Explicit Substitutions and Programming Languages

Jean-Jacques Lévy and Luc Maranget

INRIA - Rocquencourt,
`Jean-Jacques.Levy@inria.fr`
`Luc.Maranget@inria.fr`,
`http://para.inria.fr/∼{levy,maranget}`

Abstract. The λ-calculus has been much used to study the theory of substitution in logical systems and programming languages. However, with explicit substitutions, it is possible to get finer properties with respect to gradual implementations of substitutions as effectively done in runtimes of programming languages. But the theory of explicit substitutions has some defects such as non-confluence or the non-termination of the typed case. In this paper, we stress on the sub-theory of weak substitutions, which is sufficient to analyze most of the properties of programming languages, and which preserves many of the nice theorems of the λ-calculus.

1 Introduction

In the past ten years, several calculi of explicit substitutions have been proposed and studied with various motivations. In their original work, Curien [10] and Hardin [17] considered Categorical Combinators, as an algebraic definition of the syntax of the λ-calculus. In [1, 15], their calculus is simplified by using a two-sorted language, with terms and substitutions. The goal was there to study fundamental syntatic properties and applications to the design of runtime interpreters or fancy type-checkers. Unfortunately, the calculus in [1] is neither confluent (Church-Rosser property), nor strongly normalizable on the elementary first-order typed subset [28]. But it is confluent on closed terms (of explicit substitutions), which are sufficient to represent all λ-terms. Later several calculi were designed with full confluence [11], or with both properties by suppressing some of the operations of explicit substitutions such as the composition of substitutions[4, 23, 7]. Until very recently no fully expressive calculus existed with both properties of confluence and strong normalization. The termination problem, which is connected to cut elimination in linear logic[13], seemed more difficult; according to Martin-Löf or Melliès, it is due to an unlimited use of the η-expansion rule in the λ-calculus, which is known as non terminating when coupled with β-conversion. Recently, there has been a proposal for a new calculus of explicit substitutions with both the confluence and strong normalization properties [12], but this calculus seems rather complex. Therefore, one may be skeptic about the usage of these theories.

Explicit substitutions may be used for a refined study of logical systems with bound variables, for instance, when one wants to axiomatize α-conversion, in higher-order theorem provers, or in recent process algebras such as Action Calculi[30]. In some of these systems, renaming of bound variables has to be defined very carefully. This was the case with the axiomatization of the type-checker for Cardelli's Quest language, or with strategies for higher-order term matching[14].

But explicit substitutions were also introduced to have a formal theory of runtimes in programming languages, with implications for the CAM-machine (kernel of the Caml runtime) [9, 22] or Krivine's call-by-name machine. Usually, one then restricts attention to the theory of weak explicit substitutions. There is nothing really new with this remark, and much of the work at end of last decade was related to the correspondence between weak λ-calculus and runtimes of functional languages. But to our knowledge, the theory of weak explicit substitutions has not been much studied, mainly because its properties look easy, but this is quite often a "folk" statement.

In this paper, we present several weak theories, which may be consider as various exercises on weak λ-calculus, and we try to look carefully to the fundamental properties of their syntax. The claim is not that a useful theory has to be confluent or to preserve strong normalization in the typed case, but that keeping in mind these two properties could help for studying extra properties such as dependency analysis, shared evaluation, or stack allocation.

In section 2, following Çağman and Hindley in [8] for Combinatory Logic, we define a confluent calculus of weak λ-calculus, which is not a priori obvious since confluence of this weak theory often fails. This is achieved by allowing redexes under λ-abstractions to be contracted if they do not contain occurrences of bound variables. In section 3, we consider a confluent calculus of weak explicit substitutions exactly corresponding to the calculus of previous section. In section 4, we study the corresponding reductions strategies. In section 5, we map these strategies to runtime interpreters, and show how to state properties, such as stack-allocation or graph-based sharing. In section 6, we consider the ministep semantics of weak calculus of explicit substitutions, and show its connection to more traditional presentations of explicit substitutions with de Bruijn's indices. We conclude in section 7.

2 Confluence of the weak λ-calculus

As usual, the set of λ-terms is the minimum set of terms M, N defined by

$$M, N ::= x \mid MN \mid \lambda x.M$$

and the β-reduction rule is

$$(\beta) \qquad (\lambda x.M)N \to M[\![x\backslash N]\!]$$

where $M[\![x\backslash N]\!]$ is recursively defined by

$$\begin{aligned} x[\![x\backslash P]\!] &= N \\ y[\![x\backslash P]\!] &= y \\ (MN)[\![x\backslash P]\!] &= M[\![x\backslash P]\!]\ N[\![x\backslash P]\!] \\ (\lambda y.M)[\![x\backslash P]\!] &= \lambda y.M[\![x\backslash P]\!] \end{aligned}$$

In the last case, the substitution must not bind free variables in P, namely y must not be free in P. Usually in the λ-calculus (as in traditional mathematics), equality is defined up to the renaming of bound variables (α-conversion). Hence it is always possible to find y in $(\lambda y.M)$ filling the previous condition. Sometimes the substitution inside λ-abstractions is analogously defined as

$$(\lambda y.M)[\![x\backslash P]\!] = \lambda y'.M[\![y\backslash y']\!][\![x\backslash P]\!]$$

where y' is a variable not free in M and P.

In the (strong) λ-calculus, every context is active, since any sub-term may be reduced at any time. Formally, reduction is defined inductively by the following set of inference rules

$$(\xi)\ \frac{M \to M'}{\lambda x.M \to \lambda x.M'}$$

$$(\nu)\ \frac{M \to M'}{MN \to M'N} \qquad (\mu)\ \frac{N \to N'}{MN \to MN'}$$

In the weak λ-calculus, the ξ-rule is forbidden, and one cannot reduce inside λ-abstractions. This corresponds to the natural behavior in programming languages, since functions bodies cannot be evaluated without the actual values of their arguments. The ξ-rule corresponds more to partial evaluation of a function, and can be considered only as a compiling transformation.

