SUBSEXPL: A Tool for Simulating and Comparing Explicit Substitutions Calculi*

Flávio L. C. de Moura**[1] and Mauricio Ayala-Rincón***[1] and Fairouz Kamareddine[2]

[1] Departamento de Matemática, Universidade de Brasília, Brasília D.F., Brasil.
flavio@mat.unb.br, ayala@mat.unb.br
[2] School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, 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 λ-calculus and of three different styles of explicit substitutions: the $\lambda\sigma$, the λs_e and the suspension calculus. Implementations of the η-reduction are provided for each calculi. Other explicit substitutions calculi can be incorporated into the system easily due to its modular structure. Its applications include: the visualisation of the contractions of the λ-calculus, and of guided one-step reductions as well as normalisation via each of the associated substitution calculi. Many useful facilities are available: reductions can be easily recorded and stored into files or Latex outputs and several examples for dealing with arithmetic operations and computational operators such as conditionals and repetitions in the λ-calculus are available. 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 Normalisation. In addition, it has been used for teaching basic properties of the λ-calculus such as: computational adequacy, the importance of de Bruijn's notation and of making explicit substitutions in real implementations.

Keywords: λ-Calculus, Explicit Substitutions, Visualisation of β- and η-Contraction and Normalisation.

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 for practical notions such as the implementation of typed functional programming languages and higher-order proof assistants. We describe SUBSEXPL, a system developed in Ocaml, a language of the ML family, which allows for the manipulation of expressions of the λ-calculus and of three different calculi of explicit substitutions:

* Work supported by funds from CNPq (CT-INFO) 50.6598/04-7.
** Corresponding author. Supported by Brazilian CAPES Foundation.
*** Partially supported by Brazilian Research Council CNPq.

1. $\lambda\sigma$ [1] which introduces two different sets of entities: one for `terms` and one for `substitutions`.
2. λs [12] which is based on the philosophy of de Bruijn's *Automath* [20] elaborated in the new *item notation* [11]. 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 λ-calculus (cf. [13]).
3. The suspension calculus [17], which introduces three different sets of entities: `terms`, `environments` and `lists of environments`.

Each of these different styles has plus and minus points. Although various attempts have been made at comparing these styles (cf. [2, 13]), a lot remains to be explained. A better understanding of the similarities and differences of these styles may lead on one hand to solving the remaining open questions related to the various calculi, and on the other hand, to a more inclusive calculus and implementations which combine the advantages in one system. The inclusion of other calculus of explicit substitutions is also possible: the documentation provided with the source code of the system includes a file called `adding-a-new-calculus` which explains all the necessary steps.

Through SUBSEXPL, we attempt to understand the working of the rewrite rules of these calculi. We developed a full scale Ocaml implementation of the three calculi where contractions in all these calculi (as well as in the type-free λ-calculus) can be visualised in a step-wise fashion and where the behaviour of the reduction paths can be analysed. Especially, we concentrate on the one-step guided reductions and normalisation via each of the associated *substitution calculi*. However, implementation of rewriting rules is straightforward in rewriting based languages such as ELAN and Maude, we prefer to use a language of the ML family because of their natural ability to control the matching which allows for selection of redexes before contractions are done.

SUBSEXPL has been successfully used for teaching our students basic properties of the λ-calculus such as: computational adequacy, the importance of de Bruijn's notation and of making explicit substitutions in real implementations based on the λ-calculus. SUBSEXPL has also been of great importance for systematically comparing these three calculi of explicit substitutions.

Furthermore, SUBSEXPL includes adequate implementations of the rules of η-reduction for the three calculi as well as a *clean* implementation for the λs_e-calculus (cf. [2]) in the sense that no other rewriting rules than the ones strictly involved in the Eta-contraction[3] are included in one-step Eta-contraction. Work on higher-order unification (HOU) in $\lambda\sigma$ and λs_e established the importance of combining Eta-reduction or contraction (as well as expansion) with explicit substitutions. This has provided extensions of $\lambda\sigma$ and λs_e with Eta-reduction rules also referred to by $\lambda\sigma$ and λs_e (cf.[6, 3]). Eta reduction as well as expansion are necessary for working with functions and programs, since one needs to express

[3] We use the Greek letter η to refer only to the "η-rule" of the pure λ-calculus, and its name "Eta" to refer to the corresponding rules in the explicit substitutions calculi.

functional or extensional equality; i.e., when the application of two λ-terms to any 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 [9, 21, 5, 14].

