Using Fields and Explicit Substitutions to Implement Objects and Functions in a de Bruijn Setting

Eduardo Bonelli

[1] Laboratoire de Recherche en Informatique, Bât 490, Université de Paris-Sud, 91405, Orsay Cedex, France, email: bonelli@lri.fr
[2] Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina.

Abstract. We propose a calculus of explicit substitutions with de Bruijn indices for implementing objects and functions which is confluent and preserves strong normalization. We start from Abadi and Cardelli's ς-calculus [1] for the object calculus and from the λ_υ-calculus [20] for the functional calculus. The de Bruijn setting poses problems when encoding the λ_υ-calculus within the ς-calculus following the style proposed in [1]. We introduce fields as a primitive construct in the target calculus in order to deal with these difficulties. The solution obtained greatly simplifies the one proposed in [17] in a named variable setting. We also eliminate the conditional rules present in the latter calculus obtaining in this way a full non-conditional first order system.

1 Introduction

The object oriented paradigm is heavily used in the software engineering process. The simplicity of the underlying ideas makes it especially suited for resolving complex tasks. However, since no widespread consensus on its theoretical foundations has been reached, rigorous reasoning is difficult to achieve. In fact due to its success in software development the rapid evolution of object oriented languages has converted the task of formulating a formal calculus capturing the general principals of the paradigm into an interesting problem. In this direction, the calculi introduced by Abadi and Cardelli [1] constitute a simple yet powerful formalism.

The core untyped calculus presented in [1] is called the ς-calculus. This calculus defines objects as collections of methods and supports *method update*, thus providing mecanisms for inheritance by embedding. It also captures the notion of *self*, a name which allows a method to refer to its host object. These primitive constructs allow the representation of a vast amount of object oriented features, including classes, traits, and multiple inheritance. Furthermore, it may be extended into a typed setting.

Evaluation in the ς-calculus is accomplished by means of reduction rules and *substitution*. As in the lambda calculus, substitution is defined as an *atomic*

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 204–219, 1999.

operation which does not form part of the calculus. Therefore, any implementation has to deal with its computation. This is not a trivial task, in particular in a setting where variables are represented by names (as is usually done). Thus inevitably, a gap arises between theory and implementation. Calculi of explicit substitution eliminate this gap by decomposing the substitution process into more atomical parts and incorporating the behaviour of these parts as new operators in the calculus. This has the added benefit that we obtain a finer grained control on the computation of the substitution, providing for example tools for studying refinement of proofs in type theory or the theory of abstract evaluation machines.

Explicit substitution calculi arise with the study of pioneering calculus λ_σ [2]. The idea is simple: the notion of substitution used to define β-reduction in the lambda calculus takes place at a meta-level, explicit substitution calculi add new operators, and reduction rules for these new operators, so that substitutions may be computed at the object-level. Explicit substitution constructors thus implement substitution within the calculus, drawing the theory closer to the implementation level. Abadi, Cardelli, Curien and Lévy used indices, as introduced by de Bruijn in [8], to represent variables and introduced also a typed version of λ_σ. Other calculi of explicit substitutions are $\lambda_{\sigma_\Uparrow}$ [13], λ_v[20], λ_s [14], λ_d [11], λ_ζ [23], λ_χ [21], and λ_x [26]. All but the last two of these calculi have been formulated with de Bruijn indices, λ_χ uses de Bruijn levels and λ_x uses variable names. They have all been studied in the setting of the lambda calculus. Attempts to study explicit substitutions in a general setting are the Explicit CRS [5], based on the higher order rewriting formalism CRS [18], and the eXplicit Reduction Systems [24]. These formalisms although defined in a higher order rewriting setting deal with a fixed "built in" explicit substitution calculus (Σ in Explicit CRS and $\sigma_\Uparrow$ in XRS). This rises naturally the question of the generality of these formalisms as theories of explicit substitution. In particular, we shall see below that the calculus of explicit substitutions implementing the ς object oriented language fits in neither of these schemes.

In this paper we provide an implementation language for object oriented programming as formalized by the ς-calculus. We introduce the untyped ς_{DBES}-calculus, an explicit substitution calculus in a de Bruijn indice setting which is confluent and preserves strong normalization. Abadi and Cardelli's ς-calculus allows the execution of lambda calculus expressions by means of an elegant translation which requires the use of fields (see section 4). Although it does not provide fields as primitive constructions they can be simulated. A brief analysis, as discussed in section 5, shows that this simulation is not well adapted when variable names are replaced by de Bruijn indices and explicit substitutions are incorporated. The ς_{DBES}-calculus introduces fields as *primitive constructs* in the language, thus allowing to merge both the object oriented language and the functional lambda calculus in the spirit of [1].

The use of de Bruijn indices by encoding variable names with numbers avoids having to deal with α-conversion, thus simplifying the associated reduction relation.

Most importantly, the analysis pertaining to the merging of the object oriented language and the functional lambda calculus while retaining the spirit of the aforementioned translation revealed [17] that two different notions of substitution were necessary: Ordinary Substitution and Invoke Substitution. Ordinary Substitution is used to perform evaluation of methods and may be related to the usual notion of substitution which is made explicit in calculi for the lambda calculus. Whereas Invoke Substitution is used to implement functions as objects and reports a different behaviour. Also, some interaction between both types of substitutions must be specified. Therefore, higher order rewriting formalisms such as CRS [18], Explicit CRS [5] or XRS [24] do not cater for this difference. Consequently, the ς_{DBES}-calculus is not an instance of any of these formalisms.

The work reported here is very much in the spirit of [17]. There a calculus of explicit substitutions in a *named variable* setting for the ς-calculus, called ς_{ES}, is defined. The interaction between Ordinary and Invoke substitution is easier to express since one may specify conditions on free variables naturally. Whereas in a *de Bruijn setting* the situation is more complex since conditions on free variables imply adjustments on indices. The solution we have adopted by incorporating field constructs allows us, in contrast to [17], to do away with the conditions on free variables, thus obtaining a non-conditional interaction rule. Also, the calculus obtained here is a first order calculus. No binding operators are needed. As remarked above, just as the ς_{ES} calculus is not an instance of Explicit CRS, the ς_{DBES}-calculus is not an instance of an XRS.

This paper is organized as follows. Section 2 recalls the main concepts and definitions of the ς-calculus. Section 3 introduces the ς-calculus with de Bruijn indices, called ς_{DB}-calculus. Section 4 is devoted to the ς-calculus with de Bruijn indices and fields, the $\varsigma^{\bullet}_{DB}$-calculus. Here we introduce the syntax, we prove confluence and finally we show how it relates to the ς_{DB}-calculus. Also the invoke substitution is defined. Section 5 defines de $\varsigma^{\bullet}_{DB}$-calculus with explicit substitutions, called ς_{DBES}-calculus. The following section deals with the encoding of lambda calculus with explicit substitutions in the ς_{DBES}-calculus. Section 7 proves the main properties of ς_{DBES}: confluence and preservation of strong normalization. Finally, we conclude and suggest future research directions.