But such a weak λ-calculus trivially looses the Church-Rosser property (confluence), since when $N \to N'$ we have

$$\begin{tikzcd} (\lambda x.\lambda y.M)N \arrow[r] \arrow[d] & (\lambda x.\lambda y.M)N' \arrow[d] \\ (\lambda y.M[\![x\backslash N]\!]) & (\lambda y.M[\![x\backslash N']\!]) \end{tikzcd}$$

Term $(\lambda y.M[\![x\backslash N]\!])$ is in normal form and cannot be reduced, and the previous diagram does not commute. The problem has been known for a long time in combinatory logic [19], although often kept as a "folk theorem". In [8], it is specifically stated, and shown as being relevant when translating the λ-calculus into combinatory logic. One recovers confluence by adding the new inference rule

$$(\sigma)\ \frac{N \to N'}{M[\![x\backslash N]\!] \to M[\![x\backslash N']\!]}$$

However, this rule is not pleasant, since it axiomatizes more parallel reduction than single reduction steps. Let us first say that a variable x is linear in term

M iff there is a unique occurrence of the free variable x in M. Then we prefer the following variant to the σ-rule

$$(\sigma')\ \frac{N \to N' \quad (x \text{ linear in } M)}{M\llbracket x\backslash N\rrbracket \to M\llbracket x\backslash N'\rrbracket}$$

An alternative statement of (σ') is possible with the context notation. Let a context $C[\ ]$ be a λ-term with a missing sub-term, and let $C[M]$ be the corresponding term when M is placed into the hole. Say that $C[\ ]$ binds M, when a free variable of M is bound in $C[M]$. Clearly when $C[\ ]$ does not bind M, one has $C[M] = C[x]\llbracket x\backslash M\rrbracket$ for any fresh variable x not in $C[M]$. Hence, the previous rule is equivalent to

$$(\sigma'')\ \frac{M \to M' \quad (C[\ ] \text{ does not bind } M)}{C[M] \to C[M']}$$

So, inside λ-abstractions, a redex (reductible expression) does not contain free variables bound in the outside context. Namely, a redex R in M is any sub-term $R = (\lambda y.A)B$ such that $M = M_1\llbracket x\backslash R\rrbracket$ for some free x linear in M.

Now, by noticing that the σ'-rule encompasses the μ and ν-rules, it is possible to define the theory of weak λ-calculus as the set of λ-terms with the β and σ' rules. As in [3], the transitive closure of $\to$ is written $\twoheadrightarrow$. So $M \twoheadrightarrow N$ iff M can reduce in several steps (maybe none) to N.

Theorem 1. *The weak λ-calculus is confluent.*

Proof: one can follow Tait–Martin-Löf's axiomatic method used to prove the Church-Rosser property. The main remark is that the problematic previous diagram

$$\begin{tikzcd}
(\lambda x.\lambda y.M)N \arrow[r] \arrow[d] & (\lambda x.\lambda y.M)N' \arrow[d] \\
(\lambda y.M\llbracket x\backslash N\rrbracket) \arrow[r, two heads] & (\lambda y.M\llbracket x\backslash N'\rrbracket)
\end{tikzcd}$$

now commutes since y is not free in N. □

In fact, the weak λ-calculus enjoys simple syntactic properties. For instance, in the standard λ-calculus, a rather complex theorem is the so-called finite development theorem, stating that the ordering in which a given set of redexes is contracted is not relevant and thus that there is a consistent definition for parallel reductions. This property is not easy to prove, since residuals of disjoint redexes may be nested. Take for instance term $(\lambda x.Ix)(Iy)$ with $I = \lambda x.x$. Then

$$(\lambda x.\underline{Ix})(\underline{Iy}) \to \underline{I(\underline{Iy})}$$

It is not the case in the weak λ-calculus since Ix in $(\lambda x.Ix)$ contains the bound variable x and is not a redex in the weak λ-calculus. In order to formally define residuals of redexes, we will use two ways. The first one uses named redexes by extending the set of λ-terms as follows

$$M, N ::= x \mid MN \mid \lambda x.M \mid (\lambda x.M)^a N$$

where a is taken in a given alphabet of names. The calculus is defined by the same β and σ' rules with the addition of a new β'-rule for contraction of named redexes

$$(\beta') \qquad (\lambda x.M)^a N \rightarrow M[\![x\backslash N]\!]$$

and substitution is extended by the following equation

$$((\lambda y.M)^a N)[\![x\backslash P]\!] = (\lambda y.M[\![x\backslash P]\!])^a \; N[\![x\backslash P]\!]$$

In order to track redexes along reductions, redexes may be named and the residuals are redexes with the same names in the term after reduction. Notice that (named) redexes must not contain external bound variables as implied by (σ') and it has to be shown that residuals of redexes are still redexes.

The second way is based on the substitution notation. Let R be a redex in M and let $M \rightarrow M'$ by contraction of R. Then $M = M_1[\![x\backslash R]\!]$, $M' = M_1[\![x\backslash R']\!]$ with x linear in M_1 and $R \rightarrow R'$. Let S be another redex in M. Then $M = N_1[\![y\backslash S]\!]$ with y linear in N_1. Then the residuals of S in M' are defined by case analysis:

– If S contains R, then $M_1 = N_1[\![y\backslash S_1]\!]$, $S = S_1[\![x\backslash R]\!]$ where S_1 is a redex and x linear in S_1. (Clearly, S_1 is a redex in M_1 since x not bound in M_1 cannot be bound in S_1). Then $M = N_1[\![y\backslash S_1]\!][\![x\backslash R]\!]$ and $M' = N_1[\![y\backslash S_1]\!][\![x\backslash R']\!]$. So $M' = N_1[\![y\backslash S_1[\![x\backslash R']\!]]\!]$. The residual of S is this unique redex $S' = S_1[\![x\backslash R']\!]$. It is indeed a redex since $M' = N_1[\![x\backslash R']\!]$. Notice too that $S \rightarrow S'$.

– If S does not contain R. Then $N_1 = N_2[\![x\backslash R_1]\!]$ with x linear in N_2, and $R = R_1[\![y\backslash S]\!]$ where y may not appear in R_1. Then $N_1 = N_2[\![x\backslash R_1]\!] \rightarrow N_2[\![x\backslash R_1']\!] = N_1'$ with $R_1 \rightarrow R_1'$. And $M = N_1[\![y\backslash S]\!] \rightarrow N_1'[\![y\backslash S]\!] = M'$. The residuals of S are all the S-redexes appearing at each occurrence of y in N_1'. Notice then that all residuals are equal to S. Again no free variable in S may be bound in M' since $M' = N_1'[\![y\backslash S]\!]$.
– If $M_1 = N_1$ and $x = y$. Then S and R coincide and S has no residual in M'.

This second definition shows that residuals of redexes are still redexes in the weak λ-calculus. We also remark that a residual R' of any redex R by a given many-step reduction is always such that $R \twoheadrightarrow R'$, which is not true in the normal λ-calculus where one may substitute the free variables of a redex and give much more reduction power to residuals. Finally, it remains to show that the two ways of defining redexes give identical definitions. We leave this in exercise to the reader.

