Calculi of explicit substitutions: new results

Pierre LESCANNE, Zine-El-Abidine BENAISSA, and Daniel BRIAUD

Centre de Recherche en Informatique de Nancy (CNRS)
INRIA-Lorraine
Campus Scientifique, BP 239,
F54506 Vandœuvre-lès-Nancy, France

email: Pierre.Lescanne@loria.fr

Abstract. In this paper we present an approach to lambda calculus based on a careful examination of the computation process at a low level of granularity. This is called explicit substitution because it makes substitution an explicit concept in the language. In the course of the paper, we describe several calculi and we adopt the following plan. First we present calculi of explicit substitution and the main concepts which are associated to them. Then we present two applications of explicit substitutions, namely the implementation of functional programming languages and the higher order unification. We feel they are two key concepts in declarative programming and therefore appropriate to present for this conference.

1 De Bruijn's indices

For several reasons one may want to eliminate bound variables. In a quest to found mathematics, Bourbaki (1954) tried to avoid any reference to natural numbers since he did not want to use numbers before their definition for preventing vicious circle. For that, he used pointers as illustrated by Fig. 1. De Bruijn proposed instead to associate indices with each variables. The de Bruijn's index associated with a given variable position counts the number of λ one crosses before one gets to the binder of that variable. Thus if de Bruijn's indices are used, the term $(\lambda f.\lambda x.f(f(f\ x)))\ (\lambda y.y)$ is writen $\lambda(\lambda(\underline{1}(\underline{1}(\underline{1}\ \underline{0}))))\ (\lambda\underline{0})$. Indeed the variables f have to cross the λ which binds x before they get to the binder (hence the notation $\underline{1}$) when for x and for y, there is no binder to cross (hence the notation $\underline{0}$).

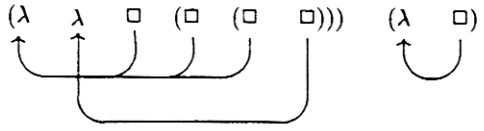


Fig. 1. Bourbaki's notations for $(\lambda f.\lambda x.f(f(f(x))))\ (\lambda x.x)$

The term $\lambda k.\mathsf{K}\ k\ k = \lambda k.((\lambda c.\lambda x.c)\ k\ k)$ written $\lambda(\lambda(\lambda\underline{1})\underline{0}\underline{0})$ with de Bruijn's indices β-reduces to the term $\lambda k.(\lambda x.k)\ k$ written $\lambda(\lambda(\underline{1})\ \underline{0})$ in de Bruijn's indices.

One notices two facts. First the same variable (here k) can be associated with different indices in the same term according to the context in which it occurs, Secund, indices must be shifted when a β-reduction is performed.

With de Bruijn's indices, one looses readability, but one gets an intrinsic and canonical notation for terms. In implementations, no symbol table is needed for getting the binder of a variable. Lescanne and Rouyer-Degli (1995) propose another notation introduced by de Bruijn (1972) namely *levels*, which makes also terms canonical, but in a more readable form, since the same variable is associated with the same natural number in the whole term where it occurs. In this case, the natural number which is associated with a variable is simply the *level* or the *depth* of is binder (see Section 3). Using indices or levels is not mandatory since Rose and Bloo (1995) used named variables to base their calculus of explicit substitution.

2 The calculus of explicit substitutions $\lambda\upsilon$

The description of the β-reduction by the rule

$$(\lambda x \cdot M)\ N \to M[N/x]$$

presents a drawback already noticed by Curry and Feys (1958). Indeed it refers to an external theory of substitution, that Curry calls the *epitheory* and which is "around" the theory of lambda-calculus. To avoid this drawback, he introduced *Combinatory Logic*, but at the same time he acknowledged the non intuitive character of that calculus. Calculi of explicit substitutions fulfill Curry's wish, since they handle the substitutions inside the lambda-calculus theory. Therefore they propose a unique and intuitive theory of lambda-calculus without epitheory. In this section we introduce a simple calculus of explicit substitution: the calculus $\lambda\upsilon$ (read *lambda-upsilon*) (Lescanne 1994; Benaissa, Briaud, Lescanne and Rouyer-Degli 1996). It is based on de Bruijn's indices; $\lambda\upsilon$ is the simpler calculus one may expect (see Kesner 1996)).

To set a calculus of explicit substitution, a notation for substitution has to be given. The tradition uses brackets; thus if one wants to assign a substitution s to a term M, one creates the term $M[s]$ and one says that $M[s]$ is a *closure*. In this context the rule β writes

$$(\lambda M)\ N \to M[N/] \qquad \text{(B)}$$

where $N/$ is a particular substitution which replaces (here in M) the index $\underline{0}$ by N and decrements the other indices of M. The behavior of the substitutions of type / is formally described by rules FVar and RVar of Fig. 2. In order to fully describe the behavior of the other substitutions, one must know how to distribute them into applications and abstractions. To distribute a substitution s in an application, one just applies it to each member of the application (rule App). To enter a substitution into an abstraction one has to modify it (rule Lambda); for that, one applies an operator $\Uparrow$ called *Lift*. $\Uparrow(s)$ is a substitution which does not modify the index $\underline{0}$, since when s was outside that abstraction in $(\lambda M)[s]$, s did not have access to that index, in other words, s in $(\lambda M)[s]$ and $\Uparrow(s)$ in $\lambda(M[\Uparrow(s)])$ should behave the same on $\underline{0}$ (rule FVarLift). On another hand, $\Uparrow(s)$ replaces the index $\underline{n+1}$ by the term which s would have assigned to $\underline{n}$, but this term is now in a new context and all the indices it contains must be incremented (rule RVarLift). That incrementation is done by applying a substitution called *Shift* and written $\uparrow$ whose behavior, namely the incrementation of the indices, is ruled by VarShift.

(B)	$(\lambda M)N \to M[N/]$
(App)	$(MN)[s] \to M[s]N[s]$
(Lambda)	$(\lambda M)[s] \to \lambda(M[\Uparrow(s)])$
(FVar)	$\underline{0}[M/] \to M$
(RVar)	$\underline{n+1}[M/] \to \underline{n}$
(FVarLift)	$\underline{0}[\Uparrow(s)] \to \underline{0}$
(RVarLift)	$\underline{n+1}[\Uparrow(s)] \to \underline{n}[s][\uparrow]$
(VarShift)	$\underline{n}[\uparrow] \to \underline{n+1}$

Fig. 2. The rewrite system $\lambda\upsilon$

In $\lambda\upsilon$ we distinguishe a specific subset, namely

$$\upsilon = \{\mathsf{App}, \mathsf{Lambda}, \mathsf{FVar}, \mathsf{RVar}, \mathsf{FVarLift}, \mathsf{RVarLift}, \mathsf{VarShift}\}$$

made of the rules which eliminate substitutions.

