# SUBSEXPL: a tool for simulating and comparing explicit substitutions calculi[1]

**F. L. C. de Moura**[†] [*] — **M. Ayala-Rincón**[‡] [*] — **F. Kamareddine**[**]

**1** *Work supported by funds from CNPq (CT-INFO) 50.6598/04-7.*
† *Supported by the Brazilian Ministry Educational Council CAPES.*
‡ *Partially supported by the Brazilian Research Council CNPq.*
[*] *Departamento de Matemática, Universidade de Brasília*
*70900-910 Brasília D.F., Brasil*
{flaviomoura,ayala}@unb.br
[**] *School of Mathematical and Computer Sciences*
*Heriot-Watt University, Edinburgh EH14 4AS, Scotland*
fairouz@macs.hw.ac.uk

ABSTRACT. *We present the system SUBSEXPL used for simulating and comparing explicit substitutions calculi. The system allows the manipulation of expressions of the $\lambda$-calculus and of three different styles of explicit substitutions: the $\lambda\sigma$, the $\lambda s_e$ and the suspension calculus. A variation of the suspension calculus, which allows for combination of steps of $\beta$-contraction is included too. Implementations of the $\eta$-reduction are provided for each style. Other explicit substitutions calculi can be easily incorporated into the system due to its modular structure. The uses of the system include: the visualization of the contractions of the $\lambda$-calculus in de Bruijn notation, and of guided one-step reductions as well as normalization via each of the associated substitution calculi. Many useful facilities are included: reductions can be easily recorded as text files, Latex outputs can be generated and several examples for dealing with arithmetic operations and computational operators such as conditionals and repetitions in the $\lambda$-calculus are available. The system can be executed over Emacs using the x-symbol mode in such a way that $\lambda$-terms and terms of the explicit substitutions calculi are represented in its natural syntax avoiding the necessity of repeatedly generating Latex outputs. The system has been of great help for systematically comparing explicit substitutions calculi, as well as for understanding properties of explicit substitutions such as the Preservation of Strong Normalization. In addition, it has been used for teaching basic properties of the $\lambda$-calculus such as: computational adequacy, the usefulness of de Bruijn's notation and of making explicit substitutions in real implementations.*

KEYWORDS: *$\lambda$-Calculus, Explicit Substitutions, Visualization of $\beta$- and $\eta$-Contraction and Normalization.*

## 1. Introduction

In the last decade, a number of explicit substitutions calculi have been developed. Most of these calculi have been claimed to be useful in practical fields such as in the implementation of typed functional programming languages and of higher-order proof assistants. We describe SUBSEXPL, a system developed in Ocaml, a language of the ML family; the system allows for the manipulation of expressions of the $\lambda$-calculus and of four different calculi of explicit substitutions:

1) $\lambda\sigma$ [ABA 91] which introduces two different sets of entities: one for `terms` and one for `substitutions`.

2) $\lambda s_e$ [KAM 97] which is based on the philosophy of de Bruijn's *Automath* [NED 94] elaborated in the new *item notation* [KAM 96]. In this framework, a term is a sequence of *items*, which can be an *application item*, an *abstraction item*, a *substitution item* or an *updating item*. The advantages of building the explicit substitutions calculus in this framework include remaining as close as possible to the familiar $\lambda$-calculus (cf. [KAM 00]).

3) The suspension calculus [NAD 99], which introduces three different sets of entities: `terms`, `environments` and `lists of environments`.

4) The "combining suspension calculus" [NAD 03, LIA 04], which is a refinement of the suspension calculus, that allows for combinations of steps of $\beta$-contraction.

Each of these different calculi has advantages and disadvantages. Although various attempts have been made at comparing these styles (cf. [AYA 05, KAM 00]), a lot remains to be explained. A better understanding of the similarities and differences of these styles may lead on the one hand to solving the remaining open questions related to the various calculi (such as preservation of strong normalization, subject reduction, etc.), and on the other hand, to a more inclusive calculus and implementations which combine the advantages in one system. The inclusion of other calculi of explicit substitutions is also possible and details are given in the documentation provided with the source code of the system, where all the necessary steps are explained.

Through SUBSEXPL, we attempt to understand the working of the rewrite rules of these calculi. We developed a full scale Ocaml implementation of the four calculi where contractions in all these calculi (as well as in the type-free $\lambda$-calculus) can be visualized in a step-wise fashion and where the behavior of the reduction paths can be analyzed. Especially, we concentrate on the one-step guided reductions and normalization via each of the associated *substitution calculi*. Although the implementation of rewriting rules is straightforward in a rewriting based language such as ELAN and Maude, we prefer to use a language of the ML family because of its natural ability to control the matching which allows for the selection of redexes before contractions are carried out.

SUBSEXPL has been successfully used for teaching our students basic properties of the $\lambda$-calculus such as: computational adequacy, the usefulness of de Bruijn's notation [BRU 72] for dealing with collisions and clashes avoiding the necessity of $\alpha$-

conversion and of making explicit substitutions in real implementations based on the $\lambda$-calculus. SUBSEXPL has also been of great importance for systematically comparing these four calculi of explicit substitutions.

Furthermore, SUBSEXPL includes adequate implementations of the rules of $\eta$-reduction for the three former calculi as well as a *clean* implementation for the $\lambda s_e$-calculus (cf. [AYA 05]) in the sense that no other rewriting rules than the ones strictly involved in Eta-contraction[1] are included in one-step Eta-contraction. Work on higher-order unification (HOU) in $\lambda\sigma$ and $\lambda s_e$ established the importance of combining Eta-reduction or contraction (as well as expansion) with explicit substitutions (cf. [DOW 00, AYA 01]). This has provided extensions of $\lambda\sigma$ and $\lambda s_e$ with Eta-reduction rules also referred to as $\lambda\sigma$ and $\lambda s_e$. Eta reduction as well as expansion is necessary for working with functions and programs, since one needs to express functional or extensional equality; i.e., when the application of two $\lambda$-terms to the same term yields the same result, then they should be considered equal. This led to various extensions of explicit substitutions calculi with an Eta-rule even before this was applied to HOU [HAR 92, RÍO 93, BRI 95, KES 00].

Input/output of $\lambda$-terms in SUBSEXPL is a difficult point because $\lambda$-expressions are difficult to write correctly and after some contractions they may become big very quickly. Inputs are given to the system in an internal syntax close to the ones of the $\lambda$-calculus in de Bruijn notation; for example, `L(_)` and `A(_,_)` stay for abstraction and application, respectively. Other symbols of the treated explicit substitutions calculi are represented in this internal syntax similarly (see the grammatical description provided with the system). Outputs are given in the same syntax, except when using the x-symbol[2] mode of the Emacs editor for which translation to symbolic notation is provided automatically. Also, in order to ease reading the outputs of the system, we provided Latex outputs which can be generated during any step of the derivations and, moreover, the generated file can be easily edited according to the user's requirement.

SUBSEXPL has been used as a tool for understanding properties of explicit substitutions calculi. Desired properties of an explicit substitutions calculus include:

(a) Simulation of one step $\beta$-reduction: whenever $a$ reduces to $b$ in the $\lambda$-calculus using one step $\beta$-reduction, we have that $a$ reduces to $b$ in the explicit substitutions calculus using one step of the explicit $\beta$-reduction (starting rule) and the substitution rules.

(b) Confluence (CR): confluence is the property that establishes that reductions do not depend on reduction strategies or in other words, that whenever a term can be reduced in two different ways, the obtained terms can be *joined* by rewriting into a common term. CR is considered for two classes of terms:

(b.1) Ground terms: these are the usual terms of the $\lambda$-calculus built from variables, applications and abstractions.

1. We use the Greek letter $\eta$ to refer only to the "$\eta$-rule" of the pure $\lambda$-calculus, and its name "Eta" to refer to the corresponding rules in the explicit substitutions calculi.
2. `http://x-symbol.sf.net`

(b.2) Open terms: in this case, the language of the explicit substitutions calculus is expanded with a new class of variables, known as meta-variables. In this setting, open terms can be seen as contexts and meta-variables as place-holders. Open terms are essential in higher-order unification and matching algorithms that use explicit substitutions [DOW 00, AYA 01, MOU 05].

(c) Strong normalization (SN) of the underlying calculus of explicit substitutions: this is the termination property of the explicit substitutions calculi without the explicit $\beta$-reduction rule; i.e., without the rule that starts the simulation of $\beta$-reduction.

(d) Preservation of SN (PSN): whenever all possible reductions starting from a pure $\lambda$-term terminate in the $\lambda$-calculus, there are no possible infinite reductions starting from this term in the explicit substitutions calculus.

Without Eta, $\lambda\sigma$ satisfies (a), (b.1), (c) and satisfies (b.2) only when the set of open terms is restricted to those which admit meta-variables of sort `terms`. Without Eta, $\lambda s$ satisfies (a)..(d) but not (b.2). However, $\lambda s$ has an extension $\lambda s_e$ (again without Eta) for which (a), (b.1) and (b.2) hold, but (d) fails and no answer to (c) is known. The suspension calculus (which does not have Eta) satisfies (a) and when restricted to well formed terms it also satisfies (b.1), (b.2) and (c), but (d) is unknown (cf. [KAM 00, NAD 98]). As a refinement of the suspension calculus, the combining suspension calculus inherits all its properties allowing, in addition, for the combination of steps of $\beta$-contraction.

SUBSEXPL has been used as a tool for examining the PSN property of two of the three calculi we consider. The system allows us to follow the counter-examples of Melliès ([MEL 95]) and Guillaume ([GUI 00]) for proving that neither the $\lambda\sigma$- nor the $\lambda s_e$-calculus preserves SN. In addition, important computational properties such as the possibility of combining $\beta$-contractions in $\lambda\sigma$ and $\lambda s_e$ have been formalized with the help of this tool.

In section 2 we describe briefly the implemented calculi of explicit substitutions in order to give some flavour of the object of study in SUBSEXPL. In section 3 we briefly describe the system and its usage and, before concluding in section 5, in section 4 we illustrate the applications of the system.

## 2. Preliminaries

Since the proposal of this work is to describe a tool that deals with explicit substitutions calculi, we briefly describe the syntax and give some intuition on the implemented calculi. For a more detailed description we refer the reader to the references given at the beginning of the introduction.

Explicit substitutions calculi are given in order to make explicit the operation of substitution, which is traditionally given in the formalization of the $\lambda$-calculus as a *meta-operation* instead of being an explicitly defined operation. These calculi attempt to be closer to implementations of the $\lambda$-calculus than the $\lambda$-calculus itself. SUBSEXPL deals with the $\lambda\sigma$-calculus, which is historically important as the first calcu-

lus of this kind; with the suspension calculus which is well-known as the underlying calculus of $\lambda$PROLOG; with a simple extension of the later calculus, that allows for combinations of $\beta$-contractions; and with the $\lambda s_e$-calculus which has a completely different philosophy from the previous ones introducing arithmetic constraints in order to propagate substitutions explicitly. The philosophies of the $\lambda\sigma$ and of the suspension calculi (and its extension) are similar because these calculi represent substitutions as lists of terms, which are simultaneously propagated over the body of expressions. For doing this, these calculi need to introduce in their syntax other sorts of objects than terms. The $\lambda\sigma$ introduces a sort of substitutions and the suspension calculus introduces two new sorts of objects: environments and environment-terms. In contrast, the $\lambda s_e$ maintains a sole sort of objects (terms) as the $\lambda$-calculus does, and uses two operators together with arithmetic constraints to propagate substitutions.

### 2.1. *The $\lambda\sigma$-Calculus*

The $\lambda\sigma$-calculus, introduced in [ABA 91], is given by a first-order rewriting system, which makes substitutions explicit by extending the language with two sorts of objects: **terms** and **substitutions**.