Proposition 1. *Residuals of disjoint redexes are disjoint redexes.*

Proof: Let R and S be two disjoint redexes in M. The only way to get a residual S' of S to go through a residual R' of R by the contraction of a redex $(\lambda x.A)B$ in M is that $A = C[R]$ and x is a free variable of R substituted by $B = C'[S]$. But then R is no longer a redex of the weak λ-calculus, since it contains the free variable x bound in the external context. □

Let $\mathcal{F}$ be a set of redexes in M. A development of $\mathcal{F}$ is any maximal reduction only contracting residuals of redexes of $\mathcal{F}$.

Proposition 2. *Developments of sets of redexes are always finite.*

Proof: To each R in $\mathcal{F}$, we associate its maximal nesting level $n(R)$, where $n(R) = \max\{1 + n(S) \mid R \text{ directly contains } S \in \mathcal{F}\}$. So $n(R) = 0$ if R contains no redex in $\mathcal{F}$. Consider the multiset $\omega(\mathcal{F}) = \{n(R) \mid R \in \mathcal{F}\}$ with the natural multiset ordering. Then each step of the development decreases this multiset, since by previous proposition no new nesting appears in the residuals $\mathcal{F}'$ of $\mathcal{F}$. However, a redex contained in the contracted redex may have several copies as residuals, but then its nesting level is less that the one of the contracted redex which disappears. As the multiset ordering is well-founded, every development is finite. □
The rest of the finite development theorem (i.e. confluence and consistency of residuals) is proved as in the standard λ-calculus. A simple way is by use of the labeled weak λ-calculus defined above, and by showing that it is confluent and strongly normalizable when contracting only with the β'-rule. (The strong normalization proof follows exactly the previous outline used for finite developments by considering the nesting levels of β'-redexes).

Proposition 3. *Let $M \to M'$ by contraction of redex R, and let S' be a redex of the weak λ-calculus, residual of S in M not inside R. Then S is also a redex in the weak λ-calculus.*

Proof: Let R be the redex contracted in $M \to M'$. Then $M = M_1[\![x\backslash R]\!]$ and $M' = M_1'[\![x\backslash R']\!]$ where $R = (\lambda zA)B$ and $R' = A[\![z\backslash B]\!]$. We work by contradiction, and suppose S is not a redex. Then a free variable y of S is bound in M. We have several cases:

– R' and S' are disjoint. Then $S' = S$ and if y in S is bound in M, clearly it is same in M'.
– S contains R'. Then $S = S_1[\![x\backslash R]\!] \to S_1[\![x\backslash R']\!] = S'$. If the free y-variable in M is bound in M. Then either y is in S_1 and remains in S', which involves that S' is not a redex of the weak λ-calculus. Either y is in R, which contradicts the fact that R is also a redex of the weak λ-calculus.

□
The statement of the previous property may seem over-complicated, but some care is needed since for instance when $I = \lambda x.x$, an easy counterexample is $(\lambda z.Iz)a \to Ia$.

This proposition allows now to state another interesting theorem of the weak λ-calculus, namely Curry's standardization theorem. A standard reduction is usually defined as reduction contracting redexes in an outside-in and left-to-right way. Precisely a reduction of the form

$$M = M_0 \to M_1 \to \dots M_n = N \quad (n \geq 0)$$

is standard when for all i and j such that $i < j$, the R_j-redex contracted at step j in M_{j-1} is not a residual of a redex internal to or to the left of the R_i-redex contracted at step i in M_{i-1}. We write $M \underset{\text{st}}{\twoheadrightarrow} N$ for the existence of a standard reduction from M to N. Notice that the leftmost outermost reduction is a standard reduction (in the usual λ-calculus), but standard reductions may be more general.

Theorem 2. *If $M \twoheadrightarrow M'$, then $M \underset{st}{\twoheadrightarrow} M'$.*

Proof: One follows the proof scheme in [20] or checks the axioms of [29]. The basic step of the proof follows from proposition 3. Take $M \to N \to P$ by contracting R in M and S in N. Suppose R and S are not in the standard ordering. Then S is residual of a redex S' in M to the left of or outside R. By proposition 3, we know that S' is a redex of the weak λ-calculus and we may contract it getting N'. By the finite development theorem, we converge to P by a finite development of the residuals of R in N'. $\square$

3 Weak explicit substitutions

Although its language is minimal, some properties of the weak λ-calculus may look non intuitive and require at least a careful analysis. However it is close to a calculus of closures, which is the language of interpreters for functional languages. We now introduce such a calculus of closures, named the calculus of weak explicit substitutions, and study its connection to the weak λ-calculus.

The language contains terms which are reductible, and programs which are constant. The new terms with respect to the weak λ-calculus are closures represented as a λ-abstraction coupled with a substitution. We use same names for variables in terms and programs, and will precise their kind when necessary. Programs correspond to all λ-terms. Substitutions are functions from variables to terms represented by their (finite) graph. Notice that the domain of a substitution is always finite

$$\begin{array}{lll} M, N ::= & & \text{term} \\ & x & \text{variable} \\ & \mid\ MN & \text{application} \\ & \mid\ (\lambda x.P)[s] & \text{closure} \end{array}$$

$$\begin{array}{rll} P, Q ::= & x \mid PQ \mid \lambda x.P & \text{programs} \\ s ::= & (x_1, M_1), (x_2, M_2), \cdots (x_n, M_n) & x_i \text{ distinct } (n \geq 0) \end{array}$$

with $domain(s) = \{x_1, x_2, \cdots x_n\}$ and $s(x_1) = M_1, s(x_2) = M_2, \dots s(x_n) = M_n$. Notice that s is explicitely written as a set of pairs representing the graph of function s. Thus the ordering in which this graph is written does not matter. Now substitutions are extended to every program in the usual way.

$$\begin{aligned} PQ[\![s]\!] &= P[\![s]\!]Q[\![s]\!] \\ (\lambda x.P)[\![s]\!] &= (\lambda x.P)[s] \\ x[\![s]\!] &= s(x) \text{ if } x \in \mathit{domain}(s) \\ x[\![s]\!] &= x \ \text{ otherwise} \end{aligned}$$

Thus substitutions are applied to every subexpression of a program, except for lambda-abstraction where it stays in the substitution part of a closure. A substitution is modified by forcing one of its value

$$\begin{aligned} s[x\backslash N](y) &= N \qquad \text{if } y = x \\ &= s(y) \qquad \text{otherwise} \end{aligned}$$

The dynamics of weak explicit substitutions can now be defined by the following β-rule and inference rules for active contexts

$$(\beta) \qquad (\lambda x.P)[s]\ N \to P[\![\, s[x\backslash N]\,]\!]$$

$$(\xi)\ \frac{s \to s'}{(\lambda x.P)[s] \to (\lambda x.P)[s']}$$

$$(\nu)\ \frac{M \to M'}{MN \to M'N} \qquad (\mu)\ \frac{N \to N'}{MN \to MN'}$$

$$(\sigma)\ \frac{s(x) \to M' \quad s' = s[x\backslash M']}{s \to s'}$$

The calculus of weak explicit substitutions is nearly a first-order orthogonal term rewriting system. It manipulates sets for substitutions, which is not allowed in a standard rewriting system where only terms in a free algebra are considered. It is also defined with a scheme of axioms (the β-rule) with respect to programs. Anyhow, the calculus has the good properties of orthogonal systems.