Input/output of λ-terms is a difficult point because λ-expressions may become big very quickly. In order to ease reading the outputs of the system, we provided Latex outputs which can be generated during any step of the reduction 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 β-reduction: whenever a reduces to b in the λ-calculus using one step β-reduction, we have that a reduces to b in the explicit substitutions calculus using one step of the explicit β-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 λ-calculus built from variables, applications and abstractions.
  (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 uses explicit substitutions [6, 3, 7].
(c) Strong normalisation (SN) of the underlying calculus of explicit substitutions: this is the termination property of the explicit substitutions calculi without the explicit β-reduction rule; i.e., without the rule that starts the simulation of the β-reduction.
(d) Preservation of SN (PSN): whenever all possible reductions starting from a pure λ-term are terminating in the λ-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, λs satisfies (a)..(d) but not (b.2). However, λs has an extension λs_e (again without Eta) for which (a), (b.1) and (b.2) holds, but (d) fails and (c) is unknown. 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. [13, 19]).

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 ([16]) and Guillaume ([8]) for proving that neither $\lambda\sigma$- nor λs_e-calculi preserve SN.

In section 2 we briefly describe the system and its usage and, before concluding, in section 3 we illustrate the applications of the system.

2 Description of SUBSEXPL

SUBSEXPL is an implementation of the rewriting rules of the three treated calculi of explicit substitutions. SUBSEXPL is an open source software, runs over GNU/Linux platforms and is available at `www.mat.unb.br/~ayala/TCgroup/`.

2.1 Use of the system

To start the system, execute the file `subsexpl.bin` (by typing `./subsexpl.bin` in a terminal). We recommend the use of the line editor ledit[4]:
`./ledit.out ./subsexpl.bin`.

Alternatively, the user can run SUBSEXPL inside a shell in the EMACS editor so that (s)he can easily cut and paste and check the balance of expressions. To do so just type within EMACS `M-x shell` and then `./subsexpl`.

The first screen is as below where option 4 gives a brief grammatical description of the input and output for each calculus.

```
************** SUBSEXPL **************

SELECT the calculus
TYPE
      0 for the Pure lambda-calculus
      1 for the Lambda sigma calculus
      2 for the Lambda s_e calculus
      3 for the Suspension calculus
      4 for the Grammatical description IN/OUT (and internal)
OR    5 for quit
>
```

Option `0` allows the user to simulate one-step β-reduction and η-reduction as well as normalisations in the pure λ-calculus, while options `1`, `2` and `3` perform simulations of reductions and normalisations in $\lambda\sigma$, λs_e and the suspension calculus, respectively.

As a complete example, we will show how to operate with the Church's numerals (cf. [4]) whose description can be found in the `Examples` file distributed with the source code. Consider the reduction $A_+C_1C_1 \rightarrow_\beta^6 C_2$, which evaluates "$1+1$" in the λ-calculus, where $A_+ = \lambda xypq.((x\ p)((y\ p)\ q))$ represents the sum operator, and $C_1 = \lambda fx.fx$ is a Church numeral. 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))))))`.

Applying this operator to add the Church numeral C_1 twice, 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))))`

After choosing option `0` in the first screen of the system, we type the above expression:

[4] `http://cristal.inria.fr/~ddr`

```
*************** SUBSEXPL ***************

SELECT the calculus
TYPE
      0 for the Pure lambda-calculus
      1 for the Lambda sigma calculus
      2 for the Lambda s_e calculus
      3 for the Suspension calculus
      4 for the Grammatical description IN/OUT (and internal)
OR    5 for quit
> 0
Give an expression (or quit): 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))))
```