DEFINITION 1. — *The syntax of the $\lambda\sigma$-calculus is given by:*

| | |
|---|---|
| ***Terms*** | $a, b ::= \underline{1} \mid (a\ b) \mid \lambda a \mid a[s]$ |
| ***Substitutions*** | $s ::= id \mid \uparrow \mid a \cdot s \mid s \circ s$ |

Substitutions are lists of terms: for example, the list $a_1 \cdot a_2 \cdot \ldots \cdot a_n \cdot \uparrow^n$ represents the substitution $\{\underline{1}/a_1, \underline{2}/a_2, \ldots, \underline{n}/a_n\}$ indicating that the de Bruijn index $\underline{j}$ must be replaced by the term $a_j$, for $j = 1..n$. The substitution $id$, a shorthand for lists of the form $\underline{1} \cdot \underline{2} \cdot \underline{n} \ldots \uparrow^n$, for $n \geq 0$, represents the identity substitution. $\uparrow$ is the substitution $\{\underline{i}/\underline{i+1}\}$ that increases by one every free de Bruijn index $\underline{i}$ in the term where it is applied. $s \circ s$ represents the composition of substitutions. Notice that de Bruijn indexes are codified in the language of the $\lambda\sigma$-calculus. In fact, the term $\underline{1}[\uparrow^n]$, where $n \in \mathbb{N}^*$, codifies the de Bruijn index $\underline{n+1}$. Notice that de Bruijn indexes are discriminated from naturals by being underlined. The term $a[b \cdot id]$ starts the simulation of the $\beta$-reduction of $(\lambda a\ b)$ in the $\lambda\sigma$-calculus. Thus, in addition to the replacement of the free occurrences of the index $\underline{1}$ by the corresponding term, free occurrences of indices should be actualized (decreased) because of the elimination of the abstractor. Table 1 includes the rewriting system of the $\lambda\sigma$-calculus augmented with an Eta rule for $\eta$-reduction, as presented in [DOW 00].

This system without (Eta) is equivalent to the one presented in [ABA 91] originally. The associated substitution calculus, denoted as $\sigma$, is the one induced by all the rules except (Beta) and (Eta), and its equality is denoted as $=_\sigma$.

| | | | |
|---|---|---|---|
| $(\lambda a\ b)$ | $\longrightarrow$ | $a[b \cdot id]$ | $(Beta)$ |
| $(a\ b)[s]$ | $\longrightarrow$ | $(a[s]\ b[s])$ | $(App)$ |
| $\underline{1}[a \cdot s]$ | $\longrightarrow$ | $a$ | $(VarCons)$ |
| $a[id]$ | $\longrightarrow$ | $a$ | $(Id)$ |
| $(\lambda a)[s]$ | $\longrightarrow$ | $\lambda(a[\underline{1} \cdot (s \circ \uparrow)])$ | $(Abs)$ |
| $(a[s])[t]$ | $\longrightarrow$ | $a[s \circ t]$ | $(Clos)$ |
| $id \circ s$ | $\longrightarrow$ | $s$ | $(IdL)$ |
| $\uparrow \circ (a \cdot s)$ | $\longrightarrow$ | $s$ | $(ShiftCons)$ |
| $(s_1 \circ s_2) \circ s_3$ | $\longrightarrow$ | $s_1 \circ (s_2 \circ s_3)$ | $(AssEnv)$ |
| $(a \cdot s) \circ t$ | $\longrightarrow$ | $a[t] \cdot (s \circ t)$ | $(MapEnv)$ |
| $s \circ id$ | $\longrightarrow$ | $s$ | $(IdR)$ |
| $\underline{1} \cdot \uparrow$ | $\longrightarrow$ | $id$ | $(VarShift)$ |
| $\underline{1}[s] \cdot (\uparrow \circ s)$ | $\longrightarrow$ | $s$ | $(Scons)$ |
| $\lambda(a\ \underline{1})$ | $\longrightarrow$ | $b$ if $a =_\sigma b[\uparrow]$ | $(Eta)$ |

**Table 1.** *The rewriting system for the $\lambda\sigma$-calculus*

### 2.2. *The $\lambda s_e$-Calculus*

In contrast to the $\lambda\sigma$-calculus, the $\lambda s_e$-calculus, introduced in [KAM 97], has a sole sort of objects maintaining a closer syntax to the $\lambda$-calculus. The $\lambda s_e$-calculus introduces two operators $\sigma$ and $\varphi$, for substitution and updating, respectively.

DEFINITION 2. — *The syntax of the $\lambda s_e$-calculus is given by:*

***Terms*** $a, b ::= \underline{n} \mid (a\ b) \mid \lambda a \mid a\,\sigma^i b \mid \varphi_k^j\, a$, *where* $n, i, j \in \mathbb{N}^*$ *and* $k \in \mathbb{N}$.

The term $a\,\sigma^i b$ represents the term $a\{\underline{i}/b\}$; i.e., the substitution of the free occurrences of $\underline{i}$ in $a$ by $b$, updating the free variables in $a$ (and in $b$). The term $\varphi_k^j\, a$ represents $j-1$ applications of the $k$-lift to the term $a$; i.e., $a^{+k^{(j-1)}}$. Table 2 contains the rewriting rules of the $\lambda s_e$-calculus augmented with the rule (Eta), as introduced in [AYA 01]. $=_{s_e}$ denotes the equality for the associated substitution calculus, denoted as $s_e$, induced by all the rules except ($\sigma$-generation) and (Eta).

### 2.3. *The Suspension Calculus*

The suspension calculus [NAD 98, NAD 99] deals with $\lambda$-terms as computational mechanisms. This was motivated by implementational questions related to $\lambda$Prolog, a logic programming language that uses typed $\lambda$-terms as data structures [NAD 88]. The suspension calculus works with three different types of entities:

| | | | |
|---|---|---|---|
| **Terms** | $a, b$ | ::= | $c \mid \underline{n} \mid \lambda a \mid (a\ b) \mid [\![a, i, j, e_1]\!]$ |
| **Environments** | $e_1, e_2$ | ::= | $nil \mid et :: e_1 \mid \{\!\{e_1, i, j, e_2\}\!\}$ |
| **Environment Terms** | $et$ | ::= | $@i \mid (a, i) \mid \langle\!\langle et, i, j, e_1 \rangle\!\rangle$ |

| | | | |
|---|---|---|---|
| $(\lambda a\ b)$ | $\longrightarrow$ | $a\,\sigma^1 b$ | ($\sigma$-generation) |
| $(\lambda a)\,\sigma^i b$ | $\longrightarrow$ | $\lambda(a\,\sigma^{i+1} b)$ | ($\sigma$-$\lambda$-transition) |
| $(a_1\ a_2)\,\sigma^i b$ | $\longrightarrow$ | $((a_1\,\sigma^i b)\ (a_2\,\sigma^i b))$ | ($\sigma$-app-transition) |
| $\underline{n}\,\sigma^i b$ | $\longrightarrow$ | $\begin{cases} \underline{n-1} & \text{if } n > i \\ \varphi_0^i\, b & \text{if } n = i \\ \underline{n} & \text{if } n < i \end{cases}$ | ($\sigma$-destruction) |
| $\varphi_k^i\,(\lambda a)$ | $\longrightarrow$ | $\lambda(\varphi_{k+1}^i\, a)$ | ($\varphi$-$\lambda$-transition) |
| $\varphi_k^i\,(a_1\ a_2)$ | $\longrightarrow$ | $((\varphi_k^i\, a_1)\ (\varphi_k^i\, a_2))$ | ($\varphi$-app-transition) |
| $\varphi_k^i\,\underline{n}$ | $\longrightarrow$ | $\begin{cases} \underline{n+i-1} & \text{if } n > k \\ \underline{n} & \text{if } n \leq k \end{cases}$ | ($\varphi$-destruction) |
| $(a_1\,\sigma^i a_2)\,\sigma^j b$ | $\longrightarrow$ | $(a_1\,\sigma^{j+1} b)\,\sigma^i(a_2\,\sigma^{j-i+1} b)$ if $i \leq j$ | ($\sigma$-$\sigma$-transition) |
| $(\varphi_k^i\, a)\,\sigma^j b$ | $\longrightarrow$ | $\varphi_k^{i-1}\, a$ if $k < j < k+i$ | ($\sigma$-$\varphi$-transition 1) |
| $(\varphi_k^i\, a)\,\sigma^j b$ | $\longrightarrow$ | $\varphi_k^i\,(a\,\sigma^{j-i+1} b)$ if $k+i \leq j$ | ($\sigma$-$\varphi$-transition 2) |
| $\varphi_k^i\,(a\,\sigma^j b)$ | $\longrightarrow$ | $(\varphi_{k+1}^i\, a)\,\sigma^j(\varphi_{k+1-j}^i\, b)$ if $j \leq k+1$ | ($\varphi$-$\sigma$-transition) |
| $\varphi_k^i\,(\varphi_l^j\, a)$ | $\longrightarrow$ | $\varphi_l^j\,(\varphi_{k+1-j}^i\, a)$ if $l+j \leq k$ | ($\varphi$-$\varphi$-transition 1) |
| $\varphi_k^i\,(\varphi_l^j\, a)$ | $\longrightarrow$ | $\varphi_l^{j+i-1}\, a$ if $l \leq k < l+j$ | ($\varphi$-$\varphi$-transition 2) |
| $\lambda(a\ \underline{1})$ | $\longrightarrow$ | $b$ if $a =_{s_e} \varphi_0^2\, b$ | $(Eta)$ |

**Table 2.** *The rewriting system of the $\lambda s_e$-calculus*

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

Rather than performing adjustments at each stage, the suspension calculus performs the adjustments into a substitution term only at the final substitution stage. Intuitively, a suspended term of the form $[\![a, i, j, e_1]\!]$ means that the first $i$ variables of the $\lambda$-term $a$ 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 $a$ used to appear within $i$ abstractions but now appears within $j$ of them.

The suspension calculus has a *generation* rule $\beta_s$, which initiates the simulation of a $\beta$-reduction (in a similar way to the corresponding rules of $\lambda\sigma$ and $\lambda s_e$, namely, the *Beta* and the $\sigma$*-generation* rules) 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 3 augmented with an Eta rule for the $\eta$-reduction, as presented in [AYA 05] in the so-called $\lambda_{\text{SUSP}}$-calculus.

## 3. Description of SUBSEXPL

SUBSEXPL is an open source software which runs over GNU/Linux platforms and is available at: `http://ayala.mat.unb.br/TCgroup/SUBSEXPL/`

In the following subsections we describe the implementation and use of the system.

| | | |
|---|---|---|
| $(\beta_s)$ | $(\lambda a\ b)$ | $\longrightarrow [\![a, 1, 0, (b, 0) :: nil]\!]$ |
| $(r_1)$ | $[\![c, ol, nl, e]\!]$ | $\longrightarrow c$, where $c$ is a constant |
| $(r_2)$ | $[\![\underline{i}, 0, nl, nil]\!]$ | $\longrightarrow \underline{i+nl}$ |
| $(r_3)$ | $[\![\underline{1}, ol, nl, @l :: e]\!]$ | $\longrightarrow \underline{nl\text{-}l}$ |
| $(r_4)$ | $[\![\underline{1}, ol, nl, (a, l) :: e]\!]$ | $\longrightarrow [\![a, 0, (nl\text{-}l), nil]\!]$ |
| $(r_5)$ | $[\![\underline{i}, ol, nl, et :: e]\!]$ | $\longrightarrow [\![\underline{i\text{-}1}, (ol\text{-}1), nl, e]\!]$, for $i > 1$ |
| $(r_6)$ | $[\![(a\ b), ol, nl, e]\!]$ | $\longrightarrow ([\![a, ol, nl, e]\!]\ [\![b, ol, nl, e]\!])$ |
| $(r_7)$ | $[\![\lambda\, a, ol, nl, e]\!]$ | $\longrightarrow \lambda\, [\![a, (ol + 1), (nl + 1), @nl :: e]\!]$ |
| $(m_1)$ | $[\![[\![a, ol_1, nl_1, e_1]\!], ol_2, nl_2, e_2]\!]$ | $\longrightarrow [\![a, ol', nl', \{\!\{e_1, nl_1, ol_2, e_2\}\!\}]\!]$, 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 $nl = m + 1$ |
| $(m_9)$ | $\langle\!\langle (a, nl), nl, ol, et :: e\rangle\!\rangle$ | $\longrightarrow ([\![a, ol, l', et :: e]\!], m)$, where $l' = \mathit{ind}(et)$ 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 $nl \neq \mathit{ind}(et)$ |
| $(Eta)$ | $\lambda\, (a\ \ \underline{1})$ | $\longrightarrow b$, if $a =_{rm} [\![b, 0, 1, nil]\!]$ |

**Table 3.** *Rewriting rules of the suspension calculus*

### 3.1. *Description of the system: design and implementation*

Since in SUBSEXPL we want to provide the user with full control of reduction we decided to use Ocaml instead a rewriting programming environment such as ELAN or Maude. This decision was made because during the process of derivation of an expression in any language of explicit substitutions we want to follow sequentially the following sub-tasks:

1) detect matches of the rewriting rules of our explicit substitution calculi;

2) allow the user to freely select the desired redex (rule and position) to be applied;

3) contract the expression according to the selection of the user.