2 The ς-Calculus

This section presents the ς-calculus as defined in [1]. We have at our disposal an infinite list of variables denoted $x, y, z, \ldots$, and an infinite list of labels denoted $l, l_i, l', \ldots$. The labels shall be used to reference methods. An object is represented as a collection of methods denoted $l_i = \varsigma(x_i).a_i$. Each method has a reference or method name l_i and a method body $\varsigma(x_i).a_i$. The labels of an object's methods are assumed to be all distinct. Operations allowed on objects are *method invocation* and *method update*. A method invocation of the method l_j in an object $[l_i = \varsigma(x_i).a_i{}^{i\in 1..n}]$ is represented by the term $[l_i = \varsigma(x_i).a_i{}^{i\in 1..n}].l_j$. As a result of a method invocation, not only the corresponding method body is returned but also, this method body is supplied with a copy of its host object.

Thus method bodies are represented as $\varsigma(x_i).a_i$ where ς is a binder that binds the variable x_i in a_i. This variable called *self* will be replaced by the host object when the associated method is invoked. It is this notion of *self* captured by the ς-calculus that makes it so versatile. The other valid operation on objects is *method update*. A method $l_j = \varsigma(x_j).a_j$ in an object o may be replaced by a new method $l'_j = \varsigma(x'_j).a'_j$, thus resulting in a new object o'.

The terms of the ς-calculus, denoted $\mathcal{T}_\varsigma$, is given by the following grammar $a ::= x \mid a.l \mid a \lhd \langle l = \varsigma(x).a \rangle \mid [l_i = \varsigma(x_i).a_i{}^{i\in 1..n}]$.

A variable convention similar to the one present in λ-calculus is adopted: terms differing only in the names of their bound variables (i.e. α-equivalent) are considered identical.

We say that x is a *variable*, $a.l$ is a *method invocation*, $a \lhd \langle l = \varsigma(x).a \rangle$ is a *method update* and $[l_i = \varsigma(x_i).a_i{}^{i\in 1..n}]$ is an *object*.

In order to introduce reduction between terms the notions of free variables and substitution are defined as in [1]. The result of substituting a free variable x in a term a for a term b shall be denoted $a\{x \leftarrow b\}$.

The semantics of the ς-calculus, referred to as the *primitive semantics* in [1], is defined by the following rewrite rules:

$$\begin{array}{lll} o.l_j & \longrightarrow_\varsigma a_j\{x_j \leftarrow o\} & j \in 1..n \\ o \lhd \langle l_j = \varsigma(x).a \rangle & \longrightarrow_\varsigma [l_j = \varsigma(x).a,\ l_i = \varsigma(x_i).a_i{}^{i\in 1..n, i\neq j}] & j \in 1..n \end{array}$$

where $o \equiv [l_i = \varsigma(x_i).a_i{}^{i\in 1..n}]$.

The first rule defines the semantics of *method invocation*. The result of invoking the method l_j (a "call" to method $\varsigma(x_j).a_j$) is the body of the method a_j where the self variable has been replaced by a copy of the host object. The second rule defines the semantics of *method update*. Note that the substitution operator is not part of the ς-calculus but rather a meta-operation.

As regards the expressive power of this calculus, it is shown in [1] that lambda terms can be encoded as objects and that β-reduction can be simulated by ς-reduction.

Definition 1. *The translation* $\prec\!\prec . \succ\!\succ$ *from* λ*-terms to* $\mathcal{T}_\varsigma$ *is defined as:*

$$\begin{array}{lll} \prec\!\prec x \succ\!\succ & =_{def} & x \\ \prec\!\prec \lambda x.a \succ\!\succ & =_{def} & [arg = \varsigma(z).z.arg, val = \varsigma(x).\ \prec\!\prec a \succ\!\succ \{x \leftarrow x.arg\}] \\ \prec\!\prec ab \succ\!\succ & =_{def} & \prec\!\prec a \succ\!\succ \bullet \prec\!\prec b \succ\!\succ \\ & & where\ c \bullet d\ =_{def} (c \lhd \langle arg = \varsigma(y).d \rangle).val\ with\ y \notin FV(d) \end{array}$$

It is then proved for λ-terms a and b that if $a \longrightarrow_\beta b$ then $\prec\!\prec a \succ\!\succ \overset{*}{\longrightarrow}_\varsigma \prec\!\prec b \succ\!\succ$.

3 The ς-Calculus à la de Bruijn (ς_{DB}-Calculus)

Here we introduce the ς-calculus in a de Bruijn setting. N.G.de Bruijn introduced a notation for lambda terms which deals with the problem of having to rename bound variables when implementing mechanized provers [8].

Instead of labelling bound variables with names (as above) variables are labelled with natural numbers. This number is usually referred to as a de Bruijn index. If a term is viewed as a tree, an index n stands for a variable bound by the n-th binder starting from the position of the index. For example, the term $[l_1 = \varsigma(x_1).[l_2 = \varsigma(y_1).x_1, l_3 = \varsigma(z_1).z_1], l_4 = \varsigma(x_2).y_2]$ is represented as $[l_1 = \varsigma([l_2 = \varsigma(2), l_3 = \varsigma(1)]), l_4 = \varsigma(2)]$. Note that free variables are represented by indices greater than the number of binders above it, thus a variable assigned an index n that has m sigmas above it refers to the $(n-m)$-th free variable (in a preestablished ordering on the set of variables). The advantage attained is that there is no longer any need to perform renaming of bound variables. Nevertheless we must take care of index adjustments: if a substitution drags a term under a binder, its indices must be adjusted in order to avoid unwanted capture of indices.

The terms of the ς-calculus à la de Bruijn (the ς_{DB}-calculus), denoted $\mathcal{T}_{\varsigma_{DB}}$, are characterized by the grammar $a ::= \underline{p} \mid a.l \mid a \lhd \langle l = \varsigma(a) \rangle \mid [l_i = \varsigma(a_i)^{\ i\in 1..n}]$ where p is a natural number ($\mathbb{N}$) greater than zero. We shall use underlined natural numbers for indices.