Example 1. The following sequences of rewrites illustrates de Bruijn indices and the system $\lambda\upsilon$. Let three be $\lambda(\lambda(\underline{1}(\underline{1}(\underline{1}\ \underline{0}))))$, let I be $(\lambda\underline{0})$ and let us reduce three I (the term of Fig. 1). In each rewriting step, the pattern of the used rule (or of the first rule in the case of a sequence of rewrites[1]) is boxed. The following sequence of reduction contains one application of rule B followed by a substitution elimination. Therefore this sequence of rules simulates one step of β-reduction. This computation may have been continued to the normal form $\lambda\underline{0}$.

$$\boxed{\lambda(\lambda(\underline{1}(\underline{1}(\underline{1}\ \underline{0})))) (\lambda\underline{0})} \xrightarrow{B} \boxed{\lambda(\underline{1}(\underline{1}(\underline{1}\ \underline{0})))[(\lambda\underline{0})/]}$$

$$\to_{\upsilon} \lambda(\boxed{(\underline{1}(\underline{1}(\underline{1}\ \underline{0})))[\Uparrow((\lambda\underline{0}))/]})$$

$$\xrightarrow{+}_{\upsilon} \lambda(\boxed{\underline{1}[\Uparrow((\lambda\underline{0})/)]}(\underline{1}[\Uparrow((\lambda\underline{0})/)](\underline{1}[\Uparrow((\lambda\underline{0})/)](\underline{0}[\Uparrow((\lambda\underline{0})/)]))))$$

$$\to_{\upsilon} \lambda(\boxed{\underline{0}[(\lambda\underline{0})/]}[\uparrow](\underline{1}[\Uparrow((\lambda\underline{0})/)](\underline{1}[\Uparrow((\lambda\underline{0})/)](\underline{0}[\Uparrow((\lambda\underline{0})/)]))))$$

$$\to_{\upsilon} \lambda(\boxed{(\lambda\underline{0})[\uparrow]}(\underline{1}[\Uparrow((\lambda\underline{0})/)](\underline{1}[\Uparrow((\lambda\underline{0})/)](\underline{0}[\Uparrow((\lambda\underline{0})/)]))))$$

[1] A non empty sequence of rewrites is denoted by $\xrightarrow{+}$ and a possibly empty sequence of rewrite is denoted by $\xrightarrow{*}$ or $\twoheadrightarrow$. The first notation is in the rewriting or relation calculus tradition whereas the second is in the lambda-calculus tradition. The reduction to normal form is written $\twoheadrightarrow_{\mathsf{N}}$.

$$\xrightarrow[\upsilon]{} \lambda(\lambda(\boxed{\underline{0}[\Uparrow(\uparrow)]})(\underline{1}[\Uparrow((\lambda\underline{0})/)](\underline{1}[\Uparrow((\lambda\underline{0})/)](\underline{0}[\Uparrow((\lambda\underline{0})/)]))))$$
$$\xrightarrow[\upsilon]{} \lambda(\lambda(\underline{0})(\boxed{\underline{1}[\Uparrow((\lambda\underline{0})/)]}(\boxed{\underline{1}[\Uparrow((\lambda\underline{0})/)]}(\underline{0}[\Uparrow((\lambda\underline{0})/)]))))$$
$$\xrightarrow[\upsilon]{+} \lambda((\lambda\underline{0})((\lambda\underline{0})((\lambda\underline{0})(\boxed{\underline{0}[\Uparrow((\lambda\underline{0})/)]}))))$$
$$\xrightarrow[\upsilon]{} \lambda((\lambda\underline{0})((\lambda\underline{0})((\lambda\underline{0})(\underline{0}))))$$

3 The calculus of explicit substitutions with levels

In the calculus $\lambda\chi$ (Lescanne and Rouyer-Degli 1995) described in this section, only variables of the form x_i are used with a very precise convention: the most external λ is λx_0 and for each i the first λ below λx_i is λx_{i+1}. Roughly speaking one has only terms of the form

$$\lambda x_0 \ldots \lambda x_1 \ldots (\lambda x_i.(\lambda x_{i+1} \ldots) \ldots) \ldots \lambda x_1.(\lambda x_i.(\lambda x_{i+1}) \ldots).$$

One associates also a "level" to a substitution, written $[\_]_i$, which is the number of λ's the substitution has crossed since it has been created. Actually a substitution is created at level 0 by application of rule B. The nine rules which deal with substitutions form the calculus χ and $\{\mathsf{B}\} \cup \chi$ is $\lambda\chi$ (see Fig. 3). $\lambda\chi$ contains four rules $\{\mathsf{App}_\alpha, \mathsf{Lambda}_\alpha, \mathsf{Var}_{\geq,\alpha}, \mathsf{Var}_{<,\alpha}\}$ which represent a form of explicit alpha-conversion. In other words, $\lambda\chi$ is explicit substitution plus explicit alpha-conversion.

(B)	$(\lambda x_i \cdot a)b \to a[b/x_i]_0$
(App)	$(a\ b)[c/x_i]_j \to a[c/x_i]_j\ b[c/x_i]_j$
(Lambda)	$(\lambda x_{i+j+1} \cdot a)[b/x_i]_j \to \lambda x_{i+j} \cdot (a[b/x_i]_{j+1})$
($\mathsf{Var}_>$)	$x_{i+k+1}[a/x_i]_j \to x_{i+k}$
($\mathsf{Var}_<$)	$x_i[a/x_{i+k+1}]_j \to x_i$
($\mathsf{Var}_=$)	$x_i[a/x_i]_j \to \alpha(a,i,j)$
(App_α)	$\alpha(a\ b,i,j) \to \alpha(a,i,j)\ \alpha(b,i,j)$
(Lambda_α)	$\alpha(\lambda x_{i+k} \cdot a,i,j) \to \lambda x_{i+k+j} \cdot \alpha(a,i,j)$
($\mathsf{Var}_{\geq,\alpha}$)	$\alpha(x_{i+k},i,j) \to x_{i+k+j}$
($\mathsf{Var}_{<,\alpha}$)	$\alpha(x_i,i+k+1,j) \to x_i$

Fig. 3. The calculus of explicit substitutions with levels $\lambda\chi$

4 Properties of calculi of explicit substitutions

This section describes some properties of $\lambda\upsilon$ and υ whose proofs may be found in (Benaissa et al. 1996). The rewrite system υ *terminates* (one also says that it is *strongly normalizing*), which means that υ admits only finite derivations. On another hand the system υ does not have *superpositions*, which means there is no term which can be rewritten at the same position into two different ways. It is also *left linear*, which means that all its left-hand sides are linear terms, i.e., terms with at most one occurrence of each variable. The conjunction of absence of superposition and left-linearity is called *orthogonality*. Moreover all patterns which contain a closure are *covered*, indeed one sees easily that all the configurations of closure can be rewritten; for instance the configuration $M[s][t]$ can be rewritten, since the term $M[s]$ is rewritten in a term without closure. Termination, orthogonality and coverture are three interesting properties, since they insure that one may always fully eliminate substitutions without using a specific strategy. No strategy will loop or block on an irreducible term. $\lambda\upsilon$ is a *faithful extension of lambda-calculus* in the sense that each $\xrightarrow[\beta]{}$ can be simulated by one step of $\xrightarrow[B]{}$ followed by a normalization by υ, more precisely

$$\xrightarrow[\beta]{} = \xrightarrow[B]{} \cdot \xrightarrow[\upsilon]{}^{!!}.$$

The system $\lambda\upsilon$ does not terminate since it contains the lambda-calculus which does not terminate. Moreover B and App superpose and admit a *critical pair*, i.e., a pair of terms

$$\langle\ M[N/][s],\ M[\Uparrow(s)][N[s]/]\ \rangle$$

created by reduction of the same term $((\lambda\, M)\ N)[s]$ (see Fig. 4). The terms $M[N/][s]$ and $M[\Uparrow(s)][N[s]/]$ cannot be reduced to the same normal form and the equality

$$M[N/][s] = M[\Uparrow(s)][N[s]/]$$

cannot be proved as an equality of the theory $\lambda\upsilon$. However this equality can be proved inductively for all the terms without variables (ground terms) even those which contain closures. It is often called the *substitution lemma*, since it corresponds to a fundamental lemma in lambda-calculus which says how substitutions can be permuted. It is worth noticing that this lemma has a very natural origin in terms of rewriting and explicit substitutions.