This work is done in a natural way in Ocaml by pattern matching through the `match _ with` structure that allows to match left-hand sides of the rewriting rules with the current expression in a natural way. Most importantly, when these matches are detected one can select the desired effect: either simply report on the detection of the redex (sub-task 1 above) or contract a previously detected redex (sub-task 3 above). With Ocaml we have full control on the reduction process in this natural way.

In contrast with rewriting computational environments, doing these sub-tasks requires some additional programming work because the philosophy of rewriting languages is to detect matches of the left-hand sides of the rules in order to apply the matched rules immediately. To change this in rewriting based environments, one can use some alternatives such as logical strategies in ELAN (to control the application of rewriting rules), but we consider this to be artificial since we would like to provide the user with full control over derivations.

Then for each calculus included in SUBSEXPL, it is essentially necessary to describe its syntax, to detect its redexes and to apply the rewriting rules.

The syntax of the terms of $\lambda\sigma$, $\lambda s_e$ and the suspension calculus (and its extension) are implemented in SUBSEXPL as follows:

1) $\lambda\sigma$-terms of the form $\underline{1}$, $\lambda a$, $(a\ b)$ and $a[s]$ are respectively represented as `One`, `L(a)`, `A(a,b)` and `Sb(a,s)`; $\lambda\sigma$-substitutions of the form $id$, $\uparrow$, $a \cdot s$ and $s \circ t$ are respectively represented as `Id`, `Up`, `Pt(a,s)` and `Cp(s,t)`.

2) $\lambda s_e$-terms of the form $\underline{n}$, $\lambda a$, $(a\ b)$, $a\sigma^i b$ and $\varphi_k^i a$, where $k \geq 0$ and $i \geq 1$ are respectively represented as `n`, `L(a)`, `A(a,b)`, `S(i,a,b)` and `P(i,k,a)`.

3) For the suspension calculus, $\lambda_{\text{SUSP}}$-terms of the form $\underline{n}$, $\lambda a$, $(a\ \ b)$ and $[\![a, i, j, e]\!]$ are respectively represented as `n`, `L(a)`, `A(a,b)` and `Sp(a,i,j,e)`; $\lambda_{\text{SUSP}}$-environments of the form $nil$, $et\ ::\ e$ and $\{\!\{e_1, i, j, e_2\}\!\}$ as `Nil`, `Con(et,e)` and `Ck(e1,i,j,e2)`; $\lambda_{\text{SUSP}}$-environment terms of the form $@i$, $(a, i)$ and $\langle\!\langle et, i, j, e\rangle\!\rangle$ are respectively represented as `Ar i`, `Paar(a,i)` and `LG(et,i,j,e)`.

4) Since the combining suspension calculus is a refinement of the suspension calculus, its terms are essentially $\lambda_{\text{SUSP}}$-term as above without environments of the form $\{\!\{e_1, i, j, e_2\}\!\}$ and without environment terms of the form $\langle\!\langle et, i, j, e_1\rangle\!\rangle$.

For example, the $\lambda s_e$-term $(\lambda\underline{1})\sigma^1(\underline{2}\ \underline{3})$ is represented in the internal language of SUBSEXPL as `S(1,L(1), A(2,3))`.

The current structure of the system is intended to allow easy inclusion of new calculi. The general structure of the system is represented in Figure 1. A stepwise description on how to include a new calculus can be found in the tutorial (see also the file `adding-a-new-calculi`) distributed with the source code of the system. A more detailed description of the dependencies of the files in the current implementation is given in Figure 2.

For each implemented calculus, there are two main parts: *matching* and *reduction*. The *matching* part is responsible for detecting and reporting all the existing redexes, for each rewriting rule of the calculus, of the current $\lambda$-term and to add the positions of these redexes into a list. The functions responsible for this work are implemented in the files `sematch`$\square$`.ml`, where $\square$ ranges over `ls, lse, sus` and `suscomb` for $\lambda\sigma$, $\lambda s_e$, the suspension calculus and its extension, respectively. The files responsible for the *reduction* part are similarly called `sered`$\square$`.ml`.

As an example of the implementation of matching and reduction, we partially present in tables 4 and 5 the corresponding functions for the rule *($\sigma$-generation)* of

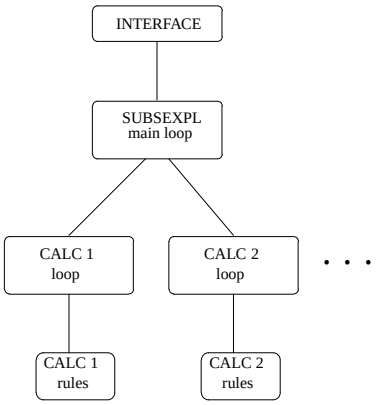


**Figure 1.** *General structure of the system SUBSEXPL*

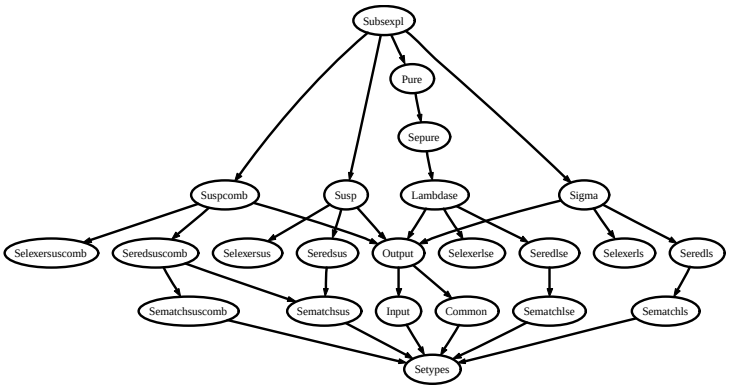


**Figure 2.** *File dependencies for the four calculi currently treated by SUBSEXPL*

$\lambda s_e$. The three parameters of the function `matchingSgeneration` in Table 4: `exp`, `l` and `pos` are for the current expression, the list of positions and the current position (w.r.t the main expression), respectively. One can observe in Table 4 the use of the `match _ with` structure of Ocaml for pattern matching occurrences of the left-hand side of the $\sigma$-generation rule $(\lambda e1 \quad e2) \longrightarrow e1\,\sigma^1 e2$ that is given in the grammar of SUBSEXPL by `A(L(e1),e2)`. Once the redex is detected, the current position `pos` is included in the list of redexes for the rule $\sigma$-generation. This list of redexes is maintained in the parameter `l`. Other possible matches are detected by subsequent searches inside the structure of the current expression; for instance, when the current expression is of the form $\lambda e_1$ or equivalently `L(e1)`, no new redex is detected (the list `l` remains unchanged) and the search for redexes should follow through the subexpression `e1` and the current position `pos` should be modified by following the branch 1 of the term $\lambda e_1$, that corresponds to the sub-term $e_1$:

```
        (matchingSgeneration e1 l (List.append pos [1])).
```

`(List.append pos [1])`, simply adds 1 to the current position.

```
let rec matchingSgeneration exp l pos =
  match exp with
    | DB i -> l
    | A(L(e1),e2) -> pos :: List.append
          (matchingSgeneration e1 l (List.append pos [1;1]))
          (matchingSgeneration e2 [] (List.append pos [2]))
    | A(e1,e2) -> List.append
          (matchingSgeneration e1 l (List.append pos [1]))
          (matchingSgeneration e2 [] (List.append pos [2]))
    | L(e1) -> (matchingSgeneration e1 l (List.append pos [1]))
    | S(i,e1,e2) -> List.append
          (matchingSgeneration e1 l (List.append pos [1]))
          (matchingSgeneration e2 [] (List.append pos [2]))
    | P(j,k,e1) -> (matchingSgeneration e1 l (List.append pos [1]))
    | _ -> assert false
```

**Table 4.** *Implementation of matching for the rule $\sigma$-generation of the $\lambda s_e$*

The *reduction* part implemented for the $\sigma$-generation rule is the function called `reductionSgeneration` presented in Table 5. This function needs only two parameters: `exp` for the expression and `pos` for the position (given as before as a sequence/list of naturals). By recursively applying the `match _ with` structure the position is deconstructed and simultaneously the corresponding sub-term of the expression reached. This is achieved when the current position is the empty list (see the subsection 3.3 for the notation of positions as sequences of naturals). Once the selected sub-term is reached, it has to be of the form `A(L(e1),e2)` (equivalently $(\lambda e_1\ e_2)$) and then, it is reduced into `S(1,e1,e2)` (or equivalently $e_1\sigma^1 e_2$). This works correctly, because all the matches were detected and the possible selection of redexes given by the user is restricted to the set of the detected redexes.

### 3.2. *The implementation of Eta contraction*

SUBSEXPL includes implementations of Eta contraction for each of the calculi of explicit substitutions treated here (the Eta-rule of the suspension calculus and its refinement coincide). The implementation follows the notion of cleanness as defined in [AYA 05]. The intuitive idea of a *clean* Eta implementation is that it does not mix isolated applications of Eta-reduction with applications of rules of the corresponding substitution calculi other than the ones strictly involved in Eta-reduction. Clean implementations of Eta allow us to reach good simulations of Eta-contraction, which implies the possibility of combining steps of Beta and Eta contraction.

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

**Table 5.** *Implementation of reduction for the rule $\sigma$-generation of the $\lambda s_e$*

The suspension calculus did not originally have an Eta-rule. In [AYA 05] this calculus was enlarged with an adequate Eta-rule in the so-called $\lambda_{\text{SUSP}}$-calculus. For the enlarged calculi $\lambda_{\text{SUSP}}$, $\lambda s_e$ and $\lambda\sigma$ we showed that there exists a correspondence among their Eta-rules which means that, when applied to pure $\lambda$-terms, these rules behave similarly (cf. [AYA 05]).

Neither the suspension calculus nor the $\lambda\sigma$-calculus has completely clean implementations of the Eta-rule. In fact, in these calculi, the implementation of the Eta-rule requires the application of some rewriting rules, not directly related to Eta contraction, but which are necessary to normalize some simple terms. Nevertheless, our implementation of the Eta-rule for $\lambda s_e$ is clean.

Eta-reduction is important to computational problems that arise in applications of the $\lambda$-calculus. For instance, $\eta$-reduction is useful in the treatment of higher-order unification and matching via explicit substitutions calculi (cf. [DOW 00, AYA 01]).

### 3.3. *Using the system*

In this section we briefly describe how SUBSEXPL can be used inside Emacs with the x-symbol mode, which provides a symbolic representation of $\lambda$-expressions. SUBSEXPL can be run from a shell in Emacs with the TeX macro mode of x-symbol activated (for details see the README file of the distribution).

As a complete example, we show how to operate with Church's numerals (cf. [BAR 84]) whose description can be found in the file `Examples` distributed with the source code. Consider the reduction $A_+C_1C_1 \rightarrow_\beta^6 C_2$, which evaluates "$1 + 1$" in the $\lambda$-calculus, where $A_+ = \lambda xypq.((x\ p)((y\ p)\ q))$ represents the sum operator, and

$C_1 = \lambda fx.fx$ is the Church numeral corresponding to 1. The $A_+$ operator is written in de Bruijn notation as $A_+ = \lambda\lambda\lambda.((\underline{4}\ \underline{2})((\underline{3}\ \underline{2})\ \underline{1}))$ which is translated to the SUBSEXPL language as `L(L(L(L(A(A(4,2),A(A(3,2),1))))))`.