Definition 2 (Ordinary Substitution). *Let a and b be pure terms and $n \geq 1$. The substitution of a by b at level n is defined as follows:*

$$
\begin{array}{ll}
[l_i = \varsigma(a_i)^{\ i\in 1..m}]\{n \leftarrow b\} & =_{def}\ [l_i = \varsigma(a_i\{n+1 \leftarrow b\})^{\ i\in 1..m}] \\
d.l\{n \leftarrow b\} & =_{def}\ d\{n \leftarrow b\}.l \\
d \lhd \langle l = \varsigma(c) \rangle \{n \leftarrow b\} & =_{def}\ d\{n \leftarrow b\} \lhd \langle l = \varsigma(c\{n+1 \leftarrow b\}) \rangle \\
\underline{p}\{n \leftarrow b\} & =_{def} \begin{cases} \underline{p-1} & if\ p > n \\ U_0^n(b) & if\ p = n \\ \underline{p} & if\ p < n \end{cases}
\end{array}
$$

where for every $i \geq 0$ and $n \geq 1$, $U_i^n(.)$ is an updating function from terms in $\mathcal{T}_{\varsigma_{DB}}$ to terms in $\mathcal{T}_{\varsigma_{DB}}$ defined as follows:

$$
\begin{array}{ll}
U_i^n([l_i = \varsigma(a_i)^{\ i\in 1..m}]) & =_{def}\ [l_i = \varsigma(U_{i+1}^n(a_i))^{\ i\in 1..m}] \\
U_i^n(a.l) & =_{def}\ U_i^n(a).l \\
U_i^n(a \lhd \langle l = \varsigma(c) \rangle) & =_{def}\ U_i^n(a) \lhd \langle l = \varsigma(U_{i+1}^n(c)) \rangle \\
U_i^n(\underline{p}) & =_{def} \begin{cases} \underline{p+n-1} & if\ p > i \\ \underline{p} & if\ p \leq i \end{cases}
\end{array}
$$

We now define the appropriate reduction rules using the notion of substitution defined above.

Definition 3 (Reduction in the ς_{DB}-calculus). *Reduction in the ς_{DB}-calculus is defined by the following rewrite rules:*

$$
\begin{array}{ll}
[l_i = \varsigma(b_i)^{\ i\in 1..n}].l_j & \longrightarrow_{\varsigma_{DB}}\ b_j\{1 \leftarrow [l_i = \varsigma(b_i)^{\ i\in 1..n}]\} \\
[l_i = \varsigma(b_i)^{\ i\in 1..n}] \lhd \langle l_j = \varsigma(c) \rangle & \longrightarrow_{\varsigma_{DB}}\ [l_j = \varsigma(c),\ l_i = \varsigma(b_i)^{\ i\in 1..n, i\neq j}]
\end{array}
$$

Notice that substitution is still a meta-operation in this calculus, completely external to the reduction rules of the formalism.

4 The ς-Calculus à la de Bruijn with Fields ($\varsigma^{\bullet}_{DB}$-Calculus)

The $\varsigma^{\bullet}_{DB}$-calculus is a straightforward extension of ς_{DB}-calculus. It is formulated in preparation for the introduction of explicit substitutions in Section 5 and shall also be used for proving some properties of this calculus of explicit substitutions.

From a general standpoint an object may be regarded as an entity encapsulating state (fields) and behaviour (methods) in an object-oriented language . These methods allow the object to modify its local state as well as interact with other objects. Let us concentrate on fields. Consider an object `calculator` that possesses a field which allows the user (another object) to store some intermediate result. For this the object interface includes a method `save(n)` where `n` is the number to be stored. Also, in order to retrieve this value it includes a method `recall`. Thus one would expect the equation `calculator.save(n).recall=n` to be true. This is characteristic of the behaviour of fields. As mentioned in [1] the ς-calculus does not include field contructs as primitive. Nevertheless, methods that do not use the self variable may be regarded as fields. Indeed, let b be a term in the ς-calculus such that it has no occurrence of a variable x. Then we have $[l = \varsigma(x).b].l \longrightarrow_{\varsigma} b\{x \leftarrow [l = \varsigma(x).b]\} \equiv b$ Thus we obtain exactly b, the body of the method $l = \varsigma(x).b$.

Now consider the setting where variables are represented no longer by variable names but by de Bruijn indices. Then we could attempt to proceed as above. Consider a term b in the ς_{DB}-calculus such that $1 \notin FV(b)$. Then we have, $[l = \varsigma(b)].l \longrightarrow_{\varsigma_{DB}} b\{1 \leftarrow [l = \varsigma(b)]\} \equiv b^{-}$ where b^{-} represents b with free indices decremented in one unit. The result obtained is not the same as the body of the method $l = \varsigma(b)$.

Thus we may simulate fields in ς_{DB}-calculus by representing them as methods $l = \varsigma(b^{+})$ where b^{+} represents b where all free indices are incremented in one unit. Nevertheless, we shall introduce fields as primitive constructs in the language. The reason for doing so is that when explicit substitutions are introduced into the calculus and the translation of (an explicit substitution version of) the λ-calculus into this extension studied, field simulation may no longer be performed (c.f. Section 5).

Therefore in our de Bruijn setting we incorporate, as a primitive notion, that of a field. The terms of the ς-calculus à la de Bruijn with fields (hereafter the $\varsigma^{\bullet}_{DB}$-calculus), denoted $\mathcal{T}_{\varsigma^{\bullet}_{DB}}$, are called *pure terms* and are characterized by the following grammar:

$$\begin{array}{lll} a & ::= & \underline{p} \mid a.l \mid a \lhd \langle m \rangle \mid [m_i{}^{i \in 1..n}] \\ m & ::= & l = g \mid l := a \\ g & ::= & \varsigma(a) \end{array}$$

where p is a natural number greater than zero.

An object is constructed by a list of methods and fields. A method is denoted "$l = g$" where l is its label and g its body. A field is denoted "$l := a$" where l is its label and a its body. Note that we may override a method with a field and viceversa.

We now define the appropriate reduction rules using the notion of substitution defined above.

Definition 4 (Reduction in the $\varsigma^{\bullet}_{DB}$-calculus). *Reduction in the $\varsigma^{\bullet}_{DB}$-calculus is defined adding the following rewrite rules to the rewrite rules of Definition 3:*

$$[l_j := a, {m_i}^{i\in 1..n, i\neq j}].l_j \longrightarrow_{\varsigma^{\bullet}_{DB}} a$$
$$[{m_i}^{i\in 1..n}] \lhd \langle l_j := a\rangle \longrightarrow_{\varsigma^{\bullet}_{DB}} [l_j := a,\ {m_i}^{i\in 1..n, i\neq j}]$$

Notice that substitution is still a meta-operation in this calculus, completely external to the reduction rules of the formalism.

The $\varsigma^{\bullet}_{DB}$-calculus is confluent. This may be proved using the proof technique presented in [27], a variation of the Tait-and-Martin Löf technique. Also, via a translation function that adjusts appropriately the indices of the bodies of fields it may be proved that the ς_{DB}-calculus can simulate the $\varsigma^{\bullet}_{DB}$-calculus. For details the reader is referred to [6].

