma_1 \rhd P : \sigma_2 \quad \Gamma, y : \sigma_1 \rhd Q : ZT(\tau)}{\Gamma, y : \sigma_1 \rhd P\langle x := Q\rangle : \sigma_2}\,(subs)}{\Gamma \rhd \lambda y : \sigma_1.P\langle x := Q\rangle : ZT(\sigma_1) \supset \sigma_2}\,(abs)$$

- $r \equiv ztvar2$. Then $M \equiv x[t := \tau]$ and $N \equiv x$ and the derivation runs

$$\dfrac{\Delta \rhd x : ZT(\rho)}{\Delta_{[t:=\tau]} \rhd x[t := \tau] : ZT(\rho)\{t \leftarrow ZT(\tau)\}}\,(tsubs)$$

where $x : \rho \in \Delta$. Let $x : \rho' \in \Delta_{[t:=\tau]}$ and therefore $\rho' =_{ZT} \rho[t := \tau]$. Then $ZT(\rho') \equiv ZT(\rho)\{t \leftarrow ZT(\tau)\}$ using lemma 3.4(2) and the fact that ZT is complete. Then $\Delta_{[t:=\tau]} \rhd x : ZT(\rho') \equiv ZT(\rho)\{t \leftarrow ZT(\tau)\}$ is a derivable type judgement (using (var)).

- $r \equiv ztlamt$. Then $M \equiv (\Lambda u.P)[t := \tau]$ and $N \equiv \Lambda u.P[t := \tau]$ and we assume by the variable convention that $u \notin FTV(\tau)$. Then the derivation runs

$$\dfrac{\dfrac{\Delta \rhd P : \rho}{\Delta \rhd \Lambda u.P : \forall u.\rho}\,(tabs)}{\Delta_{[t:=\tau]} \rhd (\Lambda u.P)[t := \tau] : (\forall u.\rho)\{t \leftarrow ZT(\tau)\}}\,(tsubs)$$

where $u \notin FTV(\Delta)$. We may assume by the variable convention that $u \notin FTV(\Delta_{[t:=\tau]})$ and thus obtain the derivation

$$\dfrac{\dfrac{\Delta \rhd P : \rho}{\Delta_{[t:=\tau]} \rhd P[t := \tau] : \rho\{t \leftarrow ZT(\tau)\}}\,(tsubs)}{\Delta_{[t:=\tau]} \rhd \Lambda u.P[t := \tau] : \forall u.\rho\{t \leftarrow ZT(\tau)\}}\,(tabs)$$

Suppose now that the reduction is internal. Then we consider the following cases according to each possible context C

- $C \equiv \lambda x : \tau.P$. We must consider two possible cases:
  - $P \rightarrow_{F_{es}} P'$ and therefore $N \equiv \lambda x : \tau.P'$. This case is handled by the induction hypothesis.
  - $\tau \rightarrow_{ZT} \tau'$ and therefore $N \equiv \lambda x : \tau'.P$. Then the derivation runs

  $$\dfrac{\Gamma, x : \tau \rhd P : \rho}{\Gamma \rhd \lambda x : \tau.P : ZT(\tau) \supset \rho}\,(abs)$$

  And we may conclude by using the context reduction lemma 4.6 on the derivation ending in the premise of (abs) followed by a new application of (abs), and noting that $ZT(\tau) \equiv ZT(\tau')$.

- $C \equiv P[t := \tau]$. Then the derivation runs

$$\dfrac{\Delta \rhd P : \rho}{\Delta_{[t:=\tau]} \rhd P[t := \tau] : \rho\{t \leftarrow ZT(\tau)\}}\,(tsubs)$$

And either $P \rightarrow_{F_{es}} P'$ and therefore $N \equiv P'[t := \tau]$, or $\tau \rightarrow_{ZT} \tau'$ and therefore $N \equiv P[t := \tau']$. For the first case we use the induction hypothesis. For the second we construct the following derivation.

$$\dfrac{\Delta \rhd P : \rho}{\Delta_{[t:=\tau]} \rhd P[t := \tau'] : \rho\{t \leftarrow ZT(\tau')\}}\,(tsubs)$$

We are left to verify that the assignment $\Delta_{[t:=\tau]}$ is in the set of type assignments $\{\ \{x_1 : \sigma'_1, \ldots, x_n : \sigma'_n\} \mid \sigma'_i =_{ZT} \sigma_i[t := \tau']\ \textit{for all}\ i \in 1..n\}$, where $\Delta \equiv \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, which is true since $\tau \rightarrow_{ZT} \tau'$.

- The remaining cases are dealt with by using the induction hypthesis.

∎

Lemma A.3 Let $\mathcal{C}$ be a family of candidates of reducibility. For every derivation of $\Gamma \rhd M : \sigma$ for some term $M \in \mathcal{T}$, for every assignment $\eta : \mathcal{V} \longrightarrow \mathcal{C}$, for every p-substitution $\phi : \Gamma \longrightarrow \mathcal{T}_{\lambda_x}$, if $\phi(x) \in [\![ZT(\Gamma(x))]\!]\eta$ for every $x \in Dom(\Gamma)$, then for every extension $\overline{\phi}$ of ϕ we have $\overline{\phi}(Erase(M)) \in [\![\sigma]\!]\eta$.

Proof.
By induction on the length of the derivation of $\Gamma \rhd M : \sigma$.

- Base case. Then the derivation consists solely of $\Gamma \rhd x : ZT(\rho)$ where $x : \rho \in \Gamma$. Now by hypothesis we know that $\phi(x) \in [\![ZT(\rho)]\!]\eta$. Let $[x_1, \ldots, x_n]$ be any ordering of the variables in $Dom(\Gamma)$ and suppose $x_j \equiv x$. Then we reason as follows

$$\begin{array}{ll}
\phi(x) \in [\![ZT(\rho)]\!]\eta & \text{(hypothesis)} \\
\phi(x)\langle x_n := \phi(x_n)\rangle \in [\![ZT(\rho)]\!]\eta & \text{(R4) and } \phi \text{ p-subs} \\
. & . \\
. & . \\
\phi(x)\langle x_{j+1} := \phi(x_{j+1})\rangle \ldots \langle x_n := \phi(x_n)\rangle \in [\![ZT(\rho)]\!]\eta & \text{(R4) and } \phi \text{ p-subs} \\
x\langle x := \phi(x)\rangle\langle x_{j+1} := \phi(x_{j+1})\rangle \ldots \langle x_n := \phi(x_n)\rangle \in [\![ZT(\rho)]\!]\eta & \text{(R3)} \\
. & \text{(R4) and } \Gamma \text{ type judg.} \\
. & \text{(R4) and } \Gamma \text{ type judg.} \\
x\langle x_1 := \phi(x_1)\rangle \ldots \langle x_n := \phi(x_n)\rangle \in [\![ZT(\rho)]\!]\eta & \text{(R4) and } \Gamma \text{ type judg.}
\end{array}$$

- Inductive case. Let $k+1$ be the length of this derivation. We analyse the last derivation rule applied,

  – (abs). Then the derivation ends as follows:

$$\frac{\Gamma, x : \tau \rhd M : \rho}{\Gamma \rhd \lambda x : \tau.M : ZT(\tau) \supset \rho}\,(abs)$$

  Let N be any term in $[\![ZT(\tau)]\!]\eta$. Let $\overline{\phi}$ be any extension of ϕ with ordering of variables $[x_1, \ldots, x_n]$. We choose a representative in the $\equiv_\alpha$-class of $[(\lambda x.Erase(M))\langle x_1 := \phi(x_1)\rangle \ldots \langle x_n := \phi(x_n)\rangle]_\alpha$, say $(\lambda x'.Erase(M'))\langle x'_1 := \phi(x_1)\rangle \ldots \langle x'_n := \phi(x_n)\rangle$[5] such that

  1. $x' \notin FV(\phi(x_i))$ for $i \in 1..n$ and $x' \notin FV(N)$ and $x_i \not\equiv x'$ for $i \in 1..n$, and
  2. $x'_i \notin FV(\phi(x_j))$ and $x'_i \notin FV(N)$, for $i, j \in 1..n$.