For discriminating sub-terms, occurrences and redexes in terms, we use the standard notation from rewriting theory [BAA 98]. In this notation terms are represented as trees and each sub-term is referenced according to its position given by the sequence of naturals determining the branches of the tree one needs to follow (starting from the root of the tree) in order to reach the *root* node of the selected sub-term. For instance, the set of valid positions of the $\lambda$-term $A_+ = \lambda xypq.((x\ p)((y\ p)\ q))$ is given by $\{\varepsilon$ $1, 1.1,\ 1.1.1,\ 1.1.1.1,\ 1.1.1.1.1,\ 1.1.1.1.1.1,\ 1.1.1.1.1.2,\ 1.1.1.1.2,\ 1.1.1.1.2.1,$ $1.1.1.1.2.1.1,\ 1.1.1.1.2.1.2,\ 1.1.1.1.2.2\}$; its sub-term at root position $\varepsilon$ (the empty sequence) is the term itself, the sub-term at position $1.1.1$ is $\lambda q.((x\ p)((y\ p)\ q))$, its sub-term at position $1.1.1.1.2.1$ is $(y\ p)$, $p$ occurs at positions $1.1.1.1.1.2$ and $1.1.1.1.2.1.2$, etc. SUBSEXPL omits the concatenation symbol " . ", because the treated expressions have only operators of arity at most two. Notice that this notation is also adequate for $\lambda$-terms in de Bruijn notation.

Applying the operator $A_+$ to add the Church numeral $C_1$ to itself gives the expression corresponding to $A_+C_1C_1$ in the SUBSEXPL grammar:

```
A(A(L(L(L(L(A(A(4,2),A(A(3,2),1))))))),L(L(A(2,1)))),L(L(A(2,1))))
```

Figure 3 shows the initial screen of the system where one can either select one of the available calculi or select the option for a grammatical description of the system. The example in this figure shows a direct reduction in the pure $\lambda$-calculus of the above term using the leftmost/outermost normalization strategy followed by the history of the derivation that lists all the intermediate terms obtained during the reduction.

The available options for this example are as follows:

1. `Beta:` Enumerates all the positions of the current term in which $\beta$-redexes occur.
2. `Eta:` Enumerates all the positions of the current term in which $\eta$-redexes occur.
3. `Leftmost/outermost normalization`: normalizes the given term choosing always the leftmost redex.
4. `Rightmost/innermost normalization`: normalizes the given term choosing always the rightmost redex.
5. `Back one step`: allows the user to return to the previous step in the current derivation.
6. `See history`: shows in the current screen the list of all expressions generated in the current reduction.
7. `Latex Output`: generates automatically a file with the latex code of the current reduction and display the .dvi file on the screen[3]
8. `Save current reduction`: allows the user to save the current reduction into a simple text file, say `my-reduction`. To load this reduction in a further section with the tool, the user should restart the system giving this file as argument:

3. We assume that the running system has latex and xdvi installed.

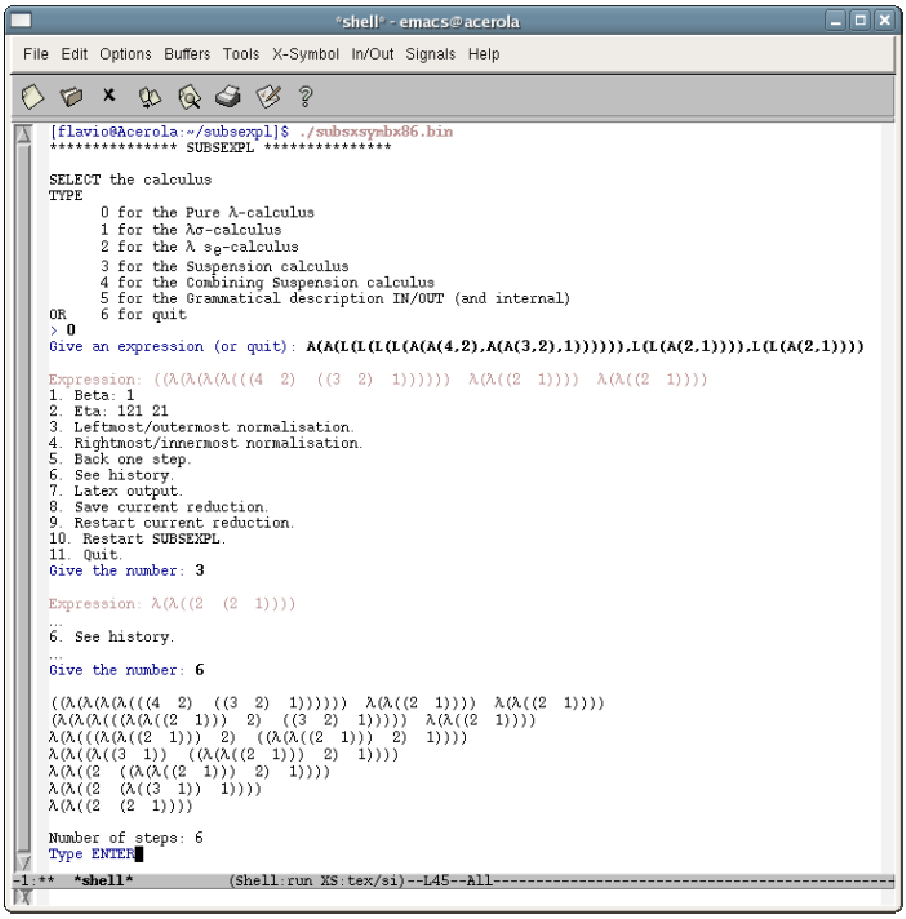


**Figure 3.** *Running an example*

`./subsexpl.bin my-reduction`.

9. `Restart current reduction`: allows the user to restart the current reduction from the beginning after asking if the user wants to save the current reduction.

10. `Restart SUBSEXPL`: restarts the system after asking if the user wants to save the current reduction.

11. `Quit`: halts the system after asking if the user wants to save the current reduction.

The generation of the latex output is an important option that is available even during the intermediate steps of a derivation. Note that in the latex output, all the reduced redexes appear underlined (remember that de Bruijn indexes are also underlined, but it is clear that they are not redexes). In Figure 4 we show the latex output generated by SUBSEXPL that corresponds to the reduction $A_+C_1C_1 \rightarrow_\beta^6 C_2$.

$$(\underline{((\lambda(\lambda(\lambda((\ \underline{4}\ \underline{2})((\ \underline{3}\ \underline{2})\ \underline{1})))))(\lambda(\lambda(\ \underline{2}\ \underline{1}))))}(\lambda(\lambda(\ \underline{2}\ \underline{1})))) \rightarrow_\beta$$
$$((\lambda(\lambda(\lambda(\underline{((\lambda(\lambda(\ \underline{2}\ \underline{1})))\ \underline{2})}((\ \underline{3}\ \underline{2})\ \underline{1})))))(\lambda(\lambda(\ \underline{2}\ \underline{1})))) \rightarrow_\beta$$
$$((\lambda(\lambda(\lambda(\underline{(\lambda(\ \underline{3}\ \underline{1}))((\ \underline{3}\ \underline{2})\ \underline{1})})))))(\lambda(\lambda(\ \underline{2}\ \underline{1})))) \rightarrow_\beta$$
$$\underline{((\lambda(\lambda(\lambda(\ \underline{2}((\ \underline{3}\ \underline{2})\ \underline{1})))))(\lambda(\lambda(\ \underline{2}\ \underline{1}))))} \rightarrow_\beta$$
$$(\lambda(\lambda(\ \underline{2}(\underline{((\lambda(\lambda(\ \underline{2}\ \underline{1})))\ \underline{2})\ \underline{1})})))) \rightarrow_\beta$$
$$(\lambda(\lambda(\ \underline{2}\underline{((\lambda(\ \underline{3}\ \underline{1}))\ \underline{1})})))) \rightarrow_\beta$$
$$(\lambda(\lambda(\ \underline{2}(\ \underline{2}\ \underline{1}))))$$

**Figure 4.** *Latex output generated by SUBSEXPL*

An interesting exercise is to simulate the derivation of Figure 4 step by step using $\lambda\sigma$, $\lambda s_e$ or the two versions of the suspension calculus. The current implementation has two normalization strategies available for simulating one full step of $\beta$-reduction in a explicit substitutions calculi: the leftmost strategy or the strategy according to the order in which the rules are given on the screen of each calculi (we call this strategy 'random'). An interesting fact is that the first step of $\beta$-reduction in this derivation, when simulated in the $\lambda\sigma$-calculus using the random normalization strategy, generates some huge $\lambda\sigma$-terms which exceeds the available memory for the latex compilation. In fact, the simulation of the first $\beta$-reduction in the $\lambda\sigma$-calculus using the 'random' strategy is done in 236 steps, while the same simulation using the leftmost strategy is performed in only 45 steps! The complete reduction using the leftmost strategy generated about 3 full pages of latex output with small fonts. In $\lambda s_e$ as well as in the suspension calculus, both strategies generate the output within about 2 pages.

Terms with internal operators of the explicit substitutions calculi may be given as input: as an example, take the $\lambda\sigma$-term $((\lambda\underline{1})\ \underline{1}[\uparrow])[\underline{1} \cdot id]$ which is written in SUBSEXPL as `Sb(A(L(1),Sb(One,Up)),Pt(One,Id))`. A partial reduction of this term is given in Figure 5.

## 4. Applications

SUBSEXPL has been successfully used to teach computational notions of the $\lambda$-calculus as well as to compare and understand some properties of explicit substitutions calculi. In this way, SUBSEXPL can be seen as a tool with both educational and research purposes. In this section we start by explaining how the system can be used for educational purposes exploring some computability notions over the $\lambda$-calculus. Afterwards, we explain how it can be used to compare calculi of explicit substitutions according to the computational effort necessary to simulate one step of $\beta$-reduction and finally we show how SUBSEXPL can be used to follow the counter-examples of Melliès and Guillaume that establish that the $\lambda\sigma$- and the $\lambda s_e$-calculus, respectively, do not preserve strong normalization.