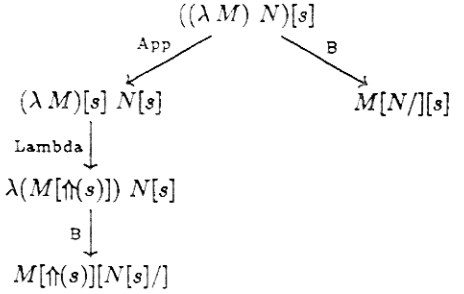
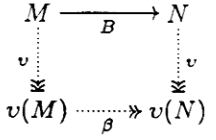
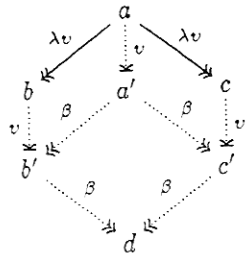


Fig. 4. Substitution lemma as a critical pair

As a consequence of the substitution lemma, one proves a *projection lemma* which says that if $M \xrightarrow[B]{} N$ then the projection of M by υ β-reduces to the projection of N by υ:



This is used to prove the *confluence* of $\lambda\upsilon$ on ground terms from the following schema:



Given a rewrite system, we say that a term M is strongly R normalizing or R-SN if every derivation which starts at M is finite. We know that there are terms that are not β-SN, but one may wonder whether a term which is β-SN is also $\lambda\upsilon$-SN. The answer is yes (Benaissa et al. 1996) and one says that $\lambda\upsilon$ *preserves strong normalization* or that $\lambda\upsilon$ is PSN, but the result is not so obvious since Melliès (1995) has shown that there are calculi of explicit substitution which are not PSN.

The system $\lambda\chi$ satisfies the same property as $\lambda\upsilon$, namely it is confluent on ground terms and PSN. On another hand, Muñoz (1996) has proposed a calculus $\lambda\zeta$ of explicit substitution which is both PSN and confluent on open terms, i.e., on terms with variables. $\lambda\zeta$ is an extension of lambda-calculus, but unfortunately in $\lambda\zeta$ the rule App is somewhat restrictive and $\lambda\zeta$ is not a faithful extention of lambda-calculus. More precisely, $\xrightarrow{}_{\beta}^{*} \subseteq \xrightarrow{}_{\lambda\zeta}^{*}$, but $\xrightarrow{}_{\beta} \neq \xrightarrow{}_{B} \cdot \xrightarrow{}_{\zeta}^{*}$ and $\xrightarrow{}_{\beta}$ is not represented in $\xrightarrow{}_{\lambda\zeta}^{*}$. This raises an interesting open problem formulated by Kristoffer Rose: does there exist a calculus of explicit substitution which is finite (i.e., only finitely many rules), PSN, confluent on open terms and a faithful extension of lambda calculus?

5 Higher Order Unification as a Typed Narrowing

Higher-order unification (HOU) solves equations where the unknowns may be functions, or, formally, equations between simply typed λ-terms. For instance,

$$x(f) == f$$

whose solutions are $x \mapsto \lambda u.u$ and $x \mapsto \lambda u.f$. As the simply typed λ-calculus is confluent and strongly normalizing, a simple and natural idea comes to mind, namely to unify via narrowing. The fact that β reduction is not a first-order rewrite relation constitutes the main difficulty.

A second difficulty is that HOU is impractical. Specifically, expressed as transformation rules (Snyder and Gallier 1989), the search tree leading to a set of unifiers is infinitely branching. To overcome this problem, Huet (1976) introduced the notion of preunifier. A given HOU problem has a preunifier if and only if it admits an HO-unifier. As a consequence, preunification, like HOU, is undecidable. Nevertheless, this is a practical and basic procedure of higher-order theorem provers, such as Paulson's (1990) Isabelle, or higher-order logic programming languages, such as λProlog (Miller and Nadathur 1986). Still, due to β-reduction and the way substitutions are usually handled, preunification is thought to be difficult to implement.

Dougherty (1993) avoids the first mentioned problem by means of a translation to simply typed combinatory logic (CL), which he called C-unification. This requires a trick since CL involves only weak β-reduction. Dougherty proves that a variant of typed narrowing is complete with respect to C-unification, but in his approach, he looses the structure of the λ-terms. Most of λ-calculi with explicit substitutions, such as $\lambda\sigma_{\uparrow\uparrow}$(Abadi, Cardelli, Curien and Lévy 1991; Ríos 1993) and $\lambda\upsilon$, aim at expressing β-reduction by means of a first order term rewriting system. Thus, they give another way of translating HOU into a first order setting and of unifying via narrowing. These calculi are quite different. For instance, $\lambda\upsilon$ tries to introduce as few operators as possible, whereas $\lambda\sigma_{\uparrow\uparrow}$ introduces derived operations such as the composition of explicit substitutions; $\lambda\sigma_{\uparrow\uparrow}$is confluent on open terms whereas $\lambda\upsilon$ is only ground confluent and PSN.

The work described in (Dowek, Hardin and Kirchner 1995) reduces HOU to a first-order equational unification in a theory presented by $\lambda\sigma_{\uparrow\uparrow}$ Then, although unification in such a calculus can be performed by narrowing, they present a specialized algorithm that computes $\beta\eta$-preunifiers for greater efficiency. Their approach could be improved in three respects: $\lambda\sigma_{\uparrow\uparrow}$is rather complex, the $\lambda\sigma_{\uparrow\uparrow}$ unification rules can be made more elementary, and their rules for β-unification are incomplete.

Departing from Dowek et al. (1995), our approach consists of two parts. First, the study in a simple setting of what we think is the heart of problem: how to decompose *in small steps* HOU by means of explicit substitutions. In other words, compute β-unifiers by means of $\lambda\upsilon$-narrowing. Second, the use of this study to design a preunification algorithm that makes small steps, easy to implement. In other words, compute $\beta\eta$-preunifiers in the spirit of $\lambda\upsilon$-narrowing. The work described here concerns the first part. Its first aim is to show that a system simpler than $\lambda\sigma$ is sufficient to do the whole job. In particular, since $\lambda\upsilon$ does not include the composition of substitutions, we show that such a composition is not mandatory to express HOU in an explicit substitutions framework. The second aim is to prove that $\lambda\upsilon$-narrowing is complete with respect to HOU and, as it is very simple to understand, to show that it provides an interesting alternative to build, at least manually, HO-unifiers.