Theorem 3. *The calculus of weak explicit substitutions is confluent.*

Proof: One proof uses the axioms in [29], another more direct proof may again follow the Tait–Martin-Löf's axiomatic technique. The only interesting cases are the two commuting diagrams

$$\begin{tikzcd} (\lambda x.P)[s]N \arrow[r] \arrow[d] & (\lambda x.P)[s']N \arrow[d] \\ P[\![\, s[x\backslash N]\,]\!] \arrow[r, two heads] & P[\![\, s'[x\backslash N]\,]\!] \end{tikzcd} \qquad \begin{tikzcd} (\lambda x.P)[s]N \arrow[r] \arrow[d] & (\lambda x.P)[s]N' \arrow[d] \\ P[\![\, s[x\backslash N]\,]\!] \arrow[r, two heads] & P[\![\, s[x\backslash N']\,]\!] \end{tikzcd}$$

when $s \to s'$ and $N \to N'$. We need then three lemmas showing that one has $s[x\backslash N] \twoheadrightarrow s'[x\backslash N]$, $P[\![s]\!] \twoheadrightarrow P[\![s']\!]$ and $s[x\backslash N] \to s[x\backslash N']$. $\square$

The standardization theorem also holds in weak explicit substitutions. The main difficulty is in its statement since residuals have to be defined, which can be done. The definition can again be done by considering named redexes, and extending the set of terms and reductions by

$$\begin{array}{lll} M, N ::= & & \text{term} \\ & \mid \ \ldots & \text{as previously} \\ & \mid \ (\lambda x.P)^a[s]\ N & \text{named redex} \end{array}$$

$$(\beta') \qquad (\lambda x.P)^a[s]\ N \to P[\![\, s[x\backslash N]\,]\!]$$

Proposition 1 can also be shown, and residuals of disjoint redexes keep disjoint. Take for instance

$$M = (\lambda x.Ix)[\,](\underline{I[\,]I[\,]}) \to \underline{I[(x, \underline{I[\,]I[\,]})](\underline{I[\,]I[\,]})} = N$$

when $I = \lambda x.x$. The external redex in N is not a residual of any redex in M. This also greatly simplifies the proof of the finite development theorem (see proposition 2).

We now consider translations of weak explicit substitutions into weak λ-calculus and vice-versa. First the former may be easily translated into the latter, since it suffices to expand substitutions through program abstractions. The translation from weak explicit substitutions to λ-calculus is

$$\begin{array}{l} \{x\} = x \\ \{MN\} = \{M\}\{N\} \\ \{(\lambda x.P)[s]\} = (\lambda x.P)[\![\,\{s\}\,]\!] \\ \{(x_1, M_1), (x_2, M_2), \cdots (x_n, M_n)\} = x_1\backslash\{M_1\}, x_2\backslash\{M_2\}, \cdots x_n\backslash\{M_n\} \end{array}$$

We assume that no variable x_j is free in a term $\{M_i\}$, which can always be achieved by renaming the x_i variables. Thus, none of the $[\![x_i\backslash M_i]\!]$ substitution interferes with another one and we may safely use the "parallel" substitution notation.

Proposition 4. *If $M \to M'$, then $\{M\} \twoheadrightarrow \{M'\}$.*

Proof: By structural induction on M. The key point is that, given a closure $(\lambda x.C[x_i])[\cdots(x_i, M)\cdots]$, either the context $C[\ ]$ does not bind x_i, or this occurrence of x_i does not refer the binding (x_i, M). $\square$

The converse translation from λ-calculus to explicit substitutions is a bit more involved. Several translations are possible for any given λ-term M. We consider the translation with maximal substitutions. Let P be a λ-term and Q a sub-term of P, so $P = C[Q]$ with the context notation. Say that Q is a free sub-term of P, iff $C[\,]$ does not bind Q. We will consider maximal free sub-terms of P. For instance, we underlined them in $\lambda x.x(\underline{y}(\lambda z.x\underline{z}))$ or in $\lambda x.x\underline{(y(\lambda z.yz))}$.

Notice that, given any λ-term P maximal free sub-terms are all disjoint. Hence, P can be written by using the natural generalization of the context notation to n holes. We get: $P = C[x_1, x_2, \cdots x_n][\![x_1\backslash P_1, x_2\backslash P_2, \cdots x_n\backslash P_n]\!]$, where P_1, P_2,… P_n are the maximal free sub-terms of P and x_1, x_2,… x_n are fresh variables all distinct. The translation from the weak λ-calculus to weak explicit substitutions is as follows

$\mathcal{I}(x) = x$
$\mathcal{I}(PQ) = \mathcal{I}(P)\mathcal{I}(Q)$
$\mathcal{I}(\lambda x.P) = (\lambda x.C[x_1, x_2, \cdots x_n])[(x_1, \mathcal{I}(P_1)), (x_2, \mathcal{I}(P_2)), \cdots (x_n, \mathcal{I}(P_n))$
where P_1, P_2,… , P_n are the maximal free sub-terms of P.

Proposition 5. *Given a λ term P, we have $\{\mathcal{I}(P)\} = P$.*

Proof: Easy, the choice of fresh variables for the x_i's is obviously crucial. $\square$

It is no surprise that the converse proposition does not hold, since $\mathcal{I}$ depends on which sub-terms are "abstracted out". Consider $M = (\lambda x.I)[\,]$, then we get $\mathcal{I}(\{M\}) = (\lambda x.y)[(y, I[\,])]$.