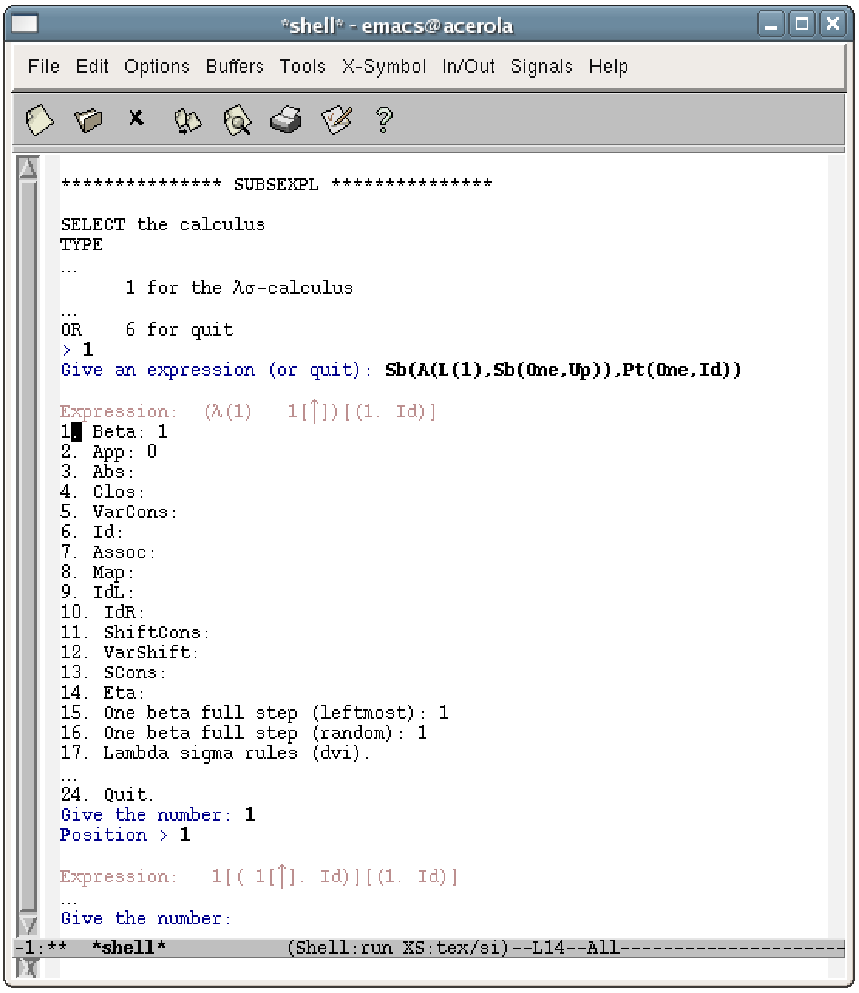


**Figure 5.** *An example for the λσ-calculus*

### 4.1. *Understanding the λ-calculus and its implementations*

We have used SUBSEXPL to explain to students questions related to the computational adequacy of the $\lambda$-calculus, the problems which arise from the usual notation with symbolic variables and the implicit notion of substitution. The main application of the tool is for persuading students about the computational advantages of the mechanics involved in the use of de Bruijn notation [BRU 72] for the treatment of $\alpha$-conversion. In fact, it is well-known that the $\lambda$-calculus in de Bruijn notation avoids $\alpha$-conversion. We consider this is a simple fact to be explained, but an unconvincing one because the $\lambda$-calculus *per se* is notationally complex and this complexity

increases when indexes rather than variable names are used. Consequently, without a good explanation students tend to start their implementations using symbolic variables, which in our opinion is not efficient. For making this point clear, notice that in implementations of the $\lambda$-calculus with names (i.e., with symbolic variables) collisions and clashes are treated by introducing fresh variable names. Because of limitations of the (symbolic) alphabet of programming languages, these new names are usually introduced by variable names with natural numbers as subscripts that are increased when new fresh variables are needed. This may produce complex intermediate outputs: Suppose, one has used $254$ different fresh variables during a computation and one needs to $\beta$-contract the $\lambda$-term with names $\lambda y.(\lambda xy.(x\ y)\ \ y)$, then capture of the bound variable $y$ is avoided by renaming it with the next available fresh variable and one obtains a term such as $\lambda y y_{255}.(y\ y_{255})$. The computational cost of these renamings is linear in the size of terms and the worst is that this may happen in all steps of a computation. In de Bruijn notation collisions are automatically avoided: the equivalent term in de Bruijn notation is given by $\lambda(\lambda\lambda(\underline{2}\ \underline{1})\ \ \underline{1})$ and it $\beta$-contracts to $\lambda\lambda(\underline{2}\ \underline{1})$ by simple updating of de Bruijn indexes. The notation with de Bruijn indexes is surely hard to be read by humans, but here we are analyzing implementations of the $\lambda$-calculus.

Other notations different from that of de Bruijn, e.g., notations based on *nominal syntax* [URB 04], are also useful for dealing with the previously mentioned problems in practical contexts. In nominal syntax, names occurring in terms may be captured within the scope of binders upon substitution by naming explicitly bound entities. In this way one has a form of substitution that preserves $\alpha$-equivalence. Also, approaches based on *director strings*, which internalize the information needed about free variables, are useful in this context. Director strings are used to implement closed reduction [FER 05a], which captures the call-by-value and call-by-name evaluation strategies in the $\lambda$-calculus with names, providing an efficient notion of reduction. However, in the setting of explicit substitutions, de Bruijn notation is of principal interest.

By using SUBSEXPL, the computational expressiveness of the $\lambda$-calculus can be illustrated by examples which range from the $\lambda$-representation of arithmetic operations such as addition, multiplication and exponentiation over Church's numerals to the $\lambda$-representation of basic data structures which include booleans, computational commands and operators such as if-then-else, iteration and recursion. All this was done in the spirit of [BAR 84].

As a concrete example, we consider an expression for computing the factorial function. This simple exercise takes a lot of effort, because students are neither familiar with the notation nor with the operational semantics of the $\lambda$-calculus. But implementing this class of exercises is necessary because this gives the real flavour of the computational power of the $\lambda$-calculus. By using SUBSEXPL over Emacs we can very quickly implement these functions: we start creating abbreviations for the needed operators and inserting the corresponding term in the system; afterwards, we

compound these operators and functions in order to complete the desired function. The factorial function is implemented by defining an iteration operator $T_H$ given by:

$$\lambda p.\langle S^+(p\ \texttt{true}), H(p\ \texttt{true})(p\ \texttt{false})\rangle$$

where $S^+$ is the successor function, i.e., $S^+ = A_+C_1$ and $H$ is a convenient function that depends on the function to be implemented. The result of applying $T_H$ to $\langle C_i, C_{f(i)}\rangle$ is the pair $\langle C_{i+1}, C_{f(i+1)}\rangle$, where $f$ references the function implemented by the iteration mechanism, the first component of the pair is a counter for the iteration step and the second one is the value of the desired function at that step.

4.1.1. *Abbreviations*

1) The Church numbers are as given before;

2) The booleans `true` and `false` correspond to the $\lambda$-terms `L(L(2))` and `L(L(1))`, respectively.

3) $\langle M, N\rangle$ represents the pair operator which is given, in the language of SUBSEXPL, by the $\lambda$-term `L(A(A(1,M),N))`. Pairs can be applied to booleans, written as $\langle M, N\rangle$`true` and $\langle M, N\rangle$`false` and the normal form of these terms are M and N, respectively.

4) For the case of the factorial function, the adequate operator `T` is given as $T_H$ above where $H$ is selected as $\lambda xy.(A_*\ y\ (S^+x))$ and $A_* = \lambda xyz.(x(y\ z))$ is the multiplication operator of Church numerals that corresponds to the term $\lambda\lambda\lambda(\underline{3}(\underline{2}\ \underline{1}))$ in de Bruijn notation. It is easy to see that this operator satisfies the property: `T`$\langle C_k, C_{k!}\rangle$ $\beta$-reduces to $\langle C_{k+1}, C_{(k+1)!}\rangle$, and so, applying repeatedly this mechanism we are counting the number of iteration in the first component of the pair and computing the associated value of the factorial in second one.

In the language of SUBSEXPL, the normal form of the operator `T` is given by:

```
L(L(A(A(1,L(L(A(2,A(A(A(4,L(L(2))),2),1))))),
L(A(A(3,L(L(1))),L(A(2,A(A(A(4,L(L(2))),2),1)))))))
```

4.1.2. *Checking parts of an implementation*

This step is useful for testing the functionality of parts of the intended implementation which allows inferring the functionality of the whole specification. For

instance, we can check that $\mathtt{T}\langle C_2, C_{2!}\rangle$ reduces to $\langle C_3, C_{3!}\rangle$. The SUBSEXPL syntax for $\mathtt{T}\langle C_2, C_{2!}\rangle$ is given by:

$$\mathtt{T}\langle C_2, C_{2!}\rangle \left\{ \begin{array}{c} \mathtt{T} \left\{ \begin{array}{c} A( \\ L(L(A(A(1, L(L(A(2, A(A(A(4, L(L(2))), 2), 1))))), \\ L(L(A(A(A(4, L(L(1))), 2), A(A(A(4, L(L(2))), \\ A(A(4, L(L(1))), 2)), 1))))))) \end{array} \right. \\ \langle C_2, C_{2!}\rangle \left\{ \begin{array}{c} L(A( \\ A(1, \\ \underbrace{L(L(A(2, A(2, 1))))}_{C_2} \\ ), \\ \underbrace{L(L(A(2, A(2, 1))))}_{C_2} \\ )) \end{array} \right. \\ ) \end{array} \right.$$

By $\beta$-normalization this part of the implementation can be checked. After normalization one obtains $\langle C_3, C_{3!}\rangle$ or equivalently `L(A(A(1,L(L(A(2,A(2,A(2,1))))))),L(L(A(2,A(2,A(2,A(2,A(2,A(2,1))))))))))`.

The repetition mechanism is completed by applying $n$ times the iteration operator starting from the pair $\langle C_0, C_{0!}\rangle$. This is done by the term:

$$A(A(C_n, \mathtt{T}), \langle C_0, C_{0!}\rangle) \tag{1}$$

which reduces to $\langle C_n, C_{n!}\rangle$.

The functionality of all the parts of the desired mechanism/function can be checked by normalization with SUBSEXPL.

### 4.1.3. *Final function*

Once enough tests have been run over SUBSEXPL, the factorial function can be written as:

$$\underbrace{L(A(\underbrace{A(A(1, \mathtt{T}), \langle C_0, C_{0!}\rangle)}_{\text{Match with term (1)}}, \underbrace{L(L(1))}_{\mathtt{false}}))}_{\text{Selection of the } 2^{nd} \text{ element of the pair}} \tag{2}$$

The term (2), when applied to the Church numeral $C_n$, $\beta$-reduces to $C_{n!}$. In fact, such an application will generate a $\beta$-redex in the root of the new term. By reducing this term, one obtains a term with sub-term (1). And this term has already been shown to reduce to the pair $\langle C_n, C_{n!}\rangle$. To get the desired result we need to select the second element of this pair which is done by applying it to `false`, as previously explained.

In the syntax of SUBSEXPL (which corresponds to that of the $\lambda$-calculus) the expression for factorial is given by (see the file `Examples` distributed with the source code of the system):

```
L(A(A(A(1,L(L(A(A(1,L(L(A(2,A(A(A(4,L(L(2))),2),1))))),
L(L(A(A(A(4,L(L(1))),2),A(A(A(4,L(L(2))),
A(A(4,L(L(1))),2)),1)))))))),L(A(A(1,L(L(1))),
L(L(A(2,1))))))),L(L(1))))
```

Similarly, other functions can be easily implemented. In fact, notice that from this construction it is easy (also for students) to infer that the sole thing to be changed in the whole repetition mechanism is the function $H$ in the definition of the iteration operator $T_H$. For instance, for computing the function $\sum_{i=0}^{n} i$, $H$ should be replaced by $\lambda xy.A_+\ y\ (S^+x)$; for computing the function $\sum_{i=0}^{n} i^2$, $H$ should be replaced by $\lambda xy.A_+y(A_*(S^+x)(S^+x))$; etc.

We believe that this kind of experiments is necessary and useful for obtaining some flavor of the computational power of the $\lambda$-calculus. A way to speed-up the generation of non elementary implementations is by using our system jointly with an editor for creating the necessary abbreviations, cutting, pasting and testing for modular constructions of "programs" or functions. In intelligent editors such as Emacs, these abbreviations can be easily incorporated in new buttons and short-cut keys, which makes the quick construction of these functions possible. Some of these experiments are included in the file `Examples`. In the current distribution, we included the use of the x-symbol mode, which allows the visualization of $\lambda$-expressions graphically over the Emacs environment directly. This makes the repeated generation and compilation of the Latex code unnecessary. Of course, there are other possibilities to help students in the understanding of the computational expressiveness of the $\lambda$-calculus, which include modularity support (e.g., naming, parameterization) in the functionality of the system itself, but instead, we opt for allowing the modular construction of elaborated functions explicitly by hand over an intelligent editor in which SUBSEXPL can be run. Our experience shows that this way more computer science engineering students are able to take the $\lambda$-calculus more seriously as a truly effective computational environment, rather than as an arid computational model.

The problem of having an implicit notion of substitution involves a complex implementational question because this is not a first-order operation. In fact, in the $\lambda$-calculus one can have variables of functional sort. The comprehension of the necessity of making substitution an explicit operation is realized only when students are asked to implement $\beta$-contraction. After illustrating the computational adequacy of the $\lambda$-calculus, problems inherent to its implementation may be easily pointed out: collisions, confusion, renaming of variables, etc. Then students realize that substitution is a meta-operation that must be carefully defined in any correct implementation of the $\lambda$-calculus and are able to truly understand the beauty and usefulness of notational solutions such as de Bruijn's indexes and the importance of explicit substitutions calculi.