5.1 Higher Order Unification

If M and N are two λ-terms of the same type then a *β-unifier* of M and N is a *substitution* θ such that $\theta(M) =_\beta \theta(N)$. A substitution is presented as a finite set of pairs (x_i, M_i). Applying such a substitution to a term N consists in replacing the free variables x_i of N by the terms M_i avoiding free variable of any M_i to be bound in the resulting term. Renaming bound variables in N prevents that capture. Such a renaming is part of the substitution process. Applying θ to N is also:

$$\theta(N) \equiv (\lambda x_1 \ldots x_n.N)M_1 \ldots M_n$$

For example, substituting x by y in $\lambda y.x$ yields $\lambda z.y$. On the contrary, first-order substitution, called *grafting* in this paper, does not take into account bound variables, hence, does not prevent the capture of free variables. For instance, grafting y for x in $\lambda y.x$ gives $\lambda y.y$. To sum up, substitution is grafting with renaming.

Let us illustrate HOU with an example. We consider a basic type i, $\tau(M)$ denotes the type of M and the following τ function,

$$\begin{array}{ll} \tau(g) = G = Y = i \to i \to i & \tau(x) = G \to Z \to Y \to i \\ \tau(a) = A = Z = i & \tau(y) = Y \\ & \tau(z) = Z \end{array}$$

The two λ-terms $xgzy$ and gaz admit as a β-unifier the substitution

$$\{(x, \lambda u_1 u_2 u_3.u_3(u_2, u_2)), (y, g), (z, a)\}$$

Generally, given a β-unification problem $M == N$, one is interested in a complete set of β-unifiers of $M == N$. Such a set, denoted $CSU_\beta(M == N)$, is a set of β-unifiers of M and N such that for any β-unifier θ of $M == N$, there is a substitution in $CSU_\beta(M == N)$ which subsumes θ.

We show now how to embed HOU into the $\lambda\upsilon$ framework. To ease the correspondence between β-unification and $\lambda\upsilon$-unification, it is convenient to abstract the free variables of the λ-terms to be unified as follows: let $\mathbf{x}$ be an ordering of the free variables of M and N, θ is a β-unifier of M and N iff θ is a β-unifier of $(\lambda\mathbf{x}.M)\mathbf{x}$ and $(\lambda\mathbf{x}.N)\mathbf{x}$. The interest of taking an ordering of the free variables of M and N comes from the equivalences:

$$\begin{aligned} \theta(M) &=_\beta \theta(N) \\ \theta((\lambda\mathbf{x}.M)\mathbf{x}) &=_\beta \theta((\lambda\mathbf{x}.N)\mathbf{x}) \\ (\lambda\mathbf{x}.M)\theta(\mathbf{x}) &=_\beta (\lambda\mathbf{x}.N)\theta(\mathbf{x}) \end{aligned}$$

$\lambda\mathbf{x}.M$ and $\lambda\mathbf{x}.N$ which contain no free variables are easily translated into terms with de Bruijn indices according to the following syntax where A is a type, f is a constant and $c_x \in C_V$.

$$\begin{aligned} Natn &::= 0 \mid n+1 \\ Pure\ Terma &::= f \mid c_x \mid \underline{n} \mid aa \mid \lambda A.a \end{aligned}$$

The set C_V is a mirror of the set of variables V. We do not detail here its motivation, which is quite technical.

The function $DB_\mathbf{x}$, where $\mathbf{x}$ is a sequence of variables, translates a λ-term to a pure term:

$$\begin{array}{lll} DB_\mathbf{x}(x) = j & & \text{if j is the index of the first occurrence of x in } \mathbf{x} \\ DB_\mathbf{x}(y) = c_y & & \text{if } y \notin \mathbf{x} \\ DB_\mathbf{x}(f) = f & & \\ DB_\mathbf{x}(MN) = DB_\mathbf{x}(M)\ DB_\mathbf{x}(N) & & \\ DB_\mathbf{x}(\lambda y.M) = \lambda A.\ DB_{y.\mathbf{x}}(M) & & \text{if } y \text{ is of type } A \end{array}$$

If $\mathbf{x}$ is void, then we shall write DB instead of $DB_\mathbf{x}$. For example,

$$\begin{aligned} DB(\lambda x.\lambda y.\lambda z.xgzy) &= \lambda X.\lambda Y.\lambda Z.\ \underline{2}\ g\ \underline{0}\ \underline{1} \\ DB(\lambda x.\lambda y.\lambda z.gaz) &= \lambda X.\lambda Y.\lambda Z.\ g\ a\ \underline{0} \end{aligned}$$

The function $BD_\mathbf{x}$, extracting a λ-term from a pure term can be defined likewise and will be used to decode $\lambda\upsilon$-unifiers. $DB_\mathbf{x}$ and $BD_\mathbf{x}$ are converse.

5.2 $\lambda\upsilon$-unification

Before introducing $\lambda\upsilon$-unification, we simply type $\lambda\upsilon$ and include variables of V.

$$\begin{array}{ll} Nat & n ::= 0 \mid n+1 \\ Term & a ::= x \mid f \mid c_x \mid \underline{n} \mid aa \mid \lambda A.a \mid a[s] \\ Subst & s ::= a/ \mid \uparrow \mid \Uparrow(s) \end{array}$$

where $x \in V$, f is a constant, $c_x \in C_V$.

Moreover we type B and *Lambda* and we add two rules to $\lambda\upsilon$ as follows:

B	$(\lambda A.x)b$	$\to x[b/]$
$Lambda$	$(\lambda A.x)[s]$	$\to \lambda A.x[\Uparrow(s)]$
$Const$	$f[s]$	$\to f$
$ConstVar$	$c_x[s]$	$\to c_x$

Like higher-order unification, $\lambda\upsilon$-unification is defined on the set $\Lambda\Upsilon$ of simply typed $\lambda\upsilon$-terms. Recall that $\lambda\upsilon$ is confluent on *ground* $\lambda\upsilon$-terms of $\Lambda\Upsilon$ and $\lambda\upsilon$ is strongly normalizing on $\Lambda\Upsilon$.

