A Flexible Framework for Visualisation of Computational Properties of General Explicit Substitutions Calculi [5]

F. L. C. de Moura, A. V. Barbosa, M. Ayala-Rincón[1,2,3]

Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Brazil

F. Kamareddine[4]

School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland

Abstract

SUBSEXPL is a system originally developed to visualise reductions, simplifications and normalisations in three important calculi of explicit substitutions and has been applied to understand and explain properties of these calculi and to compare the different styles of making explicit the substitution operation in implementations of the λ-calculus in de Bruijn notation. The system was developed in OCaml and now it can be executed inside the Emacs editor within a new *mode* which allows a very easy interaction. The use of special symbols makes its application very useful for students because the notation on the screen is as close as possible to that on the paper. In addition to dealing the λ-calculus and explicit substitutions calculi in de Bruijn notation, now it is possible to work with the λ-calculus and with several calculi of explicit substitutions using also representation of variables with names. Moreover, in contrast to the original version of the system, that was restricted to three specific calculi of explicit substitution, the new version allows the inclusion of new calculi by giving as input their grammatical descriptions. SUBSEXPL has been used with success for teaching basic properties of the λ-calculus and for illustrating the computational impact of selecting one kind of representation of variables (either names or indices) and a specific style of making explicit substitutions in real implementations based on the λ-calculus.

Keywords: Term rewriting systems, calculi of explicit substitutions, λ-calculi

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

1 Introduction

The system SUBSEXPL [14] was developed in OCaml as a system to simulate and compare calculi of explicit substitutions in de Bruijn notation (variables as indices). In this work we present an extension of this system that has many additional features that simplify the user interaction through a dedicated use inside Emacs, that allows for the treatment of calculi with variables as names too and that is much more flexible than the original system, because now it is possible to insert new calculi by giving as input their grammatical descriptions.

In the last twenty years, much work has been done in the field of explicit substitutions [27,1,7,23,31,3,21,12,18,16,2,15,26]. These developments have illustrated the usefulness of explicit substitutions calculi for practical notions like the implementation of typed functional programming languages [30,35,20] and higher-order proof assistants [11,33,9,6]. SUBSEXPL concentrates on the simulation of the application of the rewriting rules [5] of different calculi. Some of these calculi were developed using de Bruijn indexes that are very adequate for implementations because one does not have to deal with α-equivalence, other calculi use names. Named notation is good for humans but α-equivalence classes need to be treated carefully. Originally, SUBSEXPL implemented three important calculi (carefully compared in [2] with help of earlier prototypes of the system) that use de Bruijn indexes:

(i) The $\lambda\sigma$-style [1] which introduces two different sets of entities: one for terms and one for substitutions.

(ii) The λs-style [23] which makes use of the philosophy of de Bruijn's *Automath* [32] elaborated in the new item notation [22]. The philosophy states that terms are built by applications (a function applied to an argument), abstraction (a function), substitution or updating. The advantages of this philosophy include remaining as close as possible to the familiar λ-calculus (cf. [22]).

(iii) The suspension calculus [31], which introduces three different sets of entities: one for terms, one for environments and one for lists of environments.

The new extension of SUBSEXPL in this paper, so called SUBSEXPL 2.0, allows the definition of new calculi in just a few steps in both de Bruijn or named notations. In this way, the user can play with his/her own calculi, simulate reductions and normalisations, export the latex code, and have at his/her disposition many other features that were not available in the original

[1] Email: flaviomoura@unb.br
[2] Email: alexvicentebarbosa@gmail.com
[3] Email: ayala@unb.br
[4] Email: fairouz@macs.hw.ac.uk
[5] Work supported by Fundação de Apoio Pesquisa do Distrito Federal - FAP-DF 8-004/2007

implementation.

On the one hand, several of the great challenges involving explicit substitutions calculi were already solved, but on the other hand one can say that they are not completely understood because important properties of calculi are not known: it is still an open question whether the suspension calculus [31] preserves strong normalisation (PSN). In addition, it is of practical interest the development of an explicit substitution calculus in de Bruijn notation that satisfies simultaneously the properties presented below [29]. In fact, just recently a calculus of explicit substitutions that satisfies the desired computational properties was developed [26]. The desired properties of an explicit substitutions calculus, say λc, include:

(a) **Simulation of one-step β-reduction**: if $t \to_\beta t'$ then $t \to^+_{\lambda c} t'$.

