Comparing and Implementing Calculi of Explicit Substitutions with Eta-Reduction*

Mauricio Ayala-Rincón[†] Flávio L. C. de Moura[‡] Fairouz Kamareddine[§]

Abstract

The past decade has seen an explosion of work on calculi of explicit substitutions. Numerous work has illustrated the usefulness of these calculi for practical notions like the implementation of typed functional programming languages and higher order proof assistants. It has also been shown that eta-reduction is useful for adapting substitution calculi for practical problems like higher order unification. This paper concentrates on rewrite rules for eta-reduction in three different styles of explicit substitution calculi: $\lambda\sigma$, λs_e and the suspension calculus. Both $\lambda\sigma$ and λs_e when extended with eta-reduction rules, have proved useful for solving higher order unification. We enlarge the suspension calculus with an adequate eta-reduction rule which we show to preserve termination and confluence of the associated substitution calculus and to correspond to the eta rules of the other two calculi. We prove that $\lambda\sigma$ and λs_e as well as $\lambda\sigma$ and the suspension calculus are non comparable while λs_e is more adequate than the suspension calculus in simulating one step beta-reduction.

After defining the eta-reduction rule in the suspension calculus, and after comparing these three calculi of explicit substitutions (all with an eta rule), we then concentrate on the implementation of the rewrite rules of eta-reduction in these calculi. We note that it is usual practice when implementing the eta rule for substitution calculi, to mix isolated applications of eta-reduction with the application of other rules of the corresponding substitution calculi. The main disadvantage of this practice is that the eta rewrite rules so obtained are *unclean* because they have an operational semantics different from that of the eta-reduction of the λ-calculus. For the three calculi in question enlarged with adequate eta rules we show how to implement these eta rules. For the λs_e we build a *clean* implementation of the eta rule and we prove that it is not possible to follow the same approach for the $\lambda\sigma$ and λ_{SUSP}.

Keywords Explicit substitutions, λ-calculi, Eta-Reduction.

1 Introduction

Recent years have witnessed an explosion of work on expliciting substitution [1, 7, 9, 16, 17, 21, 23, 4] and on its usefulness for: automated deduction and theorem proving [32, 33], proof theory [40], programming languages [8, 20, 28, 34] and higher order unification [3, 15]. This paper studies three styles of substitutions:

1. The $\lambda\sigma$-style [1] which introduces two different sets of entities: one for terms and one for substitutions.
2. The suspension calculus [37, 34], which introduces three different sets of entities: one for terms, one for environments and one for lists of environments.
3. The λs-style [23] which uses a philosophy of de Bruijn's *Automath* [38] elaborated in the new item notation [22]. The philosophy states that terms are built by applications (a function applied to an argument), abstraction (a function), substitution or updating. The advantages of this philosophy include remaining as close as possible to the familiar λ-calculus (cf. [22]).

*Research partially supported by Grant 47488/01-6 of CNPq

[†]Departamento de Matemática, Universidade de Brasília, Brasília D.F., Brasil. Email: `ayala@mat.unb.br`. Author partially supported by the CNPq Brazilian Council

[‡]Departamento de Matemática, Universidade de Brasília, Brasília D.F., Brasil. Email: `flavio@mat.unb.br`. Author supported by the Brazilian High Education Council CAPES

[§]Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Scotland. Email: `fairouz@macs.hw.ac.uk`

Desired properties of explicit substitution calculi include a) simulation of β-reduction, b) confluence (CR) on closed terms, c) CR on open terms, d) strong normalization (SN) of explicit substitutions and e) preservation of SN of the λ-calculus. The $\lambda\sigma$-calculus (without eta) satisfies a), b), d) and satisfies c) only when the set of open terms is restricted to those which admit metavariables of sort `term`. The λs-calculus (without eta) satisfies a)..e) but not c). However, the λs-calculus has an extension λs_e (again without eta) for which a)..c) holds, but e) fails and d) is unknown. The suspension calculus (which does not have eta) satisfies a) and when restricted to well formed terms it also satisfies b)..d). For the suspension calculus, e) is unknown.

The above discussion holds for these calculi without eta-reduction. However, work on higher order unification (HOU) in λs_e and $\lambda\sigma$ established the importance of combining eta-reduction (as well as expansion) with explicit substitutions. This has provided extensions of λs_e and $\lambda\sigma$ with eta-reduction rules also referred to by λs_e and $\lambda\sigma$ (cf. [15, 3]). In fact, due to the importance of eta-reduction, calculi of explicit substitutions (including $\lambda\sigma$) have been extended with eta rules earlier than the application of $\lambda\sigma$ to HOU [19, 39, 12, 29]. Eta-reduction (as well as expansion) is necessary for working with functions and programs, since one needs to express functional or extensional equality. In particular, when the application of two lambda terms a and b to any term c yields the same result, then a and b should be considered equal.

Although λs_e and $\lambda\sigma$ have already been extended with eta-reduction, the suspension calculus still has not. This paper fills the gap and gives the first extension of the rewriting system of the suspension calculus with an eta-reduction rule bringing to it the advantages of the use of eta-reduction in substitutions calculi. Once the suspension calculus is extended with this eta-reduction rule, one can then compare these three calculi and assess the way eta-reduction should be implemented in each of them. This paper deals with three useful notions for these three calculi:

- Extending the suspension calculus with an eta-reduction rule resulting in λ_{SUSP}. We show the soundness of this rule and the confluence and strong normalisation of the underlying substitution calculus with eta.

- Comparing the *adequacy* of the reduction process of these three substitution calculi extended with eta-reduction, using the efficient simulation of β-reduction of [26] which showed that λs and $\lambda\sigma$ are non comparable. In this paper we show that λs_e and $\lambda\sigma$ as well as $\lambda\sigma$ and λ_{SUSP} are non comparable, that λs_e is more adequate than λ_{SUSP} for simulating one step beta-reduction.

- Reflecting on the correct definition and adequate implementation of the eta-reduction rewrite rules in these calculi. It is usual practice when implementing the eta rule for substitution calculi [11, 2], to mix isolated applications of eta-reduction with the application of other rules of the corresponding substitution calculi. The main disadvantage of this practice is essentially that the eta rewrite rules so obtained are *unclean* because they have an operational semantics different from the one of the eta-reduction rule of the λ-calculus: the notion of functional equivalence embedded in the eta-reduction should be interpreted modulo the semantics of the corresponding substitution calculus. For the three calculi enlarged with adequate eta rules we show how to implement in practice these eta rules without mixing the isolated application of the eta-reduction with the application of other rules of the associated substitution calculi. The definition of a successful implementation depends on an effective specification of a practical method for evaluating the conditions of these eta rules which are conditional rules of the rewriting systems of the three treated calculi. For each of these explicit substitution calculi, our implementation consists basically of a linear verification along a term of the nonexistence of occurrences of the free variable of the eta-reduction while simultaneously upgrading all other free de Bruijn indices and without applying any additional rewrite rule of the corresponding substitution calculus. The three implementations are proved complete in the sense that they effectively simulate eta-reduction over pure lambda terms.

After including the necessary notations and motivation about explicit substitutions, in the second section, we present the $\lambda\sigma$, the λs_e, and the suspension calculus. We enlarge the latter with an eta-reduction rule which is proved sound in the third section. Then, in the fourth and fifth sections, we compare the adequacy of these calculi in simulating one step beta-reduction and the appropriateness of the defined eta rewriting rules. Finally, and before concluding, we discuss the clean implementation of these eta rules in the sixth and seventh sections.

2 Preliminaries

We assume familiarity with the notion of term algebra $\mathcal{T}(\mathcal{F},\mathcal{X})$ built on a (countable) set of variables $\mathcal{X}$ and a set of operators $\mathcal{F}$. Variables in $\mathcal{X}$ are denoted by $X, Y, ...$ and for a term $a \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $var(a)$ denotes the set of variables occurring in a. Throughout, we take $a, b, c, \ldots$ to range over terms. Additionally, we assume familiarity with basic notions of rewriting as in [5]. In particular, for a *reduction relation* R over a set A, we denote with $\overset{=}{\rightarrow}_R$ the **reflexive closure** of R, with $\rightarrow_R^*$ or just $\rightarrow^*$ the **reflexive and transitive closure** of R and with $\rightarrow_R^+$ or just $\rightarrow^+$ the **transitive closure** of R. When $a \rightarrow^* b$ we say that there exists a **derivation** from a to b. By $a \rightarrow^n b$, we mean that the derivation consists of n steps of reduction and call n the **length of the derivation**. Syntactical identity is denoted by $a = b$. For a reduction relation R over A, $(A, \rightarrow_R)$, we use the standard definitions of **(locally-)confluent** or (weakly) Church Rosser **(W)CR**, normal forms and **strong** and **weak normalization/termination SN** and **WN**. Suppose R is a SN reduction relation and let t be a term, then R-nf(t) denotes its normal form. As usual we use indiscriminately either "noetherian" or "terminating" instead of SN.

A **valuation** is a mapping from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F},\mathcal{X})$. The homeomorphic extension of a valuation, θ, from its domain $\mathcal{X}$ to the domain $\mathcal{T}(\mathcal{F},\mathcal{X})$ is called the **grafting** of θ. As usual, valuations and their corresponding graftings are denoted by the same Greek letter. The application of a valuation θ or its corresponding grafting to a term $a \in \mathcal{T}(\mathcal{F},\mathcal{X})$ will be written in postfix notation $a\theta$. The **domain** of a grafting θ, is defined by $Dom(\theta) = \{X \mid X\theta \neq X, X \in \mathcal{X}\}$. Its **range**, is defined by $Ran(\theta) = \cup_{X \in Dom(\theta)} var(X\theta)$. We let $var(\theta) = Dom(\theta) \cup Ran(\theta)$. For explicit representations of a valuation and its corresponding grafting θ, we use the notation $\theta = \{X \mapsto X\theta \mid X \in Dom(\theta)\}$. Note that the notion of grafting, usually called *first order substitution*, corresponds to simple syntactic substitution without renaming.

We use notations from [6] for the λ-calculus. Let $\mathcal{V}$ be a (countable) set of variables denoted by lowercase last letters of the Roman alphabet $x, y, ...$

Definition 2.1 *Terms* $\Lambda(\mathcal{V})$ *of the* λ**-calculus with names** *are inductively defined by:* $\Lambda(\mathcal{V}) ::= x \mid (\Lambda(\mathcal{V})\ \Lambda(\mathcal{V})) \mid \lambda_x.\Lambda(\mathcal{V})$*, where* $x \in \mathcal{V}$*. We call* $\lambda_x.a$ *resp.* $(a\ b)$ **abstraction** *resp.* **application** *terms.*

Terms in $\Lambda(\mathcal{V})$ are called *closed* λ*-terms* or terms without substitution meta-variables. An abstraction $\lambda_x.a$ represents a function of formal parameter x, whose body is a. Its application $(\lambda_x.a\ b)$ to an argument b, returns the value of a, where x is replaced by b. This replacement of formal parameters with arguments is known as β**-reduction**. In the context of the first order substitution or grafting, β-reduction would be defined by $(\lambda_x.a\ b) \rightarrow a\{x \mapsto b\}$.

But in this context problems arise forcing the use of α**-conversion** to rename bound variables:

1. Let $\theta = \{x \mapsto b\}$. There are no semantic differences between the abstractions $\lambda_x.x$ and $\lambda_z.z$; both abstractions represent the identity function. But $(\lambda_x.x)\theta = \lambda_x.b$ and $(\lambda_z.z)\theta = \lambda_z.z$ are different.

2. Let $\theta = \{x \mapsto y\}$. $(\lambda_y.x)\theta = \lambda_y.y$ and $(\lambda_z.x)\theta = \lambda_z.y$, thus a capture is possible.

Consequently, β-reduction, should be defined in a way that takes care of renaming bound variables when necessary to avoid harmful capture of variables.

The λ-calculus usually considers substitution as an atomic operation leaving implicit the computational steps needed to effectively perform computational operations based on substitution such as matching and unification. In any real higher order deductive system, the substitution required by basic operations such as β-reduction should be implemented via smaller operations. Explicit substitution is an appropriate formalism for reasoning about the operations involved in real implementations of substitution. Since explicit substitution is closer to real implementations than to the classic λ-calculus, it provides a more accurate theoretical model to analyze essential properties of real systems (termination, confluence, correctness, completeness, etc.) as well as their time/space complexity. For further details of the importance of explicit substitution see [28, 4].

α-conversion should be performed before applying the substitution in the body of an abstraction. The grafting of a fresh variable avoids the possibility of capture. It is important to note that renaming selects fresh variables that have not been used previously. Moreover, since fresh variables are selected randomly, the result of the application of a substitution θ to a term a, which we denote in prefix notation θa for discriminating substitution from grafting, can be conceived as a *class* of equivalence of terms.

Definition 2.2 *β-**reduction** is the rewriting relation defined by the rewrite rule (β) and η-**reduction** is the rewriting relation defined by the rewrite rule (η), where:*

$$\begin{array}{lll} (\beta) & (\lambda_x.a\ b) & \to \{x/b\}(a) \\ (\eta) & \lambda_x.(a\ x) & \to a,\ \text{if } x \notin \mathcal{F}var(a) \end{array}$$
, where $\mathcal{F}var(a)$ denotes the free variables occurring in a.

Note that our notion of substitution is not completely satisfactory because fresh variables depend on the history of the renaming process. λ-terms with meta-variables or *open λ-terms* are given by:

Definition 2.3 *Terms $\Lambda(\mathcal{V},\mathcal{X})$, of the λ-**calculus with names** and **meta-variables** are inductively defined by: $\Lambda(\mathcal{V},\mathcal{X}) ::= x \mid X \mid (\Lambda(\mathcal{V},\mathcal{X})\ \Lambda(\mathcal{V},\mathcal{X})) \mid \lambda_x.\Lambda(\mathcal{V},\mathcal{X})$, where $x \in \mathcal{V}$ and $X \in \mathcal{X}$.*

We have seen that the names of bound variables and their corresponding abstractors play a semantically irrelevant role in the λ-calculus. So any term in $\Lambda(\mathcal{V})$ or $\Lambda(\mathcal{V},\mathcal{X})$ can be seen as a syntactical representative of its obvious equivalence class. Hence, during syntactic unification, the role that names of bound variables and their corresponding abstractors play increases the complexity of the process and creates confusion.

Avoiding names is an effective way of clarifying the meaning of λ-terms and, for the unification process, of eliminating redundant renaming. De Bruijn proposed in [14] that names of bound variables be replaced by indices which relate these bound variables to their corresponding abstractors.

It is clear that the correspondence between an occurrence of a bound variable and its associated abstractor operator is uniquely determined by its *depth*, that is the number of abstractors between them. Hence, λ-terms can be written in a term algebra over the natural numbers $\mathbb{N}$, representing depth indices, the application operator $(\_\ \_)$ and a sole abstractor operator $\lambda\_$; i.e., $\mathcal{T}(\{(\_\ \_), \lambda\_\} \cup \mathbb{N})$.

In de Bruijn's notation, indexing the occurrences of free variables is given by a *referential* according to a fixed enumeration of the set of variables $\mathcal{V}$, say $x, y, z, \ldots$, and prefixing all λ-terms with $\ldots\lambda_z.\lambda_y.\lambda_x.\_$.

Now we can define the λ-calculus in de Bruijn notation with open terms or meta-variables.

Definition 2.4 *The set $\Lambda_{dB}(\mathcal{X})$ of λ-**terms in notation of de Bruijn** is defined inductively as: $\Lambda_{dB}(\mathcal{X}) ::= \underline{\mathtt{n}} \mid X \mid (\Lambda_{dB}(\mathcal{X})\ \Lambda_{dB}(\mathcal{X})) \mid \lambda\Lambda_{dB}(\mathcal{X})$, where $X \in \mathcal{X}$ and $n \in \mathbb{N}\setminus\{0\}$.*

$\Lambda_{dB}(\mathcal{X})$-terms without meta-variables are called closed λ-terms.

We write de Bruijn indices as $\underline{\mathtt{1}}, \underline{\mathtt{2}}, \underline{\mathtt{3}}, \ldots, \underline{\mathtt{n}}, \ldots$, to distinguish them from scripts. Since all considered calculi of explicit substitutions are built over the language of $\Lambda_{dB}(\mathcal{X})$, we will use Λ to denote $\Lambda_{dB}(\mathcal{X})$.

Defining β-reduction in de Bruijn notation's as $(\lambda a\ b) \to \{\underline{\mathtt{1}}/b\}a$ (where $\{\underline{\mathtt{1}}/b\}a$ is the substitution of the index 1 in a with b) fails: 1) when eliminating the leading abstractor all indices associated with free variable occurrences in a should be decremented; 2) when propagating the substitution $\{\underline{\mathtt{1}}/b\}$ crossing abstractors through a the indices of the substitution (initially $\underline{\mathtt{1}}$) and of the free variables in b should be incremented.

Hence, we need new operators for detecting, incrementing and decrementing free variables.

Definition 2.5 *Let $a \in \Lambda_{dB}(\mathcal{X})$. The i-**lift** of a, denoted a^{+i} is defined inductively as follows:*

1) $X^{+i} = X$ *, for* $X \in \mathcal{X}$ *2)* $(a_1\ a_2)^{+i} = (a_1^{+i}\ a_2^{+i})$

3) $(\lambda a_1)^{+i} = \lambda a_1^{+(i+1)}$ *4)* $\underline{\mathtt{n}}^{+i} = \begin{cases} \underline{\mathtt{n+1}}, & \text{if } n > i \\ \underline{\mathtt{n}}, & \text{if } n \le i \end{cases}$

The **lift** of a term a is its 0-lift and is denoted briefly as a^+.

Definition 2.6 *The application of the* **substitution** *by b at the depth $n-1, n \in \mathbb{N}\setminus\{0\}$, denoted $\{\underline{\mathtt{n}}/b\}a$, on a term a in $\Lambda_{dB}(\mathcal{X})$ is defined inductively as follows:*

1) $\{\underline{\mathtt{n}}/b\}X = X$, *for* $X \in \mathcal{X}$ *2)* $\{\underline{\mathtt{n}}/b\}(a_1\ a_2) = (\{\underline{\mathtt{n}}/b\}a_1\ \{\underline{\mathtt{n}}/b\}a_2)$

3) $\{\underline{\mathtt{n}}/b\}\lambda a_1 = \lambda\{\underline{\mathtt{n+1}}/b^+\}a_1$ *4)* $\{\underline{\mathtt{n}}/b\}\underline{\mathtt{m}} = \begin{cases} \underline{\mathtt{m-1}}, & \text{if } m > n \\ b, & \text{if } m = n \\ \underline{\mathtt{m}}, & \text{if } m < n \end{cases}$ *if* $m \in \mathbb{N}\setminus\{0\}$.

Definition 2.7 *β-**reduction** in the λ-calculus with de Bruijn indices is defined as $(\lambda a\ b) \to \{\underline{\mathtt{1}}/b\}a$.*

Observe that the rewriting system of the sole β-reduction rule is left-linear and non overlapping (i.e. orthogonal). Consequently, the rewriting system defined over $\Lambda_{dB}(\mathcal{X})$ by the β-reduction rule is CR.