  We have to rename x in order to avoid name clashes (first item). But since we are working with p-substitutions this entails more renaming (second item) so that we may be able to apply the induction hypothesis when it is needed. Note that since ϕ is a p-substitution no renaming takes place inside the $\phi(x_i)$s in the representative of the α-equivalence class chosen.
  Let ϕ' be the resulting p-substitution, i.e. the substitution defined as

$$\phi'(z) =_{def} \begin{cases} \phi(x_i) & if\ z \equiv x'_i \\ N & if\ z \equiv x' \end{cases}$$

  Since the induction hypothesis holds for every derivation of length $\leq k$ and for every assignment η satisfying the conditions of the lemma, we have that for every extension $\overline{\phi'}$ of ϕ', $\overline{\phi'}(Erase(M')) \in [\![\rho]\!]\eta$. In particular, $Erase(M')\langle x'_1 := \phi(x_1)\rangle \ldots \langle x'_n := \phi(x_n)\rangle\langle x' := N\rangle \in [\![\rho]\!]\eta$.
  Then by condition (R2) we obtain that $(\lambda x'.Erase(M')\langle x'_1 := \phi(x_1)\rangle \ldots \langle x'_n := \phi(x_n)\rangle)N \in [\![\rho]\!]\eta$
  Since this is valid for every $N \in [\![ZT(\tau)]\!]\eta$ then by definition of $[\![ZT(\tau) \supset \rho]\!]\eta$ we have,
  $\lambda x'.Erase(M')\langle x'_1 := \phi(x_1)\rangle \ldots \langle x'_n := \phi(x_n)\rangle \in [\![ZT(\tau) \supset \rho]\!]\eta$
  Finally since $x' \notin FV(\phi(x_i))$ for $i \in 1..n$ (first item above) we may apply condition (R6) repeatedly obtaining, $(\lambda x'.Erase(M'))\langle x'_1 := \phi(x_1)\rangle \ldots \langle x'_n := \phi(x_n)\rangle \in [\![ZT(\tau) \supset \rho]\!]\eta$
  And by closedness under α-equivalence we conclude,
  $(\lambda x.Erase(M))\langle x_1 := \phi(x_1)\rangle \ldots \langle x_n := \phi(x_n)\rangle \equiv Erase(\lambda x : \tau.M)\langle x_1 := \phi(x_1)\rangle \ldots \langle x_n := \phi(x_n)\rangle \equiv \overline{\phi}(Erase(\lambda x : \tau.M)) \in [\![ZT(\tau) \supset \rho]\!]\eta$

  – (subs). Then the derivation ends as follows

[5] Note that renaming commutes with the erasing function.

$$\frac{\Gamma, x:\tau \rhd M_1:\sigma \quad \Gamma \rhd M_2: ZT(\tau)}{\Gamma \rhd M_1\langle x := M_2\rangle : \sigma}\,(subs)$$

Let ϕ be any p-substitution with domain Γ. Now we choose a representative of the equivalence class $[Erase(M_1)\langle x := Erase(M_2)\langle x_1 := \phi(x_1)\rangle \ldots \langle x_n := \phi(x_n)\rangle\rangle]_\alpha$
say $Erase(M_1')\langle x' := Erase(M_2)\langle x_1 := \phi(x_1)\rangle \ldots \langle x_n := \phi(x_n)\rangle\rangle$ such that

1. $x' \notin Dom(\Gamma)$, and
2. $x' \notin FV(\overline{\phi}(Erase(M_2))$ (and thus $x' \notin FV(\phi(x_j))$ with $j \in 1..n$).

Let ϕ' be the substitution $\phi' : (\Gamma, x':\tau) \longrightarrow \mathcal{T}_{\lambda_x}$ defined as

$$\phi'(z) =_{def} \begin{cases} \phi(x_i) & if\ z \equiv x_i \\ \overline{\phi}(Erase(M_2)) & if\ z \equiv x' \end{cases}$$

Note that due to the first and second item, and the fact that $x_i \notin FV(\overline{\phi}(Erase(M_2))$ with $i \in 1..n$ (since they are all bound and ϕ is a p-substitution) the resulting substitution is a p-substitution.
Since the induction hypothesis holds for any derivation of length $\leq k$ and for every assignment satisfying the conditions of the lemma, on one hand for every extension $\overline{\phi}$ of ϕ we have $\overline{\phi}(Erase(M_2)) \in [\![ZT(\tau)]\!]\eta$. Another application of the induction hypothesis to the derivation ending in the left premise of the application of (subs) allows us to infer that for every extension $\overline{\phi'}$ of ϕ' we have $\overline{\phi'}(Erase(M_1')) \in [\![\sigma]\!]\eta$. In particular we have,
$Erase(M_1')\langle x_1 := \phi(x_1)\rangle \ldots \langle x_n := \phi(x_n)\rangle\langle x' := Erase(M_2)\langle x_1 := \phi(x_1)\rangle \ldots \langle x_n := \phi(x_n)\rangle\rangle \in [\![\sigma]\!]\eta$
Now since $x' \notin FV(\phi(x_i))$ (item 2) we may apply condition (R7) repeatedly and obtain
$Erase(M_1')\langle x' := Erase(M_2)\rangle\langle x_1 := \phi(x_1)\rangle \ldots \langle x_n := \phi(x_n)\rangle \in [\![\sigma]\!]\eta$
And since $[\![\sigma]\!]\eta$ is closed under α-equivalence

$$\begin{array}{ll} & Erase(M_1)\langle x := Erase(M_2)\rangle\langle x_1 := \phi(x_1)\rangle \ldots \langle x_n := \phi(x_n)\rangle \\ \equiv & Erase(M_1\langle x := M_2\rangle)\langle x_1 := \phi(x_1)\rangle \ldots \langle x_n := \phi(x_n)\rangle \\ \equiv & \overline{\phi}(Erase(M_1\langle x := M_2\rangle)) \in [\![\sigma]\!]\eta \end{array}$$

– (tsubs). The derivation ends as follows.

$$\frac{\Delta \rhd P:\rho}{\Delta_{[t:=\tau]} \rhd P[t:=\tau] : \rho\{t \leftarrow ZT(\tau)\}}\,(tsubs)$$