To illustrate how variables are mixed with ground terms, consider for instance

$$(\lambda x.\lambda y.\lambda z.xgzy)x \underset{\beta}{\longrightarrow} \lambda y.\lambda z.xgzy$$

With $\lambda\upsilon$, the combinator $\lambda X.\lambda Y.\lambda Z.\underline{2}\ g\ \underline{0}\ \underline{1}$ applied to the *variable* x can be rewritten:

$$\begin{aligned} (\lambda X.\lambda Y.\lambda Z.\underline{2}\ g\ \underline{0}\ \underline{1})x &\underset{Beta}{\longrightarrow} (\lambda Y.\lambda Z.\underline{2}\ g\ \underline{0}\ \underline{1})\ [x/] \\ &\xrightarrow[\lambda\upsilon]{*} \lambda Y.\lambda Z.((\underline{2}\ g\ \underline{0}\ \underline{1})\ [\Uparrow^2(x/)]) \\ &\xrightarrow[\lambda\upsilon]{*} \lambda Y.\lambda Z.\underline{2}\ [\Uparrow^2(x/)]\ g\ [\Uparrow^2(x/)]\ \underline{0}\ [\Uparrow^2(x/)]\ \underline{1}\ [\Uparrow^2(x/)] \\ &\xrightarrow[\lambda\upsilon]{*} \lambda Y.\lambda Z.\underline{0}\ [x/][\uparrow][\uparrow]\ g\ \underline{0}\ \underline{1} \\ &\xrightarrow[\lambda\upsilon]{} lY.\lambda Z.x\ [\uparrow][\uparrow]\ g\ \underline{0}\ \underline{1} \end{aligned}$$

In the last $\lambda\upsilon$-term, a grafting replaces x by a $\lambda\upsilon$-term. Unlike in the $\lambda\upsilon$-term $\lambda A.x$, no capture may happen in $\lambda Y.\lambda Z.x\ [\uparrow][\uparrow]\ g\ \underline{0}\ \underline{1}$, as the two $\uparrow$ act as renaming operators.

Precisely, a *grafting* is a function $\theta : V \to Term$, identified with its unique homomorphic extension, such that $x\theta \neq x$ for only finitely many $x \in V$.

Graftings play for $\lambda\upsilon$-unification the same role as substitutions for HOU. Specifically, if a and b are two $\lambda\upsilon$-terms of the same type then a *$\lambda\upsilon$-unifier* of a and b is a grafting θ such that $a\theta =_{\lambda\upsilon} b\theta$. Such an equation is denoted $a == b$. For example, one may check that $xgzy == gaz$ admits the grafting $\{x \mapsto \lambda GZY.\underline{1}\ \underline{2}\ \underline{2}, y \mapsto g, z \mapsto a\}$ as a $\lambda\upsilon$-unifier.

5.3 Embedding HOU into $\lambda\upsilon$-unification

Having defined both kinds of unification, we are now ready to embed HOU into $\lambda\upsilon$-unification. H associates a $\lambda\upsilon$-unification problem to a β-unification problem.

Let M, N be two λ-terms of type A, and $\mathbf{x}$ an ordering of $FV(MN)$. The $\lambda\upsilon$-unification problem associated with $M == N$ is

$$H(M) == H(N)$$

where

$$H(M) = DB(\lambda\mathbf{x}.M)\mathbf{x}, H(N) = DB(\lambda\mathbf{x}.N)\mathbf{x}$$

At first sight, the associated problem depends on $\mathbf{x}$, but in fact, this is not the case.

If $M = \lambda u.z$ and $N = \lambda u.a$ are of type $i \to i$, then

$$\begin{aligned} H(M) &= (\lambda Z\lambda i.\underline{1})z =_{\lambda\upsilon} \lambda i.z[\uparrow] \\ H(N) &= (\lambda Z\lambda i.a)z =_{\lambda\upsilon} \lambda i.a \end{aligned}$$

$\uparrow$ in $H(M)$ prevents z from being captured by a grafting like $[\underline{1}/z]$.

Theorem 1. *Let $M == N$ be a β-unification problem. The β-unifiers of $M == N$ are exactly the* ground *$\lambda\upsilon$-unifiers of $H(M) == H(N)$.*

5.4 $\lambda\upsilon$-narrowing

Consider the equation $xa == ay$ with the following types:

$$\begin{aligned} \tau(a) &= Y \to B = A \\ \tau(x) &= (Y \to B) \to B \\ \tau(y) &= Y \end{aligned}$$

Expressed in the $\lambda\upsilon$-unification framework, the equation $xa == ay$ is unchanged. Here is a branch of the search space explored by $\lambda\upsilon$-narrowing on that equation:

$$\begin{array}{l} xa == ay \\ \quad \Big\downarrow {\scriptstyle Beta}\ {\scriptstyle x \mapsto \lambda A.x_1 \quad \tau(x_1)=B} \\ x_1[a/] == ay \\ \quad \Big\downarrow {\scriptstyle App}\ {\scriptstyle x_1 \mapsto x_2x_3 \quad \tau(x_2)=Y\to B,\ \tau(x_3)=Y} \\ x_2[a/]x_3[a/] == ay \\ \quad \Big\downarrow {\scriptstyle FVar}\ {\scriptstyle x_2 \mapsto \underline{0}} \\ a(x_3[a/]) == ay \\ \quad \Big\downarrow {\scriptstyle Typed\ Unif} \\ y == x_3[a/] \end{array}$$

The $\lambda\upsilon$-narrowing steps of this branch build the non-ground $\lambda\upsilon$-unifier

$$\sigma = \{x \to \lambda A.(\underline{1}\ x_3),\ y \to x_3[a/]\}.$$

Theorem 2. *$\lambda\upsilon$-narrowing is a* ground *complete method of $\lambda\upsilon$-unification.*

5.5 Translation of non-ground $\lambda\upsilon$-unifiers

The issue is now to translate the graftings produced by $\lambda\upsilon$-narrowing into β-unifiers. Ground graftings are translated by BD. What about non-ground graftings ? In the last example, we get the non-ground $\lambda\upsilon$-unifier

$$\sigma = \{x \mapsto \lambda A.(\underline{1}\ x_3),\ y \mapsto x_3[a/]\}$$

We would like to get the β-unifier

$$\{(x, \lambda u.u(x_3(u))), (y, x_3a)\}$$

To translate σ, we just need to compose it with a ground grafting, called σ_B. The subscript B in σ_B means "back translation". The rôle of σ_B is the following: given a λ-term $\lambda A.\underline{0}\ x_3$, the fact that x_3 depends on $\underline{0}$,i.e. may be replaced by a term containing $\underline{0}$, is implicit. σ_B makes this dependency explicit:

$$\sigma_B = \{x_3 \mapsto c_z\ \underline{0}\}$$

Indeed, the new constant $c_z \in C_V$ is a function of $\underline{0}$, or, put it differently, depends explicitly on $\underline{0}$. Furthering the previous example, we get:

$$\sigma\sigma_B =_{\lambda\upsilon} \{x \mapsto \lambda A.\underline{0}(c_z\underline{0}), y \mapsto c_z a, x_3 \mapsto c_z\ \underline{0}\}$$

$\sigma\sigma_B$ is a ground $\lambda\upsilon$-unifier of $xa == ay$. Hence by theorem 1, it can be translated into a β-unifier of $xa == ay$:

$$\{(x, \lambda u.u(zu)), (y, za)\}$$

Theorem 3. *Let $M == N$ be a β-unification problem. Let S the set of graftings produced by $\lambda\upsilon$-narrowing and solving $H(M) == H(N)$. One can extract a $CSU_\beta(M == N)$ out of S.*

6 Explicit substitution as a basis for Functional programming implementation

In this section, we show how explicit substitution calculi can be used as a foundation for the implementation of functional programming languages by modeling the essential ingredients of such implementations, namely *reduction strategies*, *recursion*, *recursive data structures*, and *parallel evaluation*. We show that explicit substitution calculi give a unified model of several computational aspects. Furthermore, we show how such a tool enables us to understand and generalize optimizations in implementations.