Proposition 6. *If $P \to P'$, then $\mathcal{I}(P) \to M$, with $\{M\} = P'$*

Proof: By structural induction. The key observation is as follows: let R the redex contracted in the reduction $P \to P'$, since R is a redex there exists Q_i, the maximal free sub-term of P that includes R. Then, we can apply the induction hypothesis to Q_i. $\square$

4 Reduction strategies with weak explicit substitutions

We consider three different evaluation strategies and show how they are naturally connected to executions of λ-interpreters. We start by call-by-value in weak explicit substitutions, which works on the following subset of terms

$$\begin{array}{rcll}
M, N & ::= & & \text{term} \\
 & & x & \text{variable} \\
 & \mid & MN & \text{application} \\
 & \mid & (\lambda x.P)[s_v] & \text{closure} \\
P, Q & ::= & x \mid PQ \mid \lambda x.P & \text{programs} \\
 & & & \\
V & ::= & x \mid (\lambda x.P)[s_v] & \text{values} \\
s_v & ::= & (x_1, V_1), (x_2, V_2), \cdots (x_n, V_n) & x_i \text{ distinct } (n \geq 0)
\end{array}$$

Values are either variables or closures. Notice that we take the convention that $xV_1V_2 \cdots V_n$ is not a value when $n > 0$. We could have taken a different convention, but it would have just complicated our semantics without much interest. We could also have decided that x was not a value, but this seems more speaking, especially when one adds constants to the set of terms. Now functions need values as arguments in the following β_v reduction rule.

$$(\beta_v) \qquad (\lambda x.P)[s_v]\, V \to P[\![\, s_v[x\backslash V]\,]\!]$$

For active contexts, it is sufficient to consider μ and ν rules, since ξ and σ rules can never be applied since s_v-substitutions are irreductible.

We first notice that redexes in the call-by-value strategy are innermost redexes in the calculus of weak explicit substitutions, but not all of them since we are interesting to reductions leading to values. An alternative way of expressing this strategy can be done with a bigstep SOS semantics and sequents of form $s \vdash P = V$, meaning that the result of evaluating P with substitution s is value V, as follows

$$s, (x, V) \vdash x = V$$

$$s \vdash x = x \quad (x \notin domain(s))$$

$$s \vdash \lambda x.P = (\lambda x.P)[s]$$

$$\frac{s \vdash P = (\lambda x.P')[s'] \quad s \vdash Q = V' \quad s'[x\backslash V'] \vdash P' = V}{s \vdash PQ = V}$$

Proposition 7. *$s \vdash P = V$ iff $P[\![s]\!] \twoheadrightarrow V$ in the calculus of call-by-value weak explicit substitutions.*

Proof: The proof is obvious by induction on the pair $(l, \|P\|)$ where l is the length of the reduction and $\|P\|$ is the size of P. $\square$

Obviously there are similar statements with call-by-name. The set of terms is now the full set of terms in the calculus of weak explicit substitutions. Values are variables or closures. The strategy is defined as the normal reduction $\xrightarrow[\text{norm}]{}\!\!\!\!\!\rightarrow$ which always contracts the leftmost outermost redex until reaching a value. The corresponding bigstep semantics is

$$\frac{s' \vdash P = V}{s, (x, (P, s')) \vdash x = V}$$

$$s \vdash x = x \quad (x \notin domain(s))$$

$$s \vdash \lambda x.P = (\lambda x.P)[s]$$

$$\frac{s \vdash P = (\lambda x.P')[s'] \quad s'[x\backslash (Q, s)] \vdash P' = V}{s \vdash PQ = V}$$

In fact, to model call-by-name, our bigstep semantics needed a new kind of delayed substitutions. Now substitutions may contained pairs (Q, s) for any program Q (and not only for program abstractions), which somehow correspond to the Algol 60 "thunks". A close treatment of them could be done with the calculus of gradual weak explicit substitutions exposed in section 6. A value of the bigstep semantics can be mapped to a value of the call-by-name calculus of weak explicit substitutions by replacing all sub-terms of form (Q, s) by terms $Q[\![s]\!]$. Let V^+ be the value mapped from V.

Proposition 8. *$s \vdash P = V$ iff $P[\![s]\!] \xrightarrow[norm]{} \!\!\!\!\rightarrow V^+$ in the calculus of weak explicit substitutions.*

The proof is similar to the one of proposition 7. Notice that by the standardization theorem, it is possible to show that the value computed by call-by-name is minimal, namely if $M \twoheadrightarrow V$, then $M \xrightarrow[\text{norm}]{} \!\!\!\!\rightarrow V_0 \twoheadrightarrow V$.

Call-by-need is more delicate, since one must represent some sharing of terms. Following techniques in [24, 5, 25, 27, 26], we build a confluent theory of shared reductions as follows. Terms and programs are labeled with names, names for programs are single letters a, b, c ... taken in a given alphabet, names for terms are strings α of letters which can be many-level underlined.

$$\begin{array}{rlr}
M, N ::= & x^\alpha \mid (MN)^\alpha \mid (\lambda x.P)^\alpha[s] & \text{labeled term} \\
P, Q ::= & x^b \mid (PQ)^b \mid (\lambda x.P)^b & \text{labeled programs} \\
s ::= & (x_1, M_1), (x_2, M_2), \cdots (x_n, M_n) & x_i \text{ distinct } (n \geq 0) \\
\alpha, \beta ::= & a \mid \alpha\beta \mid \underline{\alpha} & \text{labels}
\end{array}$$

The new labeled reduction rule β_l is defined as

$$(\beta_l) \qquad ((\lambda x.P)^\alpha[s]N)^\beta \to \beta \cdot (\underline{\alpha} \circ P)[\![\, s[x\backslash N]\,]\!]$$

where the labeled substitution $[\![\ ]\!]$ is defined inductively as follows in a labeled program

$$\begin{array}{rl}
x^\beta[\![s]\!] = & \beta \cdot s(x) \ \text{ if } x \in \mathit{domain}(s) \\
x^\beta[\![s]\!] = & x^\beta \ \text{ otherwise} \\
(PQ)^\beta[\![s]\!] = & (P[\![s]\!]\, Q[\![s]\!])^\beta \\
(\lambda x.P)^\beta[\![s]\!] = & (\lambda x.P)^\beta[s]
\end{array}$$

$$\begin{array}{rlrl}
\alpha \cdot x^\beta = & x^{\alpha\beta} & \alpha \circ x^b = & x^{\alpha b} \\
\alpha \cdot (MN)^\beta = & (MN)^{\alpha\beta} & \alpha \circ (PQ)^b = & ((\alpha \circ P)\, (\alpha \circ Q))^{\alpha b} \\
\alpha \cdot (\lambda x.P)^\beta[s] = & (\lambda x.P)^{\alpha\beta}[s] & \alpha \circ (\lambda x.P)^b = & (\lambda x.P)^{\alpha b}
\end{array}$$

Above, we used two external operations with labels and labeled terms to modify the external label of a term or to broadcast a label on a program. Notice that this labeled calculus is different from the labeled λ-calculus as in [3, 24, 25], which does not contain the broadcast operation. This new operation means that in the weak β-reduction we need fresh copies of application nodes from the body of the function before its application to an argument. However, it does not copy abstractions, and instead builds new closures.