### 4.2. *Comparing calculi by the simulation of $\beta$-reduction*

SUBSEXPL has been implemented with the intention of comparing different styles of explicit substitutions with respect to the effort necessary to simulate one-step $\beta$-reduction. With the help of this tool we were able to study and compare derivation examples which allow us to provide proofs of the fact that $\lambda s_e$ is more efficient than the suspension calculus and is incomparable to the $\lambda\sigma$-calculus in the simulation of one-step $\beta$-reduction [AYA 05]. The efficiency of $\lambda s_e$ is justified by the fact that the manipulation of de Bruijn indexes in $\lambda s_e$ is directly related to a built-in manipulation of natural numbers and arithmetic (which is standard in today's computational environments and programming languages) whereas in the other two calculi, this is done constructively. Of course this comparison is interesting, but not conclusive since $\lambda s_e$ is not completely adequate for combining steps of $\beta$-reduction, which is more natural in $\lambda_{\text{SUSP}}$ [LIA 02, NAD 02]. The possibility of refining the $\lambda\sigma$ for combining $\beta$-contractions and the impossibility of doing this in $\lambda s_e$ was formalized with the help of SUBSEXPL in [FER 05b] recently. But we believe this has to be investigated more carefully, since some variations of $\lambda s_e$ like $\lambda t$ ([KAM 00]), which is a calculus à la $\lambda s_e$ but which updates like $\lambda\sigma$, may allow this combination in the $\lambda\sigma$ style.

### 4.3. *Understanding properties of explicit substitutions*

SUBSEXPL has been used as a tool for understanding properties of explicit substitutions calculi. This is illustrated by examining the property of Preservation of Strong Normalization (PSN).

To illustrate the use of SUBSEXPL in understanding properties of explicit substitution calculi, we explain how one can follow(/check) papers which prove some properties of these calculi. In particular, we follow the proofs that PSN neither holds for $\lambda\sigma$ nor for $\lambda s_e$ given in [MEL 95] and [GUI 00], respectively. By examining these counter-examples in SUBSEXPL, firstly, one can animate the generation of an infinite derivation in the associated substitution calculi starting from a well typed term of the pure $\lambda$-calculus. Secondly, one can try to generate infinite derivations of $\beta$-reductions from these $\lambda$-terms, concluding that this is impossible. The latter is performed without necessarily knowing that there are no infinite ($\beta$-)derivations in the $\lambda$-calculus starting from well typed terms. In this way it is possible to simultaneously understand the importance of the PSN property as well as why it does not hold in these two calculi.

#### 4.3.1. *The counter-example of Melliès*

Melliès's counter-example for the $\lambda\sigma$-calculus consists of an infinite derivation starting from the well typed pure $\lambda$-term $\lambda((\lambda(\lambda\underline{1})((\lambda\underline{1})\underline{1}))((\lambda\underline{1})\underline{1}))$. The corresponding term in the language of SUBSEXPL is given by:

```
L(A(L(A(L(1),A(L(1),1))),A(L(1),1))).
```

The infinite reduction is generated by applying an adequate strategy which mixes rules of the associated calculus $\sigma$ with the rule **Beta** which initiates the simulation of one step $\beta$-reduction. The whole derivation, with the usual grammar of the $\lambda\sigma$-calculus, is given at the end of this subsection according to the numbering of steps given in the following tables. In this derivation, the key sub-terms, which give rise to the infinite derivation, are labeled with under-brackets.

| STEP | RULE | POSITION | STEP | RULE | POSITION |
|---|---|---|---|---|---|
| 1 | 1 | 111 | 3 | 4 | 1 |
| 2 | 1 | 1 | 4 | 8 | 12 |

The term `L(Sb(1,Cp(Pt(A(L(1),1),Id),Pt(A(L(1),1),Id))))`, in the internal notation of SUBSEXPL, or equivalently $(\lambda\underline{1}[((((\lambda\underline{1})\underline{1})\cdot id)\circ(((\lambda\underline{1})\underline{1})\cdot id))])$, in the language of the $\lambda\sigma$, is obtained after the third step.

Let us define recursively:

$s_1$ = `Pt(A(L(1),1),Id)` $(\equiv \ (((\lambda\underline{1})\underline{1})\cdot id))$

$s_2$ = `Cp(Up,Pt(Sb(1,`$s_1$`),Id))` $(\equiv \ (\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))$

= `Cp(Up,Pt(Sb(1,Pt(A(L(1),1),Id)),Id))`

$s_3$ = `Cp(Up,Pt(Sb(1,`$s_2$`),Id))`

= `Cp(Up,Pt(Sb(1,Cp(Up,Pt(Sb(1,Pt(A(L(1),1),Id)),Id))),Id))`

$(\equiv \ (\uparrow\circ(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)))$

...

$s_i$ = `Cp(Up,Pt(Sb(1,`$s_{i-1}$`)),Id))`

With this definition, we can write the current term as `L(Sb(1,Cp(`$s_1$`,`$s_1$`)))`. Notice that $s_i$ works as an abbreviation instead of an actual input to the system. But in SUBSEXPL it is always possible to deal with variable names without any problem. At this point, applying the Map transition at position 12, the sub-term $s_1$ is duplicated. And we get `L(Sb(1,Pt(Sb(A(L(1),1),` $s_1$`),Cp(Id,`$s_1$`))))` (or equivalently, $(\lambda\underline{1}[(((\lambda\underline{1})\underline{1})[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(id\circ(((\lambda\underline{1})\underline{1})\cdot id)))])$). Note that the second occurrence of $s_1$ is vacuous, in the sense that it can be easily eliminated by the rule VarCons. The key idea of Melliès is to maintain this second occurrence of $s_1$ and to propagate the first occurrence as follows:

| STEP | RULE | POSITION | STEP | RULE | POSITION |
|---|---|---|---|---|---|
| 5 | 2 | 121 | 7 | 3 | 1211 |
| 6 | 9 | 122 | | | |

One reaches $(\lambda\underline{1}[(((\lambda\underline{1}[(\underline{1}\cdot((((\lambda\underline{1})\underline{1})\cdot id)\circ\uparrow))])\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)])\cdot(((\lambda\underline{1})\underline{1})\cdot id))])$, or equivalently `L(Sb(1,Pt(A(L(Sb(1,Pt(1,Cp(`$s_1$`,Up))))),Sb(1,`$s_1$`)),`$s_1$`)))`, and again we can apply the Beta rule and then compose the two substitutions:

| STEP | RULE | POSITION | STEP | RULE | POSITION |
|---|---|---|---|---|---|
| 8 | 1 | 121 | 9 | 4 | 121 |

The next 3 steps duplicate the sub-term `Pt(Sb(1,Pt(A(L(1),1),Id)),Id)` and generate the term $s_2$ = `Cp(Up,Pt(Sb(1,Pt(A(L(1),1),Id)),Id))` which have inside an occurrence of $s_1$ (see the under-bracketed terms in the derivation at the end of this subsection):

| STEP | RULE | POSITION | STEP | RULE | POSITION |
|---|---|---|---|---|---|
| 10 | 8 | 1212 | 12 | 7 | 12122 |
| 11 | 5 | 12121 | | | |

At this point,
`L(Sb(1,Pt(Sb(1,Pt(Sb(1,`$s_1$`),Cp(`$s_1$`,`$s_2$`))),`$s_1$`)))`, or
$(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underbrace{((((\lambda\underline{1})\underline{1})\cdot id)}_{s_1}\circ\underbrace{(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))))}_{s_2})]\cdot(((\lambda\underline{1})\underline{1})\cdot id))])$
becomes the current term. It contains an occurrence of `Cp(`$s_1$`,`$s_2$`)`. By repeating the same sequence of rules we get a term with the sub-term `Cp(`$s_2$`,`$s_3$`)`.

| STEP | RULE | POSITION | STEP | RULE | POSITION |
|---|---|---|---|---|---|
| 13 | 8 | 12122 | 18 | 4 | 121221 |
| 14 | 2 | 121221 | 19 | 8 | 1212212 |
| 15 | 9 | 121222 | 20 | 5 | 12122121 |
| 16 | 3 | 1212211 | 21 | 7 | 12122122 |
| 17 | 1 | 121221 | | | |

Here, it is easy (at least easier than following the proof with paper and pencil!) to see how an infinite reduction can be built from the initial well typed term in the $\lambda\sigma$ calculus of explicit substitutions. The mechanism of this *cyclicity* may not be completely evident after the 21 steps presented here, but with SUBSEXPL the reader can further expand this infinite derivation until it is clarified. In Table 6, we give the corresponding reduction generated in Latex format by SUBSEXPL.

The steps presented in this example are stored in the file `mellies` distributed with the source code of SUBSEXPL and can be executed automatically with the command: `./subsexpl.bin mellies`.

In this case, the output dvi file automatically generated is `mellies-ls.dvi`. Note that the notation $s_1$, $s_2$, $s_3$ and the numeration of the steps are used here for ease of reading but is not automatically generated in the above `dvi` file. The latex code of the output can be found in the file `mellies-ls`.

4.3.2. *The counter example of Guillaume*

In [GUI 00], Guillaume showed that the $\lambda s_e$-calculus does not preserve strong normalization. This is realized by an infinite derivation starting from the well typed pure $\lambda$-term in de Bruijn's notation:

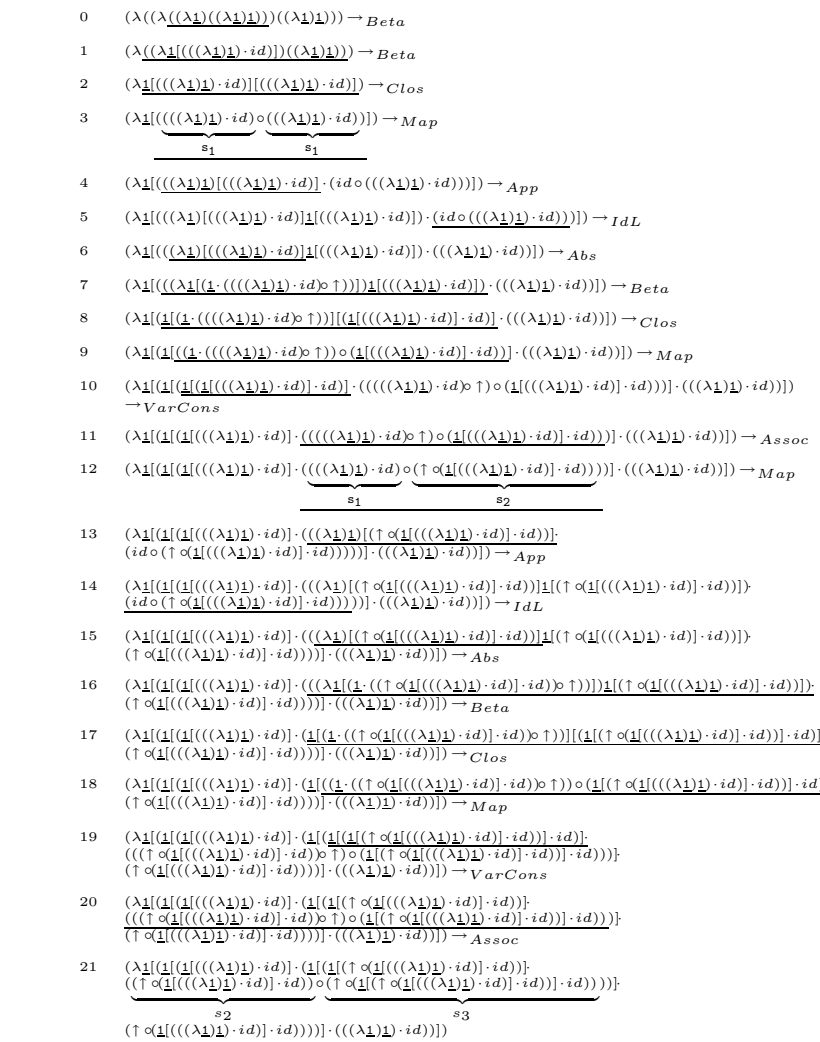

**Table 6.** *The counter-example of Melliès*

$$\lambda((\lambda((\lambda((\lambda\underline{2})\underline{3}))\underline{2}))(\lambda((\lambda\underline{2})\underline{2}))\underline{1})$$

which is written in the SUBSEXPL system as

```
L(A(L(A(L(A(L(2),3)),2)),A(L(A(L(2),2)),1)))
```

The steps of the infinite derivation, written in the usual grammar of the $\lambda s_e$-calculus, are given at the end of this subsection according to the numbering of these

steps. The key sub-terms originating the infinite derivation are under-bracketed. The initial steps are given by:

| STEP | RULE | POSITION | STEP | RULE | POSITION |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 4 | 2 | 111 |
| 2 | 1 | 11 | 5 | 3 | 1 |
| 3 | 3 | 11 | 6 | 2 | 11 |

Here, the derivation gives the sub-term `S(1,S(1,3,2),A(L(A(L(2),2)),1))` which is denoted as $u_0$, and so $u_0 := (\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})$. We recursively define the following: $u_{n+1}$ := `S(1,S(1,P(2,1,2),P(2,0,1)),`$u_n$`)` if $n \geq 0$.