After typing the expression, type ENTER. The next screen will output the current expression and the available redexes for the rules:

```
Expression: 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))))

1. Beta:  1                           7. Latex output.
2. Eta:  121  21                      8. Save current reduction.
3. Leftmost/outermost normalisation.  9. Restart current reduction.
4. Rightmost/innermost normalisation. 10. Restart SUBSEXPL.
5. Back one step.                     11. Quit.
6. See history.
Give the number:
```

To select β-reduction, type 1 and then type 1 again to select the redex at position 1. Now the current screen is:

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

1. Beta:  0  11111                    7. Latex output.
2. Eta:  1111111  21                  8. Save current reduction.
...
Give the number:
```

Note that we have two options to apply β-reduction. One at the root position of the term, written as 0, and another at position 11111. To reduce the term at position 11111, first type 1 to select `Beta` and then type the position. Continue the reduction until you get a normal term: `L(L(A(2,A(2,1))))` which corresponds to C_2.

The additional options of the system are:

`3. Leftmost/outermost normalisation`: normalises the given term choosing always the leftmost redex.

`4. Rightmost/innermost normalisation`: normalises 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[5]

`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, the user should restart the system giving this file as argument: `./ledit.out`

[5] We assume that the running system has latex and xdvi installed.

`./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.

To generate the latex output, which is possible to be generated even during the intermediate steps in a reduction, just type 7 and then give a file name without any extension. For example, `my_file`. In this case, the system will generate a dvi file named `my_file.dvi`. Note that in the latex output, all the redexes you chose during the reduction will appear underlined:

$$\underline{(((\lambda(\lambda(\lambda((\underline{42})((\underline{32})\underline{1})))))(\lambda(\lambda(\underline{21}))))}(\lambda(\lambda(\underline{21})))) \rightarrow_\beta$$

$$((\lambda(\lambda(\underline{(((\lambda(\lambda(\underline{21})))\underline{2})}((\underline{32})\underline{1})))))(\lambda(\lambda(\underline{21})))) \rightarrow_\beta$$

$$((\lambda(\lambda(\lambda\underline{((\lambda(\underline{31}))((\underline{32})\underline{1}))})))(\lambda(\lambda(\underline{21})))) \rightarrow_\beta$$

$$\underline{((\lambda(\lambda(\underline{2}((\underline{32})\underline{1})))))(\lambda(\lambda(\underline{21}))))} \rightarrow_\beta$$

$$(\lambda(\lambda(\underline{2}(\underline{(((\lambda(\lambda(\underline{21})))\underline{2})\underline{1})})))) \rightarrow_\beta$$

$$(\lambda(\lambda(\underline{2}\underline{((\lambda(\underline{31}))\underline{1})}))) \rightarrow_\beta$$

$$(\lambda(\lambda(\underline{2}(\underline{21}))))$$

An interesting exercise is to simulate such a derivation step by step using the $\lambda\sigma$, the λs_e or the suspension calculus. The current implementation has two normalisation strategies available: the leftmost/outermost strategy or the strategy according to the order of the rules given on the screen of each calculi (we call this strategy 'random'). An interesting fact is that the first step of the previous example when simulated in the $\lambda\sigma$-calculus using the random normalisation strategy generates some huge $\lambda\sigma$-terms which exceeds the available memory for the latex compilation. In fact, the simulation of the first β-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/outermost strategy generated about 3 full pages of latex output with small fonts. In the λs_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 $((\lambda\underline{1})\ \underline{1}[\uparrow])[\underline{1}.id]$ which is written in SUBSEXPL as `Sb(A(L(1),Sb(One,Up)),Pt(One,Id))`. Giving this term to the system we get the screen below, from which one can follow the reduction by selecting rules and redexes (positions).