(b) **Confluence (CR) on open terms** (and hence on closed terms): if one extends the grammar of the calculus with the so called meta-variables, and M is a term with possible occurrences of meta-variables then, if $M_1 \,{}^*_{\lambda c}\!\leftarrow M \to^*_{\lambda c} M_2$ then there exists a λ-term M_3 such that $M_1 \to^*_{\lambda c} M_3 \,{}^*_{\lambda c}\!\leftarrow M_2$.

(c) **Strong Normalisation (SN) for λc-typed terms**: If the λc-term can be typed then t does not have infinite reductions.

(d) **Preservation of SN (PSN)**: if all reductions from the λ-term t are finite in the λ-calculus then so are all the reductions from t in the λc-calculus.

(e) **Full Composition (FC)**: For all terms t, t' and variable x, $t[x/t'] \to^*_{\lambda c} t\{x/t'\}$. In other words, the implicit substitution implements the explicit one.

One of features of SUBSEXPL is that it helps the understanding of these properties.

This paper is organised as follows: a complete description of the new version of SUBSEXPL is given in Section 2. An example on how to define a new calculus in named notation is presented in subsection 2.1. A calculus in de Bruijn notation is presented in subsection 2.2 together with the well-known counter-example of Melliès. In subsection 2.3 we illustrate some educational applications of the system. We conclude in Section 4.

2 Description of SUBSEXPL

SUBSEXPL 2.0 is an OCaml implementation that uses a syntactic extension based on the revised syntax provided by the pre-processor Camlp5 (`http://pauillac.inria.fr/~ddr/camlp5/`). It permits the definition of new calculi and its rewriting rules in an easy way because it implements a generic notion of term based on a first-order signature [5] given by:

```
type expression =
    [ T of string and array expression
    | V of string ];
```

In addition, a notion of a (meta-) capture avoiding substitution, denoted by {x := N}, and of an explicit unary substitution are available. The meta-substitution is implemented according to the revised syntax as follows:

```
value rec replace (x, n) term =
    match term with
[ V _ when x = term -> n
| V _                -> term
| <:generic< L(^y,^m,^t) >> when y = x -> term
| <:generic< L(^y,^m,^t) >> when not(List.mem y
                                (Variable.free_variables n)) ->
  let m = replace (x, n) m
  in
    <:generic< L(^y,^m,^t) >>
| <:generic< L(^y,^m,^t) >>            ->
  let fv = Variable.free_variables <:generic< A(^n,^m) >> in
  let name = ref (Variable.next_name y) in
  let z = do {
    while (List.mem name.val fv) do {
      name.val := Variable.next_name name.val }
    ;
    name.val } in
  let m = replace (y, z) m in
  let m = replace (x, n) m
  in
    <:generic< L(^z,^m,^t) >>
| T name args -> T name (Array.map (replace (x, n)) args) ]
```

The implemented notion of explicit substitution, whose reserved notation is [x := N], allows the definition of new calculi of explicit substitutions that uses unary substitutions. In the next section we present, as an example, a calculus of explicit substitutions with such a unary substitution. The presentation is user friendly in the sense that it is as close as possible to the paper and pencil representation and non-ascii symbols like λ and arrows are provided by the x-symbol package that is installed with SUBSEXPL. The interface used is the GNU Emacs system (`http://www.gnu.org/software/emacs/`) which is an extensible text-editor with support text editing. A SUBSEXPL emacs mode were developed to allow an easier way to perform reductions. This emacs mode, shown in Figure 1, allows among other things, the evaluation of an expression, of a region or of the whole buffer, as well as, to start/interrupt

or kill the SUBSEXPL interactive mode. SUBSEXPL commands can be executed directly in the SUBSEXPL interactive mode, or can be typed inside a subsexpl file (i.e., file with extension `.se`) before being evaluated.

