Confluence of Extensional and Non-Extensional λ-calculi with Explicit Substitutions

Delia Kesner
LRI (CNRS URA 410)
Bât 490, Université Paris-Sud
91405 Orsay Cedex
France
e-mail:Delia.Kesner@lri.fr

Abstract

This paper studies confluence of extensional and non-extensional λ-calculi with explicit substitutions, where extensionality is interpreted by η-expansion. For that, we propose a scheme for explicit substitutions which describes those abstract properties that are sufficient to guarantee confluence. Our method makes it possible to treat at the same time many well-known calculi such as λ_σ, $\lambda_{\sigma_\Uparrow}$, λ_ϕ, λ_s, λ_v, λ_f, λ_d and λ_{dn}.

Keywords: functional programming, λ-calculi, explicit substitutions, confluence, extensionality.

1 Introduction

The λ-calculus is a convenient framework to study functional programming, where the evaluation process is modeled by β-reduction. The main mechanism used to perform β-reduction is *substitution*, which consists of the replacement of formal parameters by actual arguments. The correctness of substitution is guaranteed by a systematic renaming of bound variables, inconvenient which can be simply avoided in the λ-calculus à la de Bruijn by using natural numbers to encode the names of variables. However, substitution remains, as far as classical λ-calculus or λ-calculus à la de Bruijn is concerned, as a meta-level operation described by operations that are completely external to the language. This becomes a concrete problem in real implementations where the behavior of substitution must be described by other operations. Therefore, λ-calculi with explicit substitutions à la de Bruijn appear as a natural formalism to implement β-reduction as they are able to specify the substitution operation by symbols and reduction rules belonging to the syntax of the calculus. Nowadays, many theorems provers (HOL, ALF, Nuprl) and functional languages (Haskell, ML) use explicit substitutions in their implementations.

However, when working with functions, not only β-reduction takes place in the process of evaluation, as an *extensional* equality is also necessary to reason about programs. Extensionality means that whenever two programs yield the same result for every possible argument, then they are equal. The classical extensional axiom in the theory of λ-calculus is the η-equality, written in a classical formalism as:

$$\lambda x.Mx =_\eta M \qquad \text{if } x \text{ is not free in } M$$

This equality can be operationally interpreted from left to right, yielding the classical η-contraction rule, or in the other way round as already suggested in [Pra71, Min77, Hue76], yielding the $\widehat{\eta}$-expansion. The traditional contractive way is well behaved in the simply typed λ-calculus, but it breaks confluence in many other other λ-calculi [CDC91, DCK94, Kes97b]. On the other hand, the typed $\widehat{\eta}$-expansion can also be combined with many other reduction rules, such as expansive surjective pairing [DCK93, Aka93, Dou93, JG95], recursion [DCK93, Dou93], sums [DCK93, Dou93], algebraic rewriting systems [DCK94] and second order λ-calculus [DCK95]. All these combinations are well-behaved as they preserve confluence and strong normalization. However, none of the mentioned works covers the combination of $\widehat{\eta}$-expansion with calculi having explicit substitutions as they are all based on formalisms with β-reduction rules modelized by a meta-level substitution.

The main goal of this paper is to study *confluence* of λ-calculi with explicit substitutions in de Bruijn's notation, with or without extensional rules. For that, we keep the classical definition of $Beta$-reduction associated to β-reduction, and define the $\widehat{Eta}$-expansion rule associated to the classical $\widehat{\eta}$-expansion one, this rule being more natural than all the contractive interpretations given before in the literature [Har92, Har94, Río93, Bri95]. We study confluence of the reduction relation associated to the $Beta$-reduction alone as well as that associated to the combination of $Beta$-reduction and $\widehat{Eta}$-expansion.

Since we are interested in many different λ-calculi with explicit substitutions (such as λ_σ [ACCL91], $\lambda_{\sigma_\Uparrow}$ [HL89], λ_ϕ [Muñ97], λ_v [Les94], λ_f [FKP96], λ_s [KR95], λ_d [Kes96] and λ_{dn} [Kes96]), we develop an "abstract" proof that allows to handle in a simple and general way all these calculi, by a simple instantiation of the proof on each case-study. This is done by extracting a set of sufficient conditions to guarantee confluence, then formulating a notion of "scheme" of λ-calculi with explicit substitution, and finally translating every mentioned calculus into a calculus verifying the scheme. If the calculus itself already fits in our scheme then the translation is just the identity, otherwise another function has to be used as is done for example for λ_s.

This is an extended and detailed version of [Kes96] which is organized as follows. In Section 2 we recall the untyped and typed versions of the λ-calculus à la de Bruijn with their standard β reduction rule. We present the definition of the associated $\widehat{\eta}$-expansion rule, which is only *well-defined* in a typed framework.

In Section 3 we describe the scheme of λ-calculi with explicit substitutions, which is defined to have a concrete structure of λ-terms, but it does not specify the *full* behavior of *all* the substitution constructors. The intended meaning of the scheme is to take into the account at the same time various λ-calculi with explicit substitutions existing in the literature which fit this "abstract" specification.

We define in Section 4 the notion of $\widehat{Eta}$-expansion associated to that of $\widehat{\eta}$-expansion in classical λ-calculus, and we show that the definition is correct with respect to the corresponding equational axiomatization. Section 5 is devoted to the abstract proofs of confluence of $Beta$ and $Beta \cup \widehat{Eta}$ by using a generalized interpretation method.

In Section 6 we apply the method to some well-known λ-calculi with explicit substitutions such as λ_v, λ_σ, λ_ϕ, λ_f, λ_s, $\lambda_{\sigma_\Uparrow}$, λ_d and λ_{dn}. We finally conclude in Section 7.

2 The λ-calculus à la de Bruijn

When using the β-reduction rule to model evaluation in classical λ-calculus, bound variables need to be renamed in order to ensure the correctness of the substitution operation. Thus substitution is not really defined on terms but on α-equivalence classes.

A classical way to avoid α-conversion problems is to use the de Bruijn's notation [dB72, dB78] for λ-terms, where names of variables are replaced by natural numbers. Hence, the set of first-order representations of λ-terms, called here *pure terms* or simply *terms* to differentiate them from classical λ-terms, is defined by the following grammar, where $\underline{n}$ denotes some representation of the natural number n:

$$\begin{array}{lllll} Naturals & n & ::= & 1 \mid n+1 \\ PureTerms & a & ::= & \underline{n} \mid (a\ a) \mid \lambda a \end{array}$$

Notice that this definition is slight different from the original one [dB72, dB78], where every variable is replaced exactly by a natural number and not by a given representation of a natural number. We prefer however this definition which allows us to make the set of normal forms of a given calculus and the set of pure terms to coincide: for example, in λ_σ, the representation of 2 is the term $\underline{1}[\uparrow]$, which is considered here as a pure term.

Indeed, a given occurrence u of a variable, say x, is replaced by the (representation of the) number of symbols λ whose occurrences are between the binder of this x and u. For example, $\lambda x.\lambda y.x(\lambda z.zx)y$ is written $\lambda(\lambda(\underline{2}(\lambda(\underline{1}\ \underline{3}))\underline{1}))$. Remark that a same variable can be translated in many different ways (as the variable x is translated into $\underline{2}$ and $\underline{3}$), and that different variables can be translated into the same term (as the variables z and y are both translated into $\underline{1}$). Representation of free variables is retrieved from a given context of variables in such a way that an occurrence of a variable $\underline{n}$ represents the $(n-m)$-th free variables of the context when $n > m$ and there are m symbols λ surrounding this occurrence n. For example, the term $\lambda x.(y\ z)$ is written as $\lambda(\underline{2}\ \underline{3})$ with respect to the context y, z.

To avoid confusion we will note classical λ-terms with capital letters M, N, ... and de Bruijn's terms with lower case letters a, b, ...

Definition 2.1 (Contexts and applied terms) A *context* is a term containing a special symbol $\Box$ denoting a hole. We shall note a context as $C[\ ]$ and the textual replacement of the hole by a term a as $C[a]$. Thus, $C[a]$ denotes a term. The subterm a in a term $C[(a\ b)]$ is said to be *applied*.

2.1 β-reduction in the λ-calculus à la de Bruijn

In λ-calculus, evaluation is performed by means of the rule $(\lambda x.M)N \longrightarrow_\beta M\{x \leftarrow N\}$, where $_\{_ \leftarrow _\}$ denotes a meta-operation, usually defined by induction on terms as follows:

$$\begin{array}{lcl} (M\ L)\{x \leftarrow N\} & = & (M\{x \leftarrow N\}\ L\{x \leftarrow N\}) \\ (\lambda y.M)\{x \leftarrow N\} & = & (\lambda y.M\{x \leftarrow N\}) \\ x\{x \leftarrow N\} & = & N \\ y\{x \leftarrow N\} & = & y \end{array}$$

This operation is defined modulo α-conversion in order to avoid the capture of free variables so that in the second equation x is different from y and y is not free in N. Working

with de Bruijn's notation allows to avoid clashes of variable names and thus explicit α-conversion is no more needed in λ-calculus à la de Bruijn [dB72, dB78]. However, numbers have to be adjusted when a substitution is performed in order to guarantee correctness.

Definition 2.2 (Substitutions) Let a and b be terms and $n \geq 1$. Then, the *substitution of a by b at level n* is defined by means of an *updating function* U_i^n ($i \geq 0$ and $n \geq 1$) as follows:

$$\begin{array}{lcl} (a_1\ a_2)\{n \leftarrow b\} & = & (a_1\{n \leftarrow b\})(a_2\{n \leftarrow b\}) \\ (\lambda a)\{n \leftarrow b\} & = & \lambda(a\{n+1 \leftarrow b\}) \\ \underline{m}\{n \leftarrow b\} & = & \begin{cases} \underline{m-1} & \text{if } m > n \\ U_0^n(b) & \text{if } m = n \\ \underline{m} & \text{if } m < n \end{cases} \end{array} \qquad \begin{array}{lcl} U_i^n(a\ b) & = & U_i^n(a)U_i^n(b) \\ U_i^n(\lambda a) & = & \lambda(U_{i+1}^n(a)) \\ U_i^n(\underline{m}) & = & \begin{cases} \underline{m+n-1} & \text{if } m > i \\ \underline{m} & \text{if } m \leq i \end{cases} \end{array}$$

Now, it is possible to define the β-rule using the notions of substitution and contexts described above:

Definition 2.3 (β-reduction) Let a and b be pure terms. We say that a *β-reduces to* b, written as $a \longrightarrow_\beta b$, if and only if there is a context C and terms d and e such that $a = C[(\lambda d)e]$ and $b = C[d\{1 \leftarrow e\}]$.

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

Type information can be added to this formalism in order to model the *typed* λ-calculus. Types are built over some set of base types by means of the arrow constructor for functional types. The type information will be kept in an *environment*, which is just a list of types. *Typing judgments* have the form $\Gamma \vdash a : A$ where Γ is an environment, a is a pure term and A is a type. The *typing rules* are the following:

$$A, \Gamma \vdash \underline{1} : A \qquad\qquad \frac{\Gamma \vdash \underline{n} : B}{A, \Gamma \vdash \underline{n+1} : B}$$

$$\frac{A, \Gamma \vdash b : B}{\Gamma \vdash \lambda A.b : A \to B} \qquad\qquad \frac{\Gamma \vdash a : A \to B \qquad \Gamma \vdash b : A}{\Gamma \vdash (a\ b) : B}$$

When $\Gamma \vdash a : A$ is derivable from the previous set of typing rules, the term a is said to be *of type* A. We may omit types if they are clear from the context.

2.2 η-reduction in the λ-calculus à la de Bruijn

Due to the deep connections between λ-calculus, proof theory and category theory, studies on extensional equalities have appeared with different motivations in all these fields.

By far, the best known extensional equality in λ-calculus is the η-axiom, written as

$$(\eta) \qquad \lambda x : A.Mx = M \qquad \text{provided } x \text{ is not free in } M$$

This equality captures the fact that both functions, $\lambda x : A.Mx$ and M, return the same result when applied to the same argument in any context $C[\ ]$. This is expressed as:

$$C[(\lambda x : A.Mx)N] =_\beta C[MN]$$

When one wants to turn the η-equality into a rule there are two different choices: either from left to right as a *contraction*, called η-contraction, or in the other way round as an *expansion*, called η-expansion or $\widehat{\eta}$-expansion depending on whether some other restrictions are imposed to its application.

The η-axiom has traditionally been turned into a contraction. Such an interpretation is well behaved in the simply typed λ-calculus as it preserves confluence [Pot81]. However, η-contraction does not preserve confluence in many other λ-calculi [CDC91, DCK94, Kes97b].

Fortunately, expansions can be combined with many other higher-order reduction rules, such as expansive surjective pairing [DCK93, Aka93, Dou93, JG95], recursion [DCK93, Dou93], sums [DCK93, Dou93], algebraic rewriting systems [DCK94] and second order λ-calculus [DCK95]. All these combinations preserve confluence and strong normalization. For that, application of the η-expansion has to be restricted by some conditions in order to guarantee the strong normalization property: indeed, remark that the reduction relation is no more a congruence on terms.

Definition 2.4 (The conditional $\widehat{\eta}$-expansion)

$$C[M] \longrightarrow_{\widehat{\eta}} C[\lambda x : A.(Mx)] \quad \text{if} \begin{cases} x \notin FV(M) \\ M \text{ is of type type } A \to B \\ M \text{ is not a } \lambda\text{-abstraction} \\ M \text{ is not applied in } C[M] \end{cases}$$

This definition is adequate, *i.e.* $\widehat{\eta}$-expansion generates the same equational theory as the η-axiom, even if it seems more restrictive because of the syntactic conditions it imposes. We refer the readers to [DCK93] for a detailed discussion on these conditions.

The η-equality in the calculus *à la de Bruijn* must be able to express that for any context $C[\ ]$, the de Bruijn's representations of two terms $C[\lambda x.Mx]$ and $C[M]$ are equal. In terms of representation of variables, this means that all the free variables in M (that may be bound with respect to $C[\ ]$) traverse one more λ when M is replaced by $\lambda x.Mx$, suggesting in this way that a term a is not η-equivalent to $\lambda(a\ \underline{1})$ because of the adjustments to be made on the free variables of a. As an example, the term $\lambda(\underline{2}\ \underline{1})$, placed for example in a context $\lambda[\ ]$ is η-equivalent to the term $\lambda\underline{1}$ and not to $\lambda\underline{2}$. This is not surprising since $\lambda(\lambda(\underline{2}\ \underline{1}))$ is the de Bruijn's representation of $\lambda y.\lambda x.yx$, which is η-equivalent to $\lambda y.y$, represented by $\lambda\underline{1}$. Both conditions can be expressed using the following formalism:

Definition 2.5 (Binding depth of a position in a term) The *binding depth of a position in a term* is defined by a function h from positions and pure terms to natural numbers defined in the following way:

$$\begin{array}{lcllcl} h(\epsilon, \underline{m}) & = & 0 & h(1.\gamma, ab) & = & h(\gamma, a) \\ h(1.\gamma, \lambda a) & = & 1 + h(\gamma, a) & h(2.\gamma, ab) & = & h(\gamma, b) \end{array}$$

Definition 2.6 ($a^{\wedge_n}$ **and** a^+) Let a be a pure term and $n \geq 0$. The term $a^{\wedge_n}$ is defined as follows:

$$\begin{array}{lcl} (a\ b)^{\wedge_n} & = & (a^{\wedge_n}\ b^{\wedge_n}) \\ (\lambda a)^{\wedge_n} & = & \lambda a^{\wedge_{n+1}} \\ \underline{m}^{\wedge_n} & = & \begin{cases} \underline{m+1} & \text{if } m > n \\ \underline{m} & \text{if } m \leq n \end{cases} \end{array}$$

From now on, we note a^+ the term $a^{\wedge_0}$, *i.e.* a^+ is obtained by lifting any free variable of a by 1. This lifting is performed by replacing every index $\underline{m}$ at position γ such that $m > h(\gamma, a)$ by $\underline{m+1}$.