In the λ-calculus with names, the η-reduction rule is defined by $\lambda_x.(a\ x) \to a$, if $x \notin \mathcal{F}var(a)$. In $\Lambda_{dB}(\mathcal{X})$, the left side of this rule is written as $\lambda(a'\ 1)$, where a' stands for the corresponding translation of a under some fixed referential of variables into the language of $\Lambda_{dB}(\mathcal{X})$. "$a$ has no free occurrences of x" means, in $\Lambda(\mathcal{X})$, that there are neither occurrences in a' of the index 1 at height zero nor of the index 2 at height one nor of the index 3 at height two etc. Hence, there is in general, a term b such that $b^+ = a$.

Definition 2.8 **η-reduction** *in the λ-calculus with de Bruijn indices is:* $\lambda(a\ 1) \to b$ *if* $b^+ = a$.

3 Calculi à la $\lambda\sigma$, λs_e and the Suspension Calculus

We present $\lambda\sigma$, λs_e and the Suspension Calculus. We enlarge the latter with an eta-reduction rule which we prove to be sound and to preserve the confluence of the suspension calculus.

3.1 The $\lambda\sigma$-calculus

The $\lambda\sigma$-calculus is a first order rewriting system that contains the lambda calculus in de Bruijn notation and which makes explicit the substitutions started by β-reductions [1]. This calculus works on 2-sorted terms: *(proper) terms* (over which $a, b, \ldots$ range), and *substitutions* (over which $s, t, \ldots$ range). In this calculus, when a substitution $\{\underline{n}/b\}$ is applied to a term a: $\{\underline{n}/b\}a$, we internalise this as $a[\underline{1}.\ldots.\underline{n-1}.b.\uparrow^{n+1}]$. This means that all de Bruijn indices except $\underline{n}$ remain unchanged, while $\underline{n}$ is replaced with b. Notice that b is placed at position n of the substitution list, which allows for simultaneous substitutions; for instance, $a[b_1.b_2.\ldots]$ replaces $\underline{1}, \underline{2}, \ldots$ with $b_1, b_2, \ldots$, respectively. Operationally, this calculus applies this kind of substitution decrementing by one the size of the substitution list as well as the de Bruijn indices. When doing that the operator $\uparrow$ is reached, $a[\uparrow^k]$ internalises the k-lifting of the term a. In this calculus only $\underline{1}$ is used and the other de Bruijn indices are coded by lifting $\underline{1}$ as we will explain below. For details see [1].

Definition 3.1 *The $\lambda\sigma$-calculus is defined as the calculus of the rewriting system $\lambda\sigma$ of Table 1 where*
TERMS $a\ ::=\ \underline{1} \mid X \mid (a\ a) \mid \lambda a \mid a[s]$, *where* $X \in \mathcal{X}$ SUBS $s\ ::=\ id \mid \uparrow \mid a.s \mid s \circ s$

Table 1: The $\lambda\sigma$ Rewriting System of the $\lambda\sigma$-calculus with Eta rule

(Beta)	$(\lambda a\ b)$	$\longrightarrow$	$a\,[b \cdot id]$	*(Id)*	$a[id]$	$\longrightarrow$	a
(VarCons)	$\underline{1}\,[a \cdot s]$	$\longrightarrow$	a	*(App)*	$(a\ b)[s]$	$\longrightarrow$	$(a\,[s])\,(b\,[s])$
(Abs)	$(\lambda a)[s]$	$\longrightarrow$	$\lambda a\,[\underline{1} \cdot (s \circ \uparrow)]$	*(Clos)*	$(a\,[s])[t]$	$\longrightarrow$	$a\,[s \circ t]$
(IdL)	$id \circ s$	$\longrightarrow$	s	*(IdR)*	$s \circ id$	$\longrightarrow$	s
(ShiftCons)	$\uparrow \circ (a \cdot s)$	$\longrightarrow$	s	*(Map)*	$(a \cdot s) \circ t$	$\longrightarrow$	$a\,[t] \cdot (s \circ t)$
(Ass)	$(s \circ t) \circ u$	$\longrightarrow$	$s \circ (t \circ u)$	*(VarShift)*	$\underline{1} \cdot \uparrow$	$\longrightarrow$	id
(SCons)	$\underline{1}[s] \cdot (\uparrow \circ s)$	$\longrightarrow$	s	*(Eta)*	$\lambda(a\ \underline{1})$	$\longrightarrow$	b if $a =_\sigma b[\uparrow]$

For every substitution s we define the *iteration of the composition of* s inductively as $s^1 = s$ and $s^{n+1} = s \circ s^n$. We use s^0 to denote id. Note that the only de Bruijn index used is $\underline{1}$, but we can code $\underline{n}$ by $\underline{1}[\uparrow^{n-1}]$.

The equational theory associated with the rewriting system $\lambda\sigma$ defines a congruence denoted $=_{\lambda\sigma}$. The congruence obtained by dropping *Beta* and *Eta* is denoted $=_\sigma$. We use σ-reduction, σ-normal form, etc., with the obvious meaning, in the case when reduction is restricted to the σ-rules.

The rewriting system $\lambda\sigma$ is locally confluent [1], CR on substitution-closed terms (i.e., terms without substitution variables) [39] and not CR on open terms (i.e., terms with term and substitution variables) [13]. The possible forms of a $\lambda\sigma$-term in *$\lambda\sigma$-normal form* were given in [39] by:

1. λa, where a is a normal term;
2. $a_1 \dots a_p . \uparrow^n$, for $a_1, \dots, a_p$ normal terms and $a_p \neq \underline{\mathtt{n}}$
3. $(a\, b_1 \dots b_n)$, where a is either $\underline{\mathtt{1}}$, $\underline{\mathtt{1}}[\uparrow^n]$, X or $X[s]$ for $s \neq id$ a substitution term in normal form.

In the λ-calculus with names or de Bruijn indices, the rule $X\{y/a\} = X$, where y is an element of $\mathcal{V}$ or a de Bruijn index, respectively, is necessary because there is no way to suspend the substitution $\{y/a\}$ until X is instantiated. In the $\lambda\sigma$-calculus, the application of this substitution can be delayed, since the term $X[s]$ does not reduce to X. The fact that the application of a substitution to a meta-variable can be suspended until the meta-variable is instantiated will be used to code the substitution of variables in $\mathcal{X}$ by "$\mathcal{X}$-grafting" and explicit lifting. Consequently a notion of $\mathcal{X}$-substitution in the $\lambda\sigma$-calculus is unnecessary. Observe that the condition $a =_\sigma b[\uparrow]$ of the *Eta* rule is stronger than the condition $a = b^+$ given in Definition 2.8 as $X = X^+$, but there exists no term b such that $X =_\sigma b[\uparrow]$. Note that $\lambda\sigma$-reduction is compatible with first order substitution or grafting and hence $\mathcal{X}$-grafting and $\lambda\sigma$-reduction commute.

3.2 Calculi à la λs and the λs_e-calculus

Calculi à la λs avoid introducing two different sets of entities and insist on remaining close to the syntax of the λ-calculus using de Bruijn indices[1]. Next to λ and application, they introduce substitution σ and updating φ operators. A term containing neither substitution nor updating operators is called a *pure term*. The role of the substitution operator is to internalise the substitution. Essentially, $a\sigma^n b$ makes operational the application of the substitution $\{\underline{\mathtt{n}}/b\}$ to a. This operator is propagated into the body of the abstractors, while all free de Bruijn indices (greater than n) are decreased by one. Once an occurrence of $\underline{\mathtt{n}}$ is found, b is adequately modified (lifted) by the updating operator. The operational effect of $\varphi^i_j b$ is the $(i-1)$-lifting of all de Bruijn indices in b greater than j. For details see [23, 24].

Definition 3.2 *(The λs-calculus) Terms of the λs-calculus are given by:*

$$\Lambda s ::= \mathbb{N} \mid \Lambda s \Lambda s \mid \lambda \Lambda s \mid \Lambda s\, \sigma^i \Lambda s \mid \varphi^i_k \Lambda s \quad \textit{where} \quad i \geq 1\,, \quad k \geq 0\,.$$

The set of rules λs is given in Table 2.

Table 2: The λs-rules

σ-generation	$(\lambda a)\, b$	$\longrightarrow$	$a\, \sigma^1 b$
σ-λ-transition	$(\lambda a)\, \sigma^i b$	$\longrightarrow$	$\lambda(a\, \sigma^{i+1}\, b)$
σ-app-transition	$(a_1\, a_2)\, \sigma^i b$	$\longrightarrow$	$(a_1\, \sigma^i b)\, (a_2\, \sigma^i b)$
σ-destruction	$\mathtt{n}\, \sigma^i b$	$\longrightarrow$	$\begin{cases} \mathtt{n-1} & \text{if } n > i \\ \varphi^i_0\, b & \text{if } n = i \\ \mathtt{n} & \text{if } n < i \end{cases}$
φ-λ-transition	$\varphi^i_k(\lambda a)$	$\longrightarrow$	$\lambda(\varphi^i_{k+1}\, a)$
φ-app-transition	$\varphi^i_k(a_1\, a_2)$	$\longrightarrow$	$(\varphi^i_k\, a_1)\, (\varphi^i_k\, a_2)$
φ-destruction	$\varphi^i_k\, \mathtt{n}$	$\longrightarrow$	$\begin{cases} \mathtt{n+i-1} & \text{if } n > k \\ \mathtt{n} & \text{if } n \leq k \end{cases}$

[1]It can be argued that because we use de Bruijn indices, we remain close to de Bruijn's philosophy rather than to the syntax of the λ-calculus and that instead it is calculi like λx of [10] and $\lambda\chi$ of [30] that remain close to the syntax of the lambda calculus. So, we need to explain here that by staying with the syntax of the λ-calculus we mean that we do not introduce substitutions and other category of operators separately as in $\lambda\sigma$, but that a term for us is either an abstraction term, an application term, a substitution term or an updating term.

The λs-calculus was introduced in [23] with the aim of providing a calculus that preserves strong normalisation and has a confluent extension on open terms [24]. In [23, 25], we establish the properties of these calculi which we list in the following theorem.

Theorem 3.3 *The s-calculus is SN, the λs-calculus is confluent on closed terms and satisfies PSN. Moreover, the λs-calculus simulates β-reduction, is sound and has a confluent extension on open terms.*

We introduce the open terms and the rules that extend λs to obtain the λs_e-calculus.

Definition 3.4 *The set of* open terms, *noted Λs_{op} is given as follows:*

$$\Lambda s_{op} ::= \mathbf{V} \mid \mathbb{N} \mid \Lambda s_{op}\Lambda s_{op} \mid \lambda\Lambda s_{op} \mid \Lambda s_{op}\,\sigma^i\Lambda s_{op} \mid \varphi^i_k\Lambda s_{op} \qquad \textit{where} \quad i \geq 1\,, \quad k \geq 0$$

and where $\mathbf{V}$ *stands for a set of variables, over which X, Y, ... range. We take a, b, c to range over Λs_{op}. Furthermore,* closures, pure terms *and* compatibility *are defined as for Λs.*

Working with open terms one loses confluence as shown by the following counterexample:

$$((\lambda X)Y)\sigma^1\underline{1} \to (X\sigma^1 Y)\sigma^1\underline{1} \qquad ((\lambda X)Y)\sigma^1\underline{1} \to ((\lambda X)\sigma^1\underline{1})(Y\sigma^1\underline{1})$$

and $(X\sigma^1 Y)\sigma^1\underline{1}$ and $((\lambda X)\sigma^1\underline{1})(Y\sigma^1\underline{1})$ have no common reduct. Moreover, the above example shows that even local confluence is lost. But since $((\lambda X)\sigma^1\underline{1})(Y\sigma^1\underline{1}) \twoheadrightarrow (X\sigma^2\underline{1})\sigma^1(Y\sigma^1\underline{1})$, the solution to the problem seems at hand if one has in mind the properties of meta-substitutions and updating functions of the λ-calculus in the Bruijn notation. These properties are equalities which can be given a suitable orientation and the new rules, thus obtained, added to λs yield a rewriting system which happens to be locally confluent. For instance, the rule corresponding to the meta-substitution lemma is the *σ-σ-transition* rule. The addition of this rule solves the critical pair in our counterexample, since now we have $(X\sigma^1 Y)\sigma^1\underline{1} \to (X\sigma^2\underline{1})\sigma^1(Y\sigma^1\underline{1})$.

Definition 3.5 *The set of rules λs_e is obtained by adding the rules given in Table 3 to the set λs. The λs_e-*

Table 3: The new rules of the λs_e-calculus

σ-σ-transition	$(a\,\sigma^i b)\,\sigma^j\,c$	$\longrightarrow$	$(a\,\sigma^{j+1}\,c)\;\sigma^i\,(b\,\sigma^{j-i+1}\,c)$	if	$i \leq j$
σ-φ-transition 1	$(\varphi^i_k\,a)\,\sigma^j\,b$	$\longrightarrow$	$\varphi^{i-1}_k\,a$	if	$k < j < k+i$
σ-φ-transition 2	$(\varphi^i_k\,a)\,\sigma^j\,b$	$\longrightarrow$	$\varphi^i_k(a\,\sigma^{j-i+1}\,b)$	if	$k+i \leq j$
φ-σ-transition	$\varphi^i_k(a\,\sigma^j\,b)$	$\longrightarrow$	$(\varphi^i_{k+1}\,a)\,\sigma^j\,(\varphi^i_{k+1-j}\,b)$	if	$j \leq k+1$
φ-φ-transition 1	$\varphi^i_k\,(\varphi^j_l\,a)$	$\longrightarrow$	$\varphi^j_l\,(\varphi^i_{k+1-j}\,a)$	if	$l+j \leq k$
φ-φ-transition 2	$\varphi^i_k\,(\varphi^j_l\,a)$	$\longrightarrow$	$\varphi^{j+i-1}_l\,a$	if	$l \leq k < l+j$

calculus *is the reduction system $(\Lambda s_{op}, \to_{\lambda s_e})$ where $\to_{\lambda s_e}$ is the least compatible reduction on Λs_{op} generated by the set of rules λs_e. The* calculus of substitutions associated with the λs_e-calculus *is the rewriting system generated by the set of rules $s_e = \lambda s_e - \{\sigma\text{-generation}\}$ and we call it* s_e-calculus.

The equational theory associated to the rewriting system λs_e defines a congruence $=_{\lambda s_e}$. The congruence obtained by dropping *σ-generation* and *Eta* (that will be defined below in Table 4) is denoted by $=_{s_e}$.

Notice that for the $\lambda\sigma$-calculus we need two sorts: TERM and SUBSTITUTION [15]. The set of variables of sort TERM in a term $a \in \mathcal{T}_{\lambda s_e}(\mathcal{X})$ is denoted by $\mathcal{T}var(a)$.

We can describe the operators of the λs_e-calculus over the signature of a first order sorted term algebra $\mathcal{T}_{\lambda s_e}(\mathcal{X})$ built on $\mathcal{X}$, the set of variables of sort TERM and its subsort NAT$\subset$TERM by:

$$\begin{array}{rcrcl}
\underline{\mathtt{n}} & : & & \to & \text{NAT}, \quad \forall n \in \mathbb{N}\setminus\{0\} \\
(_\,_) & : & \text{TERM}\times\text{TERM} & \to & \text{TERM} \\
\sigma^i & : & \text{TERM}\times\text{TERM} & \to & \text{TERM}, \quad \forall i \in \mathbb{N}\setminus\{0\} \\
\lambda_ & : & \text{TERM} & \to & \text{TERM} \\
\varphi^i_k_ & : & \text{TERM} & \to & \text{TERM}, \quad \forall i \in \mathbb{N}, k \in \mathbb{N}\setminus\{0\}
\end{array}$$

In [24] we proved the following:

Theorem 3.6 (WN and CR of s_e) *The s_e-calculus is weakly normalising and confluent.*

Lemma 3.7 (Simulation of β-reduction) *Let $a, b \in \Lambda$, if $a \to_\beta b$ then $a \twoheadrightarrow_{\lambda s_e} b$.*

Theorem 3.8 (CR of λs_e) *The λs_e-calculus is confluent on open terms.*

Theorem 3.9 (Soundness) *Let $a, b \in \Lambda$, if $a \twoheadrightarrow_{\lambda s_e} b$ then $a \twoheadrightarrow_\beta b$.*

In [3] we proved that:

Proposition 3.10 *$\mathcal{X}$-grafting and λs_e-reduction commute.*

This calculus was originally introduced without the *Eta* rule that was added in [3] to deal with higher order unification problems as originally done in [15] for the $\lambda\sigma$-calculus.

Table 4: The eta rule of the λs_e-calculus

(Eta)	$\lambda(a\ \underline{1})$	$\longrightarrow$	b	if	$a =_{s_e} \varphi_0^2 b$

The characterization of the λs_e-normal forms was given in [24, 3] by: a term $a \in \Lambda s_e$ is a λs_e-nf if and only if one of the following holds:

1. $a \in \mathcal{X} \cup \mathbb{N}$;
2. $a = bc$ with b, c in λs_e-nf and b not an abstraction λd;
3. $a = \lambda b$, where b is a λs_e-nf excluding applications of the form $(c\ \underline{1})$ where $\varphi_0^2 d =_{s_e} c$ for some d;
4. $a = b\sigma^j c$, where b, c in λs_e-nf and b is of the form: X or $d\sigma^i e$, with $j < i$ or $\varphi_k^i d$, with $j \leq k$;
5. $a = \varphi_k^i b$, where b is a λs_e-nf of the form: X or $c\sigma^j d$, with $j > k+1$ or $\varphi_l^j c$, with $k < l$;

3.3 The Suspension Calculus

The suspension calculus [37, 34] deals with λ-terms as computational mechanisms. This was motivated by implementational questions related to λProlog, a logic programming language that uses typed λ-terms as data structures [36]. The suspension calculus works with three different types of entities:

SUSPENDED TERMS	M, N	::=	$C \mid \underline{n} \mid \lambda M \mid (M\ N) \mid [\![M, i, j, e_1]\!]$
ENVIRONMENTS	e_1, e_2	::=	$nil \mid et :: e_1 \mid \{\!\{e_1, i, j, e_2\}\!\}$
ENVIRONMENT TERMS	et	::=	$@i \mid (M, i) \mid \langle\!\langle et, i, j, e_1\rangle\!\rangle$

where C denotes any constant and i, j are non negative natural numbers.

As constants and de Bruijn indices are suspended terms, the suspension calculus has open terms. Rather than performing adjustments at each stage, the suspension calculus notation performs the adjustments into a substitution term only at the final substitution stage. Intuitively, a suspended term of the form $[\![M, i, j, e_1]\!]$ means that the first i variables of the λ-term M must be substituted in a way determined by the environment e_1 and its remaining bound variables must be renumbered according to the fact that M used to appear within i abstractions but now appears within j of them.

The suspension calculus owns a *generation* rule β_s, that initiates the simulation of a β-reduction (as for the $\lambda\sigma$ and the λs_e, respectively, the *Beta* and the *σ-generation* rules do) and two sets of rules for handling the suspended terms. The first set, the r rules, for reading suspensions and the second set, the m rules, for merging suspensions are given in Table 5.

As in [37] we denote by $\triangleright_{rm}$ the reduction relation defined by the r- and m-rules in Table 5. The associated substitution calculus, denoted by SUSP, is the one given by the congruence $=_{rm}$.