Weak reduction Most implementations of functional programming languages are based on weak reduction of λ-calculus which makes no reduction in abstractions. In functional programming, abstractions are codes which evaluates data (parameter). However weak reduction is not confluent as shown in Figure 5. Curien,

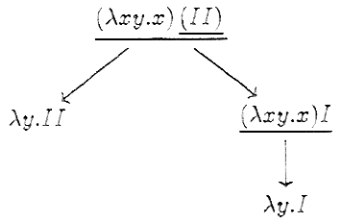


Fig. 5. Non confluence of weak reduction of λ-calculus

Hardin and Lévy (to appear) proposed a confluent weak substitution calculus $\lambda\sigma_w$ (see Figure 6). The idea behind this calculus is to forbid substitution in abstractions. The $\lambda\sigma_w$-terms are:

$$\begin{array}{rll} W ::= \underline{n} \mid WW \mid M[s] & & \text{Weak} \\ M, N ::= \lambda M \mid MN \mid \underline{n} & & \text{Pure} \\ s ::= W \cdot s \mid \mathrm{id} & & \text{Substitution} \end{array}$$

Notice that there is no substitution in the first argument (called a code) of a closure. Hence, weak $\lambda\sigma_w$ normal forms, called *values* are defined by $V ::= \lambda M[V \cdot \ldots . V] \mid \underline{n}\, V \ldots V$.

(B_w)	$(\lambda M)[s]\,W \to M[W \cdot s]$
(App)	$(MN)[s] \to M[s]\,N[s]$
(FVar)	$\underline{0}[W \cdot s] \to W$
(RVar)	$\underline{n+1}[W \cdot s] \to \underline{n}[s]$
(VarId)	$\underline{n}[\mathrm{id}] \to \underline{n}$

Fig. 6. The rewrite system $\lambda\sigma_w$.

How to model sharing ? Substitution calculi present the drawback (as for the λ-calculus) to duplicate computations. This duplication occurs in App which duplicates substitutions. In general, implementations avoid this duplication by sharing. To express this sharing in $\lambda\sigma_w$, we use the notion of *parallel reduction* (Huet 1980). The intuition behind this notion is to reduce simultaneously (in one step) a set of redexes. As a consequence we need to introduce a way to identify identical redexes. Our solution is to introduce the concept of *addresses* (Benaissa and Lescanne 1995; Rose 1996) which are *global* and *abstract* objects as pointers in heaps[2]. The use of addresses allows us to be very close to implementations. The *addressed Terms*, $\lambda\sigma_w^a$-terms, are:

$$\begin{array}{rll} T, U, V ::= E^a \mid \bot & & \text{Addressed} \\ E, F ::= M[s] \mid UV \mid \underline{n} & & \text{Evaluation Context} \\ M, N ::= \lambda M \mid MN \mid \underline{n} & & \text{Pure} \\ s, t ::= \mathrm{id} \mid U \cdot s & & \text{Substitution} \end{array}$$

By introducing addresses, we reduce in parallel redexes at the same address *i.e.*, all redexes addressed with the same address are reduced simultaneously. Figure 7 shows the addressed $\lambda\sigma$ rules. The rule App introduces two fresh addresses for the two application arguments, this rewriting corresponds to a heap allocation. There are two possibilities of Fvar rewriting, yielding two rules namely FVarE and FVarG. FVarE makes a copy of the term E at address b to the address a. Whereas FVarG makes a redirection of the pointer which refers to the address a (where the term $\underline{0}[E \cdot S]$ is) to the address b (where the term E is). In fact, this conceptual choice has a direct connection to the duplication vs sharing problem of implementations:

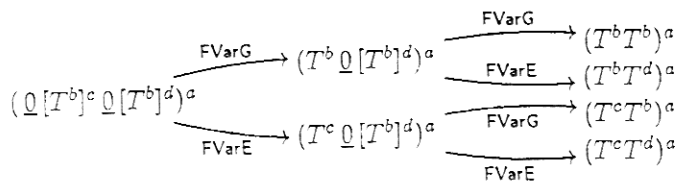
(B_w)	$((\lambda M)[s]^b\,U)^a \to M[U \cdot s]^a$	
(App)	$(MN)[s]^a \to (M[s]^b\,N[s]^c)^a$	b, c fresh
(FVarG)	$\underline{0}[E^b \cdot s]^a \to E^b$	
(FVarE)	$\underline{0}[E^b \cdot s]^a \to E^a$	
(RVar)	$\underline{n+1}[U \cdot s]^a \to \underline{n}[s]^a$	
(VarId)	$\underline{n}[\mathrm{id}]^a \to \underline{n}^a$	

Fig. 7. The rewrite system Pre-$\lambda\sigma_w^a$:

As a consequence, there is a nondeterministic choice between FVarE and

[2] in contrast with labels used by Maranget (1992) (which are local) to establish a theory of reduction and to define the call-by-need strategy.

FVarG. The following example illustrates this non determinism.



We obtain four possible different resulting terms namely $(T^bT^b)^a$, $(T^bT^d)^a$, $(T^cT^b)^a$, and $(T^cT^d)^a$. It is clear that erasing the addresses of these four terms produces the same term, namely TT: the difference between them is the amount of sharing (or shapes of the associated graph). The term $(T^bT^b)^a$ shares the two occurrences of T, whereas the two occurrences T in the three other terms are not shared. In fact the use of FVarG rule preserves sharing when FVarE unravels.

How to model strategies ? Strategies are described through restrictions of the $\lambda\sigma^a_w$-calculus. The crucial difference from traditional expositions in both cases is that we can exploit that *all redexes have a unique address*. We thus define a reduction with a strategy as a two-stage process (which can be merged in actual implementations, of course): we first give an inference system for "locating" a set of addresses where reduction can happen, and then we reduce using the original reduction constrained to just those addresses. In fact, a strategy S is a relation written $U \vdash_S A$ from addressed terms U to sets of addresses A. This set of addresses is a restriction of the following *strategy rules*:

$$\frac{}{M[s]^a \vdash \{a\}}\ \textsf{Sel} \qquad \frac{U \vdash A_1 \quad U \vdash A_2}{U \vdash A}\ A = A_1 \cup A_2\ \textsf{Par}$$

$$\frac{s \vdash A}{M[s]^a \vdash A}\ \textsf{Sub} \qquad \frac{U \vdash A}{U \cdot s \vdash A}\ \textsf{Hd} \qquad \frac{s \vdash A}{U \cdot s \vdash A}\ \textsf{Tl}$$

$$\frac{U \vdash A}{(UV)^a \vdash A}\ \textsf{Lap} \qquad \frac{V \vdash A}{(UV)^a \vdash A}\ \textsf{Rap}$$

Table 1 shows the conditions for several strategies; ✓ means "always applicable" and ÷ means "never applicable" (and blank means irrelevant as is the case for Hd and Tl when Sub is disallowed). Notice that there are two forms of non determinism in the strategies. One is due to Par, the only rule which contains the union operator and can yield more than one address for reduction. The other form of non determinism already mentioned in the previous paragraph (and here permitted only in the first strategy) is the choice between FVarE and FVarG.