The following steps are:

| STEP | RULE | POSITION | STEP | RULE | POSITION |
|---|---|---|---|---|---|
| 7 | 1 | 1 | 10 | 11 | 11 |
| 8 | 4 | 111 | 11 | 8 | 1 |
| 9 | 10 | 11 | | | |

The current term,

`L(S(1,S(2,P(2,1,2),`$u_0$`),S(1,P(2,0,A(L(A(L(2),2)),1)),`$u_0$`)))`

has the term `S(1,P(2,0,A(L(A(L(2),2)),1)),`$u_0$`)` as a sub-term, which is written as $(\varphi_0^2((\lambda((\lambda\underline{2})\underline{2}))\underline{1})\sigma^1 u_0)$ in the language of the $\lambda s_e$-calculus. This sub-term is important in the characterization of the infinite reduction.

The following steps are given in the next table:

| STEP | RULE | POSITION | STEP | RULE | POSITION |
|---|---|---|---|---|---|
| 12 | 6 | 121 | 20 | 3 | 12 |
| 13 | 5 | 1211 | 21 | 2 | 121 |
| 14 | 6 | 12111 | 22 | 1 | 12 |
| 15 | 5 | 121111 | 23 | 4 | 1211 |
| 16 | 7 | 1211111 | 24 | 10 | 121 |
| 17 | 1 | 121 | 25 | 11 | 121 |
| 18 | 3 | 121 | 26 | 8 | 12 |
| 19 | 2 | 1211 | | | |

The current term contains the sub-term `S(1,P(2,0,`$u_0$`),`$u_1$`)` which can be reduced to `S(1,P(2,0,A(L(A(L(2),2)),1)),`$u_1$`)` according to the table:

| STEP | RULE | POSITION | STEP | RULE | POSITION |
|---|---|---|---|---|---|
| 27 | 11 | 1221 | 28 | 8 | 122 |

The current term contains $\mathtt{S(1,P(2,0,A(L(A(L(2),2)),1)),}u_1\mathtt{)}$ as a sub-term which completes the first cycle of our infinite reduction. Note that we do not have a loop in these reductions because the original term is never reached again. In fact, the adequate combination of the associated calculus, named $s_e$, with the $\sigma$-generation rule permits one to start new simulations of $\beta$-reduction without finishing previous started simulations of $\beta$-reduction which suggests a kind of cycle. The same happens in the $\lambda\sigma$-calculus.

When the sub-term $\mathtt{S(1,P(2,0,A(L(A(L(2),2)),1)),}u_2\mathtt{)}$ is generated, then the next cycle is completed. This is done by repeating the same steps from 12 to 26 in adequate positions. Additional applications of the rules 11 and 8 (in this order) will be necessary to rewrite terms of the form $\mathtt{S(1,P(2,0,}u_m\mathtt{),}u_n\mathtt{)}$ as terms containing sub-terms of the form $\mathtt{S(1,P(2,0,A(L(A(L(2),2)),1)),}u_n\mathtt{)}$, for $n, m \geq 0$ (cf. [GUI 00]). The next table presents the necessary steps to complete the second cycle:

| STEP | RULE | POSITION | STEP | RULE | POSITION |
|---|---|---|---|---|---|
| 29 | 6 | 12221 | 37 | 3 | 1222 |
| 30 | 5 | 122211 | 38 | 2 | 12221 |
| 31 | 6 | 1222111 | 39 | 1 | 1222 |
| 32 | 5 | 12221111 | 40 | 4 | 122211 |
| 33 | 7 | 122211111 | 41 | 10 | 12221 |
| 34 | 1 | 12221 | 42 | 11 | 12221 |
| 35 | 3 | 12221 | 43 | 8 | 1222 |
| 36 | 2 | 122211 | | | |

Now we need to reduce the sub-term $\mathtt{S(1,P(2,0,}u_1\mathtt{),}u_2\mathtt{)}$ to a new term having $\mathtt{S(1,P(2,0,}u_0\mathtt{),}u_2\mathtt{)}$ as a sub-term, from which we get the sub-term $\mathtt{S(1,P(2,0,A(L(A(L(2),2)),1)),}u_2\mathtt{)}$. The next table includes this reduction:

| STEP | RULE | POSITION | STEP | RULE | POSITION |
|---|---|---|---|---|---|
| 44 | 11 | 122221 | 46 | 11 | 1222221 |
| 45 | 8 | 12222 | 47 | 8 | 122222 |

Note that here, two applications of rules 11 and 8 (in this order) were necessary as is shown in the previous table. To continue the derivation and generate the sub-term $\mathtt{S(1,P(2,0,A(L(A(L(2),2)),1)),}u_3\mathtt{)}$ repeat steps 29 to 47 on the adequate positions. Do not forget that additional applications of rules 11 and 8 will be necessary. Observing the under-bracketed sub-terms in the derivation, the shape of the infinite derivation can be written as

$$\begin{array}{l}\lambda((\lambda((\lambda(\lambda\underline{2})\underline{3}))\underline{2}))(\lambda((\lambda\underline{2})\underline{2}))\underline{1}) \rightsquigarrow \\ (\varphi_0^2((\lambda((\lambda\underline{2})\underline{2}))\underline{1})\sigma^1 u_0) \rightsquigarrow \\ (\varphi_0^2((\lambda((\lambda\underline{2})\underline{2}))\underline{1})\sigma^1 u_1) \rightsquigarrow \\ (\varphi_0^2((\lambda((\lambda\underline{2})\underline{2}))\underline{1})\sigma^1 u_2) \rightsquigarrow \cdots\end{array}$$

where $\rightsquigarrow$ means "leads to a term containing the following expression as a sub-term".

The latex output of the first 28 steps of the infinite derivation can be automatically generated. To do so, type the prompt shell command:

```
./subsexpl.bin guillaume
```

where `guillaume` is a file distributed with the source code of SUBSEXPL. The system will generate the `guillaume-lse.dvi` file. The result is shown in Table 7 which includes the number of the steps and some subscripts for ease of reading, such as $u_0$ and $u_1$, that do not appear in the latex output generated by the system SUBSEXPL. In this example, the latex code of the output can be found in the file called `guillaume-lse`.

The animated generation of the initial steps of these infinite derivations inside each calculus gives an easy-to-understand and more intuitive insight as to why the PSN property fails. In fact, for understanding these counter-examples (and their importance) directly from the related papers ([MEL 95, GUI 00]), the reader needs to follow a sequence of inductively proved lemmas and theorems. This is of course necessary for an adequate formalization of this fact. But when this approach is followed, one presumes previous knowledge of the reader about what PSN means. And we believe that following this course of reasoning the reader may lose the focus about the mechanics of these infinite derivations in the associated substitution calculi. Even worse, the reader may miss a very important aspect: namely, the meaning and the implications of losing PSN in these calculi. Consequently, we believe that SUBSEXPL is an adequate and useful tool for intuitively understanding the details and difficulties concerned with this and other general properties of the $\lambda$-calculus as well as of explicit substitutions calculi.

In [NAD 99, LIA 02] it has been conjectured that PSN holds in $\lambda_{\text{SUSP}}$. But until now, there is neither a formal proof nor a counter-example of this conjecture. We believe that SUBSEXPL may act as an adequate tool for reasoning about open questions like this, since every reduction (either from a pure or a non pure $\lambda$-term) can be simulated in an easy, fast and sure way in this system.

## 5. Conclusions and future work

We presented the system SUBSEXPL which is an Ocaml implementation of the rewriting rules of the $\lambda\sigma$, the $\lambda s_e$, the suspension and the combining suspension calculi of explicit substitutions, although according to the current structure the inclusion of other explicit substitutions calculi can be easily done.

We showed how the system has been applied both to educational and research purposes. Its educational uses include:

– the visualization of the mechanics of de Bruijn notation;

– the visualization of the computational adequacy of the $\lambda$-calculus via specification of numerals, numerical functions and programming operators;

– the visualization of (non trivial) properties of the $\lambda$-calculus such as non termination and the normalization theorem;

0 $(\lambda\underline{((\lambda((\lambda(\underline{2}\underline{3}))\underline{2}))((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))})$ $\rightarrow_{\sigma}$-gen

1 $(\lambda\underline{(((\lambda((\lambda\underline{2})\underline{3}))\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))})$ $\rightarrow_{\sigma}$-gen

2 $(\lambda\underline{((((\lambda\underline{2})\underline{3})\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))})$ $\rightarrow_{\sigma}$-app

3 $(\lambda\underline{(((\lambda\underline{2})\sigma^1\underline{2})(\underline{3}\sigma^1\underline{2}))\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))})$ $\rightarrow_{\sigma\lambda}$

4 $(\lambda\underline{(((\lambda(\underline{2}\sigma^2\underline{2}))(\underline{3}\sigma^1\underline{2}))\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))})$ $\rightarrow_{\sigma}$-app

5 $(\lambda\underline{(((\lambda(\underline{2}\sigma^2\underline{2}))\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))\underbrace{((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))}_{u_0}})$ $\rightarrow_{\sigma\lambda}$

6 $(\lambda\underline{((\lambda((\underline{2}\sigma^2\underline{2})\sigma^2((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))})$ $\rightarrow_{\sigma}$-gen

7 $(\lambda((\underline{(\underline{2}\sigma^2\underline{2})}\sigma^2((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))))$ $\rightarrow_{\sigma}$-des

8 $(\lambda\underline{((\varphi^2_0(\underline{2})\sigma^2((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))})$ $\rightarrow_{\sigma\varphi 2}$

9 $(\lambda\underline{(\varphi^2_0((\underline{2}\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))})$ $\rightarrow_{\varphi\sigma}$

10 $(\lambda\underline{((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))})$ $\rightarrow_{\sigma\sigma}$

11 $(\lambda((\varphi^2_1(\underline{2})\sigma^2\underbrace{((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))}_{u_0})\sigma^1(\underline{\varphi^2_0(((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))}\sigma^1\underbrace{((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))}_{u_0})))$ $\rightarrow_{\varphi}$-app

12 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1(\underline{(\varphi^2_0((\lambda((\lambda\underline{2})\underline{2})))}\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))$ $\rightarrow_{\varphi\lambda}$

13 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1(\underline{((\lambda\varphi^2_1((\lambda\underline{2})\underline{2}))}\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))$ $\rightarrow_{\varphi}$-app

14 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1(((\lambda\underline{(\varphi^2_1((\lambda\underline{2})\underline{2}))}\varphi^2_1(\underline{2}))\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))$ $\rightarrow_{\varphi\lambda}$