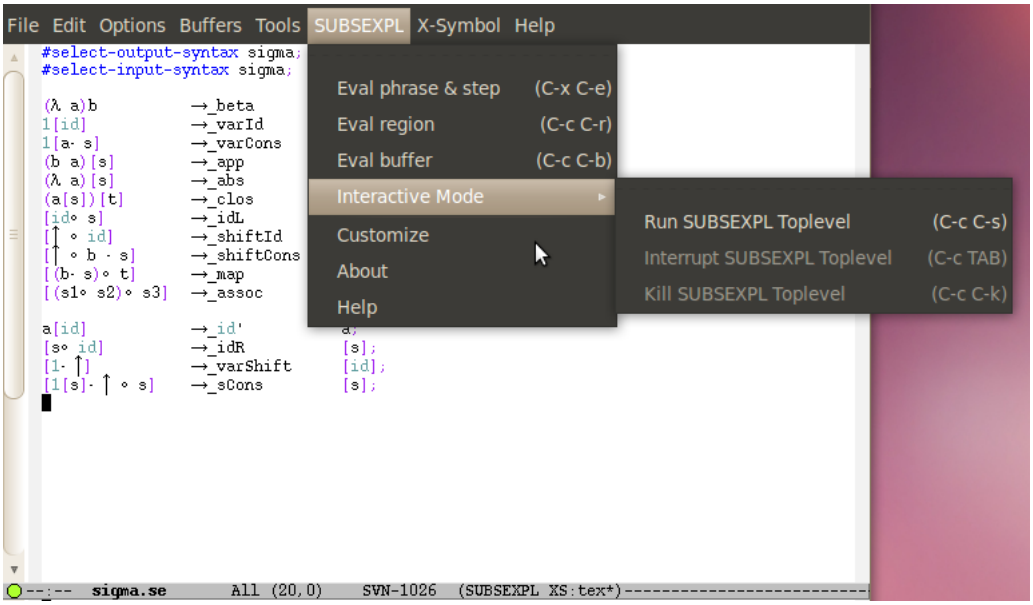

Fig. 1. The SUBSEXPL menu

In the next subsection, one can see how easy it is to define a calculus with names and unary substitution in the SUBSEXPL system.

2.1 The λex calculus

The λex calculus [26] is a calculus with explicit substitutions that uses named notation. The λex calculus is obtained by extending the λx calculus [27,34,7] with one rewriting rule to specify the composition of dependent substitutions and one equation to specify the commutation of independent substitutions, as follows:

$$t ::= x \mid (t\ t) \mid (\lambda x.t) \mid t[x/t]$$

$$t[x/u][y/v] =_{\mathtt{C}} t[y/v][x/u],\ \text{if } y \notin \mathtt{fv}(u)\ \&\ x \notin \mathtt{fv}(v)$$

$$\begin{array}{lll}
(\lambda x.t)\ u & \to_{\mathtt{B}} & t[x/u] \\
x[x/u] & \to_{\mathtt{Var}} & u \\
t[x/u] & \to_{\mathtt{Gc}} & t \qquad \text{if } x \notin \mathtt{fv}(t) \\
(t\ u)[x/v] & \to_{\mathtt{App}} & t[x/v]\ u[x/v] \\
(\lambda y.t)[x/v] & \to_{\mathtt{Lam}} & \lambda y.t[x/v] \\
t[x/u][y/v] & \to_{\mathtt{Comp}} & t[y/v][x/u[y/v]] \quad \text{if } y \in \mathtt{fv}(u)
\end{array}$$