Table 5: Rewriting rules of the suspension calculus

(β_s)	$((\lambda t_1\ t_2)$	$\longrightarrow [\![t_1, 1, 0, (t_2, 0) :: nil]\!]$
(r_1)	$[\![c, ol, nl, e]\!]$	$\longrightarrow c,$ where c is a constant
(r_2)	$[\![\underline{\mathtt{i}}, 0, nl, nil]\!]$	$\longrightarrow \underline{\mathtt{i+nl}}$
(r_3)	$[\![\underline{\mathtt{1}}, ol, nl, @l :: e]\!]$	$\longrightarrow \underline{\mathtt{nl\text{-}l}}$
(r_4)	$[\![\underline{\mathtt{1}}, ol, nl, (t, l) :: e]\!]$	$\longrightarrow [\![t, 0, (nl\text{-}l), nil]\!]$
(r_5)	$[\![\underline{\mathtt{i}}, ol, nl, et :: e]\!]$	$\longrightarrow [\![\underline{\mathtt{i\text{-}1}}, (ol\text{-}1), nl, e]\!],$ for $i > 1$
(r_6)	$[\![(t_1\ t_2), ol, nl, e]\!]$	$\longrightarrow ([\![t_1, ol, nl, e]\!]\ [\![t_2, ol, nl, e]\!])$
(r_7)	$[\![\lambda t, ol, nl, e]\!]$	$\longrightarrow \lambda [\![t, (ol+1), (nl+1), @nl :: e]\!]$
(m_1)	$[\![[\![t, ol_1, nl_1, e_1]\!], ol_2, nl_2, e_2]\!]$	$\longrightarrow [\![t, ol', nl', \{\!\{e_1, nl_1, ol_2, e_2\}\!\}]\!],$ where $ol' = ol_1 + (ol_2 \dot{-} nl_1)$ and $nl' = nl_2 + (nl_1 \dot{-} ol_2)$
(m_2)	$\{\!\{nil, nl, 0, nil\}\!\}$	$\longrightarrow nil$
(m_3)	$\{\!\{nil, nl, ol, et :: e\}\!\}$	$\longrightarrow \{\!\{nil, (nl\text{-}1), (ol\text{-}1), e\}\!\},$ for $nl, ol \geq 1$
(m_4)	$\{\!\{nil, 0, ol, e\}\!\}$	$\longrightarrow e$
(m_5)	$\{\!\{et :: e_1, nl, ol, e_2\}\!\}$	$\longrightarrow \langle\!\langle et, nl, ol, e_2\rangle\!\rangle :: \{\!\{e_1, nl, ol, e_2\}\!\}$
(m_6)	$\langle\!\langle et, nl, 0, nil\rangle\!\rangle$	$\longrightarrow et$
(m_7)	$\langle\!\langle @m, nl, ol, @l :: e\rangle\!\rangle$	$\longrightarrow @(l + (nl \dot{-} ol)),$ for $nl = m+1$
(m_8)	$\langle\!\langle @m, nl, ol, (t, l) :: e\rangle\!\rangle$	$\longrightarrow (t, (l + (nl \dot{-} ol))),$ for $nl = m+1$
(m_9)	$\langle\!\langle (t, nl), nl, ol, et :: e\rangle\!\rangle$	$\longrightarrow ([\![t, ol, l', et :: e]\!], m),$ where $l' = ind(et)$ and $m = l' + (nl \dot{-} ol)$
(m_{10})	$\langle\!\langle et, nl, ol, et' :: e\rangle\!\rangle$	$\longrightarrow \langle\!\langle et, (nl\text{-}1), (ol\text{-}1), e\rangle\!\rangle,$ for $nl \neq ind(et)$

Definition 3.11 ([37]) *The* length $len(e)$ *of an environment* e *is given by:*
$len(nil) := 0$; $len(et :: e') := len(e') + 1$ *and*
$len(\{\!\{e_1, i, j, e_2\}\!\}) := len(e_1) + (len(e_2) \dot{-} i)$.
The index $ind(et)$ *of an environment term* et*, and the* l-th index $ind_l(e)$ *of environment* e *and natural number* l*, are simultaneously defined by induction on the structure of expressions:*
$ind(@m) = m + 1$; $\quad ind((t', m)) = m$;

$$ind(\langle\!\langle et', j, k, e\rangle\!\rangle) = \begin{cases} ind_m(e) + (j \dot{-} k) & \text{if } len(e) > j \dot{-} ind(et') = m \\ ind(et') & \text{otherwise} \end{cases}$$

$ind_l(nil) = 0$; $\quad ind_0(et :: e') = ind(et)$ *and* $ind_{l+1}(et :: e') = ind_l(e')$

$$ind_l(\{\!\{e_1, j, k, e_2\}\!\}) = \begin{cases} ind_m(e_2) + (j \dot{-} k) & \text{if } l < len(e_1) \text{ and} \\ & len(e_2) > m = j \dot{-} ind_l(e_1) \\ ind_l(e_1) & \text{if } l < len(e_1) \text{ and} \\ & len(e_2) \leq m = j \dot{-} ind_l(e_1) \\ ind_{l-l_1+j}(e_2) & \text{if } l \geq l_1 = len(e_1) \end{cases}$$

The index *of an environment* e*, denoted as* $ind(e)$*, is* $ind_0(e)$.

Definition 3.12 ([37]) *An expression of the suspension calculus is said to be* well-formed *if the following conditions hold over all its subexpressions* s*:*
- *if* s *is* $[\![t, ol, nl, e]\!]$ *then* $len(e) = ol$ *and* $ind(e) \leq nl$
- *if* s *is* $et :: e$ *then* $ind(e) \leq ind(et)$
- *if* s *is* $\langle\!\langle et, j, k, e\rangle\!\rangle$ *then* $len(e) = k$ *and* $ind(et) \leq j$
- *if* s *is* $\{\!\{e_1, j, k, e_2\}\!\}$ *then* $len(e_2) = k$ *and* $ind(e_1) \leq j$.

In the sequel, we only deal with well-formed expressions of the suspension calculus.

The suspension calculus simulates β-reduction and its associated substitution calculus SUSP is CR (over closed and open terms) and SN [37]. In [34] Nadathur conjectures that the suspension calculus preserves strong normalization too but there is still no proof of this conjecture. The following lemma characterizes the $\triangleright_{rm}$-normal forms.

Lemma 3.13 ([37]) *A well-formed expression of the suspension calculus x is in its $\triangleright_{rm}$-nf if and only if one of the following affirmations holds:*
1) x is a pure λ-term in de Bruijn notation;
2) x is an environment term of the form $@l$ or (t, l), where t is a term in its $\triangleright_{rm}$-nf;
3) x is the environment nil or $et :: e$ for et and e resp. an environment term and an environment in $\triangleright_{rm}$-nf.

3.4 The suspension calculus enlarged with η-reduction: the $\lambda_{\mathbf{susp}}$-calculus

The suspension calculus was initially formulated without η-reduction. Here we introduce an adequate *Eta* rule that enlarges the suspension calculus preserving correctness, confluence, and termination of the associated substitution calculus. The suspension calculus enlarged with this *Eta* rule is denoted by λ_{SUSP} and we continue to call its associated substitution calculus SUSP. The *Eta* rule is formulated in Table 6. Intuitively *Eta* may

Table 6: The eta rule of the suspension calculus

(Eta)	$(\lambda\ (t_1\ \ \underline{1}))$	$\longrightarrow$	t_2	if	$t_1 =_{rm} [\![t_2, 0, 1, nil]\!]$

be interpreted as: when it is possible to apply the η-reduction rule to the redex $\lambda(t_1\ \underline{1})$ we obtain a term t_2 that has the same structure as t_1 with all its free de Bruijn indices decremented by one. This is possible whenever there are no free occurrences of the variable corresponding to $\underline{1}$ in t_1. Proposition 3.16 proves the correctness of *Eta* according to this interpretation.

Remark 3.14 *The reader may wonder whether this is the best formulation of Eta in the suspension calculus. Indeed, the reader may ask this question also in connection with the formulation of Eta in both the $\lambda\sigma$- and λs_e-calculi. Initially, [15] intended to use $\lambda(a[\uparrow]1) \to a$ as a formulation of Eta in the $\lambda\sigma$-calculus. However, this formulation would lead to an infinite set of critical pairs. For this reason, [15] took the formulation given in Table 1. The same reason led [3] to use a formulation of Eta in the λs_e-calculus which uses s_e convertibility (see table 4). And indeed for the suspension calculus, we also get an infinite set of critical pairs if we use $(\lambda\ ([\![t_1, 0, 1, nil]\!]\ \ \underline{1})) \longrightarrow t_1$.*

We follow [11] and [2] for $\lambda\sigma$ and λs_e respectively, and implement the *Eta* rule of the λ_{SUSP}-calculus by introducing a dummy symbol $\Diamond$, by:

$$\lambda(M\ \underline{1}) \longrightarrow_{Eta} N \qquad \text{if } N = \triangleright_{rm}\text{-nf}([\![M, 1, 0, (\Diamond, 0) :: nil]\!]) \text{ and } \Diamond \text{ does not occur in } N.$$

The correctness of this implementation is explained because an η-reduction $\lambda(M\ \underline{1}) \to_\eta N$ gives us a term N, which is obtained from M by decrementing by one all free occurrences of de Bruijn indices, as previously mentioned, and which corresponds exactly to the $\triangleright_{rm}$-normalization of the term $((\lambda M)\ \Diamond) \to_{\beta_s} [\![M, 1, 0, (\Diamond, 0) :: nil]\!]$, whenever $\Diamond$ does not appear in this normalized term.

Lemma 3.15 *Let A be a well-formed term of the suspension calculus. Then the* SUSP*-normalization of the term $[\![A, k, k+1, @k :: @k-1 :: \ldots :: @1 :: nil]\!]$ gives a term obtained from A by incrementing by one all its de Bruijn free indices greater than k and preserving unaltered all other de Bruijn indices.*

PROOF. By induction on the structure of A. The constant case is trivial.

- $\boldsymbol{A = \underline{n}}$. If $n > k$: $[\![\underline{n}, k, k+1, @k :: \ldots :: @1 :: nil]\!] \to^k_{r_5} [\![\underline{n-k}, 0, k+1, nil]\!] \to_{r_2} \underline{n+1}$.
  If $n \leq k$: $[\![\underline{n}, k, k+1, @k :: \ldots :: @1 :: nil]\!] \to^{n-1}_{r_5} [\![\underline{1}, k-n+1, k+1, @k-n+1 :: \ldots :: @1 :: nil]\!] \to_{r_3} \underline{n}$;
- $\boldsymbol{A = (B\ C)}$. we apply r_6 and induction hypothesis for B and C;

- $\boldsymbol{A = (\lambda B)}$. Since B is bounded by an abstractor, only its free variables greater than $k+1$ should be incremented by one, the other variables remain unchanged. Since $[\![(\lambda B), k, k+1, @k :: \ldots :: @1 :: nil]\!] \rightarrow_{r_7} \lambda[\![B, k+1, k+2, @k+1 :: \ldots :: @1 :: nil]\!]$, by applying induction hypothesis over the previous term we obtain the desired result.

- $\boldsymbol{A = [\![t, ol, nl, e]\!]}$. Without loss of generality A may be $\triangleright_{rm}$-normalized and by Lemma 3.13 the obtained term is of one of the forms analysed in the previous cases. □

Proposition 3.16 (Soundness of the *Eta* **rule)** *Every application of the Eta rule of λ_{SUSP} to the redex $\lambda(t_1\ \underline{1})$ gives effectively the term t_2 obtained from t_1 by decrementing all its de Bruijn free indices by one.*

PROOF. The proof is by induction over the structure of t_2 considering the premise $t_1 =_{rm} [\![t_2, 0, 1, nil]\!]$. The effect of normalizing $[\![t_2, 0, 1, nil]\!]$ is to increment by one all de Bruijn free indices occurring at t_2:

- $\boldsymbol{t_2 = \underline{\mathtt{n}}}$. $[\![\underline{\mathtt{n}}, 0, 1, nil]\!] \rightarrow_{r_2} \underline{\mathtt{n+1}} =_{rm} t_1$.

- $\boldsymbol{t_2 = (A\ B)}$. Without loss of generality we can assume that both A and B are in $\triangleright_{rm}$-nf. Observe that $[\![(A\ B), 0, 1, nil]\!] \rightarrow_{r_6} [\![A, 0, 1, nil]\!]\ [\![B, 0, 1, nil]\!]$. Now, by induction hypothesis over A and B, we have that the normalization of the suspended terms $[\![A, 0, 1, nil]\!]$ and $[\![B, 0, 1, nil]\!]$ have the desired effect and consequently the same happens with the normalization of the suspended term $[\![(A\ B), 0, 1, nil]\!]$.

- $\boldsymbol{t_2 = (\lambda A)}$. As before, assume A is in $\triangleright_{rm}$-nf. Note that $[\![(\lambda A), 0, 1, nil]\!] \rightarrow_{r_7} (\lambda[\![A, 1, 2, @1 :: nil]\!])$. By applying Lemma 3.15 to the term $[\![A, 1, 2, @1 :: nil]\!]$ we conclude that all free occurrences of de Bruijn indices greater than 1 at A are incremented by one while the other indices are unchanged.

- $\boldsymbol{t_2 = [\![t, i, j, e]\!]}$. If t is in $\triangleright_{rm}$-nf then $[\![t, i, j, e]\!] \triangleright^*_{rm} t'$, where t' is a pure λ-term in de Bruijn notation by Lemma 3.13. Hence, the analysis given in the previous three cases applies here too. □

Noetherianity of SUSP plus the *Eta* rule enables us to apply the Newman diamond lemma and the Knuth-Bendix critical pair criterion for proving its confluence.

Lemma 3.17 (susp+ *Eta* **is SN)** *The rewriting system associated to* SUSP *and the Eta rule is noetherian.*

PROOF. (Sketch) This is proved by showing that the *Eta* rule is also compatible with the well-founded partial ordering $\prec$ that is defined and proved compatible with $\triangleright_{rm}$ in [37]. □

A **simple environment** is an environment without subexpressions of the form $\{\!\{\_, \_, \_, \_\}\!\}$ or $\langle\!\langle \_, \_, \_, \_ \rangle\!\rangle$.

Lemma 3.18 ([37]) *Let e_1 be a simple environment and suppose that nl and ol are naturals such that $(nl - ind(e_1)) \geq ol$. Then $\{\!\{e_1, nl, ol, e_2\}\!\} \triangleright^*_{rm} e_1$.*

Lemma 3.19 (Local-confluence of susp+ *Eta***)** *The rewriting system of the substitution calculus* SUSP *plus the Eta rule is locally-confluent.*

PROOF. The rewrite relation $\triangleright_{rm}$, i.e., SUSP, was shown in [37] to be (locally) confluent. Thus for proving that the associated rewriting system enlarged with the *Eta* rule is locally-confluent, it is enough to show that all additional critical pairs built by overlapping between the *Eta* rule and the other rules of SUSP are joinable. Note that no critical pairs are generated from *Eta* and itself. Moreover, there is a unique overlapping between the set of rules in Table 5 (minus (β_s)) and *Eta*: namely, the one between *Eta* and (r_7).

This critical pair is $\langle [\![t_2, ol, nl, e]\!], \lambda[\![(t_1\ \underline{1}), ol+1, nl+1, @nl :: e]\!] \rangle$, where $t_1 =_{rm} [\![t_2, 0, 1, nil]\!]$. After applying the rules r_6 and r_3 the right-side term of this critical pair reduces to $\lambda([\![t_1, ol+1, nl+1, @nl :: e]\!]\ \underline{1})$.

We prove by analyzing the structure of t_1 that this critical pair is joinable. We take t_1 and t_2 as $\triangleright_{rm}$-nf's.

- $\boldsymbol{t_1 = \underline{\mathtt{n}}}$. For making possible the *Eta* application, we need that $n > 1$. According to the length of the environment $@nl :: e$ (i.e., $ol + 1$) we have the following cases:

  - $ol + 1 < n$. On one side, $\lambda([\![\underline{\mathtt{n}}, ol+1, nl+1, @nl :: e]\!]\ \underline{1}) \rightarrow^{ol+1}_{r_5} \lambda([\![\underline{\mathtt{n-ol-1}}, 0, nl+1, nil]\!]\ \underline{1}) \rightarrow_{r_2} \lambda(\underline{\mathtt{n-ol+nl}}\ \underline{1}) \rightarrow_{Eta} \underline{\mathtt{n-ol+nl-1}}$. On the other side, $t_1 =_{rm} [\![t_2, 0, 1, nil]\!]$, hence $t_2 = \underline{\mathtt{n-1}}$ and we have $[\![\underline{\mathtt{n-1}}, ol, nl, e]\!] \rightarrow^{ol}_{r_5} [\![\underline{\mathtt{n-1-ol}}, 0, nl, nil]\!] \rightarrow_{r_2} \underline{\mathtt{n-ol+nl-1}}$.

– $ol+1 \geq n$. On one side, $\lambda([\![\underline{\mathtt{n}}, ol+1, nl+1, @nl :: e]\!]\ \underline{1}) \rightarrow_{r_5}^{n-1} \lambda([\![\underline{1}, ol-n+2, nl+1, e_1 :: e']\!]\ \underline{1})$ and the subsequent derivation depends on the structure of e_1: when $e_1 = @l$ we apply r_3 obtaining $\lambda(\underline{\mathtt{nl+1-l}}\ \underline{1}) \rightarrow_{Eta} \underline{\mathtt{nl-l}}$ and on the other side, $[\![\underline{\mathtt{n-1}}, ol, nl, e]\!] \rightarrow_{r_5}^{n-2} [\![\underline{1}, ol-n+2, nl, @l :: e']\!] \rightarrow_{r_3} \underline{\mathtt{nl-l}}$; when $e_1 = (t, l)$, where without loss of generality t is supposed to be in $\rhd_{rm}$-nf, we have $\lambda([\![\underline{1}, ol-n+2, nl+1, (t,l) :: e']\!]\ \underline{1}) \rightarrow_{r_4} \lambda([\![t, 0, nl-l+1, nil]\!]\ \underline{1}) \rightarrow_{Eta}$
$\rhd_{rm}\text{-}nf([\![[\![t, 0, nl{+}1{-}l, nil]\!], 1, 0, (\Diamond, 0) :: nil]\!]) \rightarrow_{m_1}$
$\rhd_{rm}\text{-}nf([\![t, 0, nl{-}l, \{\!\{nil, nl{+}1{-}l, 1, (\Diamond, 0) :: nil\}\!\}]\!]) \rightarrow_{m_3}$
$\rhd_{rm}\text{-}nf([\![t, 0, nl-l, \{\!\{nil, nl-l, 0, nil\}\!\}]\!]) \rightarrow_{m_2} \rhd_{rm}\text{-}nf([\![t, 0, nl-l, nil]\!])$
and on the other side, $[\![\underline{1}, ol-n+2, nl, (t,l) :: e']\!] \rightarrow_{r_4} [\![t, 0, nl-l, nil]\!]$.

Since $\rhd_{rm}\text{-}nf([\![t, 0, nl-l, nil]\!])$ and $[\![t, 0, nl-l, nil]\!]$ are joinable we obtain the confluence.