```
Expression: Sb(A(L(One),Sb(One,Up)),Pt(One,Id))

1. Beta:  1       9. IdL:                          17. Back one step.
2. App:  0        10. IdR:                         18. See history.
3. Abs:           11. ShiftCons:                   19. Latex output.
4. Clos:          12. VarShift:                    20. Save current reduction.
5. VarCons:       13. SCons:                       21. Restart current reduction.
```

```
6. Id:            14. Eta:                                22. Restart SUBSEXPL.
7. Assoc:         15. One beta full step (leftmost):  1   23. Quit.
8. Map:           16. One beta full step (random):  1
Give the number:
```

2.2 Implementation of Eta contraction

SUBSEXPL includes implementations of the Eta-rule for each of the three calculi of explicit substitutions treated here. The implementation follows the notion of cleanness as defined in [2]. The intuitive idea of a *clean* Eta implementation is that it does not mix isolated applications of Eta-reduction with applications of other rules of the corresponding substitution calculi that the ones strictly involved in the Eta-reduction. Clean implementations of the Eta-rule allow us to reach good simulations of the Eta-contraction, which implies the possibility of combining steps of Beta and Eta contraction.

The suspension calculus did not originally have an Eta-rule. In [2] this calculus was enlarged with an adequate Eta-rule in the so-called λ_{SUSP} calculus. For the enlarged calculus λ_{SUSP}, λs_e and $\lambda\sigma$ we showed that there exists a correspondence among their Eta-rules which means that, when applied to pure λ-terms, these rules behave similarly (cf. [2]).

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 normalise some simple terms. Nevertheless, our implementation of the Eta-rule for λs_e is clean.

Eta-reduction is important to computational problems that arise in applications of the λ-calculus. For instance, in [6, 3] η-reduction is useful in the treatment of higher order unification and matching via explicit substitutions calculi.

3 Applications

SUBSEXPL has been successfully used to teach computational notions of the λ-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 λ-calculus. After that, we explain how it can be used to compare calculi of explicit substitutions according to the computational effort necessary to simulate one step of β-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 λs_e-calculus, respectively, do not preserve strong normalisation.

3.1 Understanding the λ-calculus and its implementations

We have used SUBSEXPL to explain to students questions related to the computational adequacy of the λ-calculus and the problems which arise from the

usual notation with symbolic variables and the implicit notion of substitution. The computational expressiveness of the λ-calculus can be illustrated by examples which range from the λ-representation of arithmetic operations such as addition, multiplication and exponentiation over Church's numerals to the λ-representation of basic data structures which include booleans and computational commands and operators such as if-then-else, iteration and recursion. All this was done in the spirit of [4].

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 λ-calculus. But implementing this class of exercises is necessary because this gives the real flavour of the computational power of the λ-calculus. By using SUBSEXPL over EMACS we can very quickly implement these functions: Initially, we create abbreviations for the needed operators and functions; afterwards, we compound these operators and functions in order to complete the desired function. We illustrate how this is done for the case of the factorial function. Basically, this function is implementing 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 does the right job. 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. This iteration operator is then used repeatedly.

Abbreviations 1. The Church numbers are as given before;
2. The booleans `true` and `false` correspond to the λ-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 λ-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 $\lambda xy.A_*\ y\ (S^+x)$. It is easy to see that this operator satisfies the property: `T`$\langle C_k, C_{k!}\rangle$ β-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` for factorial 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))))))))
```

Checking parts of the implementation This step is useful for testing the functionality of parts of the intended implementation which allows to infer the functionality of the whole specification. For instance, we can check that

$\mathtt{T}\langle C_2, C_{2!}\rangle$ reduces to $\langle C_3, C_{3!}\rangle$. In the input syntax of SUBSEXPL this is written as

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

By β-normalisation this part of the implementation can be checked obtaining the term
`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))))))))))` which corresponds to $\langle C_3, C_{3!}\rangle$

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

Functionality of all parts of the desired mechanism/function can be checked by normalisation with SUBSEXPL.

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

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

The equation (2), when applied to the Church numeral C_n, β-reduces to $C_{n!}$. In fact, such an application will generate a β-redex in the root of the new term. Reducing this new term, there is a sub-term of eq. (2) which reduces exactly to the term corresponding to eq. (1). And, this term we have already showed that reduces 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.