5 Fields and Explicit Substitutions

The ς-calculus with explicit substitutions and de Bruijn indices which we shall hereafter refer to as the ς_{DBES}-calculus is presented in this section. This calculus introduces two forms of substitution into the object language: ordinary substitution and invoke substitution. Also, the need for using explicit fields is explained.

5.1 The ς_{DBES}-Calculus

The set of terms of the ς_{DBES}-calculus, denoted $\mathcal{T}_{\varsigma_{DBES}}$, consists of terms of sort `Term` and terms of sort `Subst`. These are defined by the following grammar (sort `Term` to the left and sort `Subst` to the right)

$$a ::= \underline{p} \mid a.l \mid a \lhd \langle m\rangle \mid [{m_i}^{i\in\ 1..n}] \mid a[s]$$
$$m ::= \overline{l} = g \mid l := a$$
$$g ::= \varsigma(a) \mid g[s]$$
$$s ::= a/ \mid @l \mid \Uparrow(s) \mid \uparrow$$

where p is a natural number greater than zero.

Unless otherwise stated when we say that "a is a term in $\mathcal{T}_{\varsigma_{DBES}}$" we mean "$a$ is a term in $\mathcal{T}_{\varsigma_{DBES}}$ of sort `Term`". A *closure* is a term of the form $a[s]$. A term that does not contain occurrences of closures as subterms is called a *pure term*. A term $a[s]$ may be regarded as the term a with pending substitution s. The substitution operator .[.] is part of the calculus (at the object-level). A substitution s with an occurrence of $a/$ is called an *ordinary substitution* whereas a substitution s with an occurrence of $@l$ is called an *invoke substitution*. More on invoke substitutions shall be said in Section 7. Note that if we erase the grammar rules generating closures then we obtain the set $\mathcal{T}_{\varsigma^{\bullet}_{DB}}$.

The substitution grammar (and substitution subcalculus) for ordinary substitution is based on the calculus of explicit substitution for the lambda calculus, λ_v [20].

We shall frequently use the notation $\Uparrow^i (s)$ and $a[s]^i$ defined inductively as:

$$\begin{array}{ll} \Uparrow^0 (s) \;\; =_{def}\; s & a[s]^0 \;\; =_{def}\; a \\ \Uparrow^{i+1} (s) =_{def} \Uparrow (\Uparrow^i (s)) & a[s]^{i+1} =_{def} a[s]^i[s] \end{array}$$

The semantics of the ς_{DBES}-calculus is defined by the following rewrite rules:

$$\begin{array}{lll}
[l_j = \varsigma(a), m_i{}^{i\in 1..n, i\neq j}].l_j & \longrightarrow_{MI} & a[[l_j = \varsigma(a), m_i{}^{i\in 1..n, i\neq j}]/] \\
[l_j := a, m_i{}^{i\in 1..n, i\neq j}].l_j & \longrightarrow_{FI} & a \\
[m_i{}^{i\,\in\, 1..n}] \lhd \langle l_j = g\rangle & \longrightarrow_{MO} & [l_j = g,\ m_i^{i\in 1..n, i\neq j}] \quad j \in 1..n \\
[m_i{}^{i\,\in\, 1..n}] \lhd \langle l_j := a\rangle & \longrightarrow_{FO} & [l_j := a,\ m_i^{i\in 1..n, i\neq j}] \quad j \in 1..n \\
(\varsigma(c))[s] & \longrightarrow_{SM} & \varsigma(c[\Uparrow (s)]) \\
[m_i{}^{i\in\, 1..n}][s] & \longrightarrow_{SO} & [m_i[s]\ {}^{i\in\, 1..n}] \\
(l := a)[s] & \longrightarrow_{SF} & l := a[s] \\
(l = g)[s] & \longrightarrow_{SB} & l = g[s] \\
a.l[s] & \longrightarrow_{SI} & a[s].l \\
a \lhd \langle m\rangle[s] & \longrightarrow_{SU} & a[s] \lhd \langle m[s]\rangle \\
\underline{1}[a/] & \longrightarrow_{FVar} & a \\
\underline{p+1}[a/] & \longrightarrow_{RVar} & \underline{p} \\
\underline{1}[@l] & \longrightarrow_{FInv} & \underline{1}.l \\
\underline{p+1}[@l] & \longrightarrow_{RInv} & \underline{p+1} \\
\underline{1}[\Uparrow (s)] & \longrightarrow_{FVarLift} & \underline{1} \\
\underline{p+1}[\Uparrow (s)] & \longrightarrow_{RVarLift} & \underline{p}[s][\uparrow] \\
\underline{p}[\uparrow] & \longrightarrow_{VarShift} & \underline{p+1} \\
a[\Uparrow^i (@l_j)][\Uparrow^i ([l_j := b, m_i{}^{i\in 1..n, i\neq j}]/)] & \longrightarrow_{CO} & a[\Uparrow^i (b/)] \\
a[\Uparrow^i (@l)][\Uparrow^k (s)] & \longrightarrow_{SW} & a[\Uparrow^k (s)][\Uparrow^i (@l)] \qquad k > i
\end{array}$$

The rule *MI* activates a method invocation. The rule *FI* activates a field invocation. The rules MO, FO activate method override and field override respectively. Rules SM, SO, SF, SB, SI, SU allow the propagation of the substitution operator through method body, object, field, method, invocation and override constructors. Rules $FVar, RVar, Finv, RInv, FVarLift, RVarLift, VarShift$ allow the computation of substitutions on indices. Finally, the rule *CO* expresses a form of interaction of substitutions, and *SW* expresses a (weak) form of commutation or switching of substitutions. These two rules will be used in simulating λ_v in the ς_{DBES}-calculus.

It is interesting to compare rules *RVar* and *RInv*. The creation of a substitution of the form $b/$ is accompanied by the elimination of a binder (see rule *MI*). Hence all "free" indices should be decremented in one unit. Whereas in the case of the invoke substitution operator "@" no such adjustment is made.

The ς_{DBES}-calculus without the rules MI, MO, FI and FO is referred to as the *ESDB* rewriting system. Note that *ESDB* is not locally confluent since for example the term $\underline{1}[@l_1][[l_1 := b]/]$ reduces to two different terms by the rules

FInv and *CO* respectively, and requires *FI* to close the diagram. The *ESDB* rewriting system is responsible for performing or discarding the substitution operators and additionally allows for some interaction between ordinary and invoke substitution operators. The rewriting system obtained by eliminating the rules for substitution interaction (rules *CO* and *SW*) is called the *BES* (Basic Explicit Substitution) rewriting system. This system suffices for executing substitutions; the interaction rules shall be needed when simulating λ_υ.