Remark 2.1 If a is a pure term, then $U_0^2(a)$ is a^+. Indeed, $U_m^2(a)$ is exactly $a^{\wedge_m}$ according to definition 2.2 and thus $U_0^2(a) = a^{\wedge_0} = a^+$.

Definition 2.7 (η**-equality)** Let a and b be pure terms. Then $\lambda(a\ \underline{1}) =_\eta b$ if $a = b^+$.

The *η-contraction* rule is obtained by orienting the previous equality from left to right, while the *η-expansion* rule is obtained by orienting the equation into the other way round. Analogously to what is done in classical λ-calculus to guarantee strong normalization of extensional expansions [DCK93], we can now define a conditional $\widehat{\eta}$-expansion as follows:

Definition 2.8 (The conditional $\widehat{\eta}$-expansion) Let $C[\ ]$ be any context and let a be a pure term.

$$C[a] \longrightarrow_{\widehat{\eta}} C[\lambda A.(a^+\ \underline{1})] \qquad \text{if } \begin{cases} a \text{ is of type } A \to B \\ a \text{ is not a } \lambda\text{-abstraction} \\ a \text{ is not applied in } C[a] \end{cases}$$

Exactly as in the case of the simply typed λ-calculus, this definition is adequate, *i.e.* the limitations imposed on the reduction system do not make us loose any valid equality as they are just introduced to avoid reduction loops. The result can be obtained by proving that any $\beta\eta$-equality can be generated by the reflexive, symmetric and transitive closure of $\beta\widehat{\eta}$-reduction. We omit here the details of this (standard) proof, which can be found in [Kes97a] (theorem 2.7).

As expected, the $\widehat{\eta}$-expansion rule enjoys the subject reduction property:

Lemma 2.2 (Subject reduction for $\widehat{\eta}$) If $\Gamma \vdash a : A$ and $a \longrightarrow_{\widehat{\eta}} b$, then $\Gamma \vdash b : A$.

Proof. If $a \longrightarrow_{\widehat{\eta}} b$ is a root expansion step, then A is a functional type $B \to C$ and $b = \lambda B.(a^+\ \underline{1})$, so we have to show that $\Gamma \vdash \lambda B.(a^+\ \underline{1}) : B \to C$. For that, we first show by induction on the structure of a, that for every $n \geq 0$ and for every type B, if $\Gamma, \Delta \vdash a : A$ and n is the length of the environment Γ, then $\Gamma, B, \Delta \vdash a^{\wedge_n} : A$. As a corollary, $\Gamma \vdash a : A$ implies $B, \Gamma \vdash a^+ : A$ so that we obtain the following typing sequence:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash a : B \to C}{B, \Gamma \vdash a^+ : B \to C} \qquad B, \Gamma \vdash \underline{1} : B}{B, \Gamma \vdash (a^+\ \underline{1}) : C}}{\Gamma \vdash \lambda B.(a^+\ \underline{1}) : B \to C}$$

If $a \longrightarrow_{\widehat{\eta}} b$ is not a root expansion, then one proceeds by induction on terms.

A direct proof of confluence of $\beta \cup \widehat{\eta}$ in the λ-calculus à la de Bruijn can be found in [Kes97a], but since we know that the union of β and $\widehat{\eta}$ yields a confluent reduction relation in the typed λ-calculus, and also that the λ-calculus is isomorphic [1] to the λ-calculus à la de Bruijn [Mau85, Cur83, dB72], then we are able to conclude that

Theorem 2.3 $\longrightarrow^*_{\beta\widehat{\eta}}$ is confluent in the typed λ-calculus à la de Bruijn.

As a consequence, using the fact that $=_{\beta\eta}$ can be generated by the reflexive, symmetric and transitive closure of $\longrightarrow_{\beta\widehat{\eta}}$ we obtain

Corollary 2.4 If $a =_{\beta\eta} b$, then there is a term c such that $a \longrightarrow^*_{\beta\widehat{\eta}} c$ and $b \longrightarrow^*_{\beta\widehat{\eta}} c$.

3 A scheme for λ-calculi with explicit substitutions

When substitutions are specified at a meta-level, as in classical λ-calculus or λ-calculus à la de Bruijn, an external mechanism takes place each time a β-redex needs to be reduced. The approach is very different in λ-calculi with explicit substitutions, where the manipulation of substitutions is incorporated in an explicit way to the calculus via a set of rewrite rules. Such λ-calculi offer the possibility to have a more refined control over substitutions, by allowing for example to delay or record them, and they can be used to describe different implementations in a uniform way.

In this section we introduce a *scheme for λ-calculi with explicit substitutions* which is defined to have a concrete structure of terms, but does not specify the *full* behavior of *all* the substitution constructors. The intended meaning of this scheme is to extract from the various λ-calculi with explicit substitutions existing in the literature a uniform description of their behavior and a set of common features being sufficient to reason about confluence.

3.1 Syntax and rewriting rules

Let us consider two distinguished symbols $\mathcal{T}$ (for terms) and $\mathcal{S}$ (for substitutions).

Definition 3.1 (Substitution signatures) A *substitution declaration* is a (possibly empty) word over the alphabet $\{\mathcal{T}, \mathcal{S}\}$. A *substitution signature* is defined to be a set of symbols Σ all equipped with an arity n and a substitution declaration of length n. We usually write $\xi : \langle n, \sigma_1 \dots \sigma_n \rangle$ if the symbol ξ has arity n and substitution declaration $\sigma_1 \dots \sigma_n$.

Definition 3.2 (Substitution language) The set of objects over a substitution signature Σ constitutes a *substitution language* which is the union of objects of sort $\mathcal{T}$ and $\mathcal{S}$ constructed in the following way :

- If n is a natural number, then $\underline{n}$ is of sort $\mathcal{T}$.
- If a and b are of sort $\mathcal{T}$, then $(a\ b)$ is of sort $\mathcal{T}$.
- If a is of sort $\mathcal{T}$, then $\lambda(a)$ is of sort $\mathcal{T}$.
- If a is of sort $\mathcal{T}$ and s is of sort $\mathcal{S}$, then $a[s]$ is of sort $\mathcal{T}$.

[1] In the sense that there exist a translation $\mathcal{T}$ from λ-terms into λ-terms à la de Bruijn such that $a \longrightarrow_{\beta\eta} b$ if and only if $\mathcal{T}(a) \longrightarrow_{\beta\eta} \mathcal{T}(b)$.

- If every f_i is of sort σ_i, and $\xi : \langle n, \sigma_1 \ldots \sigma_n \rangle \in \Sigma$, then $\xi(f_1, \ldots, f_n)$ is of sort $\mathcal{S}$.

We will say that $\underline{n}$ is a *variable*, $(a\ b)$ an *application*, $\lambda(a)$ a *λ-abstraction* and $a[s]$ a *closure*. We also omit parenthesis when they are clear from the context. For each substitution language W, the set of objects of sort $\mathcal{T}$ (resp. $\mathcal{S}$) is called *the set of terms (resp. substitutions) of W*. We write Λ_W for the set of terms and substitutions of W.

As an example, the substitution signature of the v-calculus [Les94] is the set $\{\#, \Uparrow, \uparrow\}$, where $\# : \langle 1, \mathcal{T} \rangle$, $\Uparrow$: $\langle 1, \mathcal{S} \rangle$ and $\uparrow$: $\langle 0, \epsilon \rangle$, while the the substitution signature of the σ-calculus [ACCL91] is the set $\{\cdot, \circ, id, \uparrow\}$, where $\cdot : \langle 2, \mathcal{TS} \rangle$, $\circ : \langle 2, \mathcal{SS} \rangle$, $id : \langle 0, \epsilon \rangle$ and $\uparrow$: $\langle 0, \epsilon \rangle$. We refer the reader to section 6 for a more detailed presentation of these calculi.

We now introduce some of the rewriting rules required for our scheme. We start with the *Beta* rule that is the main mechanism in all the λ-calculi with explicit substitutions: it is used to initiate the computation and to explicitly introduce substitutions in the terms to be evaluated. The left-hand side of the *Beta* rule corresponds to a classical β-redex $(\lambda a)b$ while the right-hand side must be an expression used to denote the term a where the index $\underline{1}$ is replaced by the term b, the index $\underline{2}$ by $\underline{1}$, the index $\underline{3}$ by $\underline{2}$, etc. This is done in different ways according to the substitution language of the corresponding calculus, but we can, in general, write this expression as $a[cons^W(b)]$, where $cons^W$ is a cons-function from terms to substitutions that must be defined for each λ-calculus W. The *Beta* rule of the scheme can then be expressed as [2]:

$$(Beta) \qquad (\lambda a)b \longrightarrow a[cons^W(b)]$$

As an example, the cons-function of the λ_v-calculus is $cons^v(b) = \#(b)$, while the cons-function of the λ_σ-calculus is $cons^\sigma(b) = b \cdot id$.

The second rule of the scheme for λ-calculi with explicit substitutions, called the *App* rule, is used to distribute a substitution with respect to the subterms of an application term. It is expressed as:

$$(App) \qquad (a\ b)[s] \longrightarrow (a[s]\ b[s])$$

The third rule, called *Lambda*, is used when a substitution goes under a λ constructor. The left-hand side of this rule corresponds then to a redex $(\lambda a)[s]$, while the right-hand side will be written in general as $\lambda(a[lift^W(s)])$, where $lift^W$ is a lift-function from substitutions to substitutions which must be defined for each λ-calculus W. The substitution $lift^W(s)$ denotes a function which does not affect the indices $\underline{1}$ (which represent the variables which are bound by the λ-abstraction that $[s]$ has crossed within the *Lambda* rule) but only changes the other free variables making the correspondent adjustments of indices. The *Lambda* rule of the scheme can then be expressed as [3]:

$$(Lambda) \qquad (\lambda a)[s] \longrightarrow \lambda(a[lift^W(s)])$$

As an example, the lift-function of the λ_v-calculus is $lift^v(s) = \Uparrow (s)$, while the lift-function of the λ_σ-calculus is $lift^\sigma(s) = \underline{1} \cdot (s \circ \uparrow)$.

[2] Since the right hand side of the *Beta* rule depends on the substitution calculus W, we should denote this rule as $Beta_W$, but we prefer to omit the index W to make easier the notation.

[3] Since the right hand side of the *Lambda* rule depends on the substitution calculus W, we should denote this rule as $Lambda_W$, but we prefer to omit the index W to make easier the notation.

Definition 3.3 (Substitution calculus) A rewrite rule $l \to r$ is said to be *sort-preserving* if r is of sort $\mathcal{K}$ when l is of sort $\mathcal{K}$. A *Substitution Calculus*[4] consists of a signature containing the substitution constant $shift^W : \langle 0, \epsilon \rangle$, which is called a *shift-constant* and will be used to denote extensional equalities, a lift-function, a cons-function, and a set of sort-preserving rewriting rules containing *App* and *Lambda* (but not *Beta*). For simplicity we will often omit the superscript W from $shift^W$, $lift^W(s)$ and $cons^W(b)$ when they are clear from the context.

Notation 3.1 We write simply W for the set of rewriting rules of the substitution calculus W, and λ_W for the set $W \cup Beta$. The notations $\longrightarrow_{\mathcal{R}}$ and $=_{\mathcal{R}}$ are respectively used to denote $\mathcal{R}$-reduction and $\mathcal{R}$-equality. If s is a substitution, we note $[s]^n$ the concatenation of substitutions $\underbrace{[s]\ldots[s]}_{n \quad times}$, assuming that $[s]^0$ denotes the empty word. For any function h, h_i $(i \geq 0)$ is defined by $h_0(s) = s$ and $h_{n+1}(s) = h(h_n(s))$.

3.2 The scheme

The rewriting rules *Beta*, *App* and *Lambda* introduced in section 3.1 are necessary to describe some relations between terms and substitutions, but they are not sufficient to express all of them. In particular, we want to prove confluence for some reduction relations associated to all the λ-calculi with explicit substitutions satisfying our scheme, so we have to impose some additional conditions on the set of rewriting rules of a substitution calculus W. These conditions are mainly inspired from the various proofs of confluence existing in the literature for the relations $\longrightarrow_{\lambda_W}$, where W is for example the σ-calculus [ACCL91], the $\sigma_{\Uparrow}$-calculus [HL89], or the υ-calculus [Les94].

Definition 3.4 (Basic conditions) A substitution calculus W is said to be *basic* (written BSC) if it satisfies the following conditions:

1. W is strongly normalizing[5].
2. W is confluent.
3. W-normal forms of terms are pure terms.
4. $W(a\ b) = (W(a)\ W(b))$ and $W(\lambda a) = \lambda W(a)$, where $W(a)$ denotes the W-normal form of a.
5. For every substitution s in W, $\underline{1}[lift^W(s)] =_W \underline{1}$
6. For every substitution s in W and every $m \geq 1$, $\underline{m+1}[lift^W(s)] =_W \underline{m}[s][shift^W]$
7. For every term k in W and every $m \geq 1$, $\underline{m+1}[cons^W(k)] =_W \underline{m}$

[4] Another notion of substitution calculus can be found in [vOvR94].

[5] The well-known substitution calculi appearing in the literature are all strongly normalizing, and moreover, standard proof techniques like polynomial orders or semantic labellings [Zan94] are usually sufficient to show this property. However, when adding the *Beta* rule, strong normalization does not always hold, even in a typed framework [Mel95].

8. For every term k in W, $\underline{1}[cons^W(k)] =_W k$

9. For every variable $\underline{m}$ in W we have $\underline{m}[shift^W] =_W \underline{m+1}$.

It is evident that there exist even more abstract properties to ensure confluence and strong normalization of any rewriting system W, but the intention of definition 3.4 is to consider a set of sufficient tools to reason about confluence of extensional and non-extensional λ-calculi; we think that this presentation is quite reasonable to attack these confluence problems from an abstract point of view.

From conditions 1 and 2 in definition 3.4 we deduce that W-normal forms are unique[6], and this, together with condition 3 guarantees that for every object a in Λ_W and every pure term b such that $W(a) = b$, then $a \longrightarrow^*_W b$. This property will be used in several lemmas of the paper. Conditions 5,6,7,8,9 state the behavior of the main substitutions $lift$, $shift$ and $cons$ that we need in any calculus with explicit substitutions in order to implement β and η reduction. They interact between them as follows.

Corollary 3.2 Let W be a basic substitution calculus. Then for every $m \geq 1$, for every $n \geq 0$, every term k in Λ_W and every substitution s in Λ_W:

$$W(\underline{m}[lift_n(s)]) = \begin{cases} W(\underline{m-n}[s][shift]^n) & \text{if } m > n \\ \underline{m} & \text{if } m \leq n \end{cases}$$

$$W(\underline{m}[lift_n(shift)]) = \begin{cases} \underline{m+1} & \text{if } m > n \\ \underline{m} & \text{if } m \leq n \end{cases}$$

$$W(\underline{m}[lift_n(cons(k))]) = \begin{cases} \underline{m-1} & \text{if } m > n+1 \\ \underline{m} & \text{if } m < n+1 \\ W(k[shift]^n) & \text{if } m = n+1 \end{cases}$$

Now, we add some simple restrictions that are very easy to prove when working with concrete calculi, as they just concern the interaction between variables and substitutions of the substitution language.

Definition 3.5 (Scheme) We say that a basic substitution calculus W obeys the *scheme* iff for every variable $\underline{m} \in \Lambda_W$ and every function symbol $\xi \in \Sigma$ of arity q one of the two following conditions holds:

- There exist a variable $\underline{n}$, indices $i_1, \ldots, i_p$ $(p \geq 0)$ and substitutions $u_1, \ldots, u_k$ $(k \geq 0)$ such that
  - $1 \leq i_1, i_2, \ldots, i_p \leq q$ and the i_j's are all distinct, and
  - For all $s_1 \ldots s_q$ we have $\underline{m}[\xi(s_1, \ldots, s_q)] =_W \underline{n}[s_{i_1}] \ldots [s_{i_p}][u_1] \ldots [u_k]$.