Proposition 9. *In the labeled calculus of weak explicit substitutions, the following three lemmas hold*

$$\begin{array}{ll}
\textit{(i)} & (\alpha\beta) \cdot M \ = \ \alpha \cdot \beta \cdot M \\
\textit{(ii)} & M \to M' \ \Rightarrow \ \alpha \cdot M \to \alpha \cdot M' \\
\textit{(iii)} & s \to s' \ \Rightarrow \ M[\![s]\!] \twoheadrightarrow M[\![s']\!]
\end{array}$$

Proof: (i) is obvious by definition. And $(i) \Rightarrow (ii) \Rightarrow (iii)$. □

Proposition 10. *The labeled calculus of weak explicit substitutions is confluent.*

Proof: As in previous confluence proofs, we only consider local confluence, leaving the remaining part of the proof to the Tait–Martin-Löf's axiomatic method. When $s \to s'$ and $N \to N'$, we have two interesting cases corresponding to the two diagrams

$$\begin{tikzcd}
((\lambda x.P)^\alpha[s]\ N)^\beta \arrow[r] \arrow[d] & ((\lambda x.P)^\alpha[s']\ N)^\beta \arrow[d] \\
\beta \cdot (\underline{\alpha} \circ P)[\![\, s[x\backslash N] \,]\!] \arrow[r, two heads] & \beta \cdot (\underline{\alpha} \circ P)[\![\, s'[x\backslash N] \,]\!]
\end{tikzcd}$$

and

$$\begin{tikzcd}
((\lambda x.P)^\alpha[s]\ N)^\beta \arrow[r] \arrow[d] & ((\lambda x.P)^\alpha[s]\ N')^\beta \arrow[d] \\
\beta \cdot (\underline{\alpha} \circ P)[\![\, s[x\backslash N] \,]\!] \arrow[r, two heads] & \beta \cdot (\underline{\alpha} \circ P)[\![\, s[x\backslash N'] \,]\!]
\end{tikzcd}$$

which commutes by using lemmas of proposition 9. □

As in other theories of labeled λ-calculi, labels are useful for naming redexes, which are either residuals of redexes in a given initial term M, or created along reductions. The name of a redex is the string of labels on the path from its application node to its abstraction node, thus naming the interaction between these two nodes. So, the name of $((\lambda x.P)^\alpha[s]\,N)^\beta$ is α. A complete labeled reduction step $M \overset{\alpha}{\Longrightarrow} N$ is the finite development of all redexes with name α in M. We write $\Longrightarrow$ for a complete anonymous step, and $\Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow$ for several steps. Finally *Init*(M) will be the predicate which is true iff every label in M is a distinct letter. Intuitively, *Init*(M) means that term M does not contain shared sub-terms.

Proposition 11. *Let Init*(M_0) *and* $M_0 \Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow M$. *If* $M \overset{\alpha}{\Longrightarrow} N$ *and* $M \overset{\beta}{\Longrightarrow} M'$, *then* $N \overset{\beta}{\Longrightarrow} N'$ *and* $M' \overset{\alpha}{\Longrightarrow} N'$ *for some* N'.

Proof: The proof is far too complex to be exposed in this article. □
This property is nice since its shows that one has a confluent sub-theory of weak complete reductions.

Call-by-need strategies $\underset{\text{norm}}{\Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow}$ correspond to complete normal reductions in the labeled calculus of weak explicit substitutions when the initial labeled term M checks predicate *Init*(M). At each step, all redexes with the same name as the one of the leftmost outermost redex are contracted. One can show that the number of steps to get a value with this reduction is always minimal. No other reductions may get quicker any value. In difference with the theory of full λ-calculus, there is always a simple reduction $\twoheadrightarrow$ which can get the value with the same cost.

Now the goal is to define a bigstep semantics for call-by-need. We make sharing explicit by considering *stores* Σ, which are mappings from *locations*

ℓ to either thunks (P, s) or values V. Substitutions s now binds variables to locations. A store may appear as a context or as result in judgments all of the form $s \bullet \Sigma \vdash P = V \bullet \Sigma'$. Accessing to the value of a variable is now a bit more complex, since the value of its location can be of two kinds. When a value, it is as before. When a thunk, one has to evaluate it, and to modify the corresponding location in the store before returning the value and the modified store.

$$\frac{s' \bullet \Sigma, (\ell, (P, s')) \vdash P = V \bullet \Sigma', (\ell, (P, s')))}{s, (x, \ell) \bullet \Sigma, (\ell, (P, s')) \vdash x = V \bullet \Sigma', (\ell, V)}$$

$$s, (x, \ell) \bullet \Sigma, (\ell, V) \vdash x = V \bullet (\ell, V)$$

$$s \bullet \Sigma \vdash x = x \bullet \Sigma \quad (x \notin domain(s))$$

$$s \bullet \Sigma \vdash \lambda x.P = (\lambda x.P)[s] \bullet \Sigma$$

$$\frac{s \bullet \Sigma \vdash P = (\lambda x.P')[s'] \bullet \Sigma' \quad s'[x \backslash \ell] \bullet \Sigma', (\ell, (Q, s)) \vdash P' = V \bullet \Sigma''}{s \bullet \Sigma \vdash PQ = V \bullet \Sigma''}$$

In the last rule, we assume that ℓ is a fresh location.

Notice that, by contrast with Launchbury's [21], there is no need for renaming while substituting a variable in the first rule. No capture of variables can occur here. Another difference is that we make a clear distinction between variables and locations. Let now write M^*, P^*, s^* for the unlabeled terms, programs and substitutions obtained by erasing all labels within the labeled terms M, programs P, and substitutions s. Let also use the V^+ notation defined in the call-by-name subsection.