Let η be any assignment and ϕ any p-substitution with domain $\Delta_{[t:=\tau]}$ meeting the conditions of the lemma. Consider any $x:\sigma_1 \in \Delta_{[t:=\tau]}$ and its corresponding $x:\sigma_2 \in \Delta$. Then by hypothesis we have $\phi(x) \in [\![ZT(\sigma_1)]\!]\eta$ and since $\sigma_1 =_{ZT} \sigma_2[t:=\tau]$ then $ZT(\sigma_1) \equiv ZT(\sigma_2)\{t \leftarrow ZT(\tau)\}$ using the fact that ZT is complete and lemma 3.4(2). Thus $[\![ZT(\sigma_1)]\!]\eta = [\![ZT(\sigma_2)\{t \leftarrow ZT(\tau)\}]\!]\eta =_{L\ 5.4} [\![ZT(\sigma_2)]\!]\eta[t := [\![ZT(\tau)]\!]\eta]$.
We apply then the induction hypothesis with p-substitution ϕ and assignment $\eta[t := [\![ZT(\tau)]\!]\eta]$ and obtain that for every extension $\overline{\phi}$ of ϕ we have $\overline{\phi}(Erase(P)) \in [\![\rho]\!]\eta[t := [\![ZT(\tau)]\!]\eta]$. But since $Erase(P) \equiv Erase(P[t:=\tau])$ and by lemmas 5.4 and 3.4(2) we may conclude that $Erase(P[t:=\tau]) \in [\![\rho\{t \leftarrow ZT(\tau)\}]\!]\eta$.

– The remaining cases are treated similarly.

∎

Variations on Linear PCF (Extended Abstract)

Valeria de Paiva and Eike Ritter*

School of Computer Science, University of Birmingham

Abstract. This paper shows that the Girard translation between the simply typed λ-calculus and the linear λ-calculus preserves and reflects observational equality. This holds not only for the call-by-name calculi but also for the call-by-value calculi. The proof consists of adapting the well-known techniques of logical relations to a linear λ-calculus with recursion. For this purpose explicit environments turn out to be useful, as we have different kinds of variables with different properties.

1 Introduction

This paper introduces the operational semantics theory [Pit97] that goes with the calculus of *linear* explicit substitutions [GdPR98], which is the basis of our linear functional programming implementation. The development is relatively straightforward, but whenever an operational semantics is used there are choices to be made and our commitment to explicit substitutions informs our choices.

To keep the paper short, we assume that the reader is familiar with Dual Intuitionistic and Linear Logic (henceforward DILL), as proposed by Barber and Plotkin [BP97], which we take as our starting point. Again there are choices as far as a *linear* lambda calculus is concerned, but here we exploit the fact that DILL has the simply typed lambda-calculus as its subcalculus, obtained when all the assumptions are intuitionistic.

To 'linearize' PCF we add to our chosen calculus DILL the typical PCF connectives, i.e. integers, booleans, conditionals and most importantly the fixpoint combinator, Y. The rules for Linear PCF are non-controversial and are described in detail, for instance by Braüner [Bra94]. Using our choice of explicit substitutions, explained and expounded on in [RdP97,GdPR98], we describe a linear environment calculus. For this linear environment calculus, we have both a call-by-name and a call-by-value operational semantics. We describe (non-surprising) translations between call-by-name PCF and its linear counterpart, and between call-by-value PCF and its linear counterpart. Then in the interesting part of this paper, using a traditional logical relations argument, we prove the call-by-name translation fully abstract (*i.e.*, preserves and reflects observational equivalence), and a similar sort of argument yields in the following section the same full abstraction for the call-by-value translation.

* Research supported under the EPSRC project no. GR/L28296, **x-SLAM: The Explicit Substitutions Linear Abstract Machine**.

It is surprising that an operational theory so analogous to the usual PCF one can be obtained for Linear PCF, especially considering the difficulties found in the literature with the 'context lemma'. We would like to consider the naturality of the results here as evidence for the linear environment calculus we advocate here and elsewhere.

This work extends work by Hasegawa [Has99b]. Using logical relations for DILL, he proves that the embedding from multiplicative intuitionistic linear logic into Barber and Plotkin's DILL satisfies full completeness. Also in an unpublished manuscript [Has99a] (which we became aware of only after submitting this paper) Hasegawa proves fullness of the Girard translation using logical relations. In both papers he does not consider any notion of observational equivalence, nor does he consider recursion or non-termination.

Crole [Cro98] investigates the relation between observational equivalence in linear logic and bisimulation. With tensor products he gives an example of two terms which are observationally equivalent but not bisimilar. This example shows that his definition of bisimilarity for tensor product, which considers both components separately, is problematic. Hasegawa's categorically motivated logical relations for the tensor product, which use the elimination rule for the tensor product instead, should yield the desired coincidence of observational equality with logical relations.

Bierman [Bie97] takes a different view of the relationship between linear and intuitionistic logic. He uses linear logic as a metalanguage, and describes in [Bie97] translations from PCF into this metalanguage. These translations differ for call-by-name and call-by-value PCF. As a consequence he obtains different results about the relationship between observational equivalence in PCF and linear PCF. In contrast we view the linear λ-calculus as an intermediate language, and for us the translation is the same for both call-by-value and call-by-name PCF.

2 Linear PCF with environments

For completeness, we start with PCF [Plo77], which is the simply-typed λ-calculus with special expressions for natural numbers and booleans and unlimited recursion added. PCF has the following raw expressions:

$$\begin{aligned} t ::= {} & x \mid \lambda x{:}A.t \mid tu \mid Yt \\ & \mathbf{n} \mid \texttt{succ}\, t \mid \texttt{pred}\, t \mid \texttt{true} \mid \texttt{false} \mid \texttt{iszero}\, t \mid \texttt{if}\, t\, \texttt{then}\, t\, \texttt{else}\, t \end{aligned}$$

The base types are ι (for natural numbers) and o (for booleans).

Linear PCF [Bra94] adds natural numbers, booleans and recursion to a linear λ-calculus rather than to the simply-typed λ-calculus. We use here Barber and Plotkin's DILL (Dual Intuitionistic Linear Logic [BP97]) as the linear λ-calculus. This calculus has two kinds of variables: intuitionistic variables, which might be copied or discarded, and linear variables, which have to be used exactly once. This distinction is also reflected in the type system: A type $!A$ denotes a set of terms which might be copied or discarded, and a type A denotes a set of terms which must be used exactly once. The idea is that an intuitionistic variable of type A is isomorphic to a linear variable of type $!A$. Natural numbers and boolean expressions are added in such a way that functions like successor or conditionals act on arguments of linear type rather than of !-type whenever possible.

The recursion in linear PCF uses only the !-types. The Y-combinator has the type $!(!A \multimap A) \multimap A$, indicating that we may use the function we do recursion over possibly several times. This recursion does not take linearity into account; it is as powerful as the recursion in PCF.

The raw expressions of linear PCF are given by

$$\begin{aligned} t &::= x \mid a \mid \lambda a{:}\,A.t \mid tt \mid\ !t \mid \texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ t \mid Yt \mid \\ &\quad\ \mathbf{n} \mid \texttt{succ}\ t \mid \texttt{pred}\ t \mid \texttt{true} \mid \texttt{false} \mid \texttt{iszero}\ t \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \\ A &::= \iota \mid o \mid A \multimap A \mid\ !A \end{aligned}$$

where x, y, z *etc.* denote intuitionistic variables and a, b, c *etc.* denote linear variables.