[6]Even if uniqueness of normal forms can also be deduced from *weak normalization* and confluence, we require a substitution calculus to have the strong normalization property to guarantee that any evaluation strategy used to eliminate substitutions will terminate. Substitution calculi are supposed to preserve the properties of the languages they implement so it is reasonable to require strong (and not weak) normalization and confluence.

- There exists i $(1 \leq i \leq q)$ such that for all $s_1 \ldots s_q$ we have $\underline{m}[\xi(s_1, \ldots, s_q)] =_W s_i$.

We assume these equations to be well-typed, in the sense that whenever the first case holds, then $s_{i_1}, \ldots, s_{i_p}$ are substitutions, and whenever the second case holds, then s_i is a term.

This condition says that the behavior of any substitution $s = \xi(s_1, \ldots, s_q)$ with respect to a variable $\underline{m}$ is only determined by the constructor ξ and not at all by the other components $s_1 \ldots s_q$, that is, the semantics of any substitution is given by the constructor and not by their arguments. Remark that we use the equality $=_W$ instead of the rewrite relation $\longrightarrow_W$ because sometimes the scheme is verified by a rewrite step $\longrightarrow_W$ and some other times by ${}_W\longleftarrow$: indeed, in the σ-calculus, we have $\underline{m}[s \circ t]$ ${}_{Clos}\longleftarrow \underline{m}[s][t]$, while for example in the λ_v-calculus, we have $\underline{m}[\uparrow] \longrightarrow_{VarShift1} \underline{m+1}$.

To understand the (technical) motivations of definition 3.5 let us suppose that $lift$, $shift$ and $cons$ are just primitive constructors. Then, conditions 5, 6, 7 and 9 of definition 3.4 corresponds to the first case of the scheme while condition 8 corresponds to the second one. Since $lift$, $shift$ and $cons$ are not always implemented via a single primitive constructor as for example $lift^{\sigma}(s) = \underline{1} \cdot (s \circ \uparrow)$ is defined as a "macro" using three different constructors $\cdot, \circ$ and $\uparrow$, we ask all the primitive constructors of substitutions to verify the scheme. Doing so, the three "macro" substitutions $lift$, $shift$ and $cons$ as well as all the primitive constructors of the language for substitutions will follow the scheme, having in this way a certain type of "regular" behavior which makes easier to reason by induction on the structure of objects.

4 Extensional λ-calculi with explicit substitutions

In this section we define the notion of $\widehat{Eta}$-expansion associated to that of $\widehat{\eta}$-expansion in classical λ-calculus, and we show that the definition is correct with respect to the corresponding equational axiomatization via the Eta-axiom. The basic conditions given in definition 3.4 will be sufficient to show that the $\widehat{Eta}$-expansion rule is a correct restriction of the Eta-axiom, while the scheme given in definition 3.5 is also needed to show that both $Beta$ and $Beta \cup \widehat{Eta}$ are confluent reduction relations.

4.1 The Eta-axiom

The standard equation corresponding to η-equality in classical λ-calculus is usually written in the formalism of λ-calculi with explicit substitutions as the axiom:

$$(Eta) \qquad \lambda(a[shift]\ \underline{1}) = a$$

The shift-constant is then necessary to express the extensional Eta-axiom, but it will be also useful to express some relations between explicit substitutions, even if the Eta-axiom is not considered in the substitution calculus W. For example, one can express the behavior of lift-functions in terms of the shift-constant as done in definition 3.4, condition 6.

We show here a characterization of a pure term a^+ and we give the relation between updating functions and explicit substitutions.

Lemma 4.1 Let W be a basic substitution calculus. Let a be a pure term, $i \geq 0$ and $n \geq 1$. Then $U_i^n(a) = W(a[lift_i(shift)]^{n-1})$.

Proof. By induction on the structure of the term a, using corollary 3.2.

Corollary 4.2 Let W be a basic substitution calculus and let a be a pure term. Then $W(a[shift])$ is the term a^+.

Proof. The property follows from lemma 4.1 for the case $n = 2$ and $i = 0$, using the fact that $U_0^2(a)$ is a^+ (remark 2.1).

4.2 Turning the *Eta*-axiom into a reduction rule

When turning the *Eta*-axiom into a reduction rule some problems arise. If we orient this equation from left to right as the rewriting rule $\lambda(a[shift]\ \underline{1}) \longrightarrow\ a$, an infinite set of critical pairs is usually generated. As a consequence, the rule is often expressed by the following *conditional* rule [Har92, Har94, Río93]:

$$(Eta_1) \qquad \lambda(a\ \underline{1}) \longrightarrow\ b \qquad \text{if } a =_W b[shift]$$

However, the condition $a =_W b[shift]$ depends on the particular definition of the substitution calculus W, and may be difficult (or expensive) to be verified each time that an Eta_1-reduction is performed.

There is another *unconditional* rewriting rule associated to the *Eta*-axiom, proposed by Briaud [Bri95], expressed in our syntax as follows:

$$(Eta_2) \qquad \lambda(a\ \underline{1}) \longrightarrow\ a[cons(\bot)]$$

where $\bot$ is a *new* constant added to the set of terms. This rule can be expressed in classical λ-calculus as:

$$(Eta_3) \qquad \lambda x.(M\ x) \longrightarrow\ M\{x \leftarrow \bot\}$$

The η-contraction rule in classical λ-calculus coincides with the Eta_3-rule only if the variable x is not free in M. Similarly, the Eta_2-rule coincides with the conditional Eta_1-rule if the variable represented by $\underline{1}$ is not free in the term a. Otherwise, the Eta_1-rule is not allowed (because the condition is not verified), but Eta_2 is. The intended meaning of the constant $\bot$ is to denote something that is not a term, *i.e.*, a meaningless term. For this reason, every normal form involving terms with $\bot$ has to be discharged, and this requires some kind of backtracking in the computation process, which is, as we know, an expensive operation. However, the Eta_2-rule can also be interpreted as the following *conditional* rule

$$(Eta_2') \qquad \lambda(a\ \underline{1}) \longrightarrow\ a[cons(\bot)], \qquad \text{if } \bot \notin W(a[cons(\bot)])$$

This leads to perform a verification before applying the rule and therefore this solution is still expensive by means of computation steps. Also, the Eta_2' rule depends on the particular definition of the substitution calculus W exactly as in the case of the Eta_1-rule.

Fortunately, there is still another reading of the Eta-axiom that avoids all these problems which corresponds to the *unrestricted* Eta-expansion:

$$(Eta) \qquad a \longrightarrow \lambda(a[shift]\ \underline{1})$$

As explained in section 2.2, the expansive interpretation of the η-axiom has been very successful in these last years, mainly because it has been shown to be compatible with many other higher-order rules. This suggests that it would also be a good solution to define *extensional* versions of λ-calculi with explicit substitutions containing the expansive interpretation of the Eta-axiom.

As the unrestricted Eta-expansion rule is not strongly normalizing, one has to restrict its application by imposing some conditions to guarantee termination, exactly as in the case of the $\widehat{\eta}$-expansion rule. Even if we are not going to study strong normalization properties in this paper, a more restricted version of Eta-expansion will be considered here in order to model exactly the standard conditional $\widehat{\eta}$-expansion rule. For that, we know that $\widehat{\eta}$-expansion is only applied to *typed* terms of functional type which verify some syntactic restrictions: the terms are not λ-abstractions and they are not applied to other subterms. All these conditions can also be expressed when dealing with Eta-expansion in λ-calculi with explicit substitutions.

Definition 4.1 (Applied-terms and Lambda-abstractions) We say that a term a is a *Lambda-abstraction* if $a = \lambda b$, or $a = b[s]$ and b is a Lambda-abstraction. Similarly, let $C[\ ]$ be any context. We say that a subterm a is an *Applied-term* in $C[a]$, if either it is applied in $C[a]$, or $a = b[s]$ and b is an Applied-term in $C[b]$.

As an example, $(\lambda\underline{1})$ and $(\lambda\underline{1})[shift][shift]$ are Lambda-abstractions and $\underline{1}$ is an Applied-term in both $(\underline{1}\ \underline{2})$ and $(\underline{1}[shift]\ \underline{2})$.

The type system associated to each substitution calculus depends on the substitution function symbols of its signature. However, there is a set of *common* typing rules that will be found in all these systems:

$$\frac{}{A, \Gamma \vdash \underline{1} : A} \qquad \frac{}{A, \Gamma \vdash shift \triangleright \Gamma}$$

$$\frac{\Gamma \vdash \underline{n} : B}{A, \Gamma \vdash \underline{n+1} : B} \qquad \frac{\Gamma \vdash a : A \qquad \Delta \vdash s \triangleright \Gamma}{\Delta \vdash a[s] : A}$$

$$\frac{\Gamma \vdash a : A \to B \qquad \Gamma \vdash b : A}{\Gamma \vdash ab : B} \qquad \frac{A, \Gamma \vdash b : B}{\Gamma \vdash \lambda A.b : A \to B}$$

We now present the conditional $\widehat{Eta}$-expansion rule, remark that the relation generated by this rule is no longer a congruence as in the case of classical λ-calculus and λ-calculus à la de Bruijn.

Definition 4.2 The conditional $\widehat{Eta}$-expansion rule is defined as:

$$C[a] \longrightarrow_{\widehat{Eta}} C[\lambda A.(a[shift]\ \underline{1})], \text{ if } \begin{cases} a \text{ is of type } A \to B \\ a \text{ is not a Lambda-abstraction} \\ a \text{ is not an Applied-term in } C[a] \end{cases}$$

Definition 4.3 Given a basic substitution calculus W, we define its *extensional* version as the calculus containing the rules W, $Beta$ and $\widehat{Eta}$. We denote the reduction relation associated to the *extensional* substitution calculus W by $\longrightarrow_{\lambda_W \cup \widehat{Eta}}$.

As expected, the $\widehat{Eta}$-expansion rule enjoys subject reduction.

Lemma 4.3 (Subject reduction for $\widehat{Eta}$) If $\Gamma \vdash a : C$ and $a \longrightarrow_{\widehat{Eta}} b$, then $\Gamma \vdash b : C$.

To justify the definition of Applied-terms, it is sufficient to remark that given a term $C[(a[s]\ b)]$, then the $\widehat{Eta}$-expansion rule will not be terminating as the following infinite reduction sequence shows:

$$a \longrightarrow_{\widehat{Eta}} \lambda(a[shift]\ \underline{1}) \longrightarrow_{\widehat{Eta}} \lambda((\lambda(a[shift]\ \underline{1}))[shift]\ \underline{1}) \ldots$$

However, our restriction is sufficient to keep the strong normalization property:

Lemma 4.4 (Strong normalization of $\widehat{Eta}$) The reduction $\longrightarrow_{\widehat{Eta}}$ is strongly normalizing.

Proof. Using a decreasing measure as defined in [Min79], we refer the reader to [Kes97b] for details.

It is also worth noticing that even if a term a is not a Lambda-abstraction according to definition 4.1, its W-normal form may be one: for example $\underline{1}[cons(\lambda\underline{1})]$ is not a Lambda-abstraction but $W(\underline{1}[cons(\lambda\underline{1})]) = \lambda\underline{1}$ is. The same happens with the notion of Applied-terms. This remark suggests that one could also use a different notion of restriction in definition 4.2, saying that a term a is a Lambda-abstraction if $a =_W \lambda b$. This alternative notion of restricted expansion coincides exactly with that of simply typed λ-calculus, but introduces at the same time a condition that depends on the particular definition of the substitution calculus W, exactly as in the case of the Eta_1 previously mentioned. Even if our approach also introduces a condition in order to use Eta as a reduction rule, the restriction does not depend at all on the particular theory generated by the substitution calculus and there is no backtracking at all.

4.3 Adequacy of $\widehat{Eta}$-expansions

The $\widehat{Eta}$-expansion generates the same equational theory generated by the Eta-equality, even if it seems more restrictive because of the syntactic conditions to be verified. This property is called *adequacy* of $\widehat{Eta}$ with respect to Eta. We show here that definition 4.2 of $\widehat{Eta}$-expansion is adequate with respect to the equational theory generated by the Eta-equality in any λ-calculus with explicit substitutions verifying the basic conditions. For that, we first study some syntactic properties which are helpful not only in proving theorem 4.6 but also for other results of the paper.

Lemma 4.5 describes two different cases where the successive application of two different explicit substitutions on a term does not change its meaning.

Lemma 4.5 Let W be a basic substitution calculus, a be a pure term and k be a term in Λ_W. Then for every $n \geq 0$,

1. $a[lift_n(shift)][lift_n(cons(k))] \longrightarrow^*_W a$
2. $a[lift_{n+1}(shift)][lift_n(cons(\underline{1}))] \longrightarrow^*_W a$

Proof. By induction on pure terms, using corollary 3.2.

We can now show theorem 4.6 that has already been announced:

Theorem 4.6 Let W be a basic substitution calculus. Then any Eta-equality $a =_{Eta} b$ can be generated by the reflexive, symmetric and transitive closure of $\longrightarrow_{\lambda_W \cup \widehat{Eta}}$.

Proof. Suppose $a =_{Eta} b$ holds but a does not $\widehat{Eta}$-reduce to b, neither b to a. Without loss of generality we can suppose that one of the restrictions holds for a, and we reason as follows:

- a is a Lambda-abstraction according to definition 4.1. Then $a = (\lambda b)[s_1] \dots [s_n]$ and we want to show that $(\lambda b)[s_1] \dots [s_n]$ and $\lambda(((\lambda b)[s_1] \dots [s_n])[shift]\ \underline{1})$ are interreducible. Let us consider the pure term $W((\lambda b)[s_1] \dots [s_n])$ that is necessarily a λ-abstraction, say λk, so that $(\lambda b)[s_1] \dots [s_n] \longrightarrow^*_W \lambda k$. Then,

$$\begin{array}{ll} \lambda(((\lambda b)[s_1] \dots [s_n])[shift]\ \underline{1}) & \longrightarrow^*_W \\ \lambda((\lambda k)[shift]\ \underline{1}) & \longrightarrow_{Lambda} \\ \lambda((\lambda k[lift(shift)])\ \underline{1}) & \longrightarrow_{Beta} \\ \lambda(k[lift(shift)][cons(\underline{1})]) & \longrightarrow^*_W \ (by\ l\ 4.5(2)) \\ \lambda k & \end{array}$$

- a is an Applied-term according to definition 4.1. Then a is a subterm of $(a[s_1] \dots [s_n]\ c)$ and we want to show that $(a[s_1] \dots [s_n]\ c)$ and $((\lambda(a[s_1] \dots [s_n][shift]\ \underline{1}))c)$ are interreducible. Let us take the term k such that $a[s_1] \dots [s_n] \longrightarrow^*_W k$. Then,

$$\begin{array}{ll} ((\lambda(a[s_1] \dots [s_n][shift]\ \underline{1}))\ c) & \longrightarrow^*_W \\ ((\lambda(k[shift]\ \underline{1}))\ c) & \longrightarrow_{Beta} \\ (k[shift]\ \underline{1})[cons(c)] & \longrightarrow_{App} \\ (k[shift][cons(c)]\ \underline{1}[cons(c)]) & \longrightarrow_W \ (def\ 3.4(5)) \\ (k[shift][cons(c)]\ c) & \longrightarrow^*_W \ (by\ l\ 4.5(1)) \\ (k\ c) & \end{array}$$

5 Confluence for the scheme

We show in this section that extensional and non-extensional basic substitution calculi having the scheme are confluent. As we deal with extensional rules, the *interpretation method* [Har87], which is the standard technique used to prove confluence of λ-calculi with explicit substitutions, has to be slightly modified, resulting in the *generalized interpretation*

method, that allows also to deal with conditional extensional rules. It is also worth noticing that standard techniques, like the Tait and Martin-Löf's parallel moves lemma, do not work when dealing with restricted expansions, and this is the reason we prefer to use directly the interpretation method even for the non-extensional versions of our λ-calculi.