Proposition 12. *Let* $Init(P[\![s]\!])$*. Then* $s^* \bullet \emptyset \vdash P^* = V^* \bullet \Sigma$ *iff* $P[\![s]\!] \xRightarrow[norm]{} V^+$ *in the labeled calculus of weak explicit substitutions.*

Proof: The proof is again much complex to be presented here. □

5 Runtime interpreters

Sets may be represented by lists with two constructors *nil* and cons ::, and substitutions may become association lists. Substitutions may be then called environments. We do not duplicate the corresponding SOS of call-by-value with environments, which leads to the following recursive λ-evaluator

$eval(x, (x, V) :: s) = V$
$eval(x, (y, V) :: s) = eval(x, s)$
$eval(x, nil) = x$
$eval(\lambda x.P, s) = (\lambda x.P)[s]$
$eval(PQ, s) = eval(P', (x, eval(Q, s)) :: s') \quad$ if $eval(P, s) = (\lambda x.P')[s']$

Similarly with call-by-name, one gets the functional interpreter

$$\begin{aligned}
&\mathit{eval}(x, (x, (P, s')) :: s) = \mathit{eval}(P, s')\\
&\mathit{eval}(x, (y, (P, s')) :: s) = \mathit{eval}(x, s)\\
&\mathit{eval}(x, \mathit{nil}) = x\\
&\mathit{eval}(\lambda x.P, s) = (\lambda x.P)[s]\\
&\mathit{eval}(PQ, s) = \mathit{eval}(P', (x, (Q, s)) :: s') \quad \text{if } \mathit{eval}(P, s) = (\lambda x.P')[s']
\end{aligned}$$

Finally, the third interpreter is for call-by-need

$$\begin{aligned}
&\mathit{eval}(x, (x, \ell)) = V \quad \text{if } \ !\ell = (P, s') \text{ and } V = \mathit{eval}(P, s') \ \ (\text{side effect } \ell \leftarrow V)\\
&\mathit{eval}(x, (x, \ell)) = V \quad \text{if } \ !\ell = V\\
&\mathit{eval}(x, (y, \ell) :: s) = \mathit{eval}(x, s)\\
&\mathit{eval}(x, \mathit{nil}) = x\\
&\mathit{eval}(\lambda x.P, s) = (\lambda x.P)[s]\\
&\mathit{eval}(PQ, s) = \mathit{eval}(P', (x, \ell) :: s') \quad \text{if } \mathit{eval}(P, s) = (\lambda x.P')[s']\\
&\hspace{20em} \text{and } \ell = \mathrm{ref}(Q, s)
\end{aligned}$$

In the last case, we use mutable variables ℓ with the ML syntax for creation the reference ℓ and access to its content $!\ell$. The three interpreters are easy mappings of the previous SOS rules seen in the previous section, and their soundness with respect to this operational semantics is straightforward.

We now consider two problems on functional interpreters, and try more to state them within weak explicit substitutions than to give solutions, far beyond the scope of this paper.

The first one is stack allocation for closures. It is well-known that functional languages cannot be implemented with the Algol/Pascal stack discipline, since closures often need to be heap-allocated. In these imperative languages, each environment cell has an extra component, a link to the outer environment, also named "static link" in the compiler terminology. The stack is now represented by the environment at the left of each rule in our bigstep operational semantics. In the call-by-value case, the SOS semantics is now as follows

$$s, (n, (x, V)) \vdash x = V$$

$$\frac{s[1..n] \vdash x = V}{s, (n, (y, V')) \vdash x = V}$$

$$s \vdash \lambda x.P = (\lambda x.P)[\ |s|\ ]$$

$$\frac{s \vdash P = (\lambda x.P')[n] \quad s \vdash Q = V' \quad s, (n, (x, V')) \vdash P' = V}{s \vdash PQ = V}$$

where $|s|$ is the length of s, and $s[1..n]$ is the list of the first n elements of s. Intuitively, in the Algol/Pascal subset of terms, the values of arguments of functions can only refer to environments already in the stack. This is why closure values are represented as $(\lambda x.P)[n]$ where n refers to an entry in the current environment. Now it remains to show formally that this works. Clearly, it fails for any currified function. Take for instance, $s \vdash (\lambda x.\lambda y.x)Q = (\lambda y.x)[|s| + 1]$ which yields a result escaping from stack s.

In this implementation of environments with stacks, we need to leave the stack unchanged after the evaluation of each application, which makes the evaluation of currified functions failing. There are other techniques with stacks, dynamic ones as in Caml [22] when functions have arities, or with a static escape analysis as in [6]. The trick is then to try to keep the stack unchanged only after the evaluation of function bodies.

The second problem that we consider in this section is graph implementations of functional languages, which are mainly useful for lazy languages. So we are in the case of weak explicit substitutions with call-by-name. The weak labeled calculus of section 4 can be used to characterize these graphs. Clearly in the β_l-rule,

$$(\beta_l) \qquad ((\lambda x.P)^\alpha[s]N)^\beta \to \beta \cdot (\underline{\alpha} \circ P)[\![\, s[x\backslash N]\,]\!]$$

the broadcast operation $\alpha \circ P$ describes the creation of a fresh copy of the function body (except for its abstraction sub-terms). The rest of the term in which the β-reduction is performed remains unchanged, with the same sharing as before the reduction step. This operation was already considered by Wadsworth in his dissertation, but in the context of sharing for the full λ-calculus. In our case, an intuitive graph β-rule would be written

$$(\beta_g) \qquad ((\lambda x.P)[s]N) \to (\text{copy }(P))[\![\, s[x\backslash N]\,]\!]$$

However, it is fascinating how the proof of the correspondence between the labeled calculus and the straightforward graph implementation is complex. The goal is to prove that nodes connected by a same labeled path are identical in the graph implementation. Notice that this proof was already quite involved in the full λ-calculus where it required to build the so-called context semantics [16]. But one could expect a much simpler proof in the weak case.

6 Weak explicit substitutions with ministep semantics

Usually, the various authors working on explicit substitutions start here. Their goal is to represent not only bindings of variables, but also the way how substitutions are gradually pushed inside programs. Often, bindings are treated with de Bruijn numbers. Notice that we never used them, since names of variables were sufficient. This is because, in our weak calculi, substitution never cross binders, namely λ-abstractions or other substitutions. Therefore, one has not to care with α-renaming of bound variables. The second part of the motivation of the usual work on explicit substitutions is to study the progression of substitutions in terms, which we avoided since substitutions are always pushed to (free) variables or abstractions.