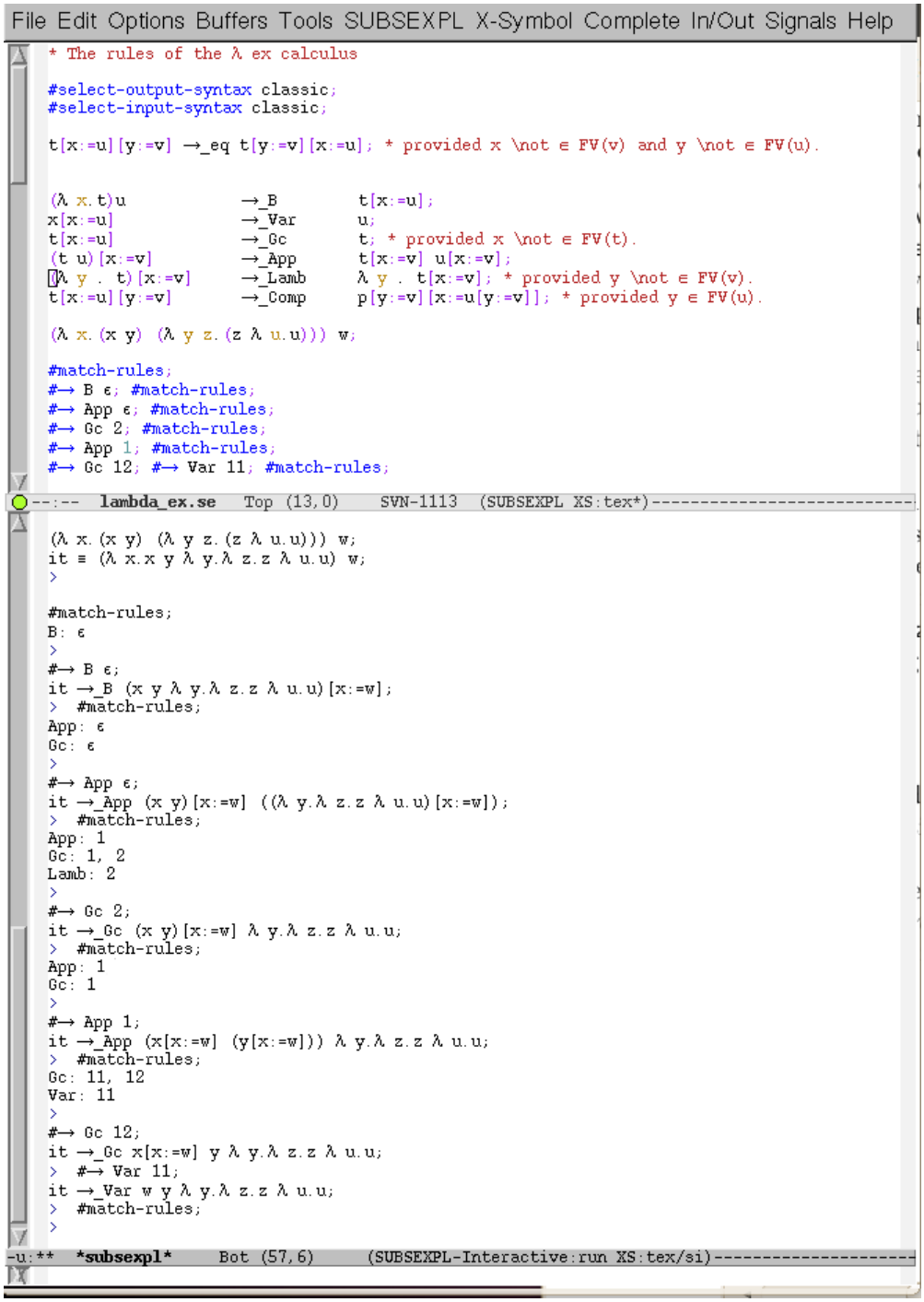

Fig. 2. A reduction in the λex calculus

The specification of the λex calculus in the SUBSEXPL system is presented in Figure 2. The selection of the `classic` syntax that is loaded in the beginning of the file `lambda_ex.se` shown on the top of Figure 2 allows the user to define a calculus in named notation. The presentation of the rules

follows exactly the "paper and pencil" one, and the current limitations of the current implementation are that equations and conditional rules cannot be expressed yet. In order to perform a correct reduction from a system with conditional rules, the system leaves to the user the decision of when or not the conditional rule is really applicable.

As an example, consider the (one-step) reduction of the term $(\lambda x.(x\ y)(\lambda yz.(z\ \lambda u.u)))w$ shown at the bottom of Figure 2. The command `#match-rules` matchs all the rules defined so far against the given term. The system lists all the rules with the corresponding positions that can be applied up to conditions; in