The interpretation method allows to reason in a quite modular way, by splitting an entire reduction relation R into two different and disjoint relations R_1 and R_2. Indeed, one has to verify that R_1 is confluent and strongly normalizing; and also that R_2 enjoys some kind of "preservation of reduction" by R_1-normal forms. These facts make it possible to put them together guaranteeing confluence of the whole reduction relation R. The method can be summarized by the following lemma:

Lemma 5.1 (Interpretation method) Let $R = R_1 \cup R_2$, where R_1 is a confluent and strongly normalizing reduction relation and R_2 an arbitrary reduction relation. If there exists a reduction relation S on the set of R_1-normal forms such that

i) $\longrightarrow^*_S \subseteq \longrightarrow^*_R$, and

ii) $a \longrightarrow_{R_2} b$ implies $R_1(a) \longrightarrow^*_S R_1(b)$

then if S is confluent, R is also confluent.

When showing confluence for any relation $\longrightarrow_{\lambda_W}$ associated to a λ-calculus with explicit substitutions, one usually takes S as β, R_1 as W and R_2 as $Beta$. However, when the $\widehat{Eta}$-expansion is also considered in the reduction system, this method is no longer useful because condition ii does not always hold, as the following example shows:

Example 5.2 Let us consider the extensional λ_v-calculus and the following $\widehat{Eta}$-expansion step: $a = \underline{1}[\#(\lambda\underline{1})] \longrightarrow_{\widehat{Eta}} (\lambda(\underline{1}[shift]\ \underline{1}))[\#(\lambda\underline{1})] = b$.

Then $W(a) = \lambda\underline{1}$ and $W(b) = \lambda((\lambda\underline{1})\ \underline{1})$, so it is neither true that $W(a) \longrightarrow^*_{\widehat{\eta}} W(b)$, nor that $W(a) \longrightarrow^*_{\beta\widehat{\eta}} W(b)$. In fact, $W(a)$ is a $\beta\widehat{\eta}$-normal form so it cannot be reduced at all.

As the previous example suggests, condition ii required by the interpretation method can be weakened to recover confluence as follows:

$$a \longrightarrow_{R_2} b \text{ implies } R_1(a) =_T R_1(b)$$

where $=_T$ stands for an equality contained in $\longrightarrow^*_S\ {}_S{}^*\longleftarrow$.

This formulation yields our generalized interpretation method, as condition ii required by lemma 5.1 is a particular case of this condition above.

Lemma 5.3 (Generalized Interpretation Method) Let $R = R_1 \cup R_2$, where R_1 is a confluent and strongly normalizing reduction relation and R_2 an arbitrary reduction relation. If there exists a reduction relation S and an equality T on the set of R_1-normal forms such that

i) $\longrightarrow^*_S \subseteq \longrightarrow^*_R$,

ii) $=_T \subseteq \longrightarrow^*_S\ {}_S{}^*\longleftarrow$ and

iii) $a \longrightarrow_{R_2} b$ implies $R_1(a) =_T R_1(b)$

Then R is confluent.

Proof. First of all, since R_1 is confluent and strongly normalizing, we know that R_1-normal forms are unique and so, from condition iii we have that $a \longrightarrow^*_R b$ implies $R_1(a) =_T R_1(b)$. Now, suppose $b \ {}_R{}^*\!\longleftarrow a \longrightarrow^*_R c$. Reduce b and c to their R_1-normal forms $R_1(b)$ and $R_1(c)$. As $R_1 \subseteq R$, then $R_1(b) \ {}_R{}^*\!\longleftarrow a \longrightarrow^*_R R_1(c)$ and by the previous remark we have that $R_1(b) = R_1(R_1(b)) =_T R_1(a) =_T R_1(R_1(c)) = R_1(c)$. By condition ii $R_1(b) \longrightarrow^*_S \ {}_S{}^*\!\longleftarrow R_1(c)$, and by condition i $R_1(b) \longrightarrow^*_R \ {}_R{}^*\!\longleftarrow R_1(c)$ so we can close the diagram as follows:

$$b \longrightarrow^*_R R_1(b) \longrightarrow^*_R \ {}_R{}^*\!\longleftarrow R_1(c) \ {}_R{}^*\!\longleftarrow c$$

When showing confluence for any relation $\longrightarrow_{\lambda_W}$, one usually takes S as β, T as β, R_1 as W and R_2 as $Beta$, while for any relation $\longrightarrow_{\lambda_W \cup \widehat{Eta}}$, one usually takes S as $\beta \cup \widehat{\eta}$, T as $\beta \cup \eta$, R_1 as W and R_2 as $Beta \cup \widehat{Eta}$. We are now able to devote the rest of this section to prove the hypothesis required by lemma 5.3, namely:

$$\longrightarrow^*_\beta \ \subseteq \longrightarrow^*_{\lambda_W} \ \text{and} \longrightarrow^*_{\beta\widehat{\eta}} \ \subseteq \longrightarrow^*_{\lambda_W \cup \widehat{Eta}} \tag{1}$$

$$=_\beta \subseteq \longrightarrow^*_\beta \ {}_\beta{}^*\!\longleftarrow \ \text{and} \ =_{\beta\eta} \subseteq \longrightarrow^*_{\beta\widehat{\eta}} \ {}_{\beta\widehat{\eta}}{}^*\!\longleftarrow \tag{2}$$

$$a \longrightarrow_{Beta} b \ \text{implies} \ W(a) =_\beta W(b) \ \text{and} \ a \longrightarrow_{Beta \cup \widehat{Eta}} b \ \text{implies} \ W(a) =_{\beta\eta} W(b) \tag{3}$$

Property 1 is quite simple to prove (see lemma 5.17) whereas the first (resp. second) part of property 2 holds by the fact that β (resp. $\beta\widehat{\eta}$) is confluent. Property 3 is the essential part of the general abstract proof that distills the different techniques used in all the proofs appearing in the literature. The main idea used to prove them comes from the case where a and b are closures $c[s]$ and $c[s']$ such that $s \longrightarrow_{Beta} s'$ (resp. $s \longrightarrow_{\widehat{Eta}} s'$). As β (resp. $\widehat{\eta}$) equality is only defined on pure terms (and W-normal forms of substitutions are not), the previous property needs to be proven together with a similar statement for substitutions, resulting in something like $s \longrightarrow_{Beta} s'$ (resp. $s \longrightarrow_{\widehat{Eta}} s'$) implies that for every pure term d, $W(d[s]) =_\beta W(d[s'])$ (resp. $W(d[s]) =_{\beta\widehat{\eta}} W(d[s'])$). See lemma 5.14 (resp. lemma 5.15) for the precise formulation of these properties.

One remark concerning property 3 above is that one might in fact prove a stronger property (as done in [Kes96]), namely that $a \longrightarrow_{Beta \cup \widehat{Eta}} b$ implies $W(a) \longrightarrow^*_{\widehat{\eta}} \ {}_\beta{}^*\!\longleftarrow W(b)$. Even if this also yields to the desired result, the proof becomes more complicated as we have to deal with a lot of details concerning the conditional reduction rule $\widehat{\eta}$.

5.1 Some technical lemmas

When showing confluence of β-reduction in classical λ-calculus, one is often faced to show the *substitution* lemma, which is usually stated as follows:

$$M[x := N][y := L] = M[y := L][x := N[y := L]], \ \text{if} \ x \notin FV(L)$$

This section is devoted to show the substitution lemma in the framework of explicit substitutions, which will be used latter in section 5.2 to show the properties required by our generalized interpretation method.

Lemma 5.4 For every basic substitution calculus W, every pure term a and every $n \geq 0$ we have $W(a[shift][lift_{n+1}(shift)]) = W(a[lift_n(shift)][shift])$.

Proof. One shows, by induction on the structure of a using corollary 3.2, that for any $n \geq k \geq 0$ $W(a[lift_k(shift)][lift_{n+1}(shift)]) = W(a[lift_n(shift)][lift_k(shift)])$. Then, the property follows from the case $k = 0$.

By n iterations of lemma 5.4 we obtain:

Corollary 5.5 Let W be a basic substitution calculus. For any pure term a, we have

$$W(a[shift]^n[lift_n(shift)]) = W(a[shift]^{n+1})$$

Lemma 5.6 Let W be a basic substitution calculus, a be a pure term and s be a substitution in Λ_W. Then $W(a[s][shift]) = W(a[shift][lift(s)])$.

Proof. We proof, by induction on the structure of a, that for every $n \geq 0$

$$W(a[lift_n(s)][lift_n(shift)]) = W(a[lift_n(shift)][lift_{n+1}(s)])$$

Then, the property follows from the case $n = 0$.

- For $a = \underline{m}$, we have two cases:
  - If $m > n$, then

$$\begin{array}{ll} W(\underline{m}[lift_n(s)][lift_n(shift)]) & =_{cor\ 3.2} \\ W(\underline{m-n}[s][shift]^n[lift_n(shift)]) & =_{cor\ 5.5} \\ W(\underline{m-n}[s][shift]^{n+1}) & =_{cor\ 3.2} \\ W(\underline{m+1}[lift_{n+1}(s)]) & =_{cor\ 3.2} \\ W(\underline{m}[lift_n(shift)][lift_{n+1}(s)]) & \end{array}$$

  - If otherwise $m \leq n$, then

$$\begin{array}{ll} W(\underline{m}[lift_n(s)][lift_n(shift)]) & =_{cor\ 3.2} \\ W(\underline{m}[lift_n(shift)]) & =_{cor\ 3.2} \\ \underline{m} & =_{cor\ 3.2} \\ W(\underline{m}[lift_{n+1}(s)]) & =_{cor\ 3.2} \\ W(\underline{m}[lift_n(shift)][lift_{n+1}(s)]) & \end{array}$$

- For $a = (a_1\ a_2)$ or $a = \lambda b$ the property immediately follows from the induction hypothesis.

Corollary 5.7 Let W be a basic substitution calculus. Let a be a pure term and s be a substitution in Λ_W. Then for every $k \geq 1$ we have $W(a[shift]^k[lift_k(s)]) = W(a[s][shift]^k)$.

Proof. By induction on k using lemma 5.6.

Lemma 5.8 Let W be a basic substitution calculus. Let a be a pure term, b be a term in Λ_W and $n \geq 0$. Then

$$W(a[lift_n(cons(b))][lift_n(s)]) = W(a[lift_{n+1}(s)][lift_n(cons(b[s]))])$$

Proof. By induction on the structure of a. We show only the interesting case which corresponds to $a = \underline{m}$.

- If $m > n+1$, then we have

$$\begin{array}{ll}
W(\underline{m}[lift_n(cons(b))][lift_n(s)]) & =_{cor\ 3.2} \\
W(\underline{m-n}[cons(b)][shift]^n[lift_n(s)]) & =_{def\ 3.4(7)} \\
W(\underline{m-n-1}[shift]^n[lift_n(s)]) & =_{cor\ 5.7} \\
W(\underline{m-n-1}[s][shift]^n]) & =_{l\ 4.5(1)} \\
W(\underline{m-n-1}[s][shift][cons(b[s])][shift]^n) & =_{cor\ 5.7} \\
W(\underline{m-n-1}[s][shift][shift]^n[lift_n(cons(b[s]))]) & =_{cor\ 3.2} \\
W(\underline{m}[lift_{n+1}(s)][lift_n(cons(b[s]))]) &
\end{array}$$

- If $m = n+1$, then we have

$$\begin{array}{ll}
W(\underline{m}[lift_n(cons(b))][lift_n(s)]) & =_{cor\ 3.2} \\
W(\underline{1}[cons(b)][shift]^n[lift_n(s)]) & =_{def\ 3.4(8)} \\
W(b[shift]^n[lift_n(s)]) & =_{l\ 5.7} \\
W(b[s][shift]^n) & =_{def\ 3.4(8)} \\
W(\underline{1}[cons(b[s])][shift]^n) & =_{cor\ 3.2} \\
W(\underline{n+1}[lift_n(cons(b[s]))]) & =_{def\ 3.4(9)} \\
W(\underline{1}[shift]^n[lift_n(cons(b[s]))]) & =_{def\ 3.4(5)} \\
W(\underline{1}[lift(s)][shift]^n[lift_n(cons(b[s]))]) & =_{cor\ 3.2} \\
W(\underline{m}[lift_{n+1}(s)][lift_n(cons(b[s]))]) &
\end{array}$$

- If otherwise $m < n+1$, then $m \leq n$ and the property holds as a consequence of corollary 3.2.

Corollary 5.9 Let W be a basic substitution calculus and let a and b be two terms in Λ_W. Then $W(a[cons(b)][s]) = W(a[lift(s)][cons(b[s])])$.

Proof. Taking $n = 0$ in lemma 5.8, we can conclude that the property holds for pure terms. Now, $W(a[cons(b)][s]) = W(W(a)[cons(b)][s])$ and $W(a)$ is a pure term by definition 3.4(3), so that $W(W(a)[cons(b)][s])$ is equal to $W(W(a)[lift(s)][cons(b[s])]) = W(a[lift(s)][cons(b[s])])$.

Lemma 5.10 gives the implementation of a classical substitution $\{n \leftarrow b\}$ applied to a term a in terms of the W-normal form of an object in W.

Lemma 5.10 Let W be a basic substitution calculus. Let a and b be pure terms. Then for every $n \geq 1$ $a\{n \leftarrow b\} = W(a[lift_{n-1}(cons(b))])$.

Proof. By induction on the structure of a, using corollary 3.2 and lemma 4.1.

The notion of β-reduction is stable by substitution, but this is no longer true when working with the $\widehat{\eta}$-expansion rule: indeed, given a term x of type $A \to B$, we have $x \longrightarrow_{\widehat{\eta}} \lambda y : A \to B.xy$, but $x\{x \leftarrow \lambda z : A \to B.z\}$ does not reduce via $\widehat{\eta}$ to $(\lambda y : A \to B.xy)\{x \leftarrow \lambda z : A \to B.z\}$. The same situation arises in the λ-calculus à la de Bruijn, and also when dealing with explicit substitutions: it is not true that $a \longrightarrow_{\widehat{\eta}} b$ implies $W(a[s]) \longrightarrow^*_{\widehat{\eta}} W(b[s])$ as the following example shows:

Example 5.11 Let us consider the following $\widehat{\eta}$-expansion step:

$$a = \underline{1} \longrightarrow_{\widehat{\eta}} (\lambda(\underline{2}\ \underline{1})) = b$$

Take $s = \lambda\underline{1}$. Then $W(a[s]) = \lambda\underline{1}$ and $W(b[s]) = \lambda((\lambda\underline{1})\ \underline{1})$ but $W(a[s]) \longrightarrow^*_{\widehat{\eta}} W(b[s])$ does not hold.

As in the case of the λ-calculus à la de Bruijn, we can state a weak property that will be sufficient for our purpose.

Lemma 5.12 Let W be a basic substitution calculus, a and a' be pure terms and s be any substitution in Λ_W. If $a =_\beta a'$, then $W(a[s]) =_\beta W(a'[s])$, and if $a =_\eta a'$, then $W(a[s]) =_\eta W(a'[s])$.