We consider the following ministep semantics for the calculus of weak explicit substitutions. The set of terms now allows closures on any program, and substitutions are represented by association lists.

$$\begin{array}{rll}
M, N ::= & & \text{term} \\
& x & \text{variable} \\
& \mid \; MN & \text{application} \\
& \mid \; P[s] & \text{extended closure} \\
\\
P, Q ::= & x \mid PQ \mid \lambda x.P & \text{programs} \\
\\
s ::= & \mathit{nil} & \text{empty substitution} \\
& \mid \; (x, M) :: s & \text{association list}
\end{array}$$

The reduction rules are defined by the following five non-overlapping left-linear rewriting rules

$$\begin{array}{rcl}
PQ[s] & \to & P[s]Q[s] \\
x[(x, M) :: s] & \to & M \\
x[(y, M) :: s] & \to & x[s] \\
x[\mathit{nil}] & \to & x \\
(\lambda x.P)[s]N & \to & P[(x, N) :: s]
\end{array}$$

Any sub-term may be reduced, as described by the following definition of active contexts

$$\frac{M \to M'}{MN \to M'N} \qquad \frac{N \to N'}{MN \to MN'}$$

$$\frac{s \to s'}{P[s] \to P[s']}$$

$$\frac{M \to M'}{(x, M) :: s \to (x, M') :: s} \qquad \frac{s \to s'}{(x, M) :: s \to (x, M) :: s'}$$

We do not detail the different proofs in this new calculus. Notice that the variables of this calculus (in the sense of term rewriting systems) are M, N, s. All other operators are constants. Thus this ministep calculus may be considered as an orthogonal system, and therefore is confluent, with the standardization theorem as stated in [29]. The normal strategy corresponds to the leftmost outermost reduction. And some simulation of the weak calculus of section 3 may easily be shown. Therefore this calculus also implements the weak λ-calculus. Finally, one can investigate sharing within this ministep semantics as in [26, 27], which leads to a ministep implementation of graph reduction, as considered at end of previous section or in interpreters of functional lazy languages [31]. Remark that then sharing works for all the set of reduction rules and not only for the β-rule.

7 Conclusion

So, before jumping in the full world of explicit substitutions we hope to have shown that the fundamental properties of the syntax of weak theories in the λ-calculus or in the weak calculus of explicit substitutions are still interesting. In this paper, we did not consider preservation of strong normalization in the typed case, but clearly it holds. We also restricted our study to the sole β-reduction, because it contains many of the problems, but δ-rules may be added in each of the three main calculi considered here, which rapidly provides the power of Plotkin's PCF or of a ML kernel. These extensions are rather easy since we have no critical pairs in the various term rewriting systems. Similarly data structures may be added (lists, records, algebraic structures) as done for records in [2].

We showed that our weak calculi are sufficiently expressive to describe the functional part of the programming languages runtimes, and could be used as a basis to model or to derive some program transformations or program analyses within compilers (stack allocation, graph implementation, dependency analysis, slicing) [6, 18, 2, 15]. But it would be very interesting to understand whether the fundamental properties of the underlining calculi are useful. For instance, how much of confluence or of the standardization property is really used? Also we would like to understand which of the three calculi presented here is the most useful.

Finally, many of the results of this paper were considered as folks theorems, rather easy to prove. We hope to have shown that some of the proofs may deserve some attention. In fact, some of them are not easy at all.

Acknowledgments

We thank Bruno Blanchet for stimulating discussions on stack allocation.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 6(2):pp. 299–327, 1996.
2. M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *Proc. of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages pp. 83–91. ACM Press, May 1996.
3. H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, 1981.
4. Z.-E.-A. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. Lambda-upsilon, a calculus of explicit substitution which preserves strong normalisation. Research Report 2477, Inria, 1995.
5. G. Berry and J.-J. Lévy. Minimal and optima l computations of recursive programs. In *Journal of the ACM*, volume 26. ACM Press, 1979.
6. B. Blanchet. Escape analysis : Correctness proof, implementation and experimental results. In *Proc. of 25th ACM Symposium on Principles of Programming Languages*. ACM Press, 1998.
7. R. Bloo and K. H. Rose. Combinatory reduction systems with explict substitutions thatpreserve strong normalization. In *In Proc. of the 1996 confence on Rewriting Techniques and Applications*. Springer, 1996.
8. N. Çağman and J. R. Hindley. Combinatory weak reduction in lambda calculus. *Theoretical Computer Science*, 198:pp. 239–249, 1998.

9. G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. In *Proc. of the second international conference on Functional programming languages and computer architecture*. ACM Press, 1985.
10. P.-L. Curien. *Categorical Combinator, Sequential Algorithms and Functional Programming*. Pitman, 1986.
11. P.-L. Curien, T. Hardin, and J.-J. L'evy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):pp. 362–397, 1996.
12. R. David and B. Guillaume. The lambda_l calculus. In *Proc. of the Second International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs*, 1999.
13. R. Di Cosmo and D. Kesner. Strong normalization of explicit substitutions via cut elimitation in proof nets. In *In Proc. of the 1997 symposium on Logics in Computer Science*, 1997.
14. G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions: the case of higher-order patterns. In M. Maher, editor, *In proc. of the joint international conference and symphosium on Logic Programming*, 1996.
15. J. Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proc. of the Seventeenth conference on Principles of Programming Languages*, volume 6, pages pp. 1–15. ACM Press, 1990.
16. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proc. of the Nineteenth conference on Principles of Programming Languages*, volume 8. ACM Press, 1992.
17. T. Hardin. Confluence results for the pure strong categorical logic ccl. lambda-calculi as subsystems of ccl. *Journal of Theoretical Computer Science*, 65:291–342, 1989.
18. T. Hardin, L. Maranget, and B. Pagano. Functional runtimes within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2), march 1998.
19. J. R. Hindley. Combinatory reductions and lambda reductions compared. *Zeit. Math. Logik*, 23:pp. 169–180, 1977.
20. J.-W. Klop. *Combinatory Reduction Systems*. PhD thesis, Mathematisch Centrum, Amsterdam, 1980.
21. J. Launchbury. A natural semantics for lazy evaluation. In *Proc. of the 1993 conference on Principles of Programming Languages*. ACM Press, 1993.
22. X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.
23. P. Lescanne. From lambda-sigma to lambda-upsilon, a journey through calculi of explicit substitutions. In *Proc. of the Twenty First conference on Principles of Programming Languages*, 1994.
24. J.-J. Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Univ. of Paris 7, Paris, 1978.
25. J.-J. Lévy. Optimal reductions in the lambda-calculus. In J. Seldin and J. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*. Academic Press, 1980. On the occasion of his 80th birthday.
26. L. Maranget. Optimal derivations in orthogonal term rewriting systems and in weak lambda calculi. In *Proc. of the Eighteenth conference on Principles of Programming Languages*. ACM Press, 1991.
27. L. Maranget. *La stratégie paresseuse*. PhD thesis, Univ. of Paris 7, Paris, 1992.
28. P.-A. Melliès. Typed lambda-calculus with explicit substitutions may not terminate. In *Proc. of the Second conference on Typed Lambda-Calculi and Applications*. Springer, 1995. LNCS 902.

29. P.-A. Melliès. *Description Abstraite des Systèmes de Réécriture*. PhD thesis, Univ. of Paris 7, december 1996.
30. R. Milner. Action calculi and the pi-calculus. In *Proc. of the NATO Summer School on Logic and Computation*. Marktoberdorf, 1993.
31. S. L. Peyton-Jones. *The implementation of Functional Programming Languages*. Prentice-Hall, 1987.