15 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1(((\lambda((\lambda\underline{\varphi^2_2(\underline{2})})\varphi^2_1(\underline{2}))\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))$ $\rightarrow_{\varphi}$-des

16 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1(\underline{((\lambda((\lambda\underline{2})\varphi^2_1(\underline{2})))\varphi^2_0(\underline{1}))}\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))$ $\rightarrow_{\sigma}$-gen

17 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1(\underline{(((\lambda\underline{2})\varphi^2_1(\underline{2}))\sigma^1\varphi^2_0(\underline{1}))}\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))$ $\rightarrow_{\sigma}$-app

18 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1(\underline{((((\lambda\underline{2})\sigma^1\varphi^2_0(\underline{1}))}(\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1})))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))$ $\rightarrow_{\sigma\lambda}$

19 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1(\underline{(((\lambda(\underline{2}\sigma^2\varphi^2_0(\underline{1})))(\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1})))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))))}))$ $\rightarrow_{\sigma}$-app

20 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1(\underline{(((\lambda(\underline{2}\sigma^2\varphi^2_0(\underline{1})))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))}$
$((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))))))$ $\rightarrow_{\sigma\lambda}$

21 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1$
$\underline{((\lambda((\underline{2}\sigma^2\varphi^2_0(\underline{1}))\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))))((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))}$) $\rightarrow_{\sigma}$-gen

22 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1(((\underline{2}\sigma^2\varphi^2_0(\underline{1}))\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1$
$((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))\sigma^1((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1(\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))$ $\rightarrow_{\sigma}$-des

23 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1(\underline{((\varphi^2_0(\varphi^2_0(\underline{1}))\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1}$
$((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))$ $\rightarrow_{\sigma\varphi 2}$

24 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1(\underline{\varphi^2_0((\varphi^2_0(\underline{1})\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))))}\sigma^1$
$((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))$ $\rightarrow_{\varphi\sigma}$

25 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1$
$\underline{((\varphi^2_1(\varphi^2_0(\underline{1}))\sigma^1\varphi^2_0(((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))))\sigma^1((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))}$) $\rightarrow_{\sigma\sigma}$

26 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1((\varphi^2_1(\varphi^2_0(\underline{1}))\sigma^2((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1$
$((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))))\sigma^1(\varphi^2_0(\underbrace{((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))}_{u_0})\sigma^1(\underbrace{((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))}_{u_1}))))$ $\rightarrow_{\varphi\sigma}$

27 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1((\varphi^2_1(\varphi^2_0(\underline{1}))\sigma^2((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1$
$((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))))\sigma^1\underline{((\varphi^2_1((\underline{3}\sigma^1\underline{2}))\sigma^1\varphi^2_0(((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))}$) $\rightarrow_{\sigma\sigma}$

28 $(\lambda((\varphi^2_1(\underline{2})\sigma^2((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))\sigma^1((\varphi^2_1(\varphi^2_0(\underline{1}))\sigma^2((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1$
$((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))))\sigma^1((\varphi^2_1((\underline{3}\sigma^1\underline{2}))\sigma^2((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))))\sigma^1\varphi^2_0(((\lambda((\lambda\underline{2})\underline{2}))\underline{1}))\sigma^1$
$\underbrace{((\varphi^2_1(\underline{2})\sigma^1\varphi^2_0(\underline{1}))\sigma^1((\underline{3}\sigma^1\underline{2})\sigma^1((\lambda((\lambda\underline{2})\underline{2}))\underline{1})))))}_{u_1}$

**Table 7.** *The counter-example of Guillaume*

– the illustration of the problem of implicitness of the substitution operator and how this is resolved in real implementations by explicit substitutions calculi; etc.

In particular, the second use was exemplified in Subsection 4.1, where we show how elaborated computational operations such as iteration are implemented in the language of SUBSEXPL and how computations of simple recursive functions are simulated. Additional simulations can be done with other functions available in the companion file of examples. Illustration of relevant non trivial properties of the $\lambda$-calculus such as the normalization theorem can be given by presenting simulations of left-most reduction for any weak normalizing $\lambda$-term from which infinite derivations are possible as well.

The research applications of SUBSEXPL include:

– assisting the analysis of non trivial properties of explicit substitutions calculi;

– comparing calculi of explicit substitutions.

The former was illustrated by showing that one can check the proofs of Melliès and Guillaume (included in the tutorial distributed with the source code of the system) of the fact that neither $\lambda\sigma$ nor $\lambda s_e$ preserve strong normalization using the system. The latter by showing how the system assisted us in the proof that $\lambda s_e$ is more efficient than the suspension calculus and is incomparable to the $\lambda\sigma$-calculus in the simulation of one-step $\beta$-reduction [AYA 05] and in formalizing the possibility and impossibility of building refinements of $\lambda\sigma$ and $\lambda s_e$, which combine $\beta$-contractions as the combining suspension calculus does [FER 05b].

Furthermore, SUBSEXPL gives correct implementations of $\eta$-reduction for each of the four explicit substitutions calculi following the principles given in [AYA 05]. For the $\lambda s_e$-calculus this implementation is also clean, but for $\lambda\sigma$ and $\lambda_{\text{SUSP}}$ (and by the nature of these calculi), the simulation of one-step $\eta$-reduction requires the use of rewriting rules that are not strictly related to this one-step simulation.

Other authors have presented tools that manipulate $\lambda$-expressions with symbolic variables in a similar way; for example Huet presented a tool and illustrated how this can be applied for assisting in the understanding of non trivial properties of the $\lambda$-calculus such as Böhm's theorem [HUE 93]. The novelty of SUBSEXPL with relation to these applications is that it follows the de Bruijn's philosophy of avoiding names, which makes our tool also adequate for assisting in the reasoning about properties of explicit substitution calculi.

SUBSEXPL is in constant development and new features should be included in future versions. Among these features, we can point out the development of the interface of the system, inclusion of new modules for dealing with the simply typed $\lambda$-terms and the $\lambda$-calculus with names.

Acknowledgements

We would like to thank Manuel Maarek and Stéphane Gimenez for the useful help with Ocaml and suggestions to improve the system.

## 6. References

[ABA 91] ABADI M., CARDELLI L., CURIEN P.-L., LÉVY J.-J., "Explicit Substitutions", *J. of Func. Programming*, vol. 1, num. 4, 1991, p. 375-416.

[AYA 01] AYALA-RINCÓN M., KAMAREDDINE F., "Unification via the $\lambda s_e$-Style of Explicit Substitution", *The Logical Journal of the IGPL*, vol. 9, num. 4, 2001, p. 489-523.

[AYA 05] AYALA-RINCÓN M., DE MOURA F., KAMAREDDINE F., "Comparing and Implementing Calculi of Explicit Substitutions with Eta-Reduction", *Annals of Pure and Applied Logic*, vol. 134, num. 1, 2005, p. 5-41, Special Issue WoLLIC 2002, Ruy de Queiroz, Bruno Poizat and S. Artemov, Eds.

[BAA 98] BAADER F., NIPKOW T., *Term Rewriting and* All That, CUP, 1998.

[BAR 84] BARENDREGT H. P., *The Lambda Calculus : Its Syntax and Semantics (revised edition)*, North Holland, 1984.

[BRI 95] BRIAUD D., "An explicit Eta rewrite rule", *Typed lambda calculi and applications*, vol. 902 of *LNCS*, Springer, 1995, p. 94-108.

[BRU 72] DE BRUIJN N., "Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem", *Indag. Mat.*, vol. 34(5), 1972, p. 381-392.

[DOW 00] DOWEK G., HARDIN T., KIRCHNER C., "Higher-order Unification via Explicit Substitutions", *Information and Computation*, vol. 157, num. 1/2, 2000, p. 183–235.

[FER 05a] FERNÁNDEZ M., MACKIE I., SINOT F.-R., "Closed reduction: explicit substitutions without $\alpha$-conversion", *Math. Struct. in Comp. Science*, vol. 15, 2005, p. 343–381.

[FER 05b] FERREIRA H., "Análise de Mecanismos para Combinar passos de Beta-Contração em Cálculos de Substituições Explícitas", Master's thesis, Departamento de Matemática, Universidade de Brasília, 2005, In Portuguese.

[GUI 00] GUILLAUME B., "The $\lambda s_e$-calculus Does Not Preserve Strong Normalization", *J. of Func. Programming*, vol. 10, num. 4, 2000, p. 321-325.

[HAR 92] HARDIN T., "Eta-conversion for the languages of explicit substitutions", *Algebraic and logic programming*, vol. 632 of *LNCS*, Springer, 1992, p. 306-321.

[HUE 93] HUET G., "An Analysis of Böhm's Theorem", *TCS*, vol. 121, 1993, p. 145-167.

[KAM 96] KAMAREDDINE F., NEDERPELT R. P., "A Useful $\lambda$-Notation", *TCS*, vol. 155, 1996, p. 85-109.

[KAM 97] KAMAREDDINE F., RÍOS A., "Extending a $\lambda$-calculus with Explicit Substitution which Preserves Strong Normalisation into a Confluent Calculus on Open Terms", *J. of Func. Programming*, vol. 7, 1997, p. 395-420.

[KAM 00] KAMAREDDINE F., RÍOS A., "Relating the $\lambda\sigma$- and $\lambda s$-Styles of Explicit Substitutions", *Journal of Logic and Computation*, vol. 10, num. 3, 2000, p. 349-380.

[KES 00] KESNER D., "Confluence of extensional and non-extensional $\lambda$-calculi with explicit substitutions", *TCS*, vol. 238, num. 1-2, 2000, p. 183-220.

[LIA 02] LIANG C., NADATHUR G., "Tradeoffs in the Intensional Representation of Lambda Terms", TISON S., Ed., *Rewriting Techniques and Applications (RTA 2002)*, vol. 2378 of *LNCS*, Spinger-Verlag, 2002, p. 192-206.

[LIA 04] LIANG C., G.NADATHUR, QI X., "Choices in Representation and Reduction Strategies for Lambda Terms in Intensional Contexts", *JAR*, vol. 33, num. 2, 2004, p. 89-132.

[MEL 95] MELLIÈS P.-A., "Typed $\lambda$-calculi with explicit substitutions may not terminate in Proceedings of TLCA'95", *LNCS*, vol. 902, 1995, p. 328-334, Springer-Verlag.

[MOU 05] DE MOURA F. L. C., KAMAREDDINE F., AYALA-RINCÓN M., "Second Order Matching via Explicit Substitutions", BAADER F., VORONKOV A., Eds., *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, vol. 3452, Springer-Verlag, 2005, p. 433-448.

[NAD 88] NADATHUR G., MILLER D., "An Overview of $\lambda$Prolog", BOWEN K., KOWALSKI R., Eds., *Proc. 5th Int. Logic Programming Conference*, MIT Press, 1988, p. 810-827.

[NAD 98] NADATHUR G., WILSON D. S., "A Notation for Lambda Terms A Generalization of Environments", *TCS*, vol. 198, 1998, p. 49-98.

[NAD 99] NADATHUR G., "A Fine-Grained Notation for Lambda Terms and Its Use in Intensional Operations", *J. of Func. and Logic Prog.*, vol. 1999, num. 2, 1999, p. 1-62.

[NAD 02] NADATHUR G., "The Suspension Notation for Lambda Terms and its Use in Metalanguage Implementations", DE QUEIROZ R., Ed., *Proceedings Ninth Workshop on Logic, Language, Information and Computation*, vol. 67 of *ENTCS*, Elsevier Sci. Publishers, 2002.

[NAD 03] NADATHUR G., QI X., "Explicit Substitutions in the Reduction of Lambda Terms", *Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2003, p. 195-206.

[NED 94] NEDERPELT R. P., GEUVERS J. H., DE VRIJER R. C., *Selected papers on Automath*, North-Holland, 1994.

[RÍO 93] RÍOS A., "Contribution à l'étude des $\lambda$-calculs avec substitutions explicites", PhD thesis, Université de Paris 7, 1993.

[URB 04] URBAN C., PITTS A. M., GABBAY M. J., "Nominal Unification", *TCS*, vol. 323, num. 1-3, 2004, p. 473-497.