Proof. By induction on the number of derivation steps of $a =_\beta a'$ or $a =_\eta a'$. If $a =_\beta a'$ (resp. $a =_\eta a'$) in 0 steps (by reflexivity), the property is trivial. Otherwise, there is a term b such that $a =_\beta b$ (resp. $a =_\eta b$) in 1 step and $b =_\beta a'$ (resp. $b =_\eta a'$) in n steps ($n \geq 0$) so that $W(b[s]) =_\beta W(a'[s])$ (resp. $W(b[s]) =_\eta W(a'[s])$) holds by induction hypothesis. Now, if $a = b$, then if the equality comes from a context closure, one proceeds by induction on the structure of a, otherwise we consider the two possible cases. If $a =_\beta b$ then $a = (\lambda a_1)a_2$, $b = a_1\{1 \leftarrow a_2\}$ and we have:

$$\begin{array}{l}
W(a[s]) = \\
W(((\lambda W(a_1))W(a_2))[s]) =_{App,Lambda} \\
(\lambda W(a_1[lift(s)]))W(a_2[s]) \qquad \longrightarrow_\beta \quad W(a_1[lift(s)])\{1 \leftarrow W(a_2[s])\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad =_{l\ 5.10} W(a_1[lift(s)][cons(a_2[s])]) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad =_{cor\ 5.9} W(a_1[cons(a_2)][s]) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad =_{l\ 5.10} W(W(a_1)\{1 \leftarrow W(a_2)\}[s]) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad = W(b[s])
\end{array}$$

If on the contrary $a =_\eta b$, then $b = \lambda(a^+\ \underline{1})$ and so we have:

$$\begin{array}{rcll}
W(b[s]) & = & W((\lambda(a^+\ \underline{1}))[s]) & =_{Lambda,App} \\
 & & \lambda W(a^+[lift(s)]\ \underline{1}[lift(s)]) & =_{def\ 3.4(5)} \\
 & & \lambda(W(a^+[lift(s)])\ \underline{1}) & =_{cor\ 4.2} \\
 & & \lambda(W(W(a[shift])[lift(s)])\ \underline{1}) & = \\
 & & \lambda(W(a[shift[lift(s)]])\ \underline{1}) & =_{l\ 5.6} \\
 & & \lambda(W(a[s][shift])\ \underline{1}) & =_{cor\ 4.2} \\
 & & \lambda(W(a[s])^+)\ \underline{1}) & =_\eta \qquad W(a[s])
\end{array}$$

In both cases we obtain $W(a[s]) =_\beta W(a'[s])$ (resp. $W(a[s]) =_\eta W(a'[s])$) by transitivity.

Corollary 5.13 Let a and b be pure terms such that $a =_{\beta\eta} b$. Then, for every substitution s of any basic substitution calculus W we have $W(a[s]) =_{\beta\eta} W(b[s])$.

5.2 The confluence proofs

We proceed now to show the main lemmas of this section, where we can find the justification of the general conditions stated in definition 3.5.

Lemma 5.14 Let W be a basic substitution calculus having the scheme and let o be an object of W. If $o \longrightarrow_{Beta} o'$, then

1. if o is a term, then $W(o) =_\beta W(o')$
2. for every pure term d such that $d[o]$ is a term, and every $n \geq 0$, we have $W(d[lift_n(o)]) =_\beta W(d[lift_n(o')])$

Proof. We show the property by induction on the lexicographic order induced by the pair $\langle o, d\rangle$.

- If o is either a variable or a substitution constant, the properties vacuously hold because o is in $Beta$-normal form. We also remark that if o is an application, an abstraction or a closure, there is nothing to prove for the second property because $d[o]$ is not a term.
- If o is an application $(a\ b)$, we then consider two cases for the first property:
  - •• If either $o' = (a'\ b)$, where $a \longrightarrow_{Beta} a'$, or $o' = (a\ b')$, where $b \longrightarrow_{Beta} b'$, then the first property immediately holds by the induction hypothesis (1) and definition 3.4(4) since $\langle a, d\rangle < \langle (a\ b), d\rangle$ and $\langle b, d\rangle < \langle (a\ b), d\rangle$.
  - •• If $o' = c[cons(b)]$, where $a = (\lambda c)$, then we have

$$\begin{array}{lll} W(o) =_{App,Lambda} (\lambda W(c))W(b) & \longrightarrow_\beta & W(c)\{1 \leftarrow W(b)\} \\ & & =_{l\ 5.10} W(W(c)[cons(W(b))]) \\ & & = W(c[cons(b)]) = W(o') \end{array}$$

- If o is a λ-abstraction λa, where $a \longrightarrow_{Beta} a'$, then the first property immediately holds by the induction hypothesis (1) and definition 3.4(4) since $\langle a, d\rangle < \langle \lambda a, d\rangle$.
- If o is a closure $a[v]$, then we consider two cases for the first property:
  - •• $o' = a'[v]$, where $a \longrightarrow_{Beta} a'$. Since a is a subterm of o, then $\langle a, d\rangle < \langle a[v], d\rangle$, so we have $W(a) =_\beta W(a')$ by the induction hypothesis (1) and $W(o) = W(W(a)[v]) =_\beta W(W(a')[v]) = W(o')$ by lemma 5.12 and the fact that W-normal forms are pure terms by definition 3.4(3).
  - •• $o' = a[v']$, where $v \longrightarrow_{Beta} v'$. Since $W(a)$ is a pure term by definition 3.4(3), then $W(o) = W(W(a)[v]) =_\beta W(W(a)[v']) = W(o')$ holds by the induction hypothesis (2) since $\langle v, W(a)\rangle < \langle a[v], W(a)\rangle$.
- If o is a substitution $\xi(s_1, \ldots, s_j, \ldots, s_q)$ $(q > 0)$, and $o' = \xi(s_1, \ldots, s'_j, \ldots, s_q)$, where $s_j \longrightarrow_{Beta} s'_j$, then there is nothing to prove for the first property as o is not a term. For the second property, we proceed by cases on the pure term d.

•• If $d = (d_1\ d_2)$ or $d = \lambda d_1$, then the property follows using the induction hypothesis (2) since $\langle o, d_i\rangle < \langle o, d\rangle$ for $i = 1, 2$.

•• If $d = \underline{m}$, we have to prove that for all $n \geq 0$ we have $W(\underline{m}[lift_n(o)]) =_\beta W(\underline{m}[lift_n(o')])$. We proceed by induction on n.

i) If $n = 0$, we proceed by cases according to definition 3.5:

i.1) Suppose that there exist a variable $\underline{l}$, indices $i_1, \ldots, i_p$ $(p \geq 0)$ and substitutions $u_1, \ldots, u_k$ $(k \geq 0)$ such that $1 \leq i_1, i_2, \ldots, i_p \leq q$, the i_j's are all distinct and for all $s_1 \ldots s_q$ $\underline{m}[\xi(s_1, \ldots, s_q)] =_W \underline{l}[s_{i_1}] \ldots [s_{i_p}][u_1] \ldots [u_k]$.

i.1.1) If $j \notin \{i_1, \ldots, i_p\}$, then the property is trivial since $W(\underline{m}[o]) = W(\underline{m}[o']) = W(\underline{l}[s_{i_1}] \ldots [s_{i_p}][u_1] \ldots [u_k])$.

i.1.2) If $j \in \{i_1, \ldots, i_p\}$, let us say $j = i_h$, then the term $W(\underline{m}[o'])$ is equal to $W(\underline{l}[s_{i_1}] \ldots [s'_{i_h}] \ldots [s_{i_p}][u_1] \ldots [u_k])$ where $W(\underline{l}[s_{i_1}] \ldots [s_{i_{h-1}}]) = e$ is a pure term by definition 3.4(3). We have $\langle s_{i_h}, e\rangle < \langle \xi(s_1, \ldots, s_q), \underline{m}\rangle$, so by the induction hypothesis (2) $W(e[s_{i_h}]) =_\beta W(e[s'_{i_h}])$. Since $W(e[s_{i_h}])$ is a pure term, we can repeatedly apply lemma 5.12 to obtain:

$$\begin{array}{rl} W(\underline{m}[o]) = & W(W(e[s_{i_h}])[s_{i_{h+1}}] \ldots [s_{i_p}][u_1] \ldots [u_k]) =_\beta \\ & W(W(e[s'_{i_h}])[s_{i_{h+1}}] \ldots [s_{i_p}][u_1] \ldots [u_k]) = \quad W(\underline{m}[o']) \end{array}$$

i.2) Suppose that there exists an index i $(1 \leq i \leq q)$ such that for all $s_1 \ldots s_q$ we have $\underline{m}[\xi(s_1, \ldots, s_q)] =_W s_i$.

i.2.1) If $i \neq j$, then we trivially have $W(\underline{m}[o']) = W(s_i) = W(\underline{m}[o'])$.

i.2.2) If $i = j$, then $W(\underline{m}[o]) = W(s_j)$ and $W(\underline{m}[o']) = W(s'_j)$. Since $\langle s_j, \underline{m}\rangle < \langle \xi(s_1, \ldots, s_j, \ldots, s_q), \underline{m}\rangle$, we get $W(s_j) =_\beta W(s'_j)$ by the induction hypothesis (1).

ii) If $n > 0$, we consider two cases:

ii.1) If $m \leq n$, then $W(\underline{m}[lift_n(o)]) = \underline{m} = W(\underline{m}[lift_n(o')])$ by corollary 3.2.

ii.2) If $m > n$, then by corollary 3.2 we have

$$W(\underline{m}[lift_n(o)]) = W(\underline{m-1}[lift_{n-1}(o)][shift])$$

$$W(\underline{m}[lift_n(o')]) = W(\underline{m-1}[lift_{n-1}(o')][shift])$$

Since variables are equivalent with respect to our ordering $\langle o, d\rangle$, we have $\langle \xi(s_1, \ldots, s_j, \ldots, s_q), \underline{m}\rangle = \langle \xi(s_1, \ldots, s_j, \ldots, s_q), \underline{m-1}\rangle$, and then we can apply the induction hypothesis on n to obtain

$$W(\underline{m-1}[lift_{n-1}(o)]) =_\beta W(\underline{m-1}[lift_{n-1}(o')])$$

Since every W-normal form of a term is a pure term by definition 3.4(3), the result finally follows from lemma 5.12, so that

$$\begin{array}{rl} W(\underline{m}[lift_n(o)]) = & W(W(\underline{m-1}[lift_{n-1}(o)])[shift]) =_\beta \\ & W(W(\underline{m-1}[lift_{n-1}(o')])[shift]) = W(\underline{m}[lift_n(o')]) \end{array}$$

Lemma 5.15 Let W be a basic substitution calculus having the scheme and let o be an object of W. If $o \longrightarrow_{\widehat{Eta}} o'$, then

1. if o is a term, then $W(o) =_{\beta\eta} W(o')$
2. for every pure term d such that $d[o]$ is a term, and every $n \geq 0$, we have $W(d[lift_n(o)]) =_{\beta\eta} W(d[lift_n(o')])$

Proof. If $o \longrightarrow_{\widehat{Eta}} \lambda(o[shift]\ \underline{1}) = o'$ is a root expansion, then o is a term and the second property vacuously holds since $d[lift_n(o)]$ is not a term. The first property holds since

$$W(o) =_{\eta} \lambda(W(o)^+\ \underline{1}) =_{cor\ 4.2} \lambda(W(W(o)[shift])\ \underline{1}) = W(\lambda(o[shift]\ \underline{1})) = W(o')$$

If $o \longrightarrow_{\widehat{Eta}} o'$ is not a root expansion, one proceeds exactly as in lemma 5.14, showing simultaneously the two properties by induction on the lexicographic ordering induced by the pair $\langle o, d\rangle$.

Corollary 5.16 Let W be a basic substitution calculus having the scheme and let a be a term of W such that $a \longrightarrow_{Beta \cup \widehat{Eta}} b$. Then $W(a) =_{\beta\eta} W(b)$.

Finally, we can obtain property (1) as follows:

Lemma 5.17 Let W be a basic substitution calculus and a be a pure term. If $a \longrightarrow^*_{\beta} b$, then $a \longrightarrow^*_{\lambda_W} b$ and if $a \longrightarrow^*_{\beta\widehat{\eta}} b$, then $a \longrightarrow^*_{\lambda_W \cup \widehat{Eta}} b$.

Proof. The proof proceeds by induction on the number of steps from a to b. We only show here the case $a \longrightarrow_{\beta} b$, the rest of the proof being straightforward. If $a \longrightarrow_{\beta} b$, then $a = C[(\lambda c)d]$ and $b = C[c\{1 \leftarrow d\}]$. By lemma 5.10 $c\{1 \leftarrow d\} = W(c[cons(d)])$, so $a = C[(\lambda c)d] \longrightarrow_{Beta} C[c[cons(d)]] \longrightarrow^*_W C[c\{1 \leftarrow d\}] = b$. Suppose now $C[a] \longrightarrow_{\widehat{\eta}} C[\lambda(a^+\ \underline{1})]$, where $a \longrightarrow_{\widehat{\eta}} \lambda(a^+\ \underline{1})$. Then a is not a λ-abstraction and is not applied to another term so by definition $a \longrightarrow_{\widehat{Eta}} \lambda(a[shift]\ \underline{1})$, and by corollary 4.2 $W(a[shift])$ is exactly a^+, so that $\lambda(a[shift]\ \underline{1}) \longrightarrow^*_W \lambda(W(a[shift])\ \underline{1}) = \lambda(a^+\ \underline{1})$ which implies $a \longrightarrow^*_{W \cup \widehat{Eta}} \lambda(a^+\ \underline{1})$ and concludes the proof.

Since every basic substitution calculus W satisfying the scheme is confluent and strongly normalizing, then we can use lemma 5.17 and corollary 5.16 to derive confluence of λ_W and $\lambda_W \cup \widehat{Eta}$ as follows:

Theorem 5.18 (λ_W is confluent) For every basic substitution calculus W satisfying the scheme, the relation λ_W is confluent.

Proof. Using lemma 5.3, where S is β, T is β, R_1 is W and R_2 is $Beta$.

Theorem 5.19 ($\lambda_W \cup \widehat{Eta}$ is confluent) For every basic substitution calculus W having the scheme, the relation $\lambda_W \cup \widehat{Eta}$ is confluent.

Proof. By lemma 5.3, where S is $\beta \cup \widehat{\eta}$, T is $\beta \cup \eta$, R_1 is W and R_2 is $Beta \cup \widehat{Eta}$.

6 Confluence for calculi with explicit substitutions

In this section we apply our method to some well-known calculi such as λ_σ, $\lambda_{\sigma_\Uparrow}$, λ_ϕ, λ_s, λ_v, λ_d, λ_{dn} and λ_f. The method applies *directly* to most of them, and needs a special treatment for λ-calculi which do not fit our definition of substitution calculus, but that can be translated to some calculus fitting the scheme. For that, we require the following:

Theorem 6.1 Let X be a calculus such that

- X is strongly normalizing.
- X-normal forms of terms are pure terms.
- $a \longrightarrow_{\beta\widehat{\eta}} b$ implies $a \longrightarrow^*_{\lambda_X \cup \widehat{Eta}} b$.
- There is a translation $\mathcal{T}$ from Λ_X to terms of some substitution calculus W such that:
  - The translation $\mathcal{T}$ is the identity on pure terms.
  - $a \longrightarrow_{\lambda_X \cup \widehat{Eta}} b$ implies $\mathcal{T}(a) \longrightarrow^*_{\lambda_W \cup \widehat{Eta}} \mathcal{T}(b)$.

Then $\lambda_X \cup \widehat{Eta}$ is confluent.

Proof. Let a be a term such that $b \; {}_{\lambda_X \cup \widehat{Eta}}{}^* \!\!\longleftarrow a \longrightarrow^*_{\lambda_X \cup \widehat{Eta}} c$. Since X is strongly normalizing we can reduce b and c to their X-normal forms $X(b) = b'$ and $X(c) = c'$ so that $b' \; {}_{\lambda_X \cup \widehat{Eta}}{}^* \!\!\longleftarrow a \longrightarrow^*_{\lambda_X \cup \widehat{Eta}} c'$. By the translation we obtain $\mathcal{T}(a) \longrightarrow^*_{\lambda_W \cup \widehat{Eta}} \mathcal{T}(c')$ and $\mathcal{T}(a) \longrightarrow^*_{\lambda_W \cup \widehat{Eta}} \mathcal{T}(b')$ and by corollary 5.16 $W(\mathcal{T}(b')) =_{\beta\widehat{\eta}} W(\mathcal{T}(a)) =_{\beta\widehat{\eta}} W(\mathcal{T}(c'))$ so that $W(\mathcal{T}(b')) \longrightarrow^*_{\beta\widehat{\eta}} k \; {}_{\beta\widehat{\eta}}{}^* \!\!\longleftarrow W(\mathcal{T}(c'))$ by theorem 2.3. Since b' and c' are pure terms by the second hypothesis, then $\mathcal{T}(b') = b'$ and $\mathcal{T}(c') = c'$ hold by the fourth hypothesis and $W(b') = b'$ and $W(c') = c'$ hold by definition of substitution calculi. We have $b' \longrightarrow^*_{\beta\widehat{\eta}} k \; {}_{\beta\widehat{\eta}}{}^* \!\!\longleftarrow c'$, which implies, by the second hypothesis, that $b' \longrightarrow^*_{\lambda_X \cup \widehat{Eta}} k \; {}_{\lambda_X \cup \widehat{Eta}}{}^* \!\!\longleftarrow c'$ and concludes the proof.