5.2 The Need for Explicit Fields

In Section 4 we saw that although the $\varsigma^\bullet_{DB}$-calculus incorporated fields as primitive constructs this is not strictly necessary as fields may be simulated in the ς_{DB}-calculus in a rather natural way. This situation no longer holds when explicit substitutions are introduced and when we attempt to translate the λ_υ-calculus into the $\varsigma^\bullet_{DBES}$-calculus using the translation in [1] (recalled in Section 2) for the λ-calculus.

Let us ignore fields as a primitive construct in the language for the moment and return to our simulation of fields as discussed in Section 4. A field b is represented as the method $l = \varsigma(b^+)$. The ς_{DBES}-calculus is then reduced to the, say, $\varsigma^\bullet_{DBES}$, where rules *FI*, *FO*, *SF* and *CO* have been eliminated.

Now when we attempt to translate the λ_υ-calculus into the $\varsigma^\bullet_{DBES}$-calculus in the style of $\prec\!\prec$. $\succ\!\succ$ we arrive naturally to the translation function k:

$$
\begin{array}{lll}
k(a/) & =_{def}\ k(a)/ & \qquad k(\underline{p}) =_{def} \underline{p} \\
k(\Uparrow(s)) & =_{def}\ \Uparrow(k(s)) & \qquad k(\uparrow) =_{def} \uparrow \\
k(a[s]) & =_{def}\ k(a)[k(s)] & \\
k(ab) & =_{def}\ (k(a) \lhd \langle arg = \varsigma(k(b)^+)\rangle).val & \\
k(\lambda a) & =_{def}\ [arg = \varsigma(1.arg), val = \varsigma(k(a)[@arg])] &
\end{array}
$$

But the meaning of $k(b)^+$ is no longer clear since $k(b)$ may have occurrences of the explicit substitution operator (it is no longer a pure term). To remedy this situation the next logical step would be to introduce an "explicit substitution version" of the $.^+$ operator which in fact we already have: the $\uparrow$ (shift) operator. Indeed, it is with the aid of the shift operator that updating is implemented explicitly (cf. Section 7 in [6]). The final clause of the definition of k is now replaced by $k(ab) =_{def} (k(a) \lhd \langle arg = \varsigma(k(b)[\uparrow])\rangle).val$

So now we proceed to verify that the translation is correct (preserves λ_υ-reduction). Consider for example the λ_υ-reduction rule $(\lambda a)b \longrightarrow_{Beta} a[b/]$ (cf. Section 6). Then we must have $k((\lambda a)b) \xrightarrow{*}_{\varsigma^\bullet_{DBES}} k(a[b/])$. We can go as far as:

$$
\begin{array}{ll}
k((\lambda a)b) & =_{def} \\
([arg = \varsigma(1.arg), val = \varsigma(k(a)[@arg])] \lhd \langle arg = \varsigma(k(b)[\uparrow])\rangle).val & \longrightarrow_{MO} \\
[arg = \varsigma(k(b)[\uparrow]), val = \varsigma(k(a)[@arg])].val & \longrightarrow_{MI} \\
k(a)[@arg][[arg = \varsigma(k(b)[\uparrow]), val = \varsigma(k(a)[@arg])]/] &
\end{array}
$$

Thus in order to arrive at $k(a)[k(b)/]$ we are in need of adding to the $\varsigma^\bullet_{DBES}$-calculus a commutation rule of the form: $a[\Uparrow^i (@l_j)][\Uparrow^i ([l_j = \varsigma(b[\uparrow$

$]), {m_i}^{i\in 1..n, i\neq j}]/)] \longrightarrow_{COM} a[\Uparrow^i (b/)]$ (taking $i = 0$ suffices for our example). But adding a rule like *COM* clearly introduces confluence problems.

A variant could be the rule $a[\Uparrow^i (@l_j)][\Uparrow^i ([l_j = \varsigma(b), {m_i}^{i\in 1..n, i\neq j}]/)] \longrightarrow_{COM'} a[\Uparrow^i (c/)]$ where $b =_{BES} c[\uparrow]$. The major drawbacks are then the fact that the rule is conditional and (computationally) expensive checking on the equational substitution theory is required (this resembles problems studied when dealing with η-contraction in explicit substitution calculi ([7],[25], [16])).

These problems stem from the fact that the formulation of rules which are subject to restrictions on the free variables in a de Bruijn index setting and in the presence of explicit substitutions is non trivial. Here, we have solved these issues by a minor change in the syntax so as to represent fields as primitive operators. In fact, the rewrite rule *CO* of the *named* ς_{ES}-calculus presented in [17] is conditional, whereas the *CO* rule presented in this work, in a *de Bruijn index* setting, is actually simpler since no condition is present.

6 Encoding λ_υ-Terms in the ς_{DBES}-Calculus

In this section we will show how to simulate the explicit substitution calculus for the lambda calculus λ_υ[20] in the ς_{DBES}-calculus. We start by augmenting the grammar productions for the terms of the ς_{DBES}-calculus in order to allow abstractions and applications as legal terms. We then define a translation from terms in the λ_υ-calculus into this augmented set of terms which preserves reduction. We recall the main definitions of the λ_υ-calculus. Terms are defined by the following grammars $t ::= \underline{p} \mid tt \mid \lambda t \mid t[s]$ and $s ::= \uparrow \mid t/ \mid \Uparrow (s)$. We recall the rules below.

$$
\begin{array}{llll}
(\lambda a)b & \longrightarrow_{Beta} & a[b/] & \underline{p+1}[a/] \longrightarrow_{RVar} \underline{p} \\
(a\ b)[s] & \longrightarrow_{app} & a[s]b[s] & \underline{1}[\Uparrow (s)] \longrightarrow_{FVarLift} \underline{1} \\
\lambda a[s] & \longrightarrow_{abs} & \lambda(a[\Uparrow (s)]) & \underline{p+1}[\Uparrow (s)] \longrightarrow_{RVarLift} \underline{p}[s][\uparrow] \\
\underline{1}[a/] & \longrightarrow_{FVar} & a & \underline{p}[\uparrow] \longrightarrow_{VarShift} \underline{p+1}
\end{array}
$$

The mixed set of terms, which we shall call $\mathcal{T}_{\lambda\varsigma_{DBES}}$ consists of the terms of sort `Term` and terms of sort `Subst` (which remain unaltered). The terms of sort `Term` are defined by the following grammar:

$$
\begin{array}{l}
a ::= \underline{p} \mid a.l \mid a \lhd \langle m \rangle \mid [{m_i}^{i\in\ 1..n}] \mid a[s] \mid \lambda a \mid (a\ a) \\
m ::= \overline{l} = g \mid l := a \\
g ::= \varsigma(a) \mid g[s]
\end{array}
$$

where p is any natural number greater than zero.