We add to linear PCF environments and closures, where environments are lists of pairs of closed terms and variables indicating which term is to be substituted for which variable. A closure is the application of an environment to a term. The use of environments and closures is motivated by the unfoldings of recursions. Each unfolding of a recursion like $Y(\lambda x{:}\,A.t) \rightsquigarrow (\lambda x{:}\,A.t)(Y(\lambda x{:}\,A.t)) \rightsquigarrow t[Y(\lambda x{:}\,A.t)/x]$ produces a new substitution for the recursion variable x. We must not use call-by-value for this application, otherwise immediate non-termination would result. Hence we need to distinguish the recursion variable x from the other variables. As a result we have three different kinds of variables, with only two of them being part of the syntax of linear PCF.

In this paper we keep this distinction in the syntax by adding to linear PCF explicit bindings of terms for all three kinds of variables in the syntax. We call this calculus linear PCF with environments or LPCF_e for short. It has terms t, s, $u, \ldots$, environments f, $g, \ldots$, and closures M, $N, \ldots$.

Definition 1. *The raw expressions of $LPCF_e$ are given by*

$$\begin{aligned} t &::= x \mid a \mid \lambda x{:}\,A.t \mid tu \mid\ !t \mid \texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ u \mid Yt \\ &\quad\ \mathbf{n} \mid \texttt{succ}\ t \mid \texttt{true} \mid \texttt{false} \mid \texttt{iszero}\ t \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \\ f &::= \langle\rangle * t \mid \langle f, M/x\rangle \mid \langle f, M/a\rangle \mid \langle f, M/x_r\rangle \\ M &::= f * t \\ A &::= \iota \mid o \mid A \multimap B \mid\ !A \end{aligned}$$

We use $a, b, c, \ldots$ for linear variables, $x, y, z \ldots$ for intuitionistic variables and x_r, y_r, z_r for recursion variables, which are special intuitionistic variables.

We identify terms up to α-equivalence, and use Barendregt's variable convention that bound and free variables are distinct. The typing rules for LPCF_e are given in Figure 1.

(i) For terms:

$$\Gamma, x{:}\, A, \Gamma' | \_ \vdash x{:}\, A \qquad \Gamma | a{:}\, A \vdash a{:}\, A$$

$$\frac{\Gamma|\Delta, a{:}\, A \vdash t{:}\, B}{\Gamma|\Delta \vdash \lambda a{:}\, A.t{:}\, A \multimap B} \quad \frac{\Gamma|\Delta_1 \vdash t{:}\, A \multimap B \qquad \Gamma|\Delta_2 \vdash u{:}\, A}{\Gamma|\Delta \vdash tu{:}\, B}$$

$$\frac{\Gamma|\_ \vdash t{:}\, A}{\Gamma|\_ \vdash !t{:}\, !A} \qquad \frac{\Gamma|\Delta_1 \vdash t{:}\, !A \qquad \Gamma, x{:}\, A|\Delta_2 \vdash u{:}\, B}{\Gamma|\Delta \vdash \mathtt{let}\ t\ \mathtt{be}\ !x\ \mathtt{in}\ u{:}\, B}$$

$$\frac{}{\Gamma|\_ \vdash \mathtt{true}{:}\, o} \qquad \frac{}{\Gamma|\_ \vdash \mathtt{false}{:}\, o}$$

$$\frac{\Gamma|\Delta_1 \vdash t_1{:}\, o \qquad \Gamma|\Delta_2 \vdash t_2{:}\, A \qquad \Gamma|\Delta_2 \vdash t_3{:}\, A}{\Gamma|\Delta_1, \Delta_2 \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3{:}\, A}$$

$$\frac{\Gamma \mid \Delta \vdash t{:}\, \iota}{\Gamma \mid \Delta \vdash \mathtt{succ}\ t{:}\, \iota} \qquad \frac{\Gamma \mid \Delta \vdash t{:}\, \iota}{\Gamma \mid \Delta \vdash \mathtt{pred}\ t{:}\, \iota}$$

$$\frac{}{\Gamma|\_ \vdash \mathbf{n}{:}\, \iota} \qquad \frac{\Gamma \mid \Delta \vdash t{:}\, \iota}{\Gamma \mid \Delta \vdash \mathtt{iszero}\ t{:}\, o}$$

$$\frac{\Gamma|\_ \vdash t{:}\, !A \multimap A}{\Gamma|\_ \vdash Y t{:}\, A}$$

In those rules with multiple premises, Δ_1, Δ_2 are disjoint and Δ is a permutation of Δ_1, Δ_2.

(ii) For environments:

$$\frac{}{\_|\_ \vdash \langle\rangle{:}\, \_|\_} \qquad \frac{\_|\_ \vdash f{:}\, \Gamma|\Delta \qquad \_|\_ \vdash g * t{:}\, A}{\_|\_ \vdash \langle f, g * t/x\rangle{:}\, (\Gamma, x{:}\, A|\Delta)}$$

$$\frac{\_|\_ \vdash f{:}\, \Gamma|\Delta \qquad \_|\_ \vdash g * t{:}\, A}{\_|\_ \vdash \langle f, g * t/x_r\rangle{:}\, (\Gamma, x_r{:}\, A|\Delta)} \quad \frac{\_|\_ \vdash f{:}\, \Gamma|\Delta \qquad \_|\_ \vdash g * t{:}\, A}{\_|\_ \vdash \langle f, t/a\rangle{:}\, \Gamma|\Delta, a{:}\, A}$$

(iii) For Closures:

$$\frac{\_|\_ \vdash f{:}\, \Gamma|\Delta \qquad \Gamma|\Delta \vdash t{:}\, A}{\_|\_ \vdash f * t{:}\, A}$$

Fig. 1. Typing rules for LPCF_e

Similarly, we extend PCF with environments and closures to obtain a calculus called PCF_e. The typing rules for environments and closures of PCF_e are

$$\frac{}{\_ \vdash \langle\rangle\colon \_} \qquad \frac{\_ \vdash f\colon \Gamma \qquad \_ \vdash g * t\colon A}{\_ \vdash \langle f, g * t/x\rangle\colon (\Gamma, x\colon A)}$$

$$\frac{\_ \vdash f\colon \Gamma \qquad \_|\_ \vdash g * t\colon A}{\_ \vdash \langle f, g * t/x_r\rangle\colon (\Gamma, x_r\colon A)} \quad \frac{\_ \vdash f\colon \Gamma \qquad \Gamma \vdash t\colon A}{\_ \vdash f * t\colon A}$$

Next, we give the Girard-translation from PCF_e into LPCF_e.