In the case of λ_σ, $\lambda_{\sigma_\Uparrow}$, λ_ϕ, λ_v, λ_d, λ_{dn} and λ_f the translation $\mathcal{T}(_)$ is simply the identity, while for the λ_s-calculus one can use for example the translation from λ_s to λ_v given in [KR95]. We omit here all the verifications concerning the treatment of λ_s but we refer the interested reader to [Kes97a] for full details.

As a summary of the calculi we analyze in this section, we give in the following table the set of substitutions as well as the representation of variables we use for each of them. In particular, the variable $\underline{1}$ is always represented by 1 but the representation of a variable $\underline{n+1}$ ($n \geq 1$) changes from one calculus to another.

When using composition, we will often assume the following notations:

Notation 6.2 We use the notation n^* for $\underbrace{succ(\ldots succ}_{n \text{ times}}(0)\ldots)$, $[\uparrow]^n$ for $\underbrace{[\uparrow]\ldots[\uparrow]}_{n \text{ times}}$. The notations $\diamond^n(s)$ and $\diamond_n(s,u)$ are defined by induction as follows:

$$\begin{array}{lcllcl} \diamond^0(s) & = & id & \diamond_0(s,u) & = & u \\ \diamond^1(s) & = & s & \diamond_{n+1}(s,u) & = & \diamond_n(s,(s \circ u)) \\ \diamond^{n+1}(s) & = & s \circ \diamond^n(s) & & & \end{array}$$

Note that $\diamond^n(s) \circ u$ is different from $\diamond_n(s,u)$ (as we have for example $\diamond^3(s) \circ u = (s \circ (s \circ s)) \circ u$ and $\diamond_3(s,u) = s \circ (s \circ (s \circ u)))$, but $\diamond^n(s) \circ u =_{Ass} \diamond_n(s,u)$. Remark also that $\diamond_n(s,s) = \diamond^{n+1}(s)$.

Calculus	Reference	Variable $\underline{n+1}$ $(n \geq 1)$	Substitutions
λ_σ	[ACCL91]	$1[\diamond^n(\uparrow)]$	$s ::= id \mid a \cdot s \mid \uparrow \mid s \circ s$
$\lambda_{\sigma_\Uparrow}$	[HL89]	$n+1$	$s ::= id \mid a \cdot s \mid \Uparrow(s) \mid \uparrow \mid s \circ s$
λ_ϕ	[Muñ97]	$1[\uparrow^{n^*}]$	$s ::= \uparrow^{n^*} \mid a \cdot s \mid s \circ s$
λ_υ	[Les94]	$n+1$	$s ::= \#(a) \mid \Uparrow(s) \mid \uparrow$
λ_f	[FKP96]	$1[\uparrow]^n$	$s ::= \#(a) \mid \Uparrow(s) \mid \uparrow$
λ_d	[Kes96]	$1[\diamond^n(\uparrow)]$	$s ::= id \mid \#(a) \mid \Uparrow(s) \mid \uparrow \mid s \circ s$
λ_{dn}	[Kes96]	$n+1$	$s ::= id \mid \#(a) \mid \Uparrow(s) \mid \uparrow \mid s \circ s$

The rewrite rules of each calculus will be given independently in different subsections, so that we present here the definition of the different functions $shift^W$, $lift^W$ and $cons^W$.

Calculus	$shift$	$lift(s)$	$cons(b)$
λ_σ	$\uparrow$	$1 \cdot (s \circ \uparrow)$	$b \cdot id$
$\lambda_{\sigma_\Uparrow}$	$\uparrow$	$\Uparrow(s)$	$b \cdot id$
λ_ϕ	$\uparrow^{succ(0)}$	$1 \cdot (s \circ \uparrow^{succ(0)})$	$b \cdot \uparrow^0$
λ_υ	$\uparrow$	$\Uparrow(s)$	$\#(b)$
λ_f	$\uparrow$	$\Uparrow(s)$	$\#(b)$
λ_d	$\uparrow$	$\Uparrow(s)$	$\#(b)$
λ_{dn}	$\uparrow$	$\Uparrow(s)$	$\#(b)$

The calculi λ_σ, $\lambda_{\sigma_\Uparrow}$ and λ_ϕ

The λ_σ calculus [ACCL91] was introduced as a bridge between the classical λ-calculus and concrete implementations of functional programming. It is inspired by de Bruijn notation [dB72, dB78] and Categorical Combinatory Logic (CCL) [Cur86], it is very useful to derive machines for the λ-calculus [HMP95] or to implement higher-order unification [DHK95]. We find in the λ_σ-calculus the main features which characterize a calculus of explicit substitutions: substitutions are incorporated to the language and manipulated explicitly, terms are divided in two sorts - terms themselves and substitutions - and β-reduction is simulated in two stages, first by the application of the *Beta* rule which activates the calculus of substitutions, then by propagation of the substitution until variables are reached. Strong normalization of the σ-calculus is proved in [CHR92], but the λ_σ-calculus does not preserve β-strong normalization of terms [Mel95], *i.e.* there exists a term which is β-strong normalizing but it is not λ_σ-strongly normalizing. The λ_σ-calculus is confluent on closed terms, and remains confluent when meta-variables for terms are added to the syntax (*i.e.*, λ_σ is confluent on semi-open terms), but is no longer confluent when variables for substitutions are also considered (confluence fails for open terms). The rewriting rules of λ_σ are:

$$
\begin{array}{lllll}
(Beta) & (\lambda a)b & \longrightarrow & a[b \cdot id] \\
(App) & (a\ b)[s] & \longrightarrow & (a[s]\ b[s]) \\
(Lambda) & (\lambda a)[s] & \longrightarrow & \lambda(a[1 \cdot (s \circ \uparrow)]) \\
(Clos) & (a[s])[t] & \longrightarrow & a[s \circ t] \\
(VarId) & 1[id] & \longrightarrow & 1 \\
(VarCons) & 1[a \cdot s] & \longrightarrow & a \\
(IdL) & id \circ s & \longrightarrow & s \\
(ShiftId) & \uparrow \circ id & \longrightarrow & \uparrow \\
(ShiftCons) & \uparrow \circ (a \cdot s) & \longrightarrow & s \\
(Ass) & (s_1 \circ s_2) \circ s_3 & \longrightarrow & s_1 \circ (s_2 \circ s_3) \\
(Map) & (a \cdot s) \circ t & \longrightarrow & a[t] \cdot (s \circ t)
\end{array}
$$

To overcome the problem of confluence of λ_σ, Hardin and Lévy introduced in [HL89] the $\lambda_{\sigma_\Uparrow}$-calculus which considers a new operator, written $\Uparrow$, that allows to recover confluence on the set of open terms [CHL92]. Indeed, the implementation of the lift-function is much more simpler in the $\lambda_{\sigma_\Uparrow}$-calculus: it only uses a unary constructor. As λ_σ, neither the $\lambda_{\sigma_\Uparrow}$-calculus preserve β-strong normalization [Mel94]. The rewriting rules of $\lambda_{\sigma_\Uparrow}$ are:

$$
\begin{array}{lllll}
(Beta) & (\lambda a)b & \longrightarrow & a[b \cdot id] \\
(App) & (a\ b)[s] & \longrightarrow & (a[s]\ b[s]) \\
(Lambda) & (\lambda a)[s] & \longrightarrow & \lambda(a[\Uparrow (s)]) \\
(Clos) & (a[s])[t] & \longrightarrow & a[s \circ t] \\
(VarShift1) & n[\uparrow] & \longrightarrow & n+1 \\
(VarShift2) & n[\uparrow \circ s] & \longrightarrow & n+1[s] \\
(FVar) & 1[a \cdot s] & \longrightarrow & a \\
(FVarLift1) & 1[\Uparrow (s)] & \longrightarrow & 1 \\
(FVarLift2) & 1[\Uparrow (s) \circ t] & \longrightarrow & 1[t] \\
(RVar) & n+1[a \cdot s] & \longrightarrow & n[s] \\
(RVarLift1) & n+1[\Uparrow (s)] & \longrightarrow & n[s \circ \uparrow] \\
(RVarLift2) & n+1[\Uparrow (s) \circ t] & \longrightarrow & n[s \circ (\uparrow \circ t)] \\
(Ass) & (s_1 \circ s_2) \circ s_3 & \longrightarrow & s_1 \circ (s_2 \circ s_3) \\
(Map) & (a \cdot s) \circ t & \longrightarrow & a[t] \cdot (s \circ t) \\
(Shift) & \uparrow \circ (a \cdot s) & \longrightarrow & s \\
(ShiftLift1) & \uparrow \circ \Uparrow (s) & \longrightarrow & s \circ \uparrow \\
(ShiftLift2) & \uparrow \circ (\Uparrow (s) \circ t) & \longrightarrow & s \circ (\uparrow \circ t) \\
(Lift1) & \Uparrow (s) \circ \Uparrow (t) & \longrightarrow & \Uparrow (s \circ t) \\
(Lift2) & \Uparrow (s) \circ (\Uparrow (t) \circ u) & \longrightarrow & \Uparrow (s \circ t) \circ u \\
(LiftEnv) & \Uparrow (s) \circ (a \cdot t) & \longrightarrow & a \cdot (s \circ t) \\
(IdL) & id \circ s & \longrightarrow & s \\
(IdR) & s \circ id & \longrightarrow & s \\
(LiftId) & \Uparrow (id) & \longrightarrow & id \\
(Id) & a[id] & \longrightarrow & a
\end{array}
$$

The λ_σ-calculus is not locally confluent on open terms, so that Ríos proposed in [Río93] a slight variant, called $\lambda_{\sigma'}$-calculus, in order to overcome this problem. The λ_ϕ-calculus [Muñ97]

is a left-linear variant of $\lambda_{\sigma'}$, which is practical for implementations and does not poses typing problems when working in higher-order type theories. As well as λ_σ, the λ_ϕ-calculus is weak normalizing and confluent on semi-open terms. The rewriting rules of λ_ϕ are:

$$
\begin{array}{lllll}
(Beta) & (\lambda a)b & \longrightarrow & a[b\cdot \uparrow^0] \\
(App) & (a\ b)[s] & \longrightarrow & (a[s]\ b[s]) \\
(Lambda) & (\lambda a)[s] & \longrightarrow & \lambda(a[1\cdot s\circ \uparrow^{succ(0)}]) \\
(Clos) & (a[s])[t] & \longrightarrow & a[s\circ t] \\
(VarCons) & 1[a\cdot s] & \longrightarrow & a \\
(Id) & a[\uparrow^0] & \longrightarrow & a \\
(Map) & (a\cdot s)\circ t & \longrightarrow & a[t]\cdot(s\circ t) \\
(IdL) & \uparrow^0 \circ s & \longrightarrow & s \\
(ShiftCons) & \uparrow^{succ(n)} \circ(a\cdot s) & \longrightarrow & \uparrow^n \circ s \\
(ShiftShift) & \uparrow^{succ(n)} \circ \uparrow^m & \longrightarrow & \uparrow^n \circ \uparrow^{succ(m)} \\
(Shift1) & 1\cdot \uparrow^{succ(0)} & \longrightarrow & \uparrow^0 \\
(Shift2) & 1[\uparrow^n]\cdot \uparrow^{succ(n)} & \longrightarrow & \uparrow^n
\end{array}
$$

The calculi λ_v and λ_f

Pierre Lescanne introduces in [Les94] the λ_v-calculus which preserves β-strong normalization [LRD94, BBLRD96], but is not confluent on (semi) open terms. The system is very simple as it has no composition operators for substitutions and has as few forms of substitutions as possible. As a consequence, it can be derived from our scheme in a very direct way: we have just to associate a constant to the shift-constant $shift^v$, a unary substitution constructor to the cons-function $cons^v$, and another one to the the lift-function $lift^v$, then each of the basic conditions 5, 6, 7, 8 and 9 becomes a rewrite rule; whereas conditions on substitutions in definition 3.5 can be immediately deduced from the calculus which is obtained from the basic conditions. The rewriting rules of λ_v are:

$$
\begin{array}{lllll}
(Beta) & (\lambda a)b & \longrightarrow & a[\#(b)] \\
(App) & (a\ b)[s] & \longrightarrow & (a[s]\ b[s]) \\
(Lambda) & (\lambda a)[s] & \longrightarrow & \lambda(a[\Uparrow (s)]) \\
(FVar) & 1[\#(a)] & \longrightarrow & a \\
(RVar) & n+1[\#(a)] & \longrightarrow & n \\
(FVarLift1) & 1[\Uparrow (s)] & \longrightarrow & 1 \\
(RVarLift1) & n+1[\Uparrow (s)] & \longrightarrow & n[s][\uparrow] \\
(VarShift1) & n[\uparrow] & \longrightarrow & n+1
\end{array}
$$

The λ_f-calculus was introduced in [FKP96] in order to show preservation of β-strong normalization of λ_d and λ_{dn}. It is in some sense the *minimal* calculus of explicit substitutions that one can expect: it correctly implements the lambda calculus and it has as few forms of substitutions and rewriting rule schemes as possible. The λ_v-calculus can also be translated to it [FKP96]. The rewriting rules of λ_f are the following, where $i \geq 0$, $\Uparrow^0 (s)$ denotes s and $\Uparrow^n (s)$ denotes $\underbrace{\Uparrow (\ldots \Uparrow}_{n\ times}(s)\ldots)$.