Observe that in the syntax of SUBSEXPL (which corresponds to the one of the λ-calculus) the expression for factorial (eq. (2)) is incomprehensible:

```
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 implemented easily. 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 a flavor of the computational power of the λ-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 of examples of the distribution.

The problem of having an implicit notion of substitution involves a complex implementational question because this is not a first-order operation. The comprehension of the necessity of making substitution an explicit operation is realised only when students are asked to implement β-contraction. After illustrating the computational adequacy of the λ-calculus, problems inherent to its implementation may be easily pointed out: collisions, confusion, renaming of variables, etc. Then students realise that substitution is a meta-operation that must be carefully defined in any correct implementation of the λ-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.

3.2 Comparing calculi by the simulation of β-reduction

SUBSEXPL has been implemented with the intention of comparing the three treated calculi of explicit substitutions with respect to the necessary effort to simulate one-step β-reduction. By applying this system we were able to conclude that λs_e is more efficient than the suspension calculus and is incomparable to the $\lambda\sigma$-calculus in the simulation of one-step β-reduction [2]. The efficiency of λs_e is justified by the fact that the manipulation of de Bruijn indexes in λs_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 λs_e is not completely adequate for combining steps of β-reduction, which is more natural in λ_{SUSP} [15, 18]. But we believe this has to be investigated more carefully, since some variations of λs_e like λt ([13]), which is a calculus à la λs_e but which updates à la $\lambda\sigma$, can allow this combination in the $\lambda\sigma$ family of calculi.

3.3 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 Normalisation (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 of non PSN of $\lambda\sigma$ and λs_e given in [16] and [8], 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 λ-calculus. Secondly, one can try to generate infinite derivations of β-reductions from these λ-terms, concluding (the most critical of them) that this is impossible. This last step is achieved without necessarily knowing that there are no infinite (β-)derivations in the λ-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. The detailed steps for running Guillaume's counter-example can be found in the tutorial distributed with the system.

The counter-example of Melliès To follow the counter-example in the $\lambda\sigma$-calculus, consider the well typed pure λ-term written in de Bruijn's notation as $\lambda((\lambda(\lambda\underline{1})((\lambda\underline{1})\underline{1}))((\lambda\underline{1})\underline{1}))$. 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 σ with the rule **Beta** which initiates the simulation of one step β-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.

STEP	RULE	POSITION
1	1	111
2	1	1
3	4	1

At this point,
`L(Sb(1,Cp(Pt(A(L(1),1),Id),Pt(A(L(1),1),Id))))` is the current term. Let us define recursively:

```
s_1 = Pt(A(L(1),1),Id)
s_2 = Cp(Up,Pt(Sb(1,s_1),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))
    ...
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)))`. 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))))`. 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
5	2	121
6	9	122
7	3	1211

Now the current term is `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
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`:

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

At this point, `L(Sb(1,Pt(Sb(1,Pt(Sb(1,s_1),Cp(s_1,s_2))),s_1)))` becomes the current term. It contains an occurrence of `Cp(s_1,s_2)`. By repeating the same sequence of rules, we will 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 to see how an infinite reduction can be built from the initial well typed term in the $\lambda\sigma$ calculus of explicit substitutions. In the following we give the corresponding reduction generated in Latex format by SUBSEXPL:

$$0\ (\lambda((\lambda\underline{1})((\lambda\underline{1})\underline{1}))((\lambda\underline{1})\underline{1}))) \rightarrow_{Beta}$$

$$1\ (\lambda\underline{((\lambda\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)])((\lambda\underline{1})\underline{1}))}) \rightarrow_{Beta}$$

$$2\ (\lambda\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)][\underline{(((\lambda\underline{1})\underline{1})\cdot id)}]) \rightarrow_{Clos}$$

$$3\ (\lambda\underline{1}[\underbrace{((((\lambda\underline{1})\underline{1})\cdot id)}_{s_1}\circ\underbrace{(((\lambda\underline{1})\underline{1})\cdot id))}_{s_1}]) \rightarrow_{Map}$$

$$4\ (\lambda\underline{1}[\underline{(((\lambda\underline{1})\underline{1})[(((\lambda\underline{1})\underline{1})\cdot id)]}\cdot(id\circ(((\lambda\underline{1})\underline{1})\cdot id))))]) \rightarrow_{App}$$

5 $(\lambda\underline{1}[(((\lambda\underline{1})[(((\lambda\underline{1})\underline{1})\cdot id)]\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)])\cdot\underline{(id\circ(((\lambda\underline{1})\underline{1})\cdot id))})]) \rightarrow_{IdL}$