Definition 2. *(i) Define the* Girard translation $\_^{lin}$ *from PCF_e into $LPCF_e$ on terms as follows:*

$$\begin{aligned}
x^{lin} &= x\\
(\lambda x\colon A.t)^{lin} &= \lambda x\colon !A^{lin}.\mathtt{let}\ !x\ \mathtt{be}\ x\ \mathtt{in}\ t^{lin}\\
(ts)^{lin} &= t^{lin}!s^{lin}\\
(Yt)^{lin} &= Yt^{lin}\\
c(t_1,\ldots,t_n)^{lin} &= c(t_1^{lin},\ldots,t_n^{lin})
\end{aligned}$$

where $c(t_1,\ldots,t_n)$ is any of the special functions in PCF_e, and on environments and closures componentwise by translating an environment of PCF_e into an environment of $LPCF_e$ with no linear variables. Define the Girard-translation of PCF_e-types into $LPCF_e$-types by

$$\begin{aligned}
\iota^{lin} &= \iota\\
o^{lin} &= o\\
(A \to B)^{lin} &= !A^{lin} \multimap B^{lin}
\end{aligned}$$

(ii) In the other direction, define a translation $(\_)^{int}$ from $LPCF_e$ to PCF_e on terms by

$$\begin{aligned}
x^{int} &= x\\
a^{int} &= a\\
(\lambda a\colon A.t)^{int} &= \lambda a\colon A^{int}.t^{int}\\
(ts)^{int} &= t^{int}s^{int}\\
(!t)^{int} &= t^{int}\\
(\mathtt{let}\ !x\ \mathtt{be}\ t\ \mathtt{in}\ s)^{int} &= s^{int}[t^{int}/x]\\
(Yt)^{int} &= Yt^{int}\\
(c(t_1,\ldots,t_n))^{int} &= c(t_1^{int},\ldots,t_n^{int})
\end{aligned}$$

and on types by

$$\begin{aligned}
\iota^{int} &= \iota\\
o^{int} &= o\\
(A \multimap B)^{int} &= A^{int} \to B^{int}\\
(!A)^{int} &= A^{int}
\end{aligned}$$

and on environments and closures componentwise.

DILL has no λ-abstraction over intuitionistic variables defined as part of the syntax. However, the term $s = \lambda a\colon !A.\mathtt{let}\ !x\ \mathtt{be}\ a\ \mathtt{in}\ t$ behaves exactly like such a λ-abstraction. Hence we will abbreviate s by $\lambda x\colon A.t$ in the sequel.

Both translations preserve typing:

Proposition 3. *(i) Assume t is a PCF-term such that $\Gamma \vdash t\colon A$. Then we have $\Gamma^{lin} \mid \_ \vdash t^{lin}\colon A^{lin}$.*
(ii) Assume t is a linear PCF-term such that $\Gamma|\Delta \vdash t\colon A$. Then $\Gamma^{int}, \Delta^{int} \vdash t^{int}\colon A^{int}$.

3 Call-by-Name

In this section we show that the Girard translation $\_^{lin}$ is fully abstract with respect to a call-by-names operational semantics.

This operational semantics specifies how closed terms are reduced to canonical terms, the so-called values. Values of type ι and o are natural numbers and booleans respectively, values of type $A \multimap B$ are closures $f * \lambda a\colon A.t$ (*i.e.*, functions) and values of type $!A$ are closures $f * !t$ (*i.e.*, terms which may be duplicated or discarded). Values are given inductively by

$$v ::= \mathtt{true} \mid \mathtt{false} \mid \mathbf{n} \mid f * \lambda a\colon A.t \mid f * !t$$

In this section we investigate a call-by-name operational semantics, where arguments of a function are reduced only when accessed. The clauses are listed in Figure 2.

The notion of observational equivalence captures formally when two terms denote the same program: the observables are the values of ground type, and two terms are called *observationally equivalent* if when evaluated in any closing context they yield the same observable (*i.e.*, value). In general it is very difficult to show that two terms are observationally equivalent according to the definition because one has to test them in all possible (arbitrarily complex) contexts. It turns out that for establishing observational equivalence it suffices to apply the two functions which we want to test for observational equivalence to all possible arguments. The fixed point combinator makes it necessary to define observational equivalence as the intersection of two approximation relations in the opposite direction. These approximation relations are the syntactic equivalent to the information order on domains, and the syntactic treatment of the fixed point combinator is analogous to its treatment as a supremum of finite approximations in this order. For an exposition of this well-known approach to showing observational equivalence see [Pit97].

The proof of the preservation of observational equivalence is structured as follows: we start by defining approximation relations

Let c be any of $\mathbf{n}$, `true` or `false`. Let f_t be the environment which contains the bindings for all intuitionistic variables of f and for all free linear variables of t.

$$\frac{}{v \Downarrow_l v}$$

$$\frac{}{f * c \Downarrow_l c}$$

$$\frac{g * t \Downarrow_l v}{\langle f, g * t/x, \boldsymbol{t/x} \rangle * x \Downarrow_l v}$$

$$\frac{g * t \Downarrow_l v}{\langle f, g * t/a, \boldsymbol{t/x} \rangle * a \Downarrow_l v}$$

$$\frac{f_t * t \Downarrow_l g * \lambda a{:}\, A.t' \qquad \langle g, f_s * s/a \rangle * t' \Downarrow_l v}{f * (ts) \Downarrow_l v}$$

$$\frac{f_t * t \Downarrow_l g{*}!u \qquad \langle f_s, g * u/x \rangle * s \Downarrow_l v}{f * \texttt{let } !x \texttt{ be } t \texttt{ in } s \Downarrow_l v}$$

$$\frac{f * t \Downarrow_l g * \lambda a{:}\, !A.t' \qquad \langle g, g * Y(\lambda a{:}\, !A.t')/x_r \rangle * t'[x_r/a] \Downarrow_l v}{f * (Yt) \Downarrow_l v}$$

$$\frac{\langle g, g * (Y(\lambda a{:}\, !A.t))/x_r \rangle * t[x_r/a] \Downarrow_l v}{\langle f, g * (Y(\lambda x{:}\, !A.t))/x_r, \boldsymbol{t/x} \rangle * x_r \Downarrow_l v}$$

$$\frac{f_{t_1} * t_1 \Downarrow_l \texttt{true} \qquad f_{t_2} * t_2 \Downarrow_l v}{f * \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow_l v}$$

$$\frac{f * t_1 \Downarrow_l \texttt{false} \qquad f_{t_3} * t_3 \Downarrow_l v}{f * \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow_l v}$$

$$\frac{f * t \Downarrow_l \mathbf{n}}{f * (\texttt{succ } t) \Downarrow_l \mathbf{n+1}}$$

$$\frac{f * t \Downarrow_l \mathbf{0}}{f * (\texttt{pred } t) \Downarrow_l \mathbf{0}}$$

$$\frac{f * t \Downarrow_l \mathbf{n+1}}{f * (\texttt{pred } t) \Downarrow_l \mathbf{n}}$$

$$\frac{f * t \Downarrow_l \mathbf{0}}{f * (\texttt{iszero } t) \Downarrow_l \texttt{true}}$$

$$\frac{f * t \Downarrow_l \mathbf{n+1}}{f * (\texttt{iszero } t) \Downarrow_l \texttt{false}}$$

Fig. 2. Operational semantics for call-by-name LPCF$_e$

$$\begin{tikzcd} PCF \arrow[loop above, "\sqsubseteq_i"] \arrow[r, shift left, "\sqsubseteq_{il}"] & LPCF \arrow[loop above, "\sqsubseteq_l"] \arrow[l, shift left, "\sqsubseteq_{li}"] \end{tikzcd}$$

The key point is to show that for each approximation relation, a PCF-term approximates itself (or the translation into linear PCF, as appropriate). In the second step we use the relations $\sqsubseteq^{il}$ and $\sqsubseteq^{li}$ to show that the Girard translation maps equivalence classes of the observational equivalence in PCF to equivalence classes of the observational equivalence in LPCF.