$(Beta)$	$(\lambda a)b$	$\longrightarrow$	$a[\#(b)]$
(App)	$(a\ b)[s]$	$\longrightarrow$	$(a[s]\ b[s])$
$(Lambda)$	$(\lambda a)[s]$	$\longrightarrow$	$\lambda(a[\Uparrow(s)])$
$(FVar)$	$1[\#(a)]$	$\longrightarrow$	a
$(FVarLift1)$	$1[\Uparrow(s)]$	$\longrightarrow$	1
$(Shift)$	$a[\Uparrow^i(\uparrow)][\Uparrow^i(\#(b))]$	$\longrightarrow$	a
$(ShiftLift)$	$a[\Uparrow^i(\uparrow)][\Uparrow^{i+1}(s)]$	$\longrightarrow$	$a[\Uparrow^i(s)][\Uparrow^i(\uparrow)]$

The calculi λ_d and λ_{dn}

Strongly inspired from the λ_τ-calculus in [Río93], which was in turn inspired by [Ehr88], we proposed in [Kes96] the λ_d-calculus by putting together the nice properties of the three previous substitution calculi: we just use unary substitution operators for the lift-function and the cons-function, keeping composition of substitutions which is one the main features needed to describe different implementations of functional programming. The rewriting rules of λ_d are:

$(Beta)$	$(\lambda a)b$	$\longrightarrow$	$a[\#(b)]$
(App)	$(a\ b)[s]$	$\longrightarrow$	$(a[s]\ b[s])$
$(Lambda)$	$(\lambda a)[s]$	$\longrightarrow$	$\lambda(a[\Uparrow(s)])$
$(Clos)$	$(a[s])[t]$	$\longrightarrow$	$a[s\circ t]$
$(FVar1)$	$1[\#(a)]$	$\longrightarrow$	a
$(FVar2)$	$1[\#(a)\circ s]$	$\longrightarrow$	$a[s]$
$(FVarLift1)$	$1[\Uparrow(s)]$	$\longrightarrow$	1
$(FVarLift2)$	$1[\Uparrow(s)\circ t]$	$\longrightarrow$	$1[t]$
(Ass)	$(s_1\circ s_2)\circ s_3$	$\longrightarrow$	$s_1\circ(s_2\circ s_3)$
$(Shift1)$	$\uparrow\circ\#(a)$	$\longrightarrow$	id
$(Shift2)$	$\uparrow\circ(\#(a)\circ s)$	$\longrightarrow$	s
$(ShiftLift1)$	$\uparrow\circ\Uparrow(s)$	$\longrightarrow$	$s\circ\uparrow$
$(ShiftLift2)$	$\uparrow\circ(\Uparrow(s)\circ t)$	$\longrightarrow$	$s\circ(\uparrow\circ t)$
$(Lift1)$	$\Uparrow(s)\circ\Uparrow(t)$	$\longrightarrow$	$\Uparrow(s\circ t)$
$(Lift2)$	$\Uparrow(s)\circ(\Uparrow(t)\circ u)$	$\longrightarrow$	$\Uparrow(s\circ t)\circ u$
(IdL)	$id\circ s$	$\longrightarrow$	s
(IdR)	$s\circ id$	$\longrightarrow$	s
$(LiftId)$	$\Uparrow(id)$	$\longrightarrow$	id
(Id)	$a[id]$	$\longrightarrow$	a

Note that the $MapEnv$ rule of the λ_τ-calculus in [Río93] $\#(a)\circ s \longrightarrow \Uparrow(s)\circ\#(a[s])$ is replaced here by $FVar2$ and $Shift2$. This may be in principle superfluous, but is essential to make our calculus fit in the scheme proposed in this paper. This difference makes it possible to prove confluence and strong normalization of d, while confluence of τ is still a conjecture and τ-strong normalization does not hold [Les]. However, the rule $MapEnv$ is essential if one wants to recover confluence on open terms [7], and is at the same time, as explained

[7] Indeed, the open term $((\lambda X)a)[s]$ reduces to both $X[\#(a)\circ s]$ and $X[\Uparrow(s)\circ\#(a[s])]$, which have no common reduct in λ_d but can be joined in the λ_τ-calculus proposed in [Río93].

by [Mel95] and [BBLRD96], the cause of non termination in many λ-calculi with explicit substitutions. Preservation of β-strong normalization holds for λ_d and λ_{dn} (see [FKP96] for details).

The λ_{dn}-calculus is just λ_d plus natural numbers to represent variables. It is a natural extension of λ_d which also satisfies the abstract properties required by the scheme. To define the set of rewriting rules of λ_{dn}, we add the following set of rules to those of λ_d.

$$\begin{array}{lllll}
(VarShift1) & n[\uparrow] & \longrightarrow & n+1 \\
(VarShift2) & n[\uparrow \circ s] & \longrightarrow & n+1[s] \\
(RVar) & n+1[\#(a)] & \longrightarrow & n \\
(RVarLift1) & n+1[\Uparrow (s)] & \longrightarrow & n[s \circ \uparrow] \\
(RVarLift2) & n+1[\Uparrow (s) \circ t] & \longrightarrow & n[s \circ (\uparrow \circ t)] \\
(RMap) & n+1[\#(a) \circ s] & \longrightarrow & n[s]
\end{array}$$

It remains now to verify for each calculus presented above not only the existence of the *Lambda*, *App* and *Beta* rules, but also the basic conditions required in definitions 3.4 and the conditions on substitutions appearing in definition 3.5. We start by studying the basic conditions.

Theorem 6.3 Every calculus $W \in \{\sigma, \sigma_{\Uparrow}, \phi, \upsilon, f, d, dn\}$ is strongly normalizing.

Proof. See [HL86, CHR92] for λ_σ, [HL89] for $\lambda_{\sigma_{\Uparrow}}$, [Les94] for λ_υ, [Muñ97] for λ_ϕ and [FKP96] for λ_f. For λ_d and λ_{dn} we proceed as follows. We first define a function $\mathcal{T}$ from objects in Λ_{dn} into objects in $\Lambda_{\sigma_{\Uparrow}}$:

$$\begin{array}{llllllllll}
\mathcal{T}(m) & = & m & \mathcal{T}(\lambda a) & = & \lambda(\mathcal{T}(a)) & \mathcal{T}(\Uparrow (s)) & = & \Uparrow (\mathcal{T}(s)) \\
\mathcal{T}(\uparrow) & = & \uparrow & \mathcal{T}(a\ b) & = & \mathcal{T}(a)\,\mathcal{T}(b) & \mathcal{T}(s \circ t) & = & \mathcal{T}(s) \circ \mathcal{T}(t) \\
\mathcal{T}(id) & = & id & \mathcal{T}(a[s]) & = & \mathcal{T}(a)[\mathcal{T}(s)] & \mathcal{T}(\#(a)) & = & \mathcal{T}(a) \cdot id
\end{array}$$

Since $\Lambda_d \subseteq \Lambda_{dn}$, then the function $\mathcal{T}$ also defines a translation from Λ_d to $\Lambda_{\sigma_{\Uparrow}}$.

Now, one shows that for every $o \in \Lambda_d$ (resp. for every $o \in \Lambda_{dn}$) such that $o \longrightarrow_d o'$ (resp. $o \longrightarrow_{dn} o'$), then $\mathcal{T}(o) \longrightarrow^+_{\sigma_{\Uparrow}} \mathcal{T}(o')$, so that strong normalization of $\sigma_{\Uparrow}$ implies that of d and dn. For that, it is sufficient to check the cases where the reduction $o \longrightarrow_d o'$ (resp. $o \longrightarrow_{dn} o'$) takes place at the root of o, verifications that we leave to the reader, for the other cases one proceeds by induction on the object o.

Theorem 6.4 Every calculus $W \in \{\sigma, \sigma_{\Uparrow}, \phi, \upsilon, f, d, dn\}$ is confluent.

Proof. A consequence of local confluence and theorem 6.3, by application of Newman's lemma.

Theorem 6.5 For every calculus $W \in \{\sigma, \sigma_{\Uparrow}, \phi, \upsilon, f, d, dn\}$, all the W-normal forms are pure terms.

Proof. See [ACCL91] for λ_σ, [HL89] for $\lambda_{\sigma_{\Uparrow}}$, [Les94] for λ_υ, [Muñ97] for λ_ϕ and [FKP96] for λ_f. For λ_d and λ_{dn} we proceed as in [Río93]:

- Let $a \in \Lambda_d$ in d-normal form. We proceed by induction on the structure of a:

  If a is 1 the property holds. Now, if a is an application or an abstraction, the property comes by induction hypothesis. So, let us suppose that a is equal to $k[s]$. If k is an application (resp. a λ-abstraction or a substituted term $k'[t]$), then App (resp. $Lambda$ and $Clos$) can be applied, so k is necessarily the variable 1.

  Let us see which are the possible cases for s. If s is $\#$, $\Uparrow$ or id, then $FVar1$, $FVarLift1$ or Id can be applied, so s is necessarily $\uparrow$ or $\circ$. If $s = \uparrow$ we are done because $1[\uparrow]$ is a pure term according to our definition. So let us suppose that $s = t \circ u$. If t is $\#$, $\Uparrow$, $\circ$ or id, then $FVar2$, $FVarLift2$, Ass or IdL can be applied, so t is necessarily $\uparrow$.

  Let us show by induction on u that $\uparrow \circ u$ is $\diamond^n(\uparrow)$ for some n.

  If u is id, $\#$ or $\Uparrow$, then IdR, $Shift1$ or $ShiftLift1$ can be applied, so u is $\uparrow$ or $u_1 \circ u_2$. If $u = \uparrow$ we are done because $\uparrow \circ \uparrow = \diamond^2(\uparrow)$. If $u = u_1 \circ u_2$, suppose that u_1 is $\circ$, id, $\#$ or $\Uparrow$. Then Ass, IdL, $Shift2$ or $ShiftLift2$ can be applied, so that u_1 is $\uparrow$ and by induction hypothesis u_2 is $\diamond^m(\uparrow)$ for some m so that $u = u_1 \circ u_2$ is equal to $\uparrow \circ \diamond^m(\uparrow) = \diamond^{m+1}(\uparrow)$.

  Now, since $s = \uparrow \circ u$ and $u = \diamond^n(\uparrow)$ for some n, then $k[s]$ is equal to $1[\diamond^{n+1}(\uparrow)]$ that is a pure term according to our definition.

- Let $a \in \Lambda_d$ in dn-normal form. We proceed by induction on the structure of a:

  If a is a variable the property holds. Now, if a is an application or an abstraction, the property comes by induction hypothesis. So, let us suppose that a is equal to $k[s]$. If k is an application (resp. a λ-abstraction or a substituted term $k_1[t]$), then App (resp. $Lambda$ and $Clos$) can be applied, so k is necessarily a variable n.

  Now, if s is $\#$, then $FVar1$ or $RVar$ applies. If s is $\Uparrow$, then $FVarLift1$ or $RVarLift1$ applies. If s is id (resp. $\uparrow$), then Id (resp. $VarShift1$) applies. So let us see the case $s = s_1 \circ s_2$. If s_1 is $\Uparrow$, then $FVarLift2$ or $RVarLift2$ applies. If s_1 is $\#$, then $RMap$ applies. If s_1 is id (resp. $\uparrow$), then IdL (resp. $VarShift2$) applies. So, $s_1 = s_3 \circ s_4$ but in that case Ass applies. We can conclude that a cannot be a term $k[s]$ and thus we are done.

Theorem 6.6 Every $W \in \{\sigma, \sigma_{\Uparrow}, \phi, \upsilon, f, d, dn\}$ verifies the basic conditions in definition 3.4.

Proof. Every calculus $W \in \{\sigma, \sigma_{\Uparrow}, \phi, \upsilon, f, d, dn\}$ is strongly normalizing (theorem 6.3) and confluent (theorem 6.4). On the other hand, W-normal forms are pure terms by theorem 6.5 and property 4 is evident for all the cases, so it remains now to verify properties 5, 6, 7, 8 and 9 in definition 3.4.

- Property 5: we have $1[lift^W(s)] =_{VarCons} 1$ for $W \in \{\sigma, \phi\}$, and $1[\Uparrow(s)] =_{FVarLift1} 1$ in $\sigma_{\Uparrow}$, d, dn, υ and f.
- Property 6: We have $m+1[\Uparrow(s)] =_{RVarLift1,Clos} m[s][\uparrow]$ in $\sigma_{\Uparrow}$ and dn, $m+1[\Uparrow(s)] =_{RVarLift1} m[s][\uparrow]$ in υ, $\underline{m+1}[\Uparrow(s)] = 1[\uparrow]^{m-1}[\uparrow][\Uparrow(s)] =_{ShiftLift} 1[\uparrow]^{m-1}[s][\uparrow] = \underline{m}[s][\uparrow]$ in f.
  - For σ we have $\underline{m+1}[1 \cdot (s \circ \uparrow)] = 1[\diamond^m(\uparrow)][1 \cdot (s \circ \uparrow)] =_{Clos,Ass} 1[\diamond_m(\uparrow, (1 \cdot (s \circ \uparrow)))]$. If $m = 1$, then

$$1[\diamond_1(\uparrow, (1 \cdot (s \circ \uparrow)))] = 1[\uparrow \circ (1 \cdot (s \circ \uparrow))] =_{ShiftCons} 1[s \circ \uparrow] =_{Clos} 1[s][\uparrow]$$

If $m > 1$, then

$$\begin{array}{rcll} 1[\diamond_m(\uparrow,(1\cdot(s\circ\uparrow)))] & = & 1[\diamond_{m-1}(\uparrow,\uparrow\circ(1\cdot(s\circ\uparrow)))] & =_{ShiftCons} \\ & & 1[\diamond_{m-1}(\uparrow,(s\circ\uparrow))] & =_{Ass} \\ & & 1[\diamond^m(\uparrow)\circ(s\circ\uparrow)] & =_{Clos} \quad \underline{m}[s][\uparrow] \end{array}$$

- For ϕ:

$$\begin{array}{lll} \underline{m+1}[1\cdot(s\circ\uparrow^{succ(0)})] & = & 1[\uparrow^{succ(m-1)}][1\cdot(s\circ\uparrow^{succ(0)})] \quad =_{Clos} \\ 1[\uparrow^{succ(m-1)}\circ(1\cdot(s\circ\uparrow^{succ(0)}))] & =_{ShiftCons} & 1[\uparrow^{m-1}\circ(s\circ\uparrow^{succ(0)})] \end{array}$$

If $m = 1$, then $1[\uparrow^{m-1}\circ(s\circ\uparrow^{succ(0)})] =_{IdL} 1[s\circ\uparrow^{succ(0)}] =_{Clos} 1[s][\uparrow]$.
If $m > 1$, then $1[\uparrow^{m-1}\circ(s\circ\uparrow^{succ(0)})] = 1[\uparrow^{succ(m-2)}\circ(s\circ\uparrow^{succ(0)})]$ which is equal by $Clos$ to $1[\uparrow^{succ(m-2)}][s][\uparrow^{succ(0)}] = \underline{m}[s][\uparrow]$.

- For d:

$$\underline{m+1}[\Uparrow(s)] = 1[\diamond^m(\uparrow)][\Uparrow(s)] =_{Clos} 1[\diamond^m(\uparrow)\circ\Uparrow(s)] =_{Ass} 1[\diamond_m(\uparrow,\Uparrow(s))]$$

If $m = 1$, then $1[\diamond_1(\uparrow,\Uparrow(s))] = 1[\uparrow\circ\Uparrow(s)] =_{ShiftLift1} 1[s\circ\uparrow] =_{Clos} 1[s][\uparrow]$.
If $m > 1$, then

$$\begin{array}{rcll} 1[\diamond_m(\uparrow,\Uparrow(s))] & = & 1[\diamond_{m-1}(\uparrow,\uparrow\circ\Uparrow(s))] & =_{ShiftLift1} \\ & & 1[\diamond_{m-1}(\uparrow,s\circ\uparrow)] & =_{Ass} \\ & & 1[\diamond^{m-1}(\uparrow)\circ(s\circ\uparrow)] & =_{Clos} \quad \underline{m}[s][\uparrow] \end{array}$$

- Property 7: we have $m+1[k\cdot id] =_{RVar,Id} m$ in $\sigma_{\Uparrow}$, $m+1[\#(k)] =_{RVar} m$ in υ and $dn\ \underline{m+1}[\#(k)] = 1[\uparrow]^{m-1}[\uparrow][\#(k)] =_{Shift} 1[\uparrow]^{m-1} = \underline{m}$ in f.
  - For σ we have $\underline{m+1}[k\cdot id] = 1[\diamond^m(\uparrow)][k\cdot id] =_{Clos,Ass} 1[\diamond_m(\uparrow,(k\cdot id))]$.
    If $m = 1$, then $1[\diamond_1(\uparrow,(k\cdot id))] = 1[\uparrow\circ(k\cdot id)] =_{ShiftCons} 1[id] =_{VarId} 1$.
    If $m > 1$, then

$$1[\diamond_m(\uparrow,(k\cdot id))] = 1[\diamond_{m-1}(\uparrow,(\uparrow\circ(k\cdot id)))] =_{ShiftCons,ShiftId} 1[\diamond^{m-1}(\uparrow)] = \underline{m}$$

  - For ϕ:

$$\underline{m+1}[b\cdot\uparrow^0] = 1[\uparrow^{succ(m-1)}][b\cdot\uparrow^0] =_{Clos,ShiftCons} 1[\uparrow^{m-1}\circ\uparrow^0] =_{ShiftId} 1[\uparrow^{m-1}] = \underline{m}$$

  - For d:

$$\underline{m+1}[\#(k)] = 1[\diamond^m(\uparrow)][\#(k)] =_{Clos} 1[\diamond^m(\uparrow)\circ\#(k)] =_{Ass} 1[\diamond_m(\uparrow,\#(k))]$$

If $m = 1$, then $1[\diamond_1(\uparrow,\#(k))] = 1[\uparrow\circ\#(k)] =_{Shift1} 1[id] =_{Id} 1$.
If $m > 1$, then

$$1[\diamond_m(\uparrow,\#(k))] = 1[\diamond_{m-1}(\uparrow,\uparrow\circ\#(k))] =_{Shift1} 1[\diamond_{m-1}(\uparrow,id)] =_{IdR} 1[\diamond^{m-1}(\uparrow)] = \underline{m}$$

- Property 8: we have $1[k \cdot id] =_{VarCons} k$ in σ, $1[k \cdot id] =_{FVar} k$ in $\sigma_{\Uparrow}$, $1[\#(k)] =_{FVar} k$ in υ and f, $1[\#(k)] =_{VarCons} k$ in ϕ, $1[\#(k)] =_{FVar1} k$ in d and dn.
- Property 9: we have $m[\uparrow] =_{VarShift1} m+1$ in $\sigma_{\Uparrow}$, υ, dn, and $\underline{m}[\uparrow] = 1[\uparrow]^{m-1}[\uparrow] = \underline{m+1}$ in f.
  - For σ and d: If $m = 1$, then $1[\uparrow] = \underline{2}$.
    If $m > 1$, then $\underline{m}[\uparrow] = 1[\diamond^{m-1}(\uparrow)][\uparrow] =_{Clos} 1[\diamond^{m-1}(\uparrow) \circ \uparrow] =_{Ass} 1[\diamond^{m}(\uparrow)] = \underline{m+1}$
  - For ϕ: If $m = 1$, then $1[\uparrow^{succ(0)}] = \underline{2}$.
    If $m > 1$, then $\underline{m}[\uparrow] = 1[\uparrow^{succ(m-2)}][\uparrow^{succ(0)}] =_{Clos} 1[\uparrow^{succ(m-2)} \circ \uparrow^{succ(0)}] =_{ShiftShift} 1[\uparrow^{0} \circ \uparrow^{succ(m-1)}] =_{IdL} 1[\uparrow^{m}] = \underline{m+1}$

Theorem 6.7 Every calculus $W \in \{\sigma, \sigma_{\Uparrow}, \phi, \upsilon, f, d, dn\}$ verifies the scheme in definition 3.5.