- $\boldsymbol{t_1 = (A\ B)}$. Since the sole rule of the λ_{SUSP} that truly "applies" applications is the β_s, we can separately consider *Eta*-reductions for A and B and then apply the induction hypothesis. That is, suppose inductively that $\lambda([\![A, ol+1, nl+1, @nl :: e]\!]\ \underline{1}) \rightarrow_{Eta} A''$ and $[\![A', ol, nl, e]\!]$, where $[\![A', 0, 1, nil]\!] =_{rm} A$ as well as $\lambda([\![B, ol+1, nl+1, @nl :: e]\!]\ \underline{1}) \rightarrow_{Eta} B''$ and $[\![B', ol, nl, e]\!]$, where $[\![B', 0, 1, nil]\!] =_{rm} B$ are joinable. Then since $\lambda([\![(A\ B), ol+1, nl+1, @nl :: e]\!]\ \underline{1}) \rightarrow_{r_6}$
$\lambda(([\![A, ol+1, nl+1, @nl :: e]\!]\ [\![B, ol+1, nl+1, @nl :: e]\!])\ \underline{1}) \rightarrow_{Eta} (A''\ B'')$ and $[\![(A'\ B'), ol, nl, e]\!] \rightarrow_{r_6}$ $([\![A', ol, nl, e]\!]\ [\![B', ol, nl, e]\!])$ we can conclude the confluence.

- $\boldsymbol{t_1 = (\lambda A)}$. By the *Eta* rule implementation, it is enough to show the joinability of the *Eta*-reduction of the term $\lambda([\![(\lambda A), ol+1, nl+1, @nl :: e]\!]\ \underline{1})$ that is $\rhd_{\text{SUSP}}$-nf$([\![[\![(\lambda A), ol+1, nl+1, @nl :: e]\!], 1, 0, (\Diamond, 0) :: nil]\!])$ and the term $[\![\ \rhd_{\text{SUSP}}\text{-nf}([\![(\lambda A), 1, 0, (\Diamond, 0) :: nil]\!]), ol, nl, e]\!]$.

  On the one side, $[\![\ \rhd_{\text{SUSP}}\text{-nf}([\![(\lambda A), 1, 0, (\Diamond, 0) :: nil]\!]), ol, nl, e]\!] \rhd_{rm}^*$
  $\rhd_{\text{SUSP}}\text{-nf}([\![[\![(\lambda A), 1, 0, (\Diamond, 0) :: nil]\!], ol, nl, e]\!]) \rightarrow_{r_7, r_7}$
  $\rhd_{\text{SUSP}}\text{-nf}((\lambda [\![[\![A, 2, 1, @0 :: (\Diamond, 0) :: nil]\!], ol+1, nl+1, @nl :: e]\!])) \rhd_{rm}^*$
  $(\lambda \rhd_{\text{SUSP}}\text{-nf}([\![[\![A, 2, 1, @0 :: (\Diamond, 0) :: nil]\!], ol+1, nl+1, @nl :: e]\!])) \rightarrow_{m_1}$
  $(\lambda \rhd_{\text{SUSP}}\text{-nf}([\![A, ol+2, nl+1, \{\!\{@0 :: (\Diamond, 0) :: nil, 1, ol+1, @nl :: e\}\!\}]\!]))$
  and we have that $\{\!\{@0 :: (\Diamond, 0) :: nil, 1, ol+1, @nl :: e\}\!\} \rightarrow_{m_5, m_5}$
  $\langle\!\langle @0, 1, ol+1, @nl :: e \rangle\!\rangle :: \langle\!\langle (\Diamond, 0), 1, ol+1, @nl :: e \rangle\!\rangle :: \{\!\{nil, 1, ol+1, @nl :: e\}\!\} \rightarrow_{m_7}$
  $@nl :: \langle\!\langle (\Diamond, 0), 1, ol+1, @nl :: e \rangle\!\rangle :: \{\!\{nil, 1, ol+1, @nl :: e\}\!\} \rightarrow_{m_{10}}$
  $@nl :: \langle\!\langle (\Diamond, 0), 0, ol, e \rangle\!\rangle :: \{\!\{nil, 1, ol+1, @nl :: e\}\!\} \rightarrow_{m_3, m_4}$
  $@nl :: \langle\!\langle (\Diamond, 0), 0, ol, e \rangle\!\rangle :: e$. Then we obtain the term
  $(\lambda \rhd_{\text{SUSP}}\text{-nf}([\![A, ol+2, nl+1, @nl :: \langle\!\langle (\Diamond, 0), 0, ol, e \rangle\!\rangle :: e]\!]))$. On the other side,
  $\rhd_{\text{SUSP}}\text{-nf}([\![[\![(\lambda A), ol+1, nl+1, @nl :: e]\!], 1, 0, (\Diamond, 0) :: nil]\!]) \rightarrow_{r_7, r_7}$
  $\rhd_{\text{SUSP}}\text{-nf}((\lambda [\![[\![A, ol+2, nl+2, @nl+1 :: @nl :: e]\!], 2, 1, @0 :: (\Diamond, 0) :: nil]\!])) \rhd_{rm}^*$
  $(\lambda \rhd_{\text{SUSP}}\text{-nf}([\![[\![A, ol+2, nl+2, @nl+1 :: @nl :: e]\!], 2, 1, @0 :: (\Diamond, 0) :: nil]\!])) \rightarrow_{m_1}$
  $(\lambda \rhd_{rm}\text{-nf}[\![A, ol+2, nl+1, \{\!\{@nl+1 :: @nl :: e, nl+2, 2, @0 :: (\Diamond, 0) :: nil]\!])$ and we have that $\{\!\{@nl+1 :: @nl :: e, nl+2, 2, @0 :: (\Diamond, 0) :: nil\}\!\} \rightarrow_{m_5, m_5}$
  $\langle\!\langle @nl+1, nl+2, 2, @0 :: (\Diamond, 0) :: nil \rangle\!\rangle :: \langle\!\langle @nl, nl+2, 2, @0 :: (\Diamond, 0) :: nil \rangle\!\rangle :: \{\!\{e, nl+2, 2, @0 :: (\Diamond, 0) :: nil\}\!\}$
  $\rightarrow_{m_7} @nl :: \langle\!\langle @nl, nl+2, 2, @0 :: (\Diamond, 0) :: nil \rangle\!\rangle :: \{\!\{e, nl+2, 2, @0 :: (\Diamond, 0) :: nil\}\!\} \rhd_{rm}^*$ (By Lemma 3.18, since we are working with well-formed terms and then) $ind(e) \leq nl$)
  $@nl :: \langle\!\langle @nl, nl+2, 2, @0 :: (\Diamond, 0) :: nil \rangle\!\rangle :: e \rightarrow_{m_{10}}$
  $@nl :: \langle\!\langle @nl, nl+1, 1, (\Diamond, 0) :: nil \rangle\!\rangle :: e \rightarrow_{m_8} @nl :: (\Diamond, nl) :: e$.

  Then we obtain the term $(\lambda \rhd_{\text{SUSP}}\text{-nf}([\![A, ol+2, nl+1, @nl :: (\Diamond, nl) :: e]\!]))$.

  The sole difference of the obtained suspended terms is the second environment term of their environments, that is $\langle\!\langle (\Diamond, 0), 0, ol, e \rangle\!\rangle$ and $(\Diamond, nl)$. But since the *Eta* rule applies, when propagating the substitution between these suspended terms, the dummy symbol and hence these second environment terms should disappear. Now we can conclude that these terms are joinable. □

Finally, since the rewriting system associated to SUSP enlarged with the *Eta* rule is locally-confluent and noetherian, we can apply the Newman diamond lemma for concluding its confluence.

Theorem 3.20 (Confluence of susp+ *Eta*) *The calculus* SUSP *jointly with the Eta rule, is confluent.*

4 Comparing the adequacy of the calculi

According to the criterion of adequacy introduced in [26] we prove that the $\lambda\sigma$ and the λ_{SUSP} as well as the $\lambda\sigma$ and the λs_e are non comparable. Additionally, we prove that the λs_e is more adequate in the simulation of one step β-reduction than the λ_{SUSP}.

Let $a, b \in \Lambda$ such that $a \to_\beta b$. A *simulation* of this β-reduction in $\lambda\xi$, for $\xi \in \{\sigma, s_e, \text{SUSP}\}$ is a $\lambda\xi$-derivation $a \to_r c \to_\xi^* \xi(c) = b$, where r is the rule starting β (*beta* for $\lambda\sigma$, σ-*generation* for λs_e, β_s for λ_{SUSP}) applied to the same redex as the redex in $a \to_\beta b$. The criterion of adequacy is defined as follow:

Definition 4.1 ([26]) (Adequacy) *Let $\xi_1, \xi_2 \in \{\sigma, s_e, \text{SUSP}\}$. The $\lambda\xi_1$-calculus is* more adequate *(in simulating one step β-reduction) than the $\lambda\xi_2$-calculus, denoted $\lambda\xi_1 \prec \lambda\xi_2$, if:*

- *for every β-reduction $a \to_\beta b$ and every $\lambda\xi_2$-simulation $a \to_{\lambda\xi_2}^n b$ there exists a $\lambda\xi_1$-simulation $a \to_{\lambda\xi_1}^m b$ such that $m \leq n$;*
- *there exists a β-reduction $a \to_\beta b$ and a $\lambda\xi_1$-simulation $a \to_{\lambda\xi_1}^m b$ such that for every $\lambda\xi_2$-simulation $a \to_{\lambda\xi_2}^n b$ we have $m < n$.*

If neither $\lambda\xi_1 \prec \lambda\xi_2$ nor $\lambda\xi_2 \prec \lambda\xi_1$, then we say that $\lambda\xi_1$ and $\lambda\xi_2$ are non comparable.

The counterexamples proving that $\lambda\sigma$ and λs are non comparable presented in [26] apply also to the incomparability of $\lambda\sigma$ and λs_e since λs_e is an extension of λs for open terms.

Proposition 4.2 *The $\lambda\sigma$- and the λs_e-calculi are non comparable.*

Lemma 4.3 *Every $\lambda\sigma$-derivation of $((\lambda\lambda\underline{2})\ \underline{1})$ to its $\lambda\sigma$-nf has length greater than or equal to* 6.

Proof. In fact, all possible derivations are of one of the following forms.

- $(\lambda\lambda\underline{1}[\uparrow])\ \underline{1} \to_{Beta} (\lambda\underline{1}[\uparrow])[\underline{1}.id] \to_{Abs} \lambda\underline{1}[\uparrow][\underline{1}.((\underline{1}.id)\circ\uparrow)] \to_{Clos}$ $\lambda\underline{1}[\uparrow\circ(\underline{1}.((\underline{1}.id)\circ\uparrow))] \to_{ShiftCons} \lambda\underline{1}[(\underline{1}.id)\circ\uparrow] \to_{Map} \lambda\underline{1}[\underline{1}[\uparrow].(id\circ\uparrow)] \to_{VarCons} \lambda\underline{1}[\uparrow] = \lambda\underline{2}$;
- $(\lambda\lambda\underline{1}[\uparrow])\ \underline{1} \to_{Beta} (\lambda\underline{1}[\uparrow])[\underline{1}.id] \to_{Abs} \lambda\underline{1}[\uparrow][\underline{1}.((\underline{1}.id)\circ\uparrow)] \to_{Clos} \lambda\underline{1}[\uparrow\circ(\underline{1}.((\underline{1}.id)\circ\uparrow))] \to_{ShiftCons}$ $\lambda\underline{1}[(\underline{1}.id)\circ\uparrow] \to_{Map} \lambda\underline{1}[\underline{1}[\uparrow].(id\circ\uparrow)] \to_{IdL} \lambda\underline{1}[\underline{1}[\uparrow].\uparrow] \to_{VarCons} \lambda\underline{1}[\uparrow] = \lambda\underline{2}$;
- $(\lambda\lambda\underline{1}[\uparrow])\ \underline{1} \to_{Beta} (\lambda\underline{1}[\uparrow])[\underline{1}.id] \to_{Abs} \lambda\underline{1}[\uparrow][\underline{1}.((\underline{1}.id)\circ\uparrow)] \to_{Clos} \lambda\underline{1}[\uparrow\circ(\underline{1}.((\underline{1}.id)\circ\uparrow))] \to_{Map}$ $\lambda\underline{1}[\uparrow\circ(\underline{1}.(\underline{1}[\uparrow].(id\circ\uparrow)))] \to_{ShiftCons} \lambda\underline{1}[\underline{1}[\uparrow].(id\circ\uparrow)] \to_{VarCons} \lambda\underline{1}[\uparrow] = \lambda\underline{2}$;
- $(\lambda\lambda\underline{1}[\uparrow])\ \underline{1} \to_{Beta} (\lambda\underline{1}[\uparrow])[\underline{1}.id] \to_{Abs} \lambda\underline{1}[\uparrow][\underline{1}.((\underline{1}.id)\circ\uparrow)] \to_{Clos} \lambda\underline{1}[\uparrow\circ(\underline{1}.((\underline{1}.id)\circ\uparrow))] \to_{Map}$ $\lambda\underline{1}[\uparrow\circ(\underline{1}.(\underline{1}[\uparrow].(id\circ\uparrow)))] \to_{ShiftCons} \lambda\underline{1}[\underline{1}[\uparrow].(id\circ\uparrow)] \to_{IdL} \lambda\underline{1}[\underline{1}[\uparrow].\uparrow] \to_{VarCons} \lambda\underline{1}[\uparrow] = \lambda\underline{2}$;
- $(\lambda\lambda\underline{1}[\uparrow])\ \underline{1} \to_{Beta} (\lambda\underline{1}[\uparrow])[\underline{1}.id] \to_{Abs} \lambda\underline{1}[\uparrow][\underline{1}.((\underline{1}.id)\circ\uparrow)] \to_{Map} \lambda\underline{1}[\uparrow][\underline{1}.(\underline{1}[\uparrow].(id\circ\uparrow))] \to_{Clos}$ $\lambda\underline{1}[\uparrow\circ(\underline{1}.(\underline{1}[\uparrow].(id\circ\uparrow)))] \to_{ShiftCons} \lambda\underline{1}[\underline{1}[\uparrow].(id\circ\uparrow)] \to_{VarCons} \lambda\underline{1}[\uparrow] = \lambda\underline{2}$;
- $(\lambda\lambda\underline{1}[\uparrow])\ \underline{1} \to_{Beta} (\lambda\underline{1}[\uparrow])[\underline{1}.id] \to_{Abs} \lambda\underline{1}[\uparrow][\underline{1}.((\underline{1}.id)\circ\uparrow)] \to_{Map} \lambda\underline{1}[\uparrow][\underline{1}.(\underline{1}[\uparrow].(id\circ\uparrow))] \to_{Clos}$ $\lambda\underline{1}[\uparrow\circ(\underline{1}.(\underline{1}[\uparrow].(id\circ\uparrow)))] \to_{ShiftCons} \lambda\underline{1}[\underline{1}[\uparrow].(id\circ\uparrow)] \to_{IdL} \lambda\underline{1}[\underline{1}[\uparrow].\uparrow] \to_{VarCons} \lambda\underline{1}[\uparrow] = \lambda\underline{2}$;
- $(\lambda\lambda\underline{1}[\uparrow])\ \underline{1} \to_{Beta} (\lambda\underline{1}[\uparrow])[\underline{1}.id] \to_{Abs} \lambda\underline{1}[\uparrow][\underline{1}.((\underline{1}.id)\circ\uparrow)] \to_{Map} \lambda\underline{1}[\uparrow][\underline{1}.(\underline{1}[\uparrow].(id\circ\uparrow))] \to_{IdL}$ $\lambda\underline{1}[\uparrow][\underline{1}.(\underline{1}[\uparrow].\uparrow)] \to_{Clos} \lambda\underline{1}[\uparrow\circ(\underline{1}.(\underline{1}[\uparrow].\uparrow))] \to_{ShiftCons} \lambda\underline{1}[\underline{1}[\uparrow].\uparrow] \to_{VarCons} \lambda\underline{1}[\uparrow] = \lambda\underline{2}$. □

In the following lemmas, $(M\ \underline{1}^n)$ is a shorthand for n applications of $\underline{1}$, i.e., $(\ldots((M\ \underline{1})\underline{1})\ldots\underline{1})$.

Lemma 4.4 *Every λ_{SUSP}-derivation of $(\lambda\lambda(\underline{2}\ \underline{2}))\ \underline{1}^n$ to its λ_{SUSP}-nf has length* $4n+5$.

Proof. In fact, note that the sole possible derivation is:
$(\lambda\lambda(\underline{2}\ \underline{2}))\ \underline{1}^n \to_{\beta_s} \llbracket(\lambda(\underline{2}\ \underline{2})), 1, 0, (\underline{1}^n, 0)::nil\rrbracket \to_{r_7} \lambda\llbracket(\underline{2}\ \underline{2}), 2, 1, @0::(\underline{1}^n, 0)::nil\rrbracket \to_{r_6}$
$\lambda(\llbracket\underline{2}, 2, 1, @0::(\underline{1}^n, 0)::nil\rrbracket\ \llbracket\underline{2}, 2, 1, @0::(\underline{1}^n, 0)::nil\rrbracket) \to_{r_5}^2$
$\lambda(\llbracket\underline{1}, 1, 1, (\underline{1}^n, 0)::nil\rrbracket\ \llbracket\underline{1}, 1, 1, (\underline{1}^n, 0)::nil\rrbracket) \to_{r_4}^2 \lambda(\llbracket\underline{1}^n, 0, 1, nil\rrbracket\ \llbracket\underline{1}^n, 0, 1, nil\rrbracket) \to_{r_6}^{2(n-1)}$
$\lambda((\llbracket\underline{1}, 0, 1, nil\rrbracket)^n\ (\llbracket\underline{1}, 0, 1, nil\rrbracket)^n) \to_{r_2}^{2n} \lambda(\underline{2}^n\ \underline{2}^n)$. □

Lemma 4.5 ([26]) *There exists a derivation of* $(\lambda\lambda(\underline{2}\ \underline{2}))\ \underline{1}^n$ *to its* $\lambda\sigma$*-nf whose length is* $n+9$.