6 $(\lambda\underline{1}[(((\underline{(\lambda\underline{1})[(((\lambda\underline{1})\underline{1})\cdot id)]}\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)])\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Abs}$

7 $(\lambda\underline{1}[(\underline{((\lambda\underline{1}[(\underline{1}\cdot((((\lambda\underline{1})\underline{1})\cdot id)\circ\uparrow))])\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)])}\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Beta}$

8 $(\lambda\underline{1}[\underline{(\underline{1}[(\underline{1}\cdot((((\lambda\underline{1})\underline{1})\cdot id)\circ\uparrow))][(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)]}\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Clos}$

9 $(\lambda\underline{1}[(\underline{1}[\underline{(\underline{1}\cdot((((\lambda\underline{1})\underline{1})\cdot id)\circ\uparrow))\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)}]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Map}$

10 $(\lambda\underline{1}[(\underline{1}[\underline{(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)]}\cdot(((((\lambda\underline{1})\underline{1})\cdot id)\circ\uparrow)\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))])$
$\rightarrow_{VarCons}$

11 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot\underline{(((((\lambda\underline{1})\underline{1})\cdot id)\circ\uparrow)\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))})]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Assoc}$

12 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underbrace{(((\lambda\underline{1})\underline{1})\cdot id)}_{s_1}\circ\underbrace{(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))}_{s_2})]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Map}$

13 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{((\lambda\underline{1})\underline{1})[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))}\}$
$\underline{(id\circ(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))}]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{App}$

14 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(((\lambda\underline{1})[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\}$
$\underline{(id\circ(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))}]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{IdL}$

15 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{((\lambda\underline{1})[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]}\}$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Abs}$

16 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{(((\lambda\underline{1}[(\underline{1}\cdot((\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))\circ\uparrow))])\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]}\}$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Beta}$

17 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{1}[(\underline{1}\cdot((\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))\circ\uparrow))][(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)}\}$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Clos}$

18 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{1}[\underline{((\underline{1}\cdot((\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))\circ\uparrow))\circ(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)}]\}$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Map}$

19 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{1}[(\underline{1}[(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)}\}$
$(((\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))\circ\uparrow)\circ(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)))\}$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{VarCons}$

20 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{1}[(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)}\}$
$\underline{(((\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))\circ\uparrow)\circ(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id)})\}$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))]) \rightarrow_{Assoc}$

21 $(\lambda\underline{1}[(\underline{1}[(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot(\underline{1}[(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))}\}$
$\underbrace{((\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))}_{s_2}\underbrace{\circ(\uparrow\circ(\underline{1}[(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id))]\cdot id))}_{s_3})\}$
$(\uparrow\circ(\underline{1}[(((\lambda\underline{1})\underline{1})\cdot id)]\cdot id)))]\cdot(((\lambda\underline{1})\underline{1})\cdot id))])$

The above steps are stored in the file `mellies` distributed with the source code of SUBSEXPL and can be executed automatically with the command `./subsexpl.bin mellies` (or `./ledit.out ./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 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`. The `mellies-README` file (distributed with the source code) explains these details and some additional information about generating the corresponding postscript file.

4 Conclusions and future work

We presented the system SUBSEXPL which is an Ocaml implementation of the rewriting rules of the $\lambda\sigma$, the λs_e and the 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 visualisation of the computational adequacy of the λ-calculus via specification of numerical functions and programming operators;
- the visualisation of (non trivial) properties of the λ-calculus such as non termination and the normalisation theorem;
- the illustration of the problem of implicitness of the substitution operator and how this is resolved in real implementations by explicit substitutions calculi; etc.

Its research applications include:

- 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 λs_e preserve strong normalisation using the system. The latter by showing how the system assisted us in the proof that λs_e is more efficient than the suspension calculus and is incomparable to the $\lambda\sigma$-calculus in the simulation of one-step β-reduction [2].

Furthermore, SUBSEXPL gives correct implementations of η-reduction for each of the three explicit substitutions calculi treated here. For the λs_e-calculus this implementation is also clean, but for $\lambda\sigma$ and λ_{SUSP} (and by the nature of these calculi), the simulation of one-step η-reduction requires the use of rewriting rules that are not strictly related to this one-step simulation.