The approximation relations are defined as logical relations, *i.e.*, we define a family of relations indexed by types in such a way that for ground types we specify the desired property, and we define two terms of function type to be related if they map related arguments to related results.

We start with the definition of the relation $\sqsubseteq^{il}$.

Definition 4. *Define the* approximation relation $\sqsubseteq^{il}_A$ *between closures of PCF$_e$ of type A and closures of LPCF$_e$ of type A^{lin} by induction over the structure of types as follows:*

G**:** *$M \sqsubseteq^{il}_G N$ if $M \Downarrow_l v$ implies $N \Downarrow_l v$.*

$A \to B$**:** *$M \sqsubseteq^{il}_{A\to B} N$ if $M \Downarrow_l f * \lambda x\colon A.t$ implies $N \Downarrow_l g * \lambda x\colon A^{lin}.s$ and for all closures M' and N' such that $M' \sqsubseteq^{il}_A N'$ we have $\langle f, M'/x\rangle * t \sqsubseteq^{il}_B \langle g, N'/x\rangle * s$.*

The key point is now to show that the Y-combinator can be approximated by certain λ-terms Y^k with

$$\begin{aligned} Y^0_A &= \lambda x\colon A \to A.Y(\lambda x\colon A.x) \\ Y^{k+1}_A &= \lambda x\colon A \to A.x(Y^k_A x) \end{aligned}$$

In PCF$_e$, we call each application of the inference rules

$$\frac{f * t \Downarrow_l g * \lambda x\colon A.s \qquad \langle g, g * Y(\lambda x\colon A.s)/x\rangle * s \Downarrow_l v}{f * (Yt) \Downarrow_l v}$$

$$\frac{\langle g, g * (Y(\lambda x\colon A.s))/x\rangle * s \Downarrow_l v}{\langle f, g * (Y(\lambda x\colon A.s))/x\rangle * x \Downarrow_l v}$$

an *unfolding* of the term Yt and $Y(\lambda x\colon A.s)$ respectively.

Note that we have two rules for the Y-combinator. The first rule reduces the term t to the function $\lambda x\colon A.s$ and does one unfolding of the recursion. Further unfoldings are done by the second rule. As an example, consider the PCF-term

$$M = (Y(\lambda x\colon \iota \to \iota.\lambda y\colon \iota.\texttt{if iszero}\ 0\ \texttt{then}\ 1\ \texttt{else}\ x(\texttt{pred}\ y))1$$

If we denote by N the term $\texttt{if iszero}\ 0\ \texttt{then}\ 1\ \texttt{else}\ x(\texttt{pred}\ y)$, then by the first unfolding rule, $M \Downarrow_l v$ if

$$\langle Y(\lambda x{:}\, \iota \to \iota.\lambda y{:}\, \iota.N)/x, 1/y\rangle * N \Downarrow_l v$$

If we denote by f the environment $\langle Y(\lambda x{:}\, \iota \to \iota.\lambda y{:}\, \iota.N)/x, 1/y\rangle$, then by the second unfolding rule, this is the case if

$$\langle Y(\lambda x{:}\, \iota \to \iota.\lambda y{:}\, \iota.N)/x, f * \texttt{pred}\ y/y\rangle * N \Downarrow_l v$$

Now it is easy to see that the second expression evaluates to 1.

If during reduction of a term a closure is unfolded at most k times, we can replace the fixed point combinator for this closure by the approximation Y^k. The following lemma gives the details.

Lemma 5. *Assume $M \Downarrow_l f * t$ in PCF_e and that there are k unfoldings of a closure $g * Yu$ with $g * u \Downarrow_l \lambda x{:}\, A.s$ during the evaluation of M. Then $M' \Downarrow_l f' * t$, where M' and f' result from M and f respectively by replacing every occurrence of $\langle g, g * (Y(\lambda x{:}\, A.s))/x\rangle$ by $g^i * (Y^{k-i}x)$ with $1 \leq i \leq k$, or with $\langle g, g * Y^k(\lambda y{:}\, A.s)/x\rangle$, where*

$$\begin{aligned} g^1 &\stackrel{\text{def}}{=} \langle g, g * \lambda y{:}\, A.s/x\rangle \\ g^{i+1} &\stackrel{\text{def}}{=} \langle g^i, g^i * x/x\rangle \end{aligned}$$

Proof. If $f * Yu \Downarrow_l v$, then the last step in the derivation must have been

$$\frac{f * u \Downarrow_l g * \lambda y{:}\, A.s \qquad \langle g, g * Y(\lambda y{:}\, A.s/x\rangle * s \Downarrow_l v}{f * (Yu) \Downarrow_l v}.$$

Now assume that there have been k occurrences of the unfolding rule for the closure $g * Y(\lambda y{:}\, A.s)$. Then show by an induction over the derivation that $\langle g, g * Y(\lambda y{:}\, A.s)/x\rangle * s \Downarrow_l f * t$ iff $\langle g, g^i * (Y^{k-i}x)/x\rangle * s \Downarrow_l f' * t$ for some i with $1 \leq i \leq k$.

The next lemma is the syntactic analogue expressed via the approximation relation of defining the semantics of Yt as the upper bound of all terms $Y^k t$.

Lemma 6. *If $f * Y^k t \sqsubseteq^{il}_A u$ for all k, then also $f * Yt \sqsubseteq^{il}_A u$.*

Proof. If $f * Yt$ diverges, the lemma is vacuously true. Hence assume there exists a k such that $f * Y^k t \Downarrow_l v$. Hence also $f * t \Downarrow_l g * \lambda x{:}\, A.u$ for some s. Now use Lemma 5 to show by an induction over the type structure that if $f' * t \sqsubseteq^{il}_A u$ for all k, where f' arises from f as described in Lemma 5, then also $f * t \sqsubseteq^{il}_A u$.

Now we can show the crucial proposition, namely the each PCF-term and its translation into LPCF are related.

Proposition 7. *For all closures M in PCF_e, we have $M \sqsubseteq^{il}_A M^{lin}$.*

Proof. Easy induction over the structure of M except for the case Yt. For this case, consider any two environments f and g such that $f * x_i \sqsubseteq^{il}_{A_i} g * x_i$. We have to show that $f * Yt \sqsubseteq^{il}_A g * Yt^{lin}$. If $f * Yt$ diverges, this is obvious. Otherwise, assume $f * t \Downarrow_l f' * \lambda x{:}\, A.t'$. Now show by an induction over k that $f' * Y^k(\lambda x{:}\, A.t') \sqsubseteq^{il}_A g * Yt^{lin}$. The case $k = 0$ is obvious because the LHS diverges. Now we turn to the induction step. We have that $f' * Y^{k+1}(\lambda x{:}\, A.t') \Downarrow_l v$ implies that $\langle f', \langle f', f' * \lambda x{:}\, A.t'/x\rangle * (Y^k x)/x\rangle * t' \Downarrow_l v$. Because x does not occur freely in Y^k, this means also that $\langle f', f' * (Y^k \lambda x{:}\, A.t')/x\rangle * t' \Downarrow_l v$. By induction hypothesis for k, $f' * (Y^k \lambda x{:}\, A.t') \sqsubseteq^{il}_A g * Yt^{lin}$. By induction hypothesis for t, we have $g * t^{lin} \Downarrow_l g' * \lambda x{:}\, A^{lin}.s'$, and $\langle f', u/x\rangle * t' \sqsubseteq^{il}_A \langle g', u'/x\rangle * s'$ if $u \sqsubseteq^{il}_A u'$. Hence if we specialise u to $f' * (Y^k \lambda x{:}\, A.t')$ and u' to $g * Yt^{lin}$, we obtain that $\langle f', f' * (Y^k \lambda x{:}\, A.t')\rangle * t' \sqsubseteq^{il}_A g * Yt^{lin}$. Because $\langle f', f' * (Y^k \lambda x{:}\, A.t')/x\rangle * t'$ and $f * Y^{k+1} t$ evaluate to the same value, we get $f * Y^{k+1} * t \sqsubseteq^{il}_A g * Yt^{lin}$. Now Lemma 6 yields the claim.