PROOF. Consider the following derivation:
$(\lambda\lambda(\underline{2}\ \underline{2}))\ \underline{1}^n = (\lambda\lambda(\underline{1}[\uparrow]\ \underline{1}[\uparrow]))\ \underline{1}^n \to_{Beta} (\lambda(\underline{1}[\uparrow]\ \underline{1}[\uparrow]))[\underline{1}^n.id] \to_{Abs}$
$\lambda((\underline{1}[\uparrow]\ \underline{1}[\uparrow])[\underline{1}.((\underline{1}^n.id)\circ\uparrow)]) \to_{Map}$
$\lambda((\underline{1}[\uparrow]\ \underline{1}[\uparrow])[\underline{1}.(\underline{1}^n[\uparrow].(id\circ\uparrow))]) \to_{App}^{n-1} \lambda((\underline{1}[\uparrow]\ \underline{1}[\uparrow])[\underline{1}.((\underline{1}[\uparrow])^n.(id\circ\uparrow))]) \to_{App}$
$\lambda((\underline{1}[\uparrow][\underline{1}.((\underline{1}[\uparrow])^n.(id\circ\uparrow))])\ (\underline{1}[\uparrow][\underline{1}.((\underline{1}[\uparrow])^n.(id\circ\uparrow))])) \to_{Clos}$
$\lambda((\underline{1}[\uparrow\circ(\underline{1}.(\underline{1}[\uparrow])^n.(id\circ\uparrow))])\ (\underline{1}[\uparrow][\underline{1}.((\underline{1}[\uparrow])^n.(id\circ\uparrow))])) \to_{ShiftCons}$
$\lambda((\underline{1}[(\underline{1}[\uparrow])^n.(id\circ\uparrow)])\ (\underline{1}[\uparrow][\underline{1}.((\underline{1}[\uparrow])^n.(id\circ\uparrow))])) \to_{VarCons}$
$\lambda((\underline{1}[\uparrow])^n\ (\underline{1}[\uparrow][\underline{1}.((\underline{1}[\uparrow])^n.(id\circ\uparrow))])) \to^3 \lambda((\underline{1}[\uparrow])^n\ (\underline{1}[\uparrow])^n) = \lambda(\underline{2}^n\ \underline{2}^n)$. □

Proposition 4.6 *The* $\lambda\sigma$*- and* λ_{SUSP}*-calculi are non comparable.*

PROOF. On one side, by Lemmas 4.4 and 4.5, there exists a simulation $(\lambda\lambda(\underline{2}\ \underline{2}))\ \underline{1}^n \twoheadrightarrow_{\lambda\sigma} \lambda(\underline{2}\ \underline{2})$ shorter than the shortest of the simulations $(\lambda\lambda(\underline{2}\ \underline{2}))\ \underline{1}^n \twoheadrightarrow_{\lambda_{\text{SUSP}}} \lambda(\underline{2}\ \underline{2})$. Then $\lambda_{\text{SUSP}} \not\prec \lambda\sigma$.

On the other side, consider the following simulation in λ_{SUSP}:
$((\lambda\lambda\underline{2})\ \underline{1}) \to_{\beta_s} [\![(\lambda\underline{2}),1,0,(\underline{1},0) :: nil]\!] \to_{r_7} \lambda[\![\underline{2},2,1,@0 :: (\underline{1},0) :: nil]\!] \to_{r_5}$
$\lambda[\![\underline{1},1,1,(\underline{1},0) :: nil]\!] \to_{r_4} \lambda[\![\underline{1},0,1,nil]\!] \to_{r_2} \lambda\underline{2}$.

This simulation together with Lemma 4.3 allows us to conclude that: $\lambda\sigma \not\prec \lambda_{\text{SUSP}}$. □

To prove that λs_e is more adequate in the simulation of one step β-reduction than λ_{SUSP} we need to estimate the lengths of derivations.

Definition 4.7 *Let* $A, B, C \in \Lambda$ *and* $k \geq 0$. *We define the functions* $M : \Lambda \to \mathbb{N}$ *and* $Q_k : \Lambda \times \Lambda \to \mathbb{N}$ *by:*

- $M(\underline{\mathtt{n}}) = 1$
- $M(\lambda A) = M(A)+1$
- $M(A\ B) = M(A)+M(B)+1$
- $Q_k((A\ B),C) = Q_k(A,C)+Q_k(B,C)+1$
- $Q_k(\underline{\mathtt{n}},B) = \begin{cases} n & \text{if } n<k \\ n+M(B) & \text{if } n=k \\ k+1 & \text{if } n>k \end{cases}$
- $Q_k(\lambda A,B) = Q_{k+1}(A,B)+1$

Lemma 4.8 *Let* $A \in \Lambda$. *Then all* s_e*-derivations of* $\varphi^i_k A$ *to its* s_e*-nf have length* $M(A)$.

PROOF. By simple induction over the structure of A. This is an easy extension of the same lemma formulated for the λs-calculus in [26]. □

Lemma 4.9 *Let* $A \in \Lambda$. *Then all* SUSP*-derivations of the well-formed term* $[\![A,i,i,@i-1 :: \ldots :: @0 :: nil]\!]$ *to its* SUSP*-nf have length greater than or equal to* $M(A)$.

PROOF. By induction over the structure of terms.

- $\boldsymbol{A = \underline{\mathtt{n}}}$. If $n > i$ then $[\![\underline{\mathtt{n}},i,i,@i-1 :: \ldots :: @0 :: nil]\!] \to^i_{r_5} [\![\underline{\mathtt{n-i}},0,i,nil]\!] \to_{r_2} \underline{\mathtt{n}}$. The length of the derivation is $i+1 \geq M(A)$. If $n \leq i$ then $[\![\underline{\mathtt{n}},i,i,@i-1 :: \ldots :: @0 :: nil]\!] \to^{n-1}_{r_5}$ $[\![\underline{1},i-n+1,i,@i-n :: \ldots :: @0 :: nil]\!] \to_{r_3} \underline{\mathtt{n}}$. The length of the derivation is $n \geq M(A)$.
- $\boldsymbol{A = (B\ C)}$. We have that $[\![(B\ C),i,i,@i-1 :: \ldots :: @0 :: nil]\!] \to_{r_6}$ $([\![B,i,i,@i-1 :: \ldots :: @0 :: nil]\!]\ [\![C,i,i,@i-1 :: \ldots :: @0 :: nil]\!])$. By the induction hypothesis we conclude that the length of the derivation is greater than or equal to $1+M(B)+M(C) = M(B\ C) = M(A)$.
- $\boldsymbol{A = (\lambda B)}$. We have that $[\![(\lambda B),i,i,@i-1 :: \ldots :: @0 :: nil]\!] \to_{r_7} \lambda[\![B,i+1,i+1,@i :: \ldots :: @0 :: nil]\!]$. By induction hypothesis we conclude that the length of the derivation is greater than or equal to $1+M(B) = M(\lambda B) = M(A)$. □

Lemma 4.10 *Let* $B \in \Lambda$ *and* $i,j \geq 0$. *The derivation of the* SUSP*-term* $[\![B,i,j,@j-1 :: e]\!]$ *to its* SUSP*-nf has length greater than or equal to* $M(B)$.

PROOF.

- Case $\boldsymbol{B = \underline{\mathtt{n}}}$, $[\![\underline{\mathtt{n}},i,j,@j-1 :: e]\!]$ rewrites to its SUSP-nf in one or more steps depending on n.

- Case $\boldsymbol{B = (C\ D)}$, we have $[\![(C\ D), i, j, @j-1::e]\!] \to_{r_6} [\![C, i, j, @j-1::e]\!]\ [\![D, i, j, @j-1::e]\!]$. By the induction hypothesis we obtain the desired result.

- Case $\boldsymbol{B = (\lambda C)}$, we have $[\![(\lambda C), i, j, @j-1::e]\!] \to_{r_7} \lambda[\![C, i+1, j+1, @j::e']\!]$, that by induction hypothesis completes the proof. $\square$

Proposition 4.11 *Let $A, B \in \Lambda$ and $k \geq 0$. Then every* SUSP*-derivation of* $[\![A, k, k-1, @k-2 :: \ldots :: @0 :: (B, l) :: nil]\!]$ *to its* SUSP*-nf has length greater than or equal to $Q_k(A, B)$.*

Proof. By structural induction over A.

- $\boldsymbol{A = \underline{\mathtt{n}}}$. If $n < k$ then $[\![\underline{\mathtt{n}}, k, k-1, @k-2::\ldots::@0::(B,l)::nil]\!] \to_{r_5}^{n-1}$ $[\![\underline{\mathtt{1}}, k-n+1, k-1, @k-n-1::\ldots::@0::(B,l)::nil]\!] \to r_3\ \underline{\mathtt{n}}$. This derivation has length $n \geq Q_k(\underline{\mathtt{n}}, B)$. If $n = k$ then $[\![\underline{\mathtt{n}}, k, k-1, @k-2::\ldots::@0::(B,l)::nil]\!] \to_{r_5}^{n-1} [\![\underline{\mathtt{1}}, 1, k-1, (B,l)::nil]\!] \to r_4$ $[\![B, 0, k-1-l, nil]\!]$. By Lemma 4.10 the last term rewrites to its SUSP-nf in $M(B)$ or more rewrite steps. The whole derivation has length greater than or equal to $n + M(B) = Q_k(\underline{\mathtt{n}}, B) = Q_k(A, B)$. If $n > k$ then $[\![\underline{\mathtt{n}}, k, k-1, @k-2::\ldots::@0::(B,l)::nil]\!] \to_{r_5}^{k} [\![\underline{\mathtt{n\text{-}k}}, 0, k\text{-}1, nil]\!] \to_{r_2} \underline{\mathtt{n-1}}$. Derivation whose length is $k+1 \geq Q_k(\underline{\mathtt{n}}, B) = Q_k(A, B)$.

- $\boldsymbol{A = (C\ D)}$. $[\![(C\ D), k, k-1, @k-2::\ldots::@0::(B,l)::nil]\!] \to_{r_6}$ $([\![C, k, k\text{-}1, @k\text{-}2::\ldots::@0::(B,0)::nil]\!]\ [\![D, k, k\text{-}1, @k\text{-}2::\ldots::@0::(B,0)::nil]\!])$. By the induction hypothesis the derivation has length greater than or equal to $1{+}Q_k(C,B){+}Q_k(D,B) = Q_k((C\ D),B) = Q_k(A,B)$.

- $\boldsymbol{A = \lambda C}$. $[\![(\lambda C), k, k-1, @k-2::\ldots::@0::(B,l)::nil]\!] \to_{r_7} \lambda[\![C, k+1, k, @k-1::\ldots::@0::(B,l)::nil]\!]$. By the induction hypothesis we can conclude that this derivation has length greater than or equal to $1 + Q_{k+1}(C, B) = Q_k(\lambda C, B) = Q_k(A, B)$. $\square$

Proposition 4.12 *Let $A, B \in \Lambda$ and $k \geq 1$. s_e-derivations of $A\sigma^k B$ to its s_e-nf have length $\leq Q_k(A, B)$.*

Proof. By structural induction over the pure lambda term A.

- $\boldsymbol{A = \underline{\mathtt{n}}}$. By applying the *$\sigma$-destruction* rule, in the case $n \neq k$, we obtain either $\underline{\mathtt{n-1}}$ or $\underline{\mathtt{n}}$ and in the case $n = k$, $\varphi_0^k B$. In the case that $n \neq k$, the derivation has length equal to $1 \leq Q_k(\underline{\mathtt{n}}, B)$. In the other case, we apply Lemma 4.8 obtaining that the complete s_e-normalization has length $1 + M(B)$. In both cases the derivation has length less than or equal to $Q_k(\underline{\mathtt{n}}, B)$.

- $\boldsymbol{A = (C\ D)}$. $(C\ D)\sigma^k B \to (C\sigma^k B\ D\sigma^k B)$. By applying the induction hypothesis we conclude that the complete derivation has length less than or equal to $1 + Q_k(C, B) + Q_k(D, B) = Q_k((C\ D), B)$.

- $\boldsymbol{A = (\lambda C)}$. $(\lambda C)\sigma^k B \to \lambda(C\sigma^{k+1} B)$. By the induction hypothesis we conclude that the whole derivation has length less than or equal to $1 + Q_{k+1}(C, B) = Q_k(\lambda C, B)$. $\square$

Theorem 4.13 ($\boldsymbol{\lambda s_e \prec \lambda_{\mathbf{susp}}}$) *The λs_e is more adequate in the simulation of one step β-reduction than the λ_{SUSP}-calculus.*

Proof. We prove the stronger result that if $A \in \Lambda$ and $A \to_{\beta_s} B \to_{\text{SUSP}}^m$ SUSP-nf(B) is a λ_{SUSP}-simulation of a β-reduction then: $A \to_{\sigma-generation} C \to_{s_e}^n s_e$-nf$(C)$ has length $n + 1 \leq m + 1$.
In λ_{SUSP}, for any redex of β_s we have $(\lambda D)\ E \to_{\beta_s} [\![D, 1, 0, (E, 0)::nil]\!] \to_{\text{SUSP}}^m$ SUSP-nf$([\![D, 1, 0, (E, 0)::nil]\!])$. In the λs_e, $(\lambda D)\ E \to_{\sigma-generation} D\sigma^1 E \to_{s_e}^n s_e$-nf$(D\sigma^1 E)$. By Propositions 4.11 and 4.12, $m \geq Q_1(D, E) \geq n$. Hence, the length of a λ_{SUSP}-simulation of a β-reduction is not shorter than that of some λs_e-simulation.

The 2nd part of being *more adequate* is shown by comparing the length of simulations. E.g., let $(\lambda\underline{\mathtt{2}})\ \underline{\mathtt{1}} \to_\beta \underline{\mathtt{1}}$. In λ_{SUSP} the only possible three steps simulation is: $(\lambda\underline{\mathtt{2}})\ \underline{\mathtt{1}} \to_{\beta_s} [\![\underline{\mathtt{2}}, 1, 0, (\underline{\mathtt{1}}, 0)::nil]\!] \to_{r_5} [\![\underline{\mathtt{1}}, 0, 0, nil]\!] \to_{r_2} \underline{\mathtt{1}}$. In λs_e the only possible two steps simulation is: $(\lambda\underline{\mathtt{2}})\ \underline{\mathtt{1}} \to_{\sigma-generation} \underline{\mathtt{2}}\sigma^1\underline{\mathtt{1}} \to_{\sigma-destruction} \underline{\mathtt{1}}$. $\square$

As mentioned in the above proof, we prove a stronger result than simple better adequacy of λs_e as in [26]. In fact, we prove that the length of all λs_e-simulations are shorter than the length of any λ_{SUSP}-simulation. Examining the proofs of Propositions 4.11 and 4.12 which relate the length of derivations with the measure operator Q_k, it appears evident that both calculi work similarly except that after having propagated suspended terms between the body of abstractors, λ_{SUSP} deals with the substitutions in a less efficient way. To explain that, compare the simulations of β-reduction from the term $(\lambda(\lambda^n \underline{\mathtt{i}}))\ \underline{\mathtt{j}}$, where $n \geq 0$:

$(\lambda(\lambda^n \underline{\mathtt{i}}))\underline{\mathtt{j}} \to_{\sigma-gen} (\lambda^n \underline{\mathtt{i}})\sigma^1 \underline{\mathtt{j}} \to^n_{\sigma-\lambda-trans} \lambda^n(\underline{\mathtt{i}}\sigma^{n+1}\underline{\mathtt{j}}) =: t_1$

$(\lambda(\lambda^n \underline{\mathtt{i}}))\underline{\mathtt{j}} \to_{\beta_s} [\![\lambda^n \underline{\mathtt{i}}, 1, 0, (\underline{\mathtt{j}},0) :: nil]\!] \to^n_{r_7} \lambda^n [\![\underline{\mathtt{i}}, n+1, n, @n\text{-}1 :: \ldots :: @0 :: (\underline{\mathtt{j}},0) :: nil]\!] =: t_2$.

After that the λs_e complete the simulation in one or two steps by checking arithmetic inequations:

$$t_1 \to_{\sigma-dest} \begin{cases} \lambda^n \underline{\mathtt{i}}, & \text{if } i < n+1 \\ \lambda^n \underline{\mathtt{i-1}}, & \text{if } i > n+1 \\ \lambda^n(\varphi_0^{n+1}\underline{\mathtt{j}}) \to_{\varphi-dest} \lambda^n \underline{\mathtt{j+n}}, & \text{if } i = n+1 \end{cases}$$

But in the λ_{SUSP} we have to destruct the environment list, environment by environment:

$$t_2 \begin{cases} \to^{i-1}_{r_5} \lambda^n [\![\underline{\mathtt{1}}, n\text{-}i+2, n, @n\text{-}i :: \ldots :: @0 :: (\underline{\mathtt{j}}, 0) :: nil]\!] \to_{r_3} \lambda^n \underline{\mathtt{i}}, & \text{if } i < n+1 \\ \to^{n+1}_{r_5} \lambda^n [\![\underline{\mathtt{i-n-1}}, 0, n, nil]\!] \to_{r_2} \lambda^n \underline{\mathtt{i-1}}, & \text{if } i > n+1 \\ \to^{i-1}_{r_5} \lambda^n [\![\underline{\mathtt{1}}, 1, n, (\underline{\mathtt{j}}, 0) :: nil]\!] \to_{r_4} \lambda^n [\![\underline{\mathtt{j}}, 0, n, nil]\!] \to_{r_2} \lambda^n \underline{\mathtt{j+n}}, & \text{if } i = n+1 \end{cases}$$

These simple considerations lead us to believe that the main difference of the two calculus (at least in the simulation of β-reduction) is given by the manipulation of indices: although λ_{SUSP} includes all de Bruijn indices, it does not profit from the existence of the built-in arithmetic for indices. These observations may be relevant for the treatment of the open question of preservation of strong normalization of λ_{SUSP} (conjectured positively in [34]), since the λs_e has been proved to answer this question negatively in [18].

5 Relating the Eta rules

[3] established the correspondence between the *Eta* rules of $\lambda\sigma$ and λs_e through the premises $t[\uparrow] =_\sigma M$ and $\varphi_0^2 t =_{s_e} M$, where $t \in \Lambda_{dB}$. This correspondence means that the effect of applying the substitution $[\uparrow]$, in $\lambda\sigma$, and the upgrading φ_0^2, in λs_e, to a pure λ-term are identical. This implies that these *Eta* rules are equivalent when applied to a pure λ-term. Hence, it remains to show that the results, in the two calculi, of applying the substitution $[\uparrow]$ and the upgrading operator φ_0^2 to a λ-term t are equal (up to the codification of the term in the internal language of the calculus). This is the case $k=0$ of the third item of the following lemma.

Lemma 5.1 (Eta correspondence of $\lambda\sigma$ and λs_e [3])

1. *Let $\underline{\mathtt{n}}$ be a de Bruijn index. Then, for $k \geq 0$, the s_e-nf of $\varphi_k^2 \underline{\mathtt{n}}$ and the σ-nf of $\underline{\mathtt{n}}[\underline{\mathtt{1}}.\underline{\mathtt{1}}[\uparrow].\underline{\mathtt{1}}[\uparrow^2]\ldots\underline{\mathtt{1}}[\uparrow^{k-1}].\uparrow^{k+1}]$ are corresponding de Bruijn indices.*
2. *Let λt an abstraction over Λ_{dB}. Then, for $k \geq 0$, $(\lambda t)[\underline{\mathtt{1}}.\underline{\mathtt{1}}[\uparrow].\underline{\mathtt{1}}[\uparrow^2].\ldots.\underline{\mathtt{1}}[\uparrow^{k-1}].\uparrow^{k+1}]$ σ-rewrites to $\lambda(t[\underline{\mathtt{1}}.\underline{\mathtt{1}}[\uparrow].\underline{\mathtt{1}}[\uparrow^2]\ldots\underline{\mathtt{1}}[\uparrow^k].\uparrow^{k+2}])$.*
3. *Let $t \in \Lambda_{dB}$ and t' its codification in the language of $\lambda\sigma$, where all de Bruijn indices $\underline{\mathtt{n}} \in \mathbb{N}$ occurring in t are replaced with $\underline{\mathtt{1}}[\uparrow^{n-1}]$. Then, for $k \geq 0$, the σ-nf of $t'[\underline{\mathtt{1}}.\underline{\mathtt{1}}[\uparrow].\underline{\mathtt{1}}[\uparrow^2].\ldots.\underline{\mathtt{1}}[\uparrow^{k-1}].\uparrow^{k+1}]$ corresponds to the s_e-nf of $\varphi_k^2 t$.*

Analogously to the previous lemma, in the next proposition we establish the correspondence between the rules *Eta* of λ_{SUSP} and λs_e; i.e., the correspondence, in the above mentioned sense, between the terms at their premises: $[\![t, 0, 1, nil]\!]$ and $\varphi_0^2 t$, for $t \in \Lambda_{dB}$. This corresponds to the case $k=0$ of the following proposition.