Other authors have presented tools that manipulate λ-expressions 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 λ-calculus such as Böhm's theorem [10]. 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.

As any modern computational system, SUBSEXPL is in constant development and new features should be included in future versions. Among these features, we can point out the inclusion of variations of the suspension calculus that combine applications of β-reduction and the development of new modules for dealing with simply typed λ-terms and λ-calculus with names. Moreover, we will develop an EMACS mode which may ease the inclusion of some common structures used to build more complex terms.

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

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *J. of Func. Programming*, 1(4):375–416, 1991.
2. M. Ayala-Rincón, F.L.C. de Moura, and F. Kamareddine. Comparing and implementing calculi of explicit substitutions with eta-reduction. *To appear in Special Issue of* Annals of Pure and Applied Logic - *WoLLIC 2002 selected papers*, 2005.
3. M. Ayala-Rincón and F. Kamareddine. Unification via the λs_e-Style of Explicit Substitution. *The Logical Journal of the Interest Group in Pure and Applied Logics*, 9(4):489–523, 2001.
4. H. P. Barendregt. *The Lambda Calculus : Its Syntax and Semantics (revised edition)*. North Holland, 1984.
5. D. Briaud. An explicit Eta rewrite rule. In *Typed lambda calculi and applications*, volume 902 of *LNCS*, pages 94–108. Springer, 1995.
6. G. Dowek, T. Hardin, and C. Kirchner. Higher-order Unification via Explicit Substitutions. *Information and Computation*, 157(1/2):183–235, 2000.
7. F. L. C. de Moura, F. Kamareddine and M. Ayala-Rincón. Second order matching via explicit substitutions. In *11th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning*, volume 3452 of *LNCS*. pages 433–448, Springer, 2005.
8. B. Guillaume. The λs_e-calculus Does Not Preserve Strong Normalization. *J. of Func. Programming*, 10(4):321–325, 2000.
9. T. Hardin. Eta-conversion for the languages of explicit substitutions. In *Algebraic and logic programming*, volume 632 of *LNCS*, pages 306–321. Springer, 1992.
10. G. Huet. An analysis of böhm's theorem. *TCS*, 121:145–167, 1993.
11. F. Kamareddine and R. P. Nederpelt. A useful λ-notation. *TCS*, 155:85–109, 1996.
12. F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with Explicit Substitutions. In *Proc. of PLILP'95*, volume 982 of *LNCS*, pages 45–62. Springer, 1995.
13. F. Kamareddine and A. Ríos. Relating the $\lambda\sigma$- and λs-Styles of Explicit Substitutions. *Journal of Logic and Computation*, 10(3):349–380, 2000.
14. D. Kesner. Confluence of extensional and non-extensional λ-calculi with explicit substitutions. *TCS*, 238(1-2):183–220, 2000.
15. C. Liang and G. Nadathur. Tradeoffs in the Intensional Representation of Lambda Terms. In S. Tison, editor, *Rewriting Techniques and Applications (RTA 2002)*, volume 2378 of *LNCS*, pages 192–206. Spinger-Verlag, 2002.
16. P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate in Proceedings of TLCA'95. *LNCS*, 902, 1995.
17. G. Nadathur. A Fine-Grained Notation for Lambda Terms and Its Use in Intensional Operations. *J. of Func. and Logic Programming*, 1999(2):1–62, 1999.
18. G. Nadathur. The Suspension Notation for Lambda Terms and its Use in Metalanguage Implementations. In *Proceedings Ninth Workshop on Logic, Language, Information and Computation (WoLLIC 2002)*, volume 67 of *ENTCS*, 2002.
19. G. Nadathur and D. S. Wilson. A Notation for Lambda Terms A Generalization of Environments. *TCS*, 198:49–98, 1998.
20. R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected papers on Automath*. North-Holland, 1994.
21. A. Ríos. *Contribution à l'étude des λ-calculs avec substitutions explicites*. PhD thesis, Université de Paris 7, 1993.