If we define a relation R on PCF$_e$closures by $M\ R\ N$ if $M \sqsubseteq^{il} N^{lin}$, then this proposition states that this relation is reflexive. The definition of $\sqsubseteq^{il}_A$ implies directly that R is transitive. Hence R is a pre-order.

Next we define the opposite approximation relation $\sqsubseteq^{li}_A$ between closures of LPCF$_e$ and PCF$_e$ in exactly the same way.

Definition 8. *Define the* approximation relation $\sqsubseteq^{li}_A$ *between closures of LPCF$_e$ of type A^{lin} and closures of PCF$_e$ of type A by induction over the structure of types as follows:*

G**:** *$M \sqsubseteq^{li}_G N$ if $M \Downarrow_l v$ implies $N \Downarrow_l v$.*
$A \to B$**:** *$M \sqsubseteq^{li}_{A \to B} N$ if $M \Downarrow_l f * \lambda x{:}\, A^{lin}.t$ implies $N \Downarrow_l g * \lambda x{:}\, A.s$ and for all closures M' and N' such that $M' \sqsubseteq^{li}_A N'$ we have $\langle f, M'/x\rangle * t \sqsubseteq^{li}_B \langle g, N'/x\rangle * s$.*

In exactly the same way we obtain

Proposition 9. *For all closures M in PCF$_e$, we have $M^{lin} \sqsubseteq^{li}_A M$.*

We denote by $\cong^{il}$ the equivalence relation on closures of PCF$_e$ given by $M \cong^{il} N$ if $M \sqsubseteq^{il}_A N^{lin}$ and $N^{lin} \sqsubseteq^{li}_A M$.

For the full abstraction of the translation, we use a characterisation of observational equivalence of PCF$_e$-closures as the intersection of $\sqsubseteq^i$ and its opposite relation, and similarly for LPCF$_e$. These approximation relations are defined by

Definition 10. *(i) Let M and N closures of PCF$_e$. We define a family of relations $\sqsubseteq^i_A$ indexed by PCF-types by*

- *$M \sqsubseteq^i_G N$ if $M \Downarrow_l v$ implies $N \Downarrow_l v$;*
- *$M \sqsubseteq^i_{A \to B} N$ if $M \Downarrow_l f * \lambda x{:}\, A.t$ implies $N \Downarrow_l g * \lambda x{:}\, A.s$, and for all closures M' and N' such that $M' \sqsubseteq^i_A N'$, we have that $\langle f, M'/x\rangle * t \sqsubseteq^i_B \langle g, N'\rangle * s$.*

(ii) Let M and N closures of LPCF$_e$. We define a family of relations $\sqsubseteq^l_A$ indexed by PCF-types by

– $M \sqsubseteq^l_G N$ *if* $M \Downarrow_l v$ *implies* $N \Downarrow_l v$*;*
– $M \sqsubseteq^l_{A \multimap B} N$ *if* $M \Downarrow_l f * \lambda a\colon A.t$ *implies* $N \Downarrow_l g * \lambda a\colon A.s$*, and for all closures* M' *and* N' *such that* $M' \sqsubseteq^l_A N'$*, we have that* $\langle f, M'/a\rangle * t \sqsubseteq^l_B \langle g, N'/a\rangle * s$*;*
– $M \sqsubseteq^l_{!A} N$ *if* $M \Downarrow_l f * !t$ *implies* $N \Downarrow_l g * !s$*, and* $f * t \sqsubseteq^l_A g * s$*.*

Similarly to $\cong^{il}$, one defines equivalence relations $\cong^i$ ($\cong^l$) as the intersection of $\sqsubseteq^i$ ($\sqsubseteq^l$) and its opposite relation. Because the languages we are considering are purely functional we have the following Theorem

Theorem 11. *The relation* $\cong^i$ *restricted to PCF-terms coincides with observational equivalence on PCF, and* $\cong^l$ *restricted to LPCF-terms coincides with observational equivalence on LPCF.*

Proof. Given two PCF-terms t and s which are related by $\cong i$, one shows by an induction over the structure of contexts $C[\_]$ that $C[t]$ and $C[s]$ are related by $\cong^i$. Proposition 9 is crucial for establishing the application case.

Now assume two PCF-terms are observationally equivalent. One shows by an induction over the type structure that they are also equivalent modulo $\cong^i$.

This argument works in the same way for linear PCF.

Crole [Cro98] uses bisimulations to reason about observational equivalence for DILL with a non-terminating term added. Because he defines bisimilarity for the tensor product $A \otimes B$ simply in terms of bisimilarity for A and for B, bisimilarity and observational equivalence do not coincide in his case. As we do not consider tensor products here, this problem does not arise. If we added tensor products, Hasegawa's approach to logical relations [Has99b], which essentially defines the logical relation for the tensor product via its elimination rule, should work.

One shows with a similar method to Proposition 9

Proposition 12. *For all closures of the linear environment calculus of type* A^{lin}*, one has* $M \cong^l (M^{int})^{lin}$*.*

Now we can show the main theorem.

Theorem 13. *The translation* $\_^{lin}$ *of call-by-name PCF into linear call-by-name PCF is fully abstract.*

Proof. First, assume that that M and N are observationally equivalent closures in the environment calculus. Hence $M \cong^i N$, and by proposition 7, we have also $M \cong^{il} M^{lin}$ and $N \cong^{il} N^{lin}$. Using Proposition 12, one shows now by induction over the PCF-types that $M^{lin} \cong^l N^{lin}$. Hence M^{lin} and N^{lin} are observationally equivalent closures in the linear environment calculus.

For the other direction one shows by an induction over PCF-types that $M^{lin} \cong^l N^{lin}$ implies $M \cong^i N$.

4 Call-by-Value

In this section we show that the results for the call-by-name calculus hold also for a call-by-value calculus.

In the call-by-value operational semantics we evaluate arguments of functions before evaluating the body. Hence environments store only values. Values and environments for a call-by-value LPCF_e are now inductively defined as follows:

$$\begin{aligned} f_v &::= \langle\rangle \mid \langle f_v, v/x\rangle \mid \langle f_v, f_v * (Yt)/x_r\rangle \\ v &::= \mathtt{true} \mid \mathtt{false} \mid \mathbf{n} \mid f_v * \lambda x{:}\, A.t \mid\ !v \end{aligned}$$

The terms are the terms of call-by-name LPCF_e. Because in the sequel only environments which are values are considered we drop the subscript in f_v. We define the call-by-value PCF_e-calculus in the same way. The operational semantics is given by the clauses in Figure 3.