Proposition 5.2 (Eta correspondence of λ_{susp} and λs_e) *Let $t \in \Lambda_{dB}$. Then, for all $k \geq 0$, the SUSP-nf of the suspended term $[\![t, k, k+1, @k :: @k-1 :: \ldots :: @1 :: nil]\!]$ corresponds to the s_e-nf of $\varphi_k^2 t$.*

PROOF. This is done by induction on the structure of t.

- $\boldsymbol{t = \underline{\mathtt{n}}}$. By Lemma 3.15 we have that for all $k \geq 0$,
  $$[\![\underline{\mathtt{n}}, k, k+1, @k :: @k-1 :: \ldots :: @1 :: nil]\!] \to \begin{cases} \underline{\mathtt{n+1}} & \text{if } n > k \\ \underline{\mathtt{n}} & \text{if } n \leq k \end{cases}$$
  This coincides with the result of applying the rule φ-*dest* to the term $\varphi_k^2 \underline{\mathtt{n}}$.

- $\boldsymbol{t = (A\ B)}$. $[\![(A\ B), k, k+1, @k :: @k-1 :: \ldots :: @1 :: nil]\!] \to_{r_6}$ $([\![A, k, k+1, @k :: \ldots :: @1 :: nil]\!]\ [\![B, k, k+1, @k :: \ldots :: @1 :: nil]\!]) \stackrel{IH}{\equiv}$ $(\varphi_k^2 A\ \varphi_k^2 B)$. Also, $\varphi_k^2 t \to_{\varphi-app} (\varphi_k^2 A\ \varphi_k^2 B)$.
- $\boldsymbol{t = (\lambda A)}$. $[\![(\lambda A), k, k+1, @k :: @k-1 :: \ldots :: @1 :: nil]\!] \to_{r_7}$ $(\lambda[\![A, k+1, k+2, @k+1 :: \ldots :: @1 :: nil]\!]) \stackrel{IH}{\equiv} (\lambda \varphi_{k+1}^2 A)$. Also, $\varphi_k^2(\lambda A) \to_{\sigma\lambda\text{-}trans} (\lambda \varphi_{k+1}^2 A)$. □

This correspondence is not obvious for open terms. In fact, let c be a constant. On one side, in λ_{SUSP}, we have that $[\![c, k, k+1, @k :: \ldots :: @1 :: nil]\!] \to_{r_1} c$. On the other side, $\varphi_k^2 c$ is irreducible in λs_e. Both terms can, in a certain sense, be considered equivalent since the upgrading operator φ_k^2 does not modify the constant c and this correspondence could be assumed in other practical contexts such as those of higher order unification via explicit substitutions.

The following notational conventions are useful for the rest of the paper:

Notation 5.3 *Let $\xi \in \{\sigma, s_e, \text{SUSP}\}$, and let $\lambda\xi$ be the corresponding explicit substitution calculus. The generation rules of $\lambda\xi$ (i.e. the Beta, σ-generation or β_s rules), will be denoted correspondingly by* $\lambda\xi$-gen. *Similarly,* Eta_ξ *denotes the corresponding* Eta *rule. ξ denotes the associated substitution calculus, that is given by the rewriting rules of the calculus $\lambda\xi$ except the* ξ-gen *and the* Eta_ξ *rules. The congruence generated by the rules of the substitution calculus ξ is denoted by $=_\xi$. By* ξ-nf(M) *we denote the ξ-normal form of the $\lambda\xi$-term M. If M has a $\lambda\xi$-gen redex at the root position then we denote by $gen_{\lambda\xi}(M, root)$ its contractum.*

Now, we establish the appropriateness of the three *Eta* rules of $\lambda\sigma$, λs_e and λ_{SUSP}. By appropriateness of a specific *Eta* rule we understand that every pure λ-terms which contains an Eta redex is reduced to the same pure λ-term by the usual η-rule as well as by the specific *Eta* rule.

Lemma 5.4 (Appropriateness of the *Eta* rules) *Let $a \in \Lambda_{dB}$. Then the following statements are equivalent:*

(a) $\lambda(a\ \underline{1}) \to_\eta b$

(b) $\lambda(a\ \underline{1}) \to_{Eta_\xi} b$*, where ξ stands for σ, s_e or* SUSP.

Proof. Suppose (a) is true. Then by structural induction on the term a:

1. λ_{SUSP}: We will show that $[\![b, 0, 1, nil]\!] =_{\text{SUSP}} a$.
   - $a = \underline{\mathtt{n}}$ $(n > 1)$: $[\![\underline{\mathtt{n}-\mathtt{1}}, 0, 1, nil]\!] \to_2 \underline{\mathtt{n}}$.
   - $a = (c\ d)$. $b^+ = (c\ d)$ means that b is obtained from $(c\ d)$ decreasing all its free indices by one. Now note that the effect of normalizing $[\![b, 0, 1, nil]\!]$ is to increase all free indices of b by one as shown in the proof of the proposition 3.16.
   - $a = \lambda c$. Suppose $b^+ = \lambda c$ so b is obtained from λc decreasing all free indices in c but $\underline{1}$ by one and conclude considering the same argument of the previous item.
2. λs_e: This is a straightforward from previous item and proposition 5.2.
3. $\lambda\sigma$: This is a straightforward from previous item and lemma 5.1.

Conversely, we will show that:

1. In λ_{SUSP} that $[\![b, 0, 1, nil]\!] =_{\text{SUSP}} b^+$:
   - $[\![\underline{\mathtt{n}-\mathtt{1}}, 0, 1, nil]\!] \to_{r_2} \underline{\mathtt{n}} = (\underline{\mathtt{n}-\mathtt{1}})^+$.
   - $[\![(c\ d), 0, 1, nil]\!] \to_{r_6} [\![c, 0, 1, nil]\!]\ [\![d, 0, 1, nil]\!] \stackrel{IH}{=} c^+\ d^+ = (c\ d)^+$.
   - $[\![\lambda c, 0, 1, nil]\!] \to_{r_7} \lambda[\![c, 1, 2, @1 :: nil]\!] \stackrel{IH}{=} \lambda c^{+1} = (\lambda c)^+$.
2. In λs_e that $\varphi_0^2(b) = b^+$:

- $\varphi_0^2(\underline{\mathtt{n-1}}) \to_{\varphi-destr} \underline{\mathtt{n}} = (\underline{\mathtt{n-1}})^+$
- $\varphi_0^2(c\ d) \to_{\varphi-app} \varphi_0^2(c)\ \varphi_0^2(d) \stackrel{IH}{=} c^+\ d^+ = (c\ d)^+$.
- $\varphi_0^2(\lambda c) \to_{\varphi-\lambda} \lambda(\varphi_1^2 c) \stackrel{IH}{=} \lambda(c^{+1}) = (\lambda c)^+$.

3. In $\lambda\sigma$ that $b[\uparrow] = b^+$.

- $\underline{\mathtt{1}}[\uparrow^{n-2}][\uparrow] \to_{Clos} \underline{\mathtt{1}}[\uparrow^{n-1}] = (\underline{\mathtt{n-1}})^+$.
- $(c\ d)[\uparrow] \to_{App} c[\uparrow]\ d[\uparrow] \stackrel{IH}{=} c^+\ d^+ = (c\ d)^+$.
- $(\lambda c)[\uparrow] \to_{Abs} \lambda(c[\underline{\mathtt{1}}.\uparrow^2]) \stackrel{IH}{=} \lambda c^{+1} = (\lambda c)^+$.

□

6 Usual implementations of Eta

In the sequel we use "η" for η-reduction, and "*Eta*" for the *Eta*-reduction rules of the explicit substitution calculi. By an "implementation" of the *Eta* rule of any of the three treated calculi of explicit substitutions we understand an effective computational mechanism of evaluation of the premisse of the conditional rewriting *Eta* rule, which allows for deciding the occurrence of *Eta*-redices and their subsequent reduction. In other words, an implementation is an effective mechanism for deciding the one step *Eta*-reduction relation.

When implementing the one step reduction of these calculi one has to take into account that the given *Eta* rule and its suggested implementation are not clean in the sense that one application of *Eta*-reduction can involve applications of other rules of the substitution calculus.

In an explicit substitution calculus $\lambda\xi$, a *clean* implementation of the η-reduction does not apply additional rules of the associated substitution calculus ξ during a one step application of the implemented η-reduction.

Definition 6.1 (Clean Implementations of η-reduction) *An implementation of η-reduction, say* ImEta_ξ*, in an explicit substitution calculus $\lambda\xi$ is said to be* clean *if for any $\lambda\xi$-term M, whenever we obtain N from M by applying this implementation of the η-reduction, denoted by $M \to_{ImEta_\xi} N$, there is no N' such that $M \to_{Eta_\xi} N'$ and $N' \to_\xi^* N$. An implementation of η-reduction that is not clean is called* unclean.

Lemma 6.2 (The *Eta* rules are unclean) *The implementations of η-reduction directly from the* Eta_ξ *rewriting rules of the three treated calculi are unclean.*

PROOF. Counterexamples are easy to formulate (e.g. see proof of Lemma 6.4) because the equational premise of all the three rules is given in terms of the corresponding ξ congruence $=_\xi$: $a =_\sigma b[\uparrow]$, $a =_{s_e} \varphi_0^2(b)$ and $a =_{\mathrm{SUSP}} [\![b, 0, 1, nil]\!]$, respectively. □

6.1 Rule implementation for $\lambda\sigma$

We used OCAML, a variation of the ML language, for implementing the rewriting rules of the three treated calculi. The code of this implementation is available at `http://www.mat.unb.br/~ayala/TCgroup/`. For $\lambda\sigma$, consider for example the rule *Abs*. We have to remark that $\lambda\sigma$ works with two different entities: terms (TERMS) and substitutions (SUBS), which should be discriminated in any implementation. $\lambda\sigma$-terms of the form $\underline{\mathtt{1}}$, λM, $(M\ N)$ and $M[S]$ are respectively represented as `One`, `L(M)`, `A(M,N)` and `Sb(M,S)` and $\lambda\sigma$-substitutions of the form id, $\uparrow$, $M.S$ and $S \circ T$ as `Id`, `Up`, `Pt(M,S)` and `Cp(S,T)`. Applications of the rules are implemented in two steps: the first one of detection of redices and the second one, after selection of a possible redex, of true reduction. Detection of redices for this rule is implemented as in Table 7. Note that the search for redices is divided in the search over terms and substitution entities. Once a redex at position `pr` of the term `exp` is detected (and selected) the application of *Abs* is done by means of the function specified in Table 8. Analogously, the application is divided in parts for terms and substitutions. All other rules are similarly implemented.

Table 7: Detection of redices for Abs of $\lambda\sigma$

```
let rec matchingAbs exp l pos =
  match exp with  Dummy -> l | One -> l | Vr c -> l |
   A(e1,e2)     -> append (matchingAbs e1 l (append pos [1])) (matchingAbs e2 [] (append pos [2])) |
   L(e1)        -> matchingAbs e1 l (append pos [1]) |
   Sb(L(e1),sb) -> pos::append(matchingAbs e1 l (append pos [1;1]))(matchingAbsSb sb [] (append pos [2])) |
   Sb(e1,sb)    -> append (matchingAbs e1 l (append pos [1])) (matchingAbsSb sb [] (append pos [2]))
and matchingAbsSb subs l pos =
  match subs with  Up -> l | Id -> l |
   Pt(e1,sb) -> append (matchingAbs e1 l (append pos [1])) (matchingAbsSb sb [] (append pos [2])) |
   Cp(s1,s2) -> append (matchingAbsSb s1 l (append pos [1])) (matchingAbsSb s2 [] (append pos [2]));;
```

Table 8: Application of Abs of $\lambda\sigma$

```
let rec absreduction exp pr =
  match pr with [] -> (match exp with Sb(L(e1),sb) -> L(Sb(e1,Pt(One,Cp(sb,Up)))) | _ -> exp)  |
               1 :: tail -> (match exp with Dummy -> exp | One -> exp | Vr c -> exp |
                  A(e1,e2) -> A((absreduction e1 tail),e2) | L(e1) -> L(absreduction e1 tail) |
                  Sb(e1,s2)  -> Sb((absreduction e1 tail),s2)) |
               2 :: tail -> (match exp with Dummy -> exp | One -> exp | Vr c -> exp |
                  L(e1) -> exp | A(e1,e2) -> A(e1,(absreduction e2 tail)) |
                  Sb(e1,s2)-> Sb(e1,(absreductionSb s2 tail))) | _ -> exp
and absreductionSb subs pr =
  match pr with [] -> subs |
               1 :: tail -> (match subs with Id -> subs | Up -> subs |
                  Cp(s1,s2) -> Cp((absreductionSb s1 tail),s2) |
                  Pt(e1,s2) -> Pt((absreduction e1 tail),s2)) |
               2 :: tail -> (match subs with Id -> subs | Up -> subs |
                  Cp(s1,s2) -> Cp(s1,(absreductionSb s2 tail)) |
                  Pt(e1,s2)-> Pt(e1,(absreductionSb s2 tail))) | _ -> subs;;
```

6.2 Rule implementation for λs_e

The implementation for λs_e is simpler since we have to consider a sole entity, that is the one of (lambda) terms. λs_e-terms are of the form $\underline{\mathtt{n}}$, $(M\ N)$, λM, $M\sigma^i N$ and $\varphi^i_k M$ and are represented in OCAML respectively as `DB n`, `A(M,N)`, `L(M)`, `S(i,M,N)` and `P(k,i,M)`. Searching for redices of the σ-λ-*transition* and its application for a selected redex `pr` are given in Tables 9 and 10, respectively.

Table 9: Detection of redices for σ-λ-*transition* of λs_e

```
let rec matchingSLtransition exp l pos =
  match exp with Dummy ->l | DB i ->l | Vr c ->l |
    A(e1,e2)->append (matchingSLtransition e1 l(append pos [1]))
                     (matchingSLtransition e2 [] (append pos [2])) |
    L(e1) -> (matchingSLtransition e1 l (append pos [1])) |
    S(i,L(e1),e2)->pos::append(matchingSLtransition e1 l (append pos [1;1]))
                              (matchingSLtransition e2 [] (append pos [2])) |
    S(i,e1,e2) -> append (matchingSLtransition e1 l (append pos [1]))
                         (matchingSLtransition e2 [] (append pos [2])) |
    P(j,k,e1) -> (matchingSLtransition e1 l (append pos [1]));;
```

6.3 Rule implementation for $\lambda_{\mathbf{susp}}$

Expressions in λ_{SUSP} can be of three different types: (suspended) terms, environments and environment terms. Terms of the form C, $\underline{\mathtt{n}}$, $(M\ N)$, λM and $[\![t,i,j,e]\!]$ are represented by `Vr c`, `DB n`, `A(M,N)`, `L(M)` and `Sp(t,i,j,e)`; environments of the form nil, $et :: e$ and $\{\!\{env1,i,j,env2\}\!\}$ by `Nilen`, `Con(et,e)` and `Ck(env1,i,j,env2)`; and environment terms of the form $@n$, (t,l) and $\langle\!\langle envt,i,j,env\rangle\!\rangle$ by `Ar(n)`, `Paar(t,l)` and `LG(envt,i,j,env)`, respectively. The search for redices of the rule (r_7) is given in Table 11 and for its application in a selected position in Table 12. Note that the search for redices and the application of the rule is divided in the search over suspended terms, environments and environment terms.

Table 10: Application of σ-λ-*transition* of λs_e

```
let rec sltransition exp pr =
match pr with  [] -> (match exp with S(i,L(e1),e2) -> L(S(i+1,e1,e2)) | _ -> exp) |
               1 :: tail -> (match exp with
                  A(e1,e2)  -> A((sltransition e1 tail),e2)   |
                  L(e1)     -> L(sltransition e1 tail)        |
                  S(i,e1,e2)-> S(i,(sltransition e1 tail),e2) |
                  P(j,k,e1) -> P(j,k,(sltransition e1 tail))  |  _ -> exp )  |
               2 :: tail -> (match exp with
                  A(e1,e2)  -> A(e1,(sltransition e2 tail))   |
                  S(i,e1,e2)-> S(i,e1,(sltransition e2 tail)) |   _ -> exp ) | _ -> exp;;
```

Table 11: Detection of redices for r_7 of λ_{SUSP}

```
let rec matching_r7 exp l pos = match exp with Dummy ->l | DB i ->l | Vr c ->l |
    A(e1,e2) -> append (matching_r7 e1 l (append pos [1]))(matching_r7 e2 [] (append pos [2])) |
    L(e1)  -> (matching_r7 e1 l (append pos [1])) |
    Sp(L(e1),_,_,env)->pos::append(matching_r7 e1 l (append pos [1;1]))
                                  (matchingEnv_r7 env [] (append pos [2]))|
    Sp(e1,_,_,env) -> append (matching_r7 e1 l (append pos [1]))
                             (matchingEnv_r7 env [] (append pos [2]))
and matchingEnv_r7 env l pos = match env with Nilen -> l |
    Con(envt, env1) -> append (matchingEt_r7 envt l (append pos [1]))
                             (matchingEnv_r7 env1 [] (append pos [2])) |
    Ck(env1,_,_,env2) -> append (matchingEnv_r7 env1 l (append pos [1]))
                               (matchingEnv_r7 env2 [] (append pos [2]))
and matchingEt_r7 envt l pos = match envt with Ar i -> l |
    LG(envt1,_,_,env1) -> append (matchingEt_r7 envt1 l (append pos [1]))
                                (matchingEnv_r7 env1 [] (append pos [2])) |
    Paar(e1,i) -> (matching_r7 e1 l (append pos [1]));;
```

6.4 Implementations by ξ-normalization of Eta are unclean

Observe that except for the *Eta* rule, deciding the applicability of all other rewrite rules of the three calculi (cf. Table 1 for $\lambda\sigma$; 2, 3 and 4 for λs_e; 5 and 6 for λ_{SUSP}) is straightforward, since these rules are either non conditional rules or their premises are simple arithmetic conditions easy to decide by means of built-in arithmetic mechanisms that are embedded in all modern computational systems.

Nevertheless, the applicability of the *Eta* rules of the three calculi depends on checking a condition over the congruence of the rewrite system, which can, in the first instance be implemented following a suggestion by Borovanský in [11] for $\lambda\sigma$ and used in [2] for λs_e. Note that the η-reduction $\lambda(M\ \underline{1}) \to_\eta N$ gives a term N resulting from M by decrementing all its free de Bruijn indices by one. And the suggestion is that this corresponds to the normalization, after the application at the root position of the generation rule of the considered calculus of the term $((\lambda M)\ \Diamond)$ whenever $\Diamond$ does not occur in this normalization. The implementation of this suggestion is presented for the three calculi in the following definition.

Definition 6.3 (ξ-nf implementation of the η-reduction) *For the three treated calculi, the direct implementation of the rewrite rule*

$$\lambda(M\ \underline{1}) \longrightarrow_{nfEta_\xi} N \quad \textit{if} \quad N = \xi\text{-}nf(gen_{\lambda\xi}(((\lambda M)\ \Diamond), root)) \textit{ and } \Diamond \textit{ does not occur in } N$$

is called the implementation by ξ-normalization *of the η-reduction, denoted by* nfEta_ξ.