Proof. We have to verify that for every variable $\underline{m}$ and every constructor ξ one of the conditions described in definition 3.5 holds. So let us analyze each constructor in detail:

- For id: one verifies for $W \in \{\sigma, \sigma_{\Uparrow}, d, dn\}$, that $\underline{m}[id] =_{Id} \underline{m}$, so that the first case of definition 3.5 applies for $\underline{n} = \underline{m}$, $k = 0$ and $p = 0$.
- For $\circ$: one verifies for $W \in \{\sigma, \sigma_{\Uparrow}, \phi, d, dn\}$, that for every substitutions s and t we have $\underline{m}[s \circ t] =_{Clos} \underline{m}[s][t]$, so the first case of definition 3.5 applies for $\underline{n} = \underline{m}$, $k = 0$, $p = 2$, $i_1 = 1$ and $i_2 = 2$.
- For $\#$: let us consider $W \in \{d, dn, f, \upsilon\}$ where we have defined $cons^W(a) = \#(a)$. If $\underline{m} = 1$, then by definition 3.4 (8) we have $1[cons^W(k)] =_W k$, so that the second case of definition 3.5 applies for $i = 1$. Otherwise, by definition 3.4 (7) we have $\underline{m+1}[\#(a)]) =_W \underline{m}$, so the first case of definition 3.5 applies for $\underline{n} = \underline{m}$, $p = 0$ and $k = 0$.
- For $\Uparrow$: let us consider $W \in \{d, dn, f, \upsilon, \sigma_{\Uparrow}\}$ where we have defined $lift^W(s) = \Uparrow(s)$. If $\underline{m} = 1$, then by definition 3.4 (5) we have $1[\Uparrow(s)] =_W 1$, so the first case of definition 3.5 applies for $\underline{n} = 1$, $p = 0$ and $k = 0$. Otherwise, by definition 3.4 (6) we have $\underline{m+1}[\Uparrow(s)] =_W \underline{m}[s][\uparrow]$, so the first case of definition 3.5 applies for $\underline{n} = \underline{m}$, $p = 1$, $i_1 = 1$, $k = 1$ and $u_1 = \uparrow$.
- For $\cdot$: let us consider $W \in \{\phi, \sigma, \sigma_{\Uparrow}\}$. If $\underline{m} = 1$, then $1[a \cdot s] =_W a$, so the second case of definition 3.5 applies for $i = 1$. Otherwise, $\underline{m+1}[a \cdot s] =_W \underline{m}[s]$ so that the first case of definition 3.5 applies for $\underline{n} = \underline{m}$, $p = 1$, $i_1 = 2$ and $k = 0$.
- For $\uparrow$: let us consider $W \in \{\sigma, \sigma_{\Uparrow}, f, d, dn, \upsilon\}$ where $shift^W = \uparrow$. Then $\underline{m}[\uparrow] =_W \underline{m+1}$ by definition 3.4 (9), so that the first case of definition 3.5 applies for $\underline{n} = \underline{m+1}$, $p = 0$ and $k = 0$.
- For $\uparrow^{k^*}$: this constructor only concerns the ϕ-calculus, where $\underline{m}[\uparrow^{k^*}] =_{\phi} \underline{m+k}$, so that the first case of definition 3.5 applies for $\underline{n} = \underline{m+k}$, $p = 0$ and $k = 0$.

Theorem 6.8 The non-extensional and extensional versions of λ_σ, $\lambda_{\sigma_{\Uparrow}}$, λ_ϕ, λ_υ, λ_d, λ_{dn} and λ_f are all confluent.

Proof. The proof comes from theorems 6.6 and 6.7, by application of theorems 5.18 and 5.19.

7 Conclusion

We give a general and homogeneous treatment of λ-calculi with explicit substitutions. Such treatment is essential in the field of implementation of functional programming because of the growing number of such calculi proposed in the literature. The major contributions of this paper can be summarized by the following points:

- We propose a scheme for λ-calculi with explicit substitutions which allows to reason about explicit substitutions in an abstract way. In particular, we are able to deal with many different calculi such as λ_σ, $\lambda_{\sigma_\Uparrow}$, λ_υ, λ_f, λ_ϕ, λ_d, λ_{dn} and λ_s, using the same formalism and tools.
- We study confluence of extensional and non-extensional λ-calculi with explicit substitutions, where extensionality is interpreted by η-expansion. This reading of extensionality turns out to be much more natural and less complex than previous interpretations via η-contraction.

Our approach turns out to be also useful to derive *new* calculi with explicit substitutions having nice properties. In particular, λ_d and λ_{dn} were derived in [Kes96] from the scheme using an automatic tool to perform completion.

We think that an abstract approach in the spirit of the one proposed in this paper would also be useful to study confluence on *open terms* and preservation of strong normalization, the challenge being the definition of an appropriate scheme associated to those properties. The scheme proposed in this paper could also be combined with Combinatory Reduction Systems [Klo80, KvOvR93], generalizing in that way the formalisms in [Pag98, BR96] which are defined to just cover one particular explicit substitution calculus.

Acknowledgments

I would like to thank Pierre Lescanne and Gilles Dowek for interesting discussions, Roberto Di Cosmo and Pierre-Louis Curien for useful comments, and Alejandro Ríos and Paul-André Mellies for explaining me some key points in the theory of calculi with explicit substitutions.

References

[ACCL91] Martin Abadi, Luca Cardelli, Pierre Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 4(1):375–416, 1991.

[Aka93] Yohji Akama. On mints' reductions for ccc-calculus. In *Proceedings of Int. Conf. on Typed Lambda Calculi and Applications (TLCA)*, number 664 in LNCS. Springer Verlag, 1993.

[BBLRD96] Zine-El-Abidine Benaissa, Daniel Briaud, Pierre Lescanne, and Jocelyne Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, 1996.

[BR96] Roel Bloo and Kristoffer Rose. Combinatory reduction systems with explicit substitution that preserve strong normalisation. In Harald Ganzinger, editor, *Proc. of the Seventh International Conference on Rewriting Techniques and Applications (RTA)*, number 1103 in LNCS, 1996.

[Bri95] Daniel Briaud. An explicit *eta* rewrite rule. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Proceedings of Int. Conf. on Typed Lambda Calculi and Applications (TLCA)*, number 902 in LNCS. Springer-Verlag, 1995.

[CDC91] Pierre-Louis Curien and Roberto Di Cosmo. A confluent reduction system for the λ-calculus with surjective pairing and terminal object. In Leach, Monien, and Artalejo, editors, *Intern. Conf. on Automata, Languages and Programming (ICALP)*, number 510 in LNCS, pages 291–302. Springer-Verlag, 1991.

[CHL92] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. Technical Report 1617, INRIA-Rocquencourt, 1992.

[CHR92] Pierre-Louis Curien, Thérèse Hardin, and Alejandro Ríos. Strong normalisation of substitutions. In *MFCS'92*, number 629 in LNCS, pages 209–218, 1992.

[Cur83] Pierre-Louis Curien. *Combinateurs Catégoriques, algorithmes séquentiels et programmation applicative.* Thèse d'etat, Université Paris VII, 1983.

[Cur86] Pierre-Louis Curien. *Categorical combinators, sequential algorithms and functional programming.* Progress in Theoretical Computer Science. Birkhaüser, 1986. first edition.

[dB72] N. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indag. Mat.*, 5(35):381–392, 1972.

[dB78] N. de Bruijn. Lambda-calculus notation with namefree formulas involving symbols that represent reference transforming mappings. *Indag. Mat.*, 40:384–356, 1978.

[DCK93] Roberto Di Cosmo and Delia Kesner. A confluent reduction for the extensional typed $\lambda-$calculus with pairs, sums, recursion and terminal object. In Andrzej Lingas, editor, *Intern. Conf. on Automata, Languages and Programming (ICALP)*, number 700 in LNCS. Springer-Verlag, 1993.

[DCK94] Roberto Di Cosmo and Delia Kesner. Combining first order algebraic rewriting systems, recursion and extensional typed lambda calculi. In *Intern. Conf. on Automata, Languages and Programming (ICALP)*, number 820 in LNCS. Springer-Verlag, 1994.

[DCK95] Roberto Di Cosmo and Delia Kesner. Rewriting with extensional polymorphic λ-calculus (extended abstract). In *Proceedings of Annual Conference of the European Association for Computer Science Logic (CSL)*, number 1092 in LNCS, 1995.

[DHK95] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In *Proceedings of the Symposium on Logic in Computer Science (LICS)*, 1995.

[Dou93] Daniel Dougherty. Some lambda calculi with categorical sums and products. In *Proc. of the Fifth International Conference on Rewriting Techniques and Applications (RTA)*, number 690 in LNCS, 1993.

[Ehr88] Thomas Ehrhard. *Une sémantique catégorique des types dépendants. Application au calcul des constructions.* Thèse de doctorat, Université de Paris VII, 1988.

[FKP96] Maria C.F. Ferreira, Delia Kesner, and Laurence Puel. λ-calculi with explicit substitutions and composition which preserve β-strong normalization (extended abstract). In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Proc. of the Fifth International Conference on Algebraic and Logic Programming (ALP)*, number 1139 in LNCS, pages 284–298, 1996.

[Har87] Thérèse 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.

[Har92] Thérèse Hardin. η-reduction for explicit substitutions. In *Algebraic and Logic Programming'92*, number 632 in LNCS, 1992.

[Har94] Thérèse Hardin. Eta-conversion for the languages of explicit substitutions. *Applicable Alg. in Engineering, Communication and Computing*, 5:317–341, 1994.

[HL86] Thérèse Hardin and Aain Laville. Proof of termination of the rewriting system subst on c.c.l. *Theoretical Computer Science*, 1986.

[HL89] Thérèse Hardin and Jean-Jacques Lévy. A confluent calculus of substitutions. In *France-Japan Artificial Intelligence and Computer Science Symposium*, 1989.

[HMP95] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional back-ends and compilers within the lambda-sigma calculus, 1995. Draft.

[Hue76] Gérad Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. Thèse de doctorat d'état, Université Paris VII, 1976.

[JG95] Colin Barry Jay and Neil Ghani. The Virtues of Eta-expansion. *Journal of Functional Programming*, 5(2):135–154, April 1995.

[Kes96] Delia Kesner. Confluence properties of extensional and non-extensional λ-calculi with explicit substitutions. In Harald Ganzinger, editor, *Proc. of the Seventh International Conference on Rewriting Techniques and Applications (RTA)*, number 1103 in LNCS, pages 184–199, 1996.

[Kes97a] Delia Kesner. Confluence of extensional and non-extensional λ-calculi with explicit substitutions. Technical Report 1103, LRI, Université Paris-Sud, 1997.

[Kes97b] Delia Kesner. Reasoning about redundant patterns. *Journal of Functional and Logic Programming*, 1997. To appear.

[Klo80] Jan Willem Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. CWI, Amsterdam, 1980. PhD Thesis.

[KR95] Fairouz Kamareddine and Alejandro Ríos. A λ-calculus à la de bruijn with explicit substitutions. In *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, number 982 in LNCS. Springer-Verlag, 1995.

[KvOvR93] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

[Les] Pierre Lescanne. Personal Communication, 1996.

[Les94] Pierre Lescanne. From λ_σ to λ_υ, a journey through calculi of explicit substitutions. In *Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 60–69. ACM, 1994.

[LRD94] Pierre Lescanne and Jocelyne Rouyer-Degli. The calculus of explicit substitutions $\lambda\nu$. Technical report, INRIA, Lorraine, 1994.

[Mau85] Michel Mauny. *Compilation des langages fonctionnels dans les combinateurs catégoriques. Applications au langage ML.* Thèse 3eme cycle, Université Paris VII, 1985.

[Mel94] Paul-André Mellies. Four typed-lambda calculi with explicit substitutions may not terminate:the first examples, 1994. Draft.

[Mel95] Paul-André Mellies. Typed λ-calculi with explicit substitutions may not terminate. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Proceedings of Int. Conf. on Typed Lambda Calculi and Applications (TLCA)*, volume 902 of *Lecture Notes in Computer Science*, April 1995.

[Min77] Grigori Mints. Closed categories and the theory of proofs. *Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta im. V.A. Steklova AN SSSR*, 68:83–114, 1977.

[Min79] Grigori Mints. Teorija categorii i teoria dokazatelstv.I. *Aktualnye problemy logiki i metodologii nauky*, pages 252–278, 1979.

[Muñ97] César Muñoz. A left-linear variant of $\lambda\sigma$. In Michael Hanus, J. Heering, and K. (Karl) Meinke, editors, *Proceedings of the 6th International Conference on Algebraic and Logic Programming (ALP'97)*, number 1298 in LNCS. Springer-Verlag, September 1997.

[Pag98] Bruno Pagano. *Des calculs de substitution explicite et de leur application à la compilation des langages fonctionnels.* Thèse de doctorat, Université Pierre et Marie Curie, 1998.

[Pot81] Garrel Pottinger. The Church Rosser Theorem for the Typed lambda-calculus with Surjective Pairing. *Notre Dame Journal of Formal Logic*, 22(3):264–268, 1981.

[Pra71] Dag Prawitz. Ideas and results in proof theory. *Proceedings of the 2nd Scandinavian Logic Symposium*, pages 235–307, 1971.

[Río93] Alejandro Ríos. *Contribution à l'étude des λ-calculus avec substitutions explicites.* Thèse de doctorat, Université de Paris VII, 1993.

[vOvR94] Vincent van Oostrom and Femke van Raamsdonk. Weak orthogonality implies confluence: the higher-order case. In *Proc. of the third International Symposium on Logical Foundations of Computer Science*, pages 379–392, 1994.

[Zan94] Hans Zantema. Termination on term rewriting: interpretation and type elimination. *Journal of Symbolic Computation*, 1(17):23–50, 1994.