The first strategy, "✓," allows every rule and gives us the full $\lambda\sigma^a_w$-calculus. CBN and CBV are just the call-by-name and call-by-value strategies of Plotkin (1975). Indeed CBN allows only Lap and disallows FVarG. In other words, reduce always the left redex without any sharing. CBV at most allows B_w if the argument U is a value and at most enables rap if the right argument U of an application is a value which defines in a small step the call-by-value. The next two

Strategy	B_w	FVarE	FVarG	Sel	Par	Sub	Hd	Tl	Lap	Rap
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CBN	✓	✓	÷	✓	÷	÷			✓	÷
CBV	U whnf	✓	÷	✓	÷	÷			÷	U whnf
CBNeedE	✓	E whnf	÷	✓	÷	$M=\underline{0}$	✓	÷	✓	÷
CBNeedG	✓	÷	✓	✓	÷	÷			✓	÷
Spine-$\|\|_n$	✓	A whnf	÷	✓	$\#A \leq n$	✓	✓	✓	✓	÷
Specul-$\|\|_n$	✓	A whnf	÷	✓	$\#A \leq n$	✓	✓	✓	✓	✓

Table 1. Strategies for $\lambda\sigma^a_w$

strategies are like CBN but add sharing in the way used by functional language implementation: CBNeedE is the call-by-need strategy (Launchbury 1993; Ariola, Felleisen, Maraist, Odersky and Wadler 1995) which is issued from *environment like machines*; CBNeedG is an adaption of Wadsworth's (1971) "λ-graph reduction" to weak reduction which is clearly issued from *graph like reduction machines*. CBNeedG is defined as CBN but it uses FVarG instead of FVarE. This rule preserves the sharing by using pointer redirection technique, keeping a single argument of a β-redex for many instances of it. CBNeedE uses FVarE. The following example shows this two different implementations of call by need strategy. Consider the term $t = (0[(II)[\mathrm{id}]^b]^c 0[(II)[\mathrm{id}]^b]^d)^a$.

$$\begin{aligned}
\text{CBNeedE:}\quad t &\xrightarrow[App]{b} (0[\underline{(I[\mathrm{id}]^f I[\mathrm{id}]^g)^b}]^c 0[\underline{(I[\mathrm{id}]^f I[\mathrm{id}]^g)^b}]^d)^a \\
&\xrightarrow[\mathsf{B}_w]{b} (0[\underline{0[I[\mathrm{id}]^g]^b}]^c 0[\underline{0[I[\mathrm{id}]^g]^b}]^d)^a \\
&\xrightarrow[FVarE]{b} (\underline{0[I[\mathrm{id}]^b]^c}\ 0[I[\mathrm{id}]^b]^d)^a \\
&\xrightarrow[FVarE]{c} (I[\mathrm{id}]^c 0[(I[\mathrm{id}]^b]^d)^a \\
\text{CBNeedG:}\quad t &\xrightarrow[FVarG]{c} (\underline{(II)[\mathrm{id}]^c}\ 0[\underline{(II)[\mathrm{id}]^c}]^e)^a \\
&\xrightarrow[App]{b} (\underline{(I[\mathrm{id}]^f I[\mathrm{id}]^g)^b}\ 0[\underline{(I[\mathrm{id}]^f I[\mathrm{id}]^g)^b}]^d)^a \\
&\xrightarrow[\mathsf{B}_w]{b} (\underline{0[I[\mathrm{id}]^g]^b}\ 0[\underline{0[I[\mathrm{id}]^g]^b}]^d)^a \\
&\xrightarrow[FVarG]{b} (I[\mathrm{id}]^g 0[I[\mathrm{id}]^g]^d)^a
\end{aligned}$$

The last two strategies realize *n-parallelism with sharing*. The *Speculative*-$||_n$ picks addresses everywhere in the term expecting that some reductions will be useful and the *Spine*-$||_n$ selects addresses in the head subterm disallowing Rap rule in order to compute the weak head normal form.

How to model data structures ? Algebraic data structures are added in the form of constructors that can be investigated using a 'case' selector statement. We extend pure $\lambda\sigma^a_w$-terms to include

$$M, N ::= \cdots \mid c_i(M_1, \ldots, M_m) \mid \langle c_1 : M_1, \ldots, c_m : M_m \rangle$$

where c_i's range over some finite set of constructors, and we add the rule

$$\begin{aligned}
&(c_i(M_1, \ldots, M_m)[s]^b\ \langle c_1 : N_1, \ldots, c_n : N_n \rangle\,[t]^c)^a \\
&\qquad \to N_i[M_1[s]^{d_1} \cdot \ldots \cdot M_m[s]^{d_m} \cdot t]^a \qquad d_1, \ldots, d_m \text{ fresh}
\end{aligned}$$ Case

The only reduction involving a data applies a selector $c_i(M_1,\ldots,M_m)[s]^b$ to a data $\langle c_1 : N_1,\ldots,c_n : N_n\rangle\,[t]^c$ resulting in an entry. For instance, consider the *list data structure* which contains two constructors namely Nil and $Cons(M,N)$. The following sequence of rewrites illustrates case on the list constructors.

$$\begin{aligned}
&(Cons(\underline{0}\,,N)\,\langle Nil : M_1, Cons : \underline{0}\rangle)[I[\mathrm{id}]^b]^a\\
&\quad\xrightarrow[App]{} (Cons(\underline{0}\,,N)[I[\mathrm{id}]^b]^c\,\langle Nil : M_1, Cons : \underline{0}\rangle\,[I[\mathrm{id}]^b]^d)^a\\
&\quad\xrightarrow[Case]{} \underline{0}\,[\underline{0}\,[I[\mathrm{id}]^b]^e\cdot I[\mathrm{id}]^b]^a\\
&\quad\xrightarrow[FVarE]{} I[\mathrm{id}]^a
\end{aligned}$$

How to model recursion ? Recursion in $\lambda\sigma^a_w$ is introduced by an explicit fixed point operator. $\lambda\sigma^a_w$-terms are extended to include the μ operator and use the *term unfolding rule* defined by

$$\mu M[s]^a \to M[\mu M[s]^a\cdot s]^b \qquad b \text{ fresh} \qquad \text{Unfold}$$

Unfold must of course be applied lazily to avoid infinite unfolding. However, this solution does not share redexes in M, for instance, if $\mu M[s]$ is unfolded 100 times and R is a redex in M then R is computed 100 times. Another solution consists in delaying unfolding until needed. The trick is to augment the explicit "horizontal" sharing that we have introduced in previous sections through addresses with explicit "vertical" sharing (using the terminology of Ariola and Klop 1994). We have chosen to do this using the $\bullet^a$ "backpointer" syntax of Rose (1996) (the notation was first described by Felleisen and Friedman 1989): reducing a fixed point operator places a $\bullet$ at the location where unfolding should happen when (if) it is needed. The difference is illustrated in Figure 8. Consider the initial

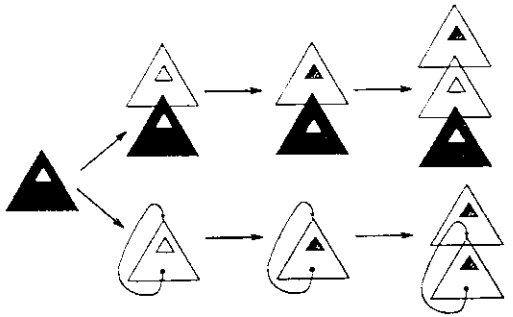


Fig. 8. Term unfolding versus verical sharing

term with a large (shaded) μ-redex containing a smaller (white) redex. Now, we wish to reduce the outer and then the inner redex. The top reduction shows how this will happen with Unfold: the redex is duplicated before it is reduced. In contrast using an explicit backpointer, illustrated at the bottom, makes it possible to share the redex by delaying the unfolding until the reduction has happened. The price we pay is that we no longer have the property that all redexes of a term are present in any representation of it, since backpointers can block particular redexes. The following rule extend $\lambda\sigma^a_w$ to include the cyclic reference.

$$\begin{aligned}
\mu M[s]^a &\to M[\bullet^a\cdot s]^a && \text{Cycle}\\
((\lambda M)[s]^b\,U)^a &\to M[U\cdot(s\{\bullet^b := \lambda M[s]^b\})]^a && \mathsf{B}^\bullet_w\\
\underline{0}\,[E^b\cdot s]^a &\to (E\{\bullet^a := \underline{0}\,[E^b\cdot s]^a\})^b && \text{FvarG}^\bullet\\
\underline{0}\,[E^b\cdot s]^a &\to (E\{\bullet^b := E^b\})^a && \text{FvarE}^\bullet
\end{aligned}$$

and Case from above changes as follows:

$$\begin{aligned}
&(c_i(M_1,\ldots,M_m)[s]^b\ \langle c_1 : N_1,\ldots,c_n : N_n\rangle\,[t]^c)^a\\
&\qquad\to N_i[M_1[s']^{d_1}\cdot\ldots\cdot M_m[s']^{d_m}\cdot t']^a \qquad d_1,\ldots,d_m \text{ fresh}
\end{aligned}\qquad \text{Case}^\bullet$$

7 Conclusion

This paper has presented calculi of explicit substitutions and two applications interesting in declarative programming, namely higher order unification and implementation of programming languages. It has profited of interesting discussions with Kristoffer Rose and Frederic Lang whom we would like to thank.

References

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

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. and Wadler, P. (1995). A call-by-need lambda calculus. *22nd POPL*. San Francisco, California. pp. 233–246.

Ariola, Z. M. and Klop, J. W. (1994). Cyclic lambda graph rewriting. *LICS*. IEEE Computer Society Press. Paris, France. pp. 416–425.

Benaissa, Z., Briaud, D., Lescanne, P. and Rouyer-Degli, J. (1996). λv, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming* . à paraître.

Benaissa, Z.-E.-A. and Lescanne, P. (1995). Triad machine; a general computational model for the description of abstract machines. *Technical Report 95-R-410*. Centre de Recherche en Informatique de Nancy.

Bourbaki, N. (1954). *Éléments de mathématiques: Théories des ensembles*. Vol. 1. Hermann & C^{ie}.

Curien, P.-L., Hardin, T. and Lévy, J.-J. (to appear). Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM* . also as 1992 INRIA report 1617.

Curry, H. B. and Feys (1958). *Combinatory Logic*. Vol. 1. Elsevier Science Publishers B. V. (North-Holland). Amsterdam.

de Bruijn, N. G. (1972). Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. Koninkl. Nederl. Akademie van Wetenschappen* 75(5): 381–392.

Dougherty, D. J. (1993). Higher-order unification via combinators. *Theoretical Computer Science* 114: 273–298.

Dowek, G., Hardin, T. and Kirchner, C. (1995). Higher-order unification via explicit substitutions, extended abstract. *In* Kozen, D. (ed.), *Proceedings of LICS'95*. San Diego. pp. 366–374.

Felleisen, M. and Friedman, D. P. (1989). A syntactic theory of sequential state. *Theoretical Computer Science* 69: 243–287.

Huet, G. (1976). *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Thèse de Doctorat d'Etat. Université de Paris 7 (France).

Huet, G. (1980). Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM* 27(4): 797–821. Preliminary version in 18th Symposium on Foundations of Computer Science, IEEE, 1977.

Kesner, D. (1996). Confluence properties of extensional and non-extensional λ-calculi with explicit substitutions. *In* Ganzinger, H. (ed.), *Proceedings 7th Conference on Rewriting Techniques and Applications, New Brunswick (New Jersey, USA)*. Springer-Verlag.

Launchbury, J. (1993). A natural semantics for lazy evaluation. *20th POPL*. pp. 144–154.

Lescanne, P. (1994). From λσ to λυ, a journey through calculi of explicit substitutions. *In* Boehm, H. (ed.), *Proceedings of the 21st Annual ACM Symposium on Principles Of Programming Languages, Portland (Or., USA)*. ACM. pp. 60–69.

Lescanne, P. and Rouyer-Degli, J. (1995). Explicit substitutions with de Bruijn's levels. *In* Hsiang, J. (ed.), *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*. Vol. 914 of *Lecture Notes in Computer Science*. Springer-Verlag. pp. 294–308.

Maranget, L. (1992). *La stratégie paresseuse*. Thèse de Doctorat d'Université. Université Paris VII.

Melliès, P.-A. (1995). Typed λ-calculi with explicit substitutions may not terminate. *In* Dezani, M. (ed.), *Int. Conf. on Typed Lambda Calculus and Applications*.

Miller, D. A. and Nadathur, G. (1986). Higher-order logic programming. *In* Shapiro, E. (ed.), *Proceedings of the Third International Logic Programming Conference*. Vol. 225 of *Lecture Notes in Computer Science*. Springer-Verlag. pp. 448–462.

Muñoz, C. (1996). Confluence and preservation of strong normalisation in an explicit substitutions calculus. *Proceedings 11th IEEE Symposium on Logic in Computer Science, New Brunswick (New Jersey, USA)*. also INRIA Tech-Report 2762.

Paulson, L. C. (1990). Isabelle: the next 700 theorem provers. *In* Odifreddi, P. (ed.), *Logic in Computer Science*. Academic Press.

Plotkin, G. D. (1975). Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science* 1: 125–159.

Ríos, A. (1993). *Contributions à l'étude des λ-calculs avec des substitutions explicites*. Thèse de Doctorat d'Université. U. Paris VII.

Rose, K. and Bloo, R. (1995). Deriving requirements for preservation of strong normalisation in lambda calculi with explicit substitution. Available as `ftp://ftp.diku.dk/diku/users/kris/Explicit-PSN.ps`.

Rose, K. H. (1996). *Operational Reduction Models for Functional Programming Languages*. PhD thesis. DIKU. Universitetsparken 1, DK-2100 København Ø. DIKU report 96/1.

Snyder, W. and Gallier, J. (1989). Higher order unification revisited: Complete sets of tranformations. *Journal of Symbolic Computation* 8(1 & 2): 101–140. Special issue on unification. Part two.

Wadsworth, C. (1971). *Semantics and pragmatics of the lambda calculus*. PhD thesis. Oxford.