This implementation is sound for $\lambda\sigma$ (cf. [11]) as well as for λs_e (cf. [2]). However this implementation is unclean because during ξ-normalization, rules of the substitution calculi not strictly involved in η-reduction can be applied. For instance, the λs_e-term $\lambda((\underline{4}\sigma^1\underline{1})\ \underline{1}) \to_{nfEta_{s_e}} \underline{2}$, but $\lambda((\underline{4}\sigma^1\underline{1})\ \underline{1}) \not\to_\eta \underline{2}$. Of course, $\lambda((\underline{4}\sigma^1\underline{1})\ \underline{1}) \to_{\sigma-dest} \lambda(\underline{3}\ \underline{1}) \to_\eta \underline{2}$ (as well as $\lambda(\underline{3}\ \underline{1}) \to_{nfEta_{s_e}} \underline{2}$). Observe here that the *Eta* rule (table 4) does not correspond to the intended operational semantics of the η rule: $\lambda(M\ \underline{1}) \to_\eta N$ means that M and N are functionally equivalent.

Table 12: Application of r_7 of λ_{SUSP}

```
let rec r7_reduction exp pr =
match pr with [] -> (match exp with Sp(L(e1),i,j,env) -> L(Sp(e1,i+1,j+1,Con(Ar(j),env))) | _ -> exp ) |
              1 :: tail -> (match exp with
                 (e1,e2)  -> A((r7_reduction e1 tail),e2)  |
                 L(e1)    -> L(r7_reduction e1 tail)       |
                 Sp(e1,i,j,env) -> Sp((r7_reduction e1 tail),i,j,env) | _ -> exp) |
              2 :: tail -> (match exp with
                A(e1,e2) -> A(e1,(r7_reduction e2 tail))  |
                Sp(e1,i,j,env) -> Sp(e1,i,j,(r7_reductionEnv env tail)) | _ -> exp)
and r7_reductionEnv env pr = match pr with
              1 :: tail -> (match env with
                Con(envt,env1) -> Con((r7_reductionEt envt tail),env1) |
                Ck(env1,i,j,env2) -> Ck((r7_reductionEnv env1 tail),i,j,env2) | _ -> env) |
              2 :: tail -> (match env with
                Con(envt,env1) -> Con(envt,(r7_reductionEnv env1 tail)) |
                Ck(env1,i,j,env2) -> Ck(env1,i,j,(r7_reductionEnv env2 tail)) | _ -> env)
and r7_reductionEt envt pr = match pr with
              1 :: tail -> (match envt with
                Paar(e1,i) -> Paar((r7_reduction e1 tail),i) |
                LG(envt1,i,j,env1) -> LG((r7_reductionEt envt1 tail),i,j,env1) | _ -> envt) |
              2 :: tail -> (match envt with
                LG(envt1,i,j,env1) -> LG(envt1,i,j,(r7_reductionEnv env1 tail))| _ -> envt);;
```

Lemma 6.4 (*nfEta*$_\xi$ implementations of the η-reduction are unclean) *The implementations of the η-reduction by ξ-normalization for the three treated calculi are unclean.*

PROOF.

- For the $\lambda\sigma$, consider the reduction $\lambda((\underline{1}[\uparrow^3][\underline{1}[\uparrow].id])\underline{1}) \to_{nfEta_\sigma} \underline{1}[\uparrow] = \underline{2}$. But $\lambda((\underline{1}[\uparrow^3][\underline{1}[\uparrow].id])\underline{1}) \to_{Eta_\sigma} \underline{1}[\uparrow^2][\underline{1}.id] \to_\sigma^* \underline{2}$.
- For the λs_e, consider the reduction $\lambda((\underline{4}\sigma^1\underline{2})\underline{1}) \to_{nfEta_{s_e}} \underline{2}$. But $\varphi_0^2(\underline{3}\sigma^1\underline{1}) =_{s_e} \underline{4}\sigma^1\underline{2}$ and so $\lambda((\underline{4}\sigma^1\underline{2})\underline{1}) \to_{Eta_{s_e}} \underline{3}\sigma^1\underline{1} \to_{s_e} \underline{2}$.
- For the λ_{SUSP}, consider the reduction $\lambda([\![\underline{4}, 1, 0, (\underline{2}, 0) :: nil]\!]\ \underline{1}) \to_{nfEta_{\text{SUSP}}} \underline{2}$. But $[\![[\![\underline{3}, 1, 0, (\underline{1}, 0) :: nil]\!], 0, 1, nil]\!] =_{\text{SUSP}} [\![\underline{4}, 1, 0, (\underline{2}, 0) :: nil]\!]$ and so $\lambda([\![\underline{4}, 1, 0, (\underline{2}, 0) :: nil]\!]\ \underline{1}) \to_{Eta_{\text{SUSP}}} [\![\underline{3}, 1, 0, (\underline{1}, 0) :: nil]\!] \to_{\text{SUSP}}^* \underline{2}$. $\square$

In the sequel, we present a cleaner way to implement the *Eta* rules avoiding the application of other rules of the substitution calculi than the ones strictly involved in the η-reduction.

7 Clean implementations of Eta

We will adapt the above implementation idea, but will restrict the ξ-normalization of the term $\text{gen}_{\lambda\xi}((\lambda M)\ \Diamond)$. The restricted ξ-normalization, called ξ-pseudo-normalization, should propagate the dummy symbol between the structure of the term M without applying extra rules of the substitution calculus.

Essentially the idea for avoiding the application of extra rules of the substitution calculi during the verification of the premise via pseudo-normalization is to apply rules only when occurrences of $\Diamond$ are detected:

$$l \to r \text{ if } \Diamond \text{ occurs in } l$$

As for all the other rules previously illustrated, our OCAML implementation divides the application of an *Eta* rule in two parts: detection of redices and reduction. For $\lambda\sigma$, $\text{gen}_{\lambda\sigma}((\lambda M)\ \Diamond) = M[\Diamond.id]$. The σ-pseudo-nf($M[\Diamond.id]$) has been implemented as the function `sig-norm` in Table 13, where the `occurdummy`'s checks search in linear time the occurrence of `Dummy` in `exp`. Note that in `sig-norm` except for the rules `IdL`, `IdR` and `Clos`, non trivial reductions are possible only `if` $\Diamond$ occurs. In case these rules had been conditioned like the others, it should be impossible to normalize very simple terms as for instance, $\underline{1}[\uparrow \circ id]$

that are necessary for pseudo-normalizations as $((\lambda \underline{1}[\uparrow^2])\ \Diamond) \to_{\beta_s} \underline{1}[\uparrow^2][\Diamond.id] \to_{Clos} \underline{1}[\uparrow^2 \circ(\Diamond.id)] \to_{Assoc}$ $\underline{1}[\uparrow \circ(\uparrow \circ(\Diamond.id))] \to_{ShiftCons} \underline{1}[\uparrow \circ id] \to_{IdR} \underline{1}[\uparrow]$. Since our objective is to propagate the dummy symbol between the structure of the normalized term that non restricted application of these rules may be pointed out as a deficiency because extra rules may be applied during the σ-pseudo-normalization.

Table 13: σ-pseudo-normalization

```
let rec sig-norm exp = match exp with Dummy -> Dummy | One -> One | Vr c -> Vr c |
(*App*) Sb(A(e1,e2),sb)->(if occurdummy1sb(sb) then A(Sb(e1,sig-normsb(sb)),Sb(e2,sig-normsb(sb)))
                                             else exp)|
(*Abs*) Sb(L(e1),sb)-> (if occurdummy1sb(sb) then L(Sb(e1,sig-normsb(Pt(One,Cp(sb,Up))))) |
(*Clos*) Sb(Sb(e1,s1),s2) -> Sb(e1,sig-normsb(Cp(s1,s2))) |
(*VarCons*) Sb(One,Pt(e1,sb)) -> (if (occurdummy1(e1) || occurdummy1sb(sb)) then sig-norm(e1) else exp) |
(*Id*) Sb(e1,Id) -> (if occurdummy1(e1) then sig-norm(e1) else exp)
and sig-normsb subs = match subs with Up -> Up | Id -> Id             |
(*SCons*) Pt(Sb(One,s1),Cp(Up,s2)) -> (if ((s1 = s2)&&(occurdummy1sb(s1))) then sig-normsb(s1) else subs) |
(*ShiftCons*) Cp(Up,Pt(e1,sb)) -> (if (occurdummy1(e1) || occurdummy1sb(sb)) then sig-normsb(sb)
                                                                                  else subs) |
(*IdL*) Cp(Id,sb) -> sig-normsb(sb) |
(*IdR*) Cp(sb,Id) -> sig-normsb(sb) |
(*Map*) Cp(Pt(e1,s1),s2) -> (if (occurdummy1(e1) || occurdummy1sb(s1) || occurdummy1sb(s2)) then
                              sig-normsb(Pt(sig-norm(Sb(e1,s2)),sig-normsb(Cp(s1,s2)))) else subs) |
(*Assoc*) Cp(Cp(s1,s2),sb3) ->  (if (occurdummy1sb(s1) || occurdummy1sb(s2) || occurdummy1sb(sb3)) then
                              sig-normsb(Cp(s1,sig-normsb(Cp(s2,sb3)))) else subs) | _ -> subs;;
```

For λs_e, we have $\text{gen}_{\lambda s_e}((\lambda M)\ \Diamond) = M\sigma^1\Diamond$. And the s_e-pseudo-normalization of a λs_e-term, `exp`, is given by the function `se-norm` in Table 14. This pseudo-normalization is simpler than the previous one, since we are dealing with a sole entity and additionally the λs_e rewrite rules preserve, in a certain way, the structure of terms: the symbol $\Diamond$ remains always as last argument of the term to be normalized. As a consequence of this regularity, implementation of the pseudo-normalization is done via unconditional rewrite rules (without premises "`if occurrdumy`"). Clearly, this represents an advantage over the other two calculi.

Table 14: s_e-pseudo-normalization

```
let rec se-norm exp = match exp with Dummy -> Dummy | DB i -> DB i | Vr c->Vr c|S(i,Vr c,Dummy)-> exp |
(*si-dest*) S(i,DB j,Dummy) -> (if j<i then DB j else (if j>i then (DB (j-1)) else P(0,i,Dummy))) |
(*si-app*) S(i,A(e1,e2),Dummy) -> A((se-norm (S(i,e1,Dummy))),(se-norm (S(i,e2,Dummy)))) |
(*si-lambda*) S(i,L(e1),Dummy) -> L(se-norm (S(i+1,e1,Dummy))) |
(*si-si*) S(i,S(j,e1,e2),Dummy)->(if i >= j then S(j,(se-norm(S(i+1,e1,Dummy))),
                                                  (se-norm(S(i-j+1,e2,Dummy))))
                                        else exp) |
(*si-phi*) S(i,P(k,n,e),Dummy)->(if i>=k+n then P(k,n,(se-norm(S(i-n+1,e,Dummy))))
                                       else (if i>k then P(k,n-1,e) else exp)) | _ -> exp;;
```

In λ_{SUSP} this implementation is very similar to the one of $\lambda\sigma$. We have that $\text{gen}_{\lambda_{\text{SUSP}}}((\lambda M)\ \Diamond) = [\![M, 1, 0, (\Diamond, 0) :: nil]\!]$. The function `susp-norm` in Table 15 implements the SUSP-pseudo-normalization of a λ_{SUSP} expression `exp`. Observations done for the `sig-norm` of $\lambda\sigma$ apply for the `susp-norm` of λ_{SUSP}: except for three rules, one step reduction is decided via the `occurdummy`'s check that runs in linear time on the size of `exp`. Rules r_2 and r_3 should be implemented without any `Dummy`. As for $\lambda\sigma$, this implies that other rules than those essential for the propagation of the $\diamond$ symbol may be applied during this pseudo-normalization.

One may think there is a tradeoff because of the inclusion conditionals, but the verification of occurrences of the `Dummy` symbol can be performed simultaneously when solving the matching without additional cost.

Definition 7.1 (ξ-pse-nf implementation of the η-reduction) *For the calculi $\lambda\sigma, \lambda s_e$ and λ_{SUSP} the previously proposed implementation of the η-reduction, that is formulated as the rewrite rule*

$$\lambda(M\ \underline{1}) \longrightarrow_{pse\text{-}nfEta_\xi} N \quad \textit{if} \quad N = \xi\textit{-pse-nf}(gen_{\lambda\xi}(((\lambda M)\ \Diamond), root)) \ \textit{and}\ \Diamond \ \textit{does not occur in}\ N$$

is called the implementation by ξ-pseudo normalization *of the η-reduction, denoted by* pse-nfEta$_\xi$.

Table 15: SUSP-pseudo-normalization

```
let rec susp-norm exp = match exp with Dummy -> Dummy | DB i -> DB i | Vr c  -> Vr c |
(*r1*) Sp(Dummy,i,j,env) -> Dummy               |
(*r2*) Sp(DB i,0,j,Nilen) -> DB (i+j)           |
(*r3*) Sp(DB 1,i,j,Con(Ar(k),env)) -> DB (j-k)  |
(*r4*) Sp(DB 1,i,j,Con(Paar(e1,k),env)) -> (if (occurdummy3 e1) then susp-norm(Sp(e1,0,j-k,Nilen))
                                                                 else exp) |
(*r5*) Sp(DB i,j,k,Con(envt,env)) -> (if((occurdummy3_Et envt) || (occurdummy3_Env env))
                                     then susp-norm(Sp(DB (i-1),j-1,k,env))else exp) |
(*r6*) Sp(A(e1,e2),i,j,env) -> (if ((occurdummy3 e1) || (occurdummy3 e2) || (occurdummy3_Env env))
                                then A(susp-norm(Sp(e1,i,j,env)),susp-norm(Sp(e2,i,j,env))) else exp) |
(*r7*) Sp(L(e1),i,j,env)->(if ((occurdummy3 e1) || (occurdummy3_Env env))
                           then L(susp-norm(Sp(e1,i+1,j+1,Con(Ar(j),env))))  else exp) |_ -> exp;;
```

From the argumentations before the previous definition, one can conclude that the implementation of η-reduction by λs_e-pseudo-normalization is cleaner and more efficient than the corresponding implementations of η-reduction for $\lambda\sigma$ and λ_{SUSP}.

Lemma 7.2 (*pse-nfEta*$_{\mathbf{susp}}$ and *pse-nfEta*$_\sigma$ implementations of the η-reduction are unclean) *The implementations of η-reduction by* SUSP- *and σ-pseudo normalization are unclean.*

PROOF. Observing the pseudo-normalization rules for these two calculi we can see that, for $\lambda\sigma$, the rules named `Clos`, `IdL` and `IdR` must be implemented without conditional as the others, i.e., these rules do not propagate the $\Diamond$ symbol. The justification for this can be found in the third paragraph of Section 7.

An analogous argument is used in the case of λ_{SUSP}. □

Lemma 7.3 (*pse-nfEta*$_{s_e}$ implementation of the η-reduction is clean) *The implementation of η-reduction by s_e-pseudo normalization is clean.*

PROOF. By direct inspection of the pseudo-normalization rules of the λs_e-calculus (Table 15). Note that all applied rules just propagate the $\Diamond$ symbol. □

The following three propositions show the completeness of the implementations of the *Eta* rules based on these pseudo-normalizations, denoted by Eta_ξ for $\xi \in \{\sigma, s_e, \text{SUSP}\}$, restricted for pure lambda terms.

Lemma 7.4 *Let $M \in \Lambda_{dB}$. The σ-pseudo-nf of $M[\underline{1}.\underline{1}[\uparrow].\ldots.\underline{1}[\uparrow^{k-2}].\Diamond[\uparrow^{k-1}].\uparrow^{k-1}]$ gives a term that preserves all occurrences of terms in M corresponding to variables less than k unchanged, replaces all occurrences corresponding to the the k^{th} variable with $\Diamond[\uparrow^{k-1}]$ and decrements by one all occurrences corresponding to variables greater than k.*

PROOF. We use the word variable for occurrences of $\underline{1}[\uparrow^{k-1}]$. By induction on the structure of M:

- $\boldsymbol{M = \underline{n}}$. If $n<k$ then $\underline{1}[\uparrow^{n-1}][\underline{1}.\underline{1}[\uparrow]\ldots\underline{1}[\uparrow^{k-2}].\Diamond[\uparrow^{k-1}].\uparrow^{k-1}] \to_{Clos}$
  $\underline{1}[\uparrow^{n-1}\circ(\underline{1}.\underline{1}[\uparrow]\ldots\underline{1}[\uparrow^{k-2}].\Diamond[\uparrow^{k-1}].\uparrow^{k-1})] \to^{n-2}_{Assoc}$
  $\underline{1}[\uparrow\circ(\uparrow\circ(\ldots(\uparrow\circ(\underline{1}.\underline{1}[\uparrow]\ldots\underline{1}[\uparrow^{k-2}].\Diamond[\uparrow^{k-1}].\uparrow^{k-1}))))] \to^{n-1}_{ShiftCons} \underline{1}[\uparrow^{n-1}]$.

  If $n=k$ then $\underline{1}[\uparrow^{n-1}][\underline{1}.\underline{1}[\uparrow]\ldots\underline{1}[\uparrow^{k-2}].\Diamond[\uparrow^{k-1}].\uparrow^{k-1}]$
  $\to_{Clos} \underline{1}[\uparrow^{n-1}\circ(\underline{1}.\underline{1}[\uparrow]\ldots\underline{1}[\uparrow^{k-2}].\Diamond[\uparrow^{k-1}].\uparrow^{k-1})] \to^{n-2}_{Assoc}$
  $\underline{1}[\uparrow\circ(\uparrow\circ(\ldots(\uparrow\circ(\underline{1}.\underline{1}[\uparrow]\ldots\underline{1}[\uparrow^{k-2}].\Diamond[\uparrow^{k-1}].\uparrow^{k-1}))))] \to^{n-1}_{ShiftCons} \Diamond[\uparrow^{n-1}]$.

  If $n>k$ then $\underline{1}[\uparrow^{n-1}][\underline{1}.\underline{1}[\uparrow]\ldots\underline{1}[\uparrow^{k-2}].\Diamond[\uparrow^{k-1}].\uparrow^{k-1}]$
  $\to_{Clos} \underline{1}[\uparrow^{n-1}\circ(\underline{1}.\underline{1}[\uparrow]\ldots\underline{1}[\uparrow^{k-2}].\Diamond[\uparrow^{k-1}].\uparrow^{k-1})] \to^{n-2}_{Assoc}$
  $\underline{1}[\uparrow\circ(\uparrow\circ(\ldots(\uparrow\circ(\underline{1}.\underline{1}[\uparrow]\ldots\underline{1}[\uparrow^{k-2}].\Diamond[\uparrow^{k-1}].\uparrow^{k-1}))))] \to^{n-1}_{ShiftCons} \underline{1}[\uparrow^{n-1-k}\circ\uparrow^{k-1}] = \underline{1}[\uparrow^{n-2}]$.

- $\boldsymbol{M = (A\ B)}$. Directly by the induction hypothesis.