In a call-by-value calculus the distinction between recursive variables and intuitionistic variables is important: if we treated recursive variables like intuitionistic ones and replaced *e.g.*, the first unfolding rule by

$$\frac{f * t \Downarrow_v g * \lambda a{:}\,!A.t' \qquad g * Y(\lambda a{:}\,!A.t') \Downarrow_v v' \qquad \langle g, v'/x_r\rangle * t'[x_r/a] \Downarrow_v v}{f * (Yt) \Downarrow_v v}$$

each closure $f * Y(\lambda a{:}\,!A.t)$ would always diverge: by this rule, one condition for $f * Y(\lambda a{:}\,!A.t) \Downarrow_v v$ to hold is exactly $f * Y(\lambda a{:}\,!A.t) \Downarrow_v v$. Hence we treat recursive variables as in the call-by-name calculus.

The full abstraction result is still true for the call-by-value calculi. The same argument as in the previous section works. Note that in particular the approximants for the fixed point are defined using recursive variables. In this way the treatment of the fixed point combinator in the call-by-name calculus carries over directly to the call-by-value case.

Theorem 14. *The translation $\_^{lin}$ of call-by-value PCF into linear call-by-value PCF is fully abstract.*

Proof. Similar to proof of Theorem 13.

5 Conclusions

This paper is part of a collection, which describe the results of the xSLAM project, currently being undertaken in the School of Computer Science, University of Birmingham. This project's main goal is to improve the efficiency of functional programming, by exploring the notion of linearity, as a logical tool. The general structure of the project is as follows:

Let c be any of $\mathbf{n}$, `true` or `false`. Let f_t be the environment which contains the bindings for all intuitionistic variables of f and for all free linear variables of t.

$$\frac{}{v \Downarrow_v v}$$

$$\frac{}{f * c \Downarrow_l c}$$

$$\frac{}{\langle f, v/x, \boldsymbol{t/x} \rangle * x \Downarrow_v v}$$

$$\frac{}{\langle f, v/a, \boldsymbol{t/x} \rangle * x \Downarrow_v v}$$

$$\frac{f_t * t \Downarrow_v g * \lambda a\colon A.t' \qquad f_s * s \Downarrow_v v' \qquad \langle g, v'/a \rangle * t' \Downarrow_v v}{f * (ts) \Downarrow_v v}$$

$$\frac{f_t * t \Downarrow_v !v \qquad \langle f_s, v/x \rangle * s \Downarrow_v v'}{f * \mathtt{let}\ !x\ \mathtt{be}\ t\ \mathtt{in}\ s \Downarrow_v v'}$$

$$\frac{f * t \Downarrow_v v}{f * !t \Downarrow_v !v}$$

$$\frac{f * t \Downarrow_v g * \lambda a\colon !A.t' \qquad \langle g, g * Y(\lambda a\colon !A.t')/x_r \rangle * t'[x_r/a] \Downarrow_v v}{f * (Yt) \Downarrow_v v}$$

$$\frac{\langle g, g * (Y(\lambda a\colon !A.t))/x_r \rangle * t[x_r/a] \Downarrow_v v}{\langle f, g * (Y(\lambda x\colon !A.t))/x_r, \boldsymbol{t/x} \rangle * x_r \Downarrow_v v}$$

$$\frac{f_{t_1} * t_1 \Downarrow_v \mathtt{true} \qquad f_{t_2} * t_2 \Downarrow_v v}{f * \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 \Downarrow_v v}$$

$$\frac{f * t_1 \Downarrow_v \mathtt{false} \qquad f_{t_3} * t_3 \Downarrow_v v}{f * \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 \Downarrow_v v}$$

$$\frac{f * t \Downarrow_v \mathbf{n}}{f * (\mathtt{succ}\ t) \Downarrow_v \mathbf{n+1}}$$

$$\frac{f * t \Downarrow_v \mathbf{0}}{f * (\mathtt{pred}\ t) \Downarrow_v \mathbf{0}}$$

$$\frac{f * t \Downarrow_v \mathbf{n+1}}{f * (\mathtt{pred}\ t) \Downarrow_v \mathbf{n}}$$

$$\frac{f * t \Downarrow_v \mathbf{0}}{f * (\mathtt{iszero}\ t) \Downarrow_v \mathtt{true}}$$

$$\frac{f * t \Downarrow_v \mathbf{n+1}}{f * (\mathtt{iszero}\ t) \Downarrow_v \mathtt{false}}$$

Fig. 3. Operational semantics for call-by-value LPCF$_e$

(1) Categorical Semantics for (Linear) Explicit Substitutions
(2) Calculus of Linear Explicit Substitutions
(3) Operational Semantics
(4) Abstract Reduction Machine xLIN
(5) Applications

We have described our proposed calculus of Linear Explicit Substitutions in [GdPR98] and our (categorical) models, which work as a guiding line in [GdPR99]. Here we discuss the operational theory. We show that the transition from PCF to linear PCF preserves and reflects observational equivalence. This seems to justify to use linear PCF to optimise the execution of PCF-programs. In fact, we would like to have a stronger result. To exploit linearity properly we require a translation of PCF into linear PCF which uses as many linear variables as possible. If we restrict ourselves to refinements of the Girard translation we expect the results of this paper to hold in this case as well.

Clearly the fixpoint combinator, as generally construed in the literature [Bra94] and as we use it here is not *linearised* as much as one would like it to be. But it is an important feature of Girard's linear logic that we can keep the (intuitionistic) traditional results that were obtained before. This note follows this line and obtains traditional full abstraction results for linear lambda-calculi using conventional (logical relations) arguments. We hope for an improved understanding of linear recursion in the near future.

References

[Bie97] G.M. Bierman. Observations on a linear PCF. Technical Report 412, University of Cambridge Computer Laboratory, 1997.

[BP97] A. Barber and G. Plotkin. Dual intuitionistic linear logic. Technical report, LFCS, University of Edinburgh, 1997.

[Bra94] T. Braüner. The Girard translation extended with recursion. In *Proc. of CSL'94*, volume 933 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[Cro98] R.L. Crole. Soundness and incompleteness of bisimilarity for contextual equivalences in linear theories. Submitted, 1998.

[GdPR98] N. Ghani, V. de Paiva, and E. Ritter. Linear explicit substitutions. In *Proc. of Westapp'98*, 1998. Full version submitted for publication.

[GdPR99] Neil Ghani, Valeria de Paiva, and Eike Ritter. Categorical models of explicit subsitutions. In *Proc. of FoSSaCS'99*, Lecture Notes in Computer Science, 1999.

[Has99a] M. Hasegawa. Girard translation is full. Submitted, 1999.

[Has99b] M. Hasegawa. Logical predicates for intuitionistic linear type theories. In *Proceedings, Fourth International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[Pit97] A.M. Pitts. Operationally based theory of program equivalence. In P. Dybjer and A.M. Pitts, editors, *Semantics and Logics of Computation*. Cambridge University Press, 1997.

[Plo77] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[RdP97] E. Ritter and V. de Paiva. On explicit substitution and names (extended abstract). In *Proc. of ICALP'97*, LNCS 1256, pages 248–258, 1997.