The rewrite rules of the $\lambda\varsigma_{DBES}$-calculus consists of the rewrite rules of the ς_{DBES}-calculus together with the rules *Beta*, *abs* and *app* of the λ_υ-calculus (note that the remaining rules of λ_υ already belong to the ς_{DBES}-calculus). The resulting system may be proved confluent using the interpretation technique [12] and the fact that the corresponding system with meta-level substitutions is an orthogonal rewrite system.

The encoding of λ_υ-terms into $\lambda\varsigma_{DBES}$-terms makes use of the invoke explicit substitution operator "@" and fields.

Definition 5 (Translation of $\lambda\varsigma_{DBES}$-terms into terms in $\mathcal{T}_{\varsigma_{DBES}}$). *The translation $\prec . \succ$ from $\lambda\varsigma_{DBES}$-terms into terms in $\mathcal{T}_{\varsigma_{DBES}}$ is defined as*

$$
\begin{array}{llll}
\prec \underline{p} \succ & =_{def} \underline{p} & \prec \varsigma(a) \succ & =_{def} \varsigma(\prec a \succ) \\
\prec a.l \succ & =_{def} \prec a \succ .l & \prec a/ \succ & =_{def} \prec a \succ / \\
\prec a \lhd \langle m \rangle \succ & =_{def} \prec a \succ \lhd \langle \prec m \succ \rangle & \prec \Uparrow (s) \succ & =_{def} \Uparrow (\prec s \succ) \\
\prec [m_i{}^{\,i \in 1..n}] \succ & =_{def} [\prec m_i \succ {}^{\,i \in 1..n}] & \prec \uparrow \succ & =_{def} \uparrow \\
\prec l = g \succ & =_{def} l =\prec g \succ & \prec @l \succ & =_{def} @l \\
\prec l := a \succ & =_{def} l :=\prec a \succ & \prec \lambda a \succ & =_{def} c \\
\prec a[s] \succ & =_{def} \prec a \succ [\prec s \succ] & \prec ab \succ & =_{def} \prec a \succ \bullet \prec b \succ
\end{array}
$$

where c is $[arg = \varsigma(1.arg), val = \varsigma(\prec a \succ [@arg])]$ and $p \bullet q \ =_{def} (p \lhd \langle arg := q \rangle).val$

The translation interprets the lambda expressions abstraction and application into objects leaving the rest of the constructions without modifications. The translation of an abstraction introduces the invoke substitution. Note that the index level 1 (to which the invoke substitution applies) is bound. This reveals a difference as regards the behaviour of ordinary and invoke substitutions, as discussed above. Ordinary substitution is of no use since its index adjusting mechanism does not exhibit the desired behaviour.

The principal motivation behind the introduction of the rules describing the interaction of ordinary substitution and invoke substitution lies in the following proposition.

Proposition 1 (ς_{DBES} simulates λ_v). *If $a \longrightarrow_{\lambda v} b$ then $\prec a \succ \xrightarrow{*}_{\varsigma_{DBES}} \prec b \succ$.*

Proof. The proof is done by structural induction on the λ_v-term. The key cases are $\prec (\lambda a)b \succ \xrightarrow{*}_{\varsigma_{DBES}} \prec a[b/] \succ$ (Case 1) and $\prec \lambda a[s] \succ \xrightarrow{*}_{\varsigma_{DBES}} \prec \lambda(a[\Uparrow (s)]) \succ$ (Case 2).

Case 1.

$$
\begin{array}{ll}
\prec (\lambda a)b \succ & =_{def} \\
([arg = \varsigma(1.arg), val = \varsigma(\prec a \succ [@arg])] \lhd \langle arg :=\prec b \succ \rangle).val & \longrightarrow_{MO} \\
[arg :=\prec b \succ, val = \varsigma(\prec a \succ [@arg])].val & \longrightarrow_{MI} \\
\prec a \succ [@arg][[arg :=\prec b \succ, val = \varsigma(\prec a \succ [@arg])]/] & \longrightarrow_{CO} \\
\prec a \succ [\prec b \succ /] & =_{def} \\
\prec a[b/] \succ &
\end{array}
$$

Case 2.

$$
\begin{array}{ll}
\prec (\lambda a)[s] \succ & =_{def} \\
[arg = \varsigma(1.arg), val = \varsigma(\prec a \succ [@arg])][\prec s \succ] & \longrightarrow_{SO} \\
[arg = (\varsigma(1.arg))[\prec s \succ], val = (\varsigma(\prec a \succ [@arg]))[\prec s \succ]] & \xrightarrow{*}_{BES} \\
[arg = \varsigma(1.arg[\Uparrow (\prec s \succ)]), val = \varsigma(\prec a \succ [@arg][\Uparrow (\prec s \succ)])] & \xrightarrow{*}_{BES} \\
[arg = \varsigma(1.arg), val = \varsigma(\prec a \succ [@arg][\Uparrow (\prec s \succ)])] & \longrightarrow_{SW} \\
[arg = \varsigma(1.arg), val = \varsigma(\prec a \succ [\Uparrow (\prec s \succ)][@arg])] & =_{def} \\
\prec \lambda(a[\Uparrow (s)]) \succ &
\end{array}
$$

We may therefore conclude that λ_v-derivations may be translated into ς_{DBES}-reductions sequences, thereby implementing objects and functions at the same time.

7 Confluence and PSN of the ς_{DBES}-Calculus

When dealing with calculi of explicit substitutions some basic properties have to be considered, namely, strong normalization of the substitution calculus ($ESDB$), confluence of the full calculus and preservation of strong normalization, that is, that every strongly normalizing term in $\varsigma^{\bullet}_{DB}$-calculus must also be strongly normalizing in the ς_{DBES}-calculus. The history of calculi of explicit substitution has revealed that this last property is by no means trivial. One of the first calculi of explicit substitution for the λ-calculus, called λ_σ [2] introduced in 1991 was long believed to satisfy the aforementioned property. Surprisingly in 1995 Melliès provided a counterexample [22], exhibiting a (pure typable) term that is strongly β-normalizing yet admits an infinite λ_σ-reduction sequence. Since we allow some interaction between substitutions this property is essential in our current setting.

Strong normalization of the substitution calculus is obtained by the polynomial interpretation technique. As for the confluence of the ς_{DBES}-calculus we have the following result which is proved using the Interpretation Method [12].

Proposition 2. *The ς_{DBES}-calculus is confluent.*

Proving preservation of strong normalization is more complicated. We shall obtain the desired result by using a technique introduced by Bloo and Geuvers in [4]. As remarked before this property is an essential ingredient in any explicit substitution implementation of a calculus, more so if there is some form of interaction between substitutions as is our case.

Definition 6 (Strongly normalising pure terms of $\mathcal{T}_{\varsigma_{DBES}}$). *Let $SN_{\varsigma^{\bullet}_{DB}}$ denote the set of all the $\varsigma^{\bullet}_{DB}$-strongly normalizing pure terms of $\mathcal{T}_{\varsigma_{DBES}}$. Then we may define $\mathcal{F}$ as $\mathcal{F} = \{a \in \mathcal{T}_{\varsigma_{DBES}} |$ for all $b \subseteq a$ of sort $\mathtt{Term}, BES(b) \in SN_{\varsigma^{\bullet}_{DB}}\}$.*

The notation $b \subseteq a$ is used to denote that b is a subterm of a. Next we show that $\mathcal{F}$ is closed with respect to reduction in the ς_{DBES}-calculus.

Lemma 1. *Let $a, b \in \mathcal{T}_{\varsigma^{\bullet}_{DB}}$. If $a \in \mathcal{F}$ and $a \longrightarrow_{\varsigma_{DBES}} b$ then $b \in \mathcal{F}$.*

Proof. We show that for every $e \subseteq b$ we have $BES(e) \in SN_{\varsigma^{\bullet}_{DB}}$. The proof is by induction on a.

Definition 7 (Labelled terms). *We define the set of terms $\mathcal{T}_l$ over the alphabet $\mathcal{A} = \{\star, \circ, ..{}_n., \lhd(.,.), . < . >_n, .[\![.]\!]_n, \varsigma(.), [.], =, :=\}$ and for n a natural number greater or equal to zero, by the following grammar:*

$$
\begin{array}{l}
t ::= \star \mid t.{}_n\circ \mid t < t >_n \mid t[\![\circ]\!]_n \mid \lhd(t,u) \mid [{u_i}^{i \in 1..n}] \\
u ::= \circ = f \mid \circ := t \\
f ::= \varsigma(t) \mid f < t >_n
\end{array}
$$

Definition 8 (Translation from $\mathcal{F}$ to $\mathcal{T}_l$). *The translation $\mathcal{S}(.) : \mathcal{F} \to \mathcal{T}_l$ is defined as follows:*

$$
\begin{array}{lll}
\mathcal{S}(\underline{p}) & =_{def} \star & \\
\mathcal{S}([\overline{m_i^{\,i\,\in 1..n}}]) & =_{def} [\mathcal{S}(m_i)^{i\,\in 1..n}] & \\
\mathcal{S}(l = g) & =_{def} \mathcal{S}(l) = \mathcal{S}(g) & \\
\mathcal{S}(l := a) & =_{def} \mathcal{S}(l) := \mathcal{S}(a) & \\
\mathcal{S}(\varsigma(a)) & =_{def} \varsigma(\mathcal{S}(a)) & \\
\mathcal{S}(a.l) & =_{def} \mathcal{S}(a).{}_n\mathcal{S}(l) & \text{where } n = maxred_{\varsigma^{\bullet}_{DB}}(BES(a.l)) \\
\mathcal{S}(a \lhd \langle m \rangle) & =_{def} \lhd(\mathcal{S}(a), \mathcal{S}(m)) & \\
\mathcal{S}(a[\Uparrow^i (b/)]) & =_{def} \mathcal{S}(a) < \mathcal{S}(b) >_n & \text{where } n = maxred_{\varsigma^{\bullet}_{DB}}(BES(a[\Uparrow^i (b/)])) \\
\mathcal{S}(a[\Uparrow^i (\uparrow)]) & =_{def} \mathcal{S}(a) & \\
\mathcal{S}(a[\Uparrow^i (@l)]) & =_{def} \mathcal{S}(a)[\![\mathcal{S}(l)]\!]_n & \text{where } n = maxred_{\varsigma^{\bullet}_{DB}}(BES(a[\Uparrow^i (@l)]))
\end{array}
$$

where $\mathcal{S}(l) = \circ$.

We define a precedence (partial ordering) on the set of operators of $\mathcal{A}$ as follows: $..{}_{n+1}. \gg . < . >_n \gg .[\![.]\!]_n \gg ..{}_n. \gg \lhd(.,.) \gg \varsigma(.), =, :=, [], \star, \circ$. Then since $\gg$ is well-founded the induced Recursive Path Ordering (RPO) '$\succ_{\mathcal{T}_l}$' defined below is well-founded on $\mathcal{T}_l$ [10].

Lemma 2. *Let $a \in \mathcal{F}$. Then $a \longrightarrow_R a'$ implies $\mathcal{S}(a) \succ_{\mathcal{T}_l} \mathcal{S}(a')$ where $R = \{MI, FI, MO, FO\}$ and $\mathcal{S}(a) \succeq_{\mathcal{T}_l} \mathcal{S}(a')$ if $R = \varsigma_{DBES} - \{MI, FI, MO, FO\}$.*

Proof. The proof is by structural induction on a using lemmas 1 and additional technical lemmata (see [6]).

We may now prove the main proposition of this section, namely, the proposition of preservation of strong normalization for the ς_{DBES}-calculus.

Proposition 3 (PSN of the ς_{DBES}-calculus). *The ς_{DBES}-calculus preserves strong normalization.*

Proof. Suppose that the ς_{DBES}-calculus does not preserve strong normalization. Thus there is a pure term a which is strongly $\varsigma^{\bullet}_{DB}$-normalizing but which possesses an infinite reduction sequence in the ς_{DBES}-calculus. Since the rewriting system $S \equiv ESDB \cup \{MO, FO, FI\}$ is strongly normalizing [6] this reduction sequence must have the form $a \equiv a_1 \xrightarrow{*}_S a_2 \longrightarrow_{MI} a_3 \xrightarrow{*}_S a_4 \longrightarrow_{MI} a_5 \ldots$ where the reductions $a_{2k} \longrightarrow_{MI} a_{2k+1}$ for $k \geq 1$ occur infinitely many times. Now since a is in $\mathcal{F}$, and since by lemma 1 the set $\mathcal{F}$ is closed under reduction in $\varsigma^{\bullet}_{DB}$ we obtain an infinite sequence

$$\mathcal{S}(a) \equiv \mathcal{S}(a_1) \succeq_{\mathcal{T}_l} \mathcal{S}(a_2) \succ_{\mathcal{T}_l} \mathcal{S}(a_3) \succeq_{\mathcal{T}_l} \mathcal{S}(a_4) \succ_{\mathcal{T}_l} \mathcal{S}(a_5) \ldots$$

This contradicts the well-foundedness of the recusive path ordering $\succ_{\mathcal{T}_l}$.

8 Conclusions and Future Work

We have proposed a first order calculus based on de Bruijn indices and explicit substitutions for implementing objects and functions. The encoding of functions

as objects in the spirit of [1] has led us to consider fields as primitive constructs in the language. The resulting calculus has been shown to correctly simulate the object calculus (ς) and the function calculus (λ_v), and also that it satisfies the properties of confluence and preservation of strong normalization. As in [17] two different forms of substitution are present in the calculus: ordinary substitution and invoke substitution. In the named variable calculus presented in [17], called ς_{ES}, this distinction is based on constraints associated with types (in an invoke substitution the type of the method invocation $x.l$ to be substituted for x differ) and the free variable property. In fact, since ς_{ES} is untyped the type constraint may be minimized. In contrast, and as already hinted in [17], in the ς_{DBES}-calculus this distinction is based on different index adjusting mecanisms, thus fully justifying the need for different substitution operators.

Interaction between substitutions such as composition or permutation of substitutions usually renders the property of preservation of strong normalization non trivial. Indeed since a weak form of interaction between both forms of substitutions is present in the ς_{DBES}-calculus the proof of the property of preservation of strong normalization has resulted a key issue.

Finally, rules possessing conditions on free variables in a named variable setting generally pose problems when expressed in a de Bruijn indice setting as may be seen for example when dealing with η-reduction ([7], [25], [16]). The use of fields as a primitive construct has allowed us to replace the conditional rules present in the ς_{ES} with non-conditional rules, thus simplifying the resulting calculus.

As already discussed the ς_{DBES}-calculus is not an instance of the de Bruijn index based higher order rewriting formalism XRS[1] [24]. XRSs provide a fixed substitution calculus ($\sigma_{\Uparrow}$) for computing ordinary substitutions. Thus an interesting approach is to generalize this framework to a formalism where various forms of substitution may be defined with possible interaction between them.

Also, in view of the importance of the typing discipline the consideration of type systems for the ς_{DBES}-calculus is required.

Acknowledgements. I would like to thank Delia Kesner, Alejandro Ríos and Pablo E. Martínez López for valuable discussions, advice and encouragement, and the anonymous referees for their comments.

References

1. M.Abadi and L.Cardelli. *A Theory of Objects.* Springer Verlag, 1996.
2. M.Abadi, L.Cardelli, P.-L.Curien, and J.-J.Lévy. *Explicit Substitutions.* Journal of Functional Programming, 4(1):375-416, 1991.
3. R.Bloo. *Preservation of Termination for Explicit Substitution.* PhD. Thesis, Eindhoven University, 1997.

[1] Strictly speaking it is a first order rewriting formalism but allows to express binding mecanisms.

4. R.Bloo and H.Geuvers. *Explicit Substitution: on the edge of strong normalization.* Theoretical Computer Science, 204, 1998.
5. R.Bloo, K.Rose. *Combinatory Reduction Systems with explicit substitution that preserve strong normalization.* In RTA'96, LNCS 1103, 1996.
6. E. Bonelli. *Using fields and explicit substitutions to implement objects and functions in a de Bruijn setting.* Full version obtainable by ftp at `ftp://ftp.lri.fr/LRI/articles/bonelli/objectdbfull.ps.gz`.
7. D.Briaud. *An explicit eta rewrite rule.* In M.Dezani Ed., Int. Conference on Typed Lambda Calculus and Applications, LNCS vol. 902, 1995.
8. N.G.de Bruijn. *Lambda calculus notation with nameless dummies, a tool for automatic formual manipulation with application to the Church-Rosser theorem.* Koninklijke Nederlandse Akademie van Wetenschappen, Series A, Mathematical Sciences, 75:381-392, 1972.
9. P.-L.Curien, T.Hardin, and J.-J.Lévy. *Confluence properties of weak and strong calculi of explicit substitutions.* Technical Report, Centre d'Etudes et de Recherche en Informatique, CNAM, 1991.
10. N. Dershowitz. *Orderings for term rewriting systems.* Theoretical Computer Science, 17(3):279-301, 1982.
11. M.Ferreira, D.Kesner. and L.Puel. *Lambda-calculi with explicit substitutions and composition which preserve beta-strong normalization.* Proceedings of the 5th International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139, 1996.
12. T.Hardin. *Résultats de confluence pour les règles fortes de la logique combinatoire catégorique et liens avec les lambda-calculs.* Thèse de doctorat, Université de Paris VII, 1987.
13. T.Hardin and J-J.Lévy. *A confluent calculus of substitutions.* In France-Japan Artificial Intelligence and Computer Science Symposium, 1989.
14. F. Kamareddine and A.Ríos. *A lambda calculus a la de Bruijn with Explicit Substitutions.* Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'95), LNCS 982, 1995.
15. F. Kamareddine and A.Ríos. *Extending a λ-calculus with Explicit Substitutions which Preserves Strong Normalization into a Confluent Calculus on Open Terms.* In Journal of Functional Programming, Vol.7 No.4 , 1997.
16. D.Kesner. *Confluence properties of extensional and non-extensional lambda-calculi with explicit substitutions.* Proceedings of the 7th International Conference on Rewriting Techniques and Applications (RTA'96), LNCS 1103, 1996.
17. D.Kesner, P.E.Martínez López. *Explicit Substitutions for Objects and Functions.* Proceedings of the Joint International Symposiums: Programming Languages, Implementations, Logics and Program (PLILP'98) and Algebraic and Logic Programming (ALP), LNCS 1490,pp. 195-212, Sept. 1998.
18. J.W.Klop. *Combinatory Reduction Systems.* PhD Thesis, University of Utrecht, 1980.
19. J.W.Klop. *Term Rewriting Systems.* In S. Abramsky, D. Gabbay, and T.Maibaum, editors, Handbook of Logic in Computer Science, Volume II, Oxford University Press, 1992.
20. P. Lescanne. *From λ_σ to λ_υ, a journey through calculi of explicit substitutions.* In Ann. ACM Symp. on Principles of Programming Languages (POPL), pp. 60-69. ACM, 1994.
21. P. Lescanne and J.Rouyer Degli. *Explicit substitutions with de Bruijn's levels.* In P.Lescanne editor, RTA'95, LNCS 914, 1995.

22. P.A.Melliès. *Typed λ-calculi with explicit substitutions may not terminate.* In TLCA'95, LNCS 902, 1995.
23. C.A.Muñoz. *Confluence and Preservation of Strong Normalisation in an Explicit Substitutions Calculus.* Proceedings of the Eleven Annual IEEE Symposium on Logic in Computer Science, 1996.
24. B.Pagano. *Des calculs de substitution explicit et leur application à la compilation des langages fonctionnels.* Thèse de doctorat, Université de Paris VII, 1998.
25. A. Ríos. *Contribution à l'étude des λ-calculs avec substitutions explicites.* Ph.D Thesis, Université de Paris VII, 1993.
26. K.Rose. *Explicit Cyclic Substitutions.* In CTRS'92, LNCS 656, 1992.
27. M. Takahashi. *Parallel reduction in the λ-calculus.* Journal of Symbolic Computation, 7:113-123, 1989.