- $\boldsymbol{M = (\lambda A)}$. Then
  $(\lambda A)[\underline{1}.\underline{1}[\uparrow]\ldots\underline{1}[\uparrow^{k-2}].\Diamond[\uparrow^{k-1}].\uparrow^{k-1}] \to_{Abs} \lambda A[\underline{1}.((\underline{1}.\underline{1}[\uparrow]\ldots\underline{1}[\uparrow^{k-2}].\Diamond[\uparrow^{k-1}].\uparrow^{k-1})\circ\uparrow)] \to^{k}_{Map}$
  $\lambda A[\underline{1}.(\underline{1}[\uparrow].\underline{1}[\uparrow][\uparrow]\ldots\underline{1}[\uparrow^{k-2}][\uparrow].\Diamond[\uparrow^{k-1}][\uparrow].(\uparrow^{k-1}\circ\uparrow))] \to_{Clos} \lambda A[\underline{1}.\underline{1}[\uparrow].\underline{1}[\uparrow^2]\ldots\underline{1}[\uparrow^{k-1}].\Diamond[\uparrow^k].\uparrow^k]$. And by the induction hypothesis we can conclude. □

Proposition 7.5 (Completeness of *pse-nfEta*$_\sigma$**)**
Let $M \in \Lambda_{dB}$*. If* $\lambda(M\ \underline{1}) \to_\eta N$ *then* $\lambda(M\ \underline{1}) \to_{pse\text{-}nfEta_\sigma} N$*.*

PROOF. Here we are interpreting the de Bruijn index $\underline{k}$ in the language of $\lambda\sigma$ as usual by $\underline{1}[\uparrow^{k-1}]$. The proof is by induction on the structure of M.

- $\boldsymbol{M = \underline{n}}$. If $n \neq 1$ then on the one side, $\lambda(\underline{n}\ \underline{1}) \to_\eta \underline{n-1}$. On the other side we have to that $\underline{n}[\Diamond.id] = \underline{1}[\uparrow^{n-1}][\Diamond.id]$ σ-pseudo-normalizes to $\underline{n-1}$. In fact, $\underline{1}[\uparrow^{n-1}][\Diamond.id] \to_{Clos} \underline{1}[\uparrow^{n-1} \circ (\Diamond.id)] \to_{Assoc} \underline{1}[\uparrow^{n-2} \circ (\uparrow \circ (\Diamond.id))] \to_{ShiftCons} \underline{1}[\uparrow^{n-2} \circ (id)] \to_{IdR} \underline{1}[\uparrow^{n-2}] = \underline{n-1}$.
- $\boldsymbol{M = (A\ B)}$. For A and B without occurrences of the free de Bruijn index $\underline{1}$, by the condition for the application of the η-reduction to $(A\ B)$, we have that $\lambda(A\ \underline{1}) \to_\eta A'$ and $\lambda(B\ \underline{1}) \to_\eta B'$, where A' and B' are obtained from A and B by decrementing all the free variables by one. Also, $(A\ B)[\Diamond.id] \to_{App} A[\Diamond.id]\ B[\Diamond.id]$. By the induction hypothesis the σ-pseudo-nf of $A[\Diamond.id]$ and $B[\Diamond.id]$ corresponds respectively to A' and B'.
- $\boldsymbol{M = (\lambda A)}$. A does not own occurrences of terms corresponding to the free de Bruijn index $\underline{2}$. Then $\lambda((\lambda A)\ \underline{1}) \to_\eta \lambda A''$, where A'' is obtained from A by decrementing all its free variables except $\underline{1}$ by one. Thus applying Lemma 7.4 to the term $M[\Diamond.id]$, we obtain the desired result. □

Lemma 7.6 *Let* $M \in \Lambda_{dB}$*. Then the* s_e*-pseudo-nf of* $M\sigma^i\Diamond$ *gives a term obtained from* M *by preserving all free de Bruijn indices less than* i *unchanged, replacing the occurrences of the* i^{th} *free de Bruijn index with* $\varphi_0^i\Diamond$ *and decrementing all the free occurrences of de Bruijn indices greater than* i *by one.*

PROOF. Induction on the structure of M.

- $\boldsymbol{M = \underline{n}}$. If $n < i$ then $\underline{n}\sigma^i\Diamond \to_{\sigma-dest} \underline{n}$. If $n = i$ then $\underline{n}\sigma^i\Diamond \to_{\sigma-dest} \varphi_0^i\Diamond$. If $n > i$ then $\underline{n}\sigma^i\Diamond \to_{\sigma-dest} \underline{n-1}$.
- $\boldsymbol{M = (A\ B)}$. $(A\ B)\sigma^i\Diamond \to_{\sigma-app} (A\sigma^i\Diamond)\ (B\sigma^i\Diamond)$. And by induction hypothesis we can conclude.
- $\boldsymbol{M = (\lambda A)}$. $(\lambda A)\sigma^i\Diamond \to_{\sigma-\lambda} \lambda A\sigma^{i+1}\Diamond$. And by induction hypothesis we can conclude. □

Proposition 7.7 (Completeness of *pse-nfEta*$_{s_e}$**)**
Let $M \in \Lambda_{dB}$*. If* $\lambda(M\ \underline{1}) \to_\eta N$ *then* $\lambda(M\ \underline{1}) \to_{pse\text{-}nfEta_{s_e}} N$*.*

PROOF. Induction on the structure of M.

- $\boldsymbol{M = \underline{n}}$. If $n > 1$ then $\underline{n}\sigma^1\Diamond \to_{\sigma-dest} \underline{n-1}$.
- $\boldsymbol{M = (A\ B)}$. For A and B without free occurrences of the de Bruijn index $\underline{1}$, we have that $\lambda(A\ \underline{1}) \to_\eta A'$ and $\lambda(B\ \underline{1}) \to_\eta B'$, where A' and B' are obtained from A and B by decrementing all their free occurrences of de Bruijn indices by one. Also, $(A\ B)\sigma^1\Diamond \to_{\sigma-app} (A\sigma^1\Diamond)\ (B\sigma^1\Diamond)$, and by the induction hypothesis we have that $(A\sigma^1\Diamond) \to_{Eta_{s_e}} A'$ and $(B\sigma^1\Diamond) \to_{Eta_{s_e}} B'$.
- $\boldsymbol{M = (\lambda A)}$. For A without free occurrences of the de Bruijn index $\underline{2}$, $\lambda((\lambda A)\ \underline{1}) \to_\eta \lambda A''$, where A'' is obtained from A by decrementing all its free de Bruijn indices except $\underline{1}$ by one. Also, $(\lambda A)\sigma^1\Diamond \to_{\sigma-\lambda} \lambda A\sigma^2\Diamond$. Now by Lemma 7.6 we get the desired result. □

Lemma 7.8 *Let* A *and* B *be well-formed* λ_{SUSP}*-terms and* $k \geq 0$*. Then the rm-normalization of the well-formed term* $[\![A, k, k-1, @k-2 :: \ldots :: @0 :: (B, l) :: nil]\!]$ *gives a term by decrementing by one all free de Bruijn indices greater than* k *occurring at* A*, replacing the* k^{th} *free variable of* A *with* B *(actualized according to the context of the term) and keeps unchanged all other free occurrences of de Bruijn indices.*

PROOF. Similar to the proof of Lemma 3.15. □

Proposition 7.9 (Completeness of *pse-nfEta*$_{\textbf{susp}}$**)**
Let $M \in \Lambda_{dB}$*. If* $\lambda(M\ \underline{1}) \to_\eta N$ *then* $\lambda(M\ \underline{1}) \to_{pse\text{-}nfEta_{\text{SUSP}}} N$*.*

PROOF. By induction on the structure of M.

- $\boldsymbol{M = \underline{\mathtt{n}}}$. If $n > 1$ then $[\![\underline{\mathtt{n}}, 1, 0, (\Diamond, 0) :: nil]\!] \rightarrow_{r_5} [\![\underline{\mathtt{n-1}}, 0, 0, nil]\!] \rightarrow_{r_2} \underline{\mathtt{n-1}}$.
- $\boldsymbol{M = (A\ B)}$. Similar to Lemma 7.8 using that $[\![(A\ B), 1, 0, (\Diamond, 0) :: nil]\!] \rightarrow_{r_6} [\![A, 1, 0, (\Diamond, 0) :: nil]\!]$ $[\![B, 1, 0, (\Diamond, 0) :: nil]\!]$ and IH: $[\![A,1,0,(\Diamond,0) :: nil]\!] \rightarrow_{Eta_{\text{SUSP}}} A'$ and $[\![B, 1, 0, (\Diamond, 0) :: nil]\!] \rightarrow_{Eta_{\text{SUSP}}} B'$.
- $\boldsymbol{M = (\lambda A)}$. For A without free occurrences of the de Bruijn index $\underline{2}$, $\lambda((\lambda A)\ \underline{1}) \rightarrow_\eta \lambda A''$, where A'' is obtained from A decrementing by one all its free de Bruijn indices except $\underline{1}$.
  Now use $[\![(\lambda A), 1, 0, (\Diamond,0) :: nil]\!] \rightarrow_{r_7} \lambda[\![A, 2, 1, @0 :: (\Diamond, 0) :: nil]\!]$ and Lemma 7.8. □

8 Future Work and Conclusion

[15, 3] showed that η-reduction is of great interest for adapting substitution calculi ($\lambda\sigma$ and λs_e) for important practical problems like higher order unification. In this paper, we have enlarged the suspension calculus of [37, 34] with an adequate *Eta* rule for η-reduction and showed that this extended suspension calculus, named λ_{SUSP}, enjoys confluence and termination of the associated substitution calculus SUSP (with *Eta*).

Additionally, we used the notion of adequacy of [26] for comparing these three calculi when simulating one step β-reduction. We concluded that $\lambda\sigma$ and $\lambda\xi$ are mutually non comparable for $\xi \in \{s_e, \text{SUSP}\}$ but that λs_e is more adequate than λ_{SUSP} in simulating one step beta-reduction. After all, although $\lambda\sigma$ is a first order calculus and the other two calculi are second order, comparing them is not unfair since the use of (built-in) arithmetic is standard in all modern programming environments. Recently Liang and Nadathur pointed out the importance of having the possibility to combine steps of beta-reduction in practical implementations, which resumes to the ability of the calculus to compose substitutions [31, 35]. This results in natural applications for $\lambda\sigma$ and the suspension calculus in contrast to the λs_e. Consequently, it will be of great importance to study possible adaptations of the λs_e which enable this property. In particular, this would be interesting if the work carried out for λs_e on HOU, can be mapped into the λt [26] which is a calculus à la λs_e but which updates à la $\lambda\sigma$. That is, λt does partial updating, like $\lambda\sigma$ and the suspension calculus, whereas, λs_e does global updating. We leave this for future work.

Moreover, we established the correspondence of these *Eta* rules of the three calculi. This correspondence means that the operational effects of applying these *Eta* rules over pure λ-terms in the three calculi are identical. For the three calculi in question enlarged with adequate eta rules we showed how to implement these eta rules. For the λs_e we build a *clean* implementation of the eta rule, that is, avoiding the application of other rules of the substitution calculi than the ones strictly involved in the verification of the η-redices. And we proved that it is not possible to follow the same approach for the $\lambda\sigma$ and λ_{SUSP}. We proved that these implementations are *complete* in the sense that any η-reduction for dealing with pure λ-terms in de Bruijn notation can be simulated by these *Eta* implementations. For the three treated calculi, the main advantage of our clean eta implementation approach is that it is closer than previous implementations to the operational semantics of the usual η-reduction of the λ-calculus. Additionally, we have pointed out that for λ_{SUSP} as well as for the $\lambda\sigma$-calculus, in these *Eta* implementations, the application of rules not strictly involved with the η-reduction is necessary, but that this is not the case for λs_e. We have also showed that for the former two calculi, conditional rewriting rules whose premises are decided in linear time in the size of the terms in normalization are necessary while for λs_e this is done via non conditional rules whose applicability is decided by simple matching of their left-hand sides. Our *Eta* implementation is being incorporated into an ELAN prototype for simply-typed higher order unification via λs_e.

An immediate work to be done is to study two open questions: whether the s_e-calculus has strong normalization (SN) [27] , and whether λ_{SUSP} preserves SN. Interesting points arise in this context since: λs_e is more adequate in the simulation of one step β-reduction than λ_{SUSP}; λs_e does not preserves SN [18]; and the substitution calculus of λ_{SUSP} has SN.

References

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

[2] M. Ayala-Rincón and F. Kamareddine. On Applying the λs_e-Style of Unification for Simply-Typed Higher Order Unification in the Pure lambda Calculus. In *Pre-Proceedings Eighth Workshop on Logic, Language, Information and Computation (WoLLIC 2001)*, pages 41–54, 2001.

[3] M. Ayala-Rincón and F. Kamareddine. Unification via the λs_e-Style of Explicit Substitution. *The Logic Journal of the Interest Group in Pure and Applied Logics*, 9(4):489–523, 2001.

[4] M. Ayala-Rincón and C. Muñoz. Explicit Substitutions and All That. *Revista Colombiana de Computación*, 1(1):47–71, 2000.

[5] F. Baader and T. Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[6] H. Barendregt. *The Lambda Calculus : Its Syntax and Semantics (revised edition)*. North Holland, 1984.

[7] Z.-el-A. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a Calculus of Explicit Substitutions which Preserves Strong Normalization. *J. of Functional Programming*, 6(5):699–722, 1996.

[8] Z.-el-A. Benaissa, P. Lescanne, and K. H. Rose. Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution. In *Programming Languages: Implementations, Logics and Programs PLILP'96*, volume 1140 of *LNCS*, pages 393–407. Springer, 1996.

[9] R. Bloo. *Preservation of Termination for Explicit Substitution*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1997.

[10] R. Bloo and K. Rose. Combinatory Reduction Systems with Explicit Substitution that Preserve Strong Normalisation. In H. Ganzinger, editor, *Seventh International Conference on Rewriting Techniques and Applications (RTA-96)*, volume 1103 of *LNCS*, pages 169–183. Springer-Verlag, 1996.

[11] P. Borovanský. Implementation of Higher-Order Unification Based on Calculus of Explicit Substitutions. In M. Bartošek, J. Staudek, and J. Wiedermann, editors, *Proceedings of the SOFSEM'95: Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 363–368. Springer Verlag, 1995.

[12] D. Briaud. An explicit Eta rewrite rule. In *Typed lambda calculi and applications*, volume 902 of *LNCS*, pages 94–108. Springer, 1995.

[13] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence Properties of Weak and Strong Calculi of Explicit Substitutions. *J. of the ACM*, 43(2):362–397, 1996. Also as *Rapport de Recherche* INRIA 1617, 1992.

[14] N. G. de Bruijn. Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indag. Mat.*, 34(5):381–392, 1972.

[15] G. Dowek, T. Hardin, and C. Kirchner. Higher-order Unification via Explicit Substitutions. *Information and Computation*, 157(1/2):183–235, 2000.

[16] M. C. F. Ferreira, D. Kesner, and L. Puel. Lambda-Calculi with Explicit Substitutions and Composition which Preserve Beta-Strong Normalisation. In *Algebraic and Logic Programming, ALP'96*, volume 1139 of *LNCS*, pages 284–298. Springer, 1996.

[17] B. Guillaume. *Un calcul des substitutions avec etiquettes*. PhD thesis, Université de Savoie, Chambéry, 1999.

[18] B. Guillaume. The λs_e-calculus Does Not Preserve Strong Normalization. *J. of Functional Programming*, 10(4):321–325, 2000.

[19] T. Hardin. Eta-conversion for the languages of explicit substitutions. In *Algebraic and logic programming*, volume 632 of *LNCS*, pages 306–321. Springer, 1992.

[20] T. Hardin, L. Maranget, and B. Pagano. Functional runtime systems within the lambda-sigma calculus. *J. of Functional Programming*, 8(2):131–176, 1998.

[21] F. Kamareddine and R. P. Nederpelt. On stepwise explicit substitution. *International Journal of Foundations of Computer Science*, 4(3):197–240, 1993.

[22] F. Kamareddine and R. P. Nederpelt. A useful λ-notation. *Theoretical Computer Science*, 155:85–109, 1996.

[23] F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with Explicit Substitutions. In *Proc. of PLILP'95*, volume 982 of *LNCS*, pages 45–62. Springer, 1995.

[24] F. Kamareddine and A. Ríos. Extending a λ-calculus with Explicit Substitution which Preserves Strong Normalisation into a Confluent Calculus on Open Terms. *J. of Functional Programming*, 7:395–420, 1997.

[25] F. Kamareddine and A. Ríos. Bridging de Bruijn indices and variable names in explicit substitutions calculi. *The Logic Journal of the Interest Group of Pure and Applied Logic*, 6(6):843–874, 1998.

[26] F. Kamareddine and A. Ríos. Relating the $\lambda\sigma$- and λs-Styles of Explicit Substitutions. *Journal of Logic and Computation*, 10(3):349–380, 2000.

[27] F. Kamareddine and A. Ríos. Explicit Substitutions à la de Bruijn: the local and global way. Electronic Notes in Theoretical Computer Science, 85(7), 2003.

[28] F. Kamareddine, A. Ríos, and J.B. Wells. Calculi of Generalised β-reduction and explicit substitution: Type Free and Simply Typed Versions. *J. of Functional and Logic Programming*, 1998(Article 5):1–44, 1998.

[29] D. Kesner. Confluence of extensional and non-extensional λ-calculi with explicit substitutions. *Theoretical Computer Science*, 238(1-2):183–220, 2000.

[30] P. Lescanne and J. Rouyer-Degli. Explicit substitutions with de Bruijn's levels. In J. Hsiang, editor, *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA-95)*, volume 914 of *LNCS*, pages 294–308, Chapel Hill, North Carolina, 1995. Springer-Verlag.

[31] C. Liang and G. Nadathur. Tradeoffs in the Intensional Representation of Lambda Terms. In S. Tison, editor, *Rewriting Techniques and Applications (RTA 2002)*, volume 2378 of *LNCS*, pages 192–206. Spinger-Verlag, 2002.

[32] L. Magnusson. *The implementation of ALF - a proof editor based on Martin Löf's Type Theory with explicit substitutions.* PhD thesis, Chalmers, 1995.

[33] C. Muñoz. *Un calcul de substitutions pour la représentation de preuves partielles en théorie de types.* PhD thesis, Université Paris 7, 1997. English version in *Rapport de recherche* INRIA RR-3309, 1997.

[34] G. Nadathur. A Fine-Grained Notation for Lambda Terms and Its Use in Intensional Operations. *J. of Functional and Logic Programming*, 1999(2):1–62, 1999.

[35] G. Nadathur. The Suspension Notation for Lambda Terms and its Use in Metalanguage Implementations. In *Proceedings Ninth Workshop on Logic, Language, Information and Computation (WoLLIC 2002)*, volume 67 of *ENTCS*. Elsevier Science Publishers, 2002.

[36] G. Nadathur and D. Miller. An Overview of λProlog. In K.A. Bowen and R.A. Kowalski, editors, *Proc. 5th Int. Logic Programming Conference*, pages 810–827. MIT Press, 1988.

[37] G. Nadathur and D. S. Wilson. A Notation for Lambda Terms A Generalization of Environments. *TCS*, 198:49–98, 1998.

[38] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected papers on Automath.* North-Holland, 1994.

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

[40] R. Vestergaard and J.B. Wells. Cut Rules and Explicit Substitutions. *Mathematical Structures in Computer Science*, 11(1):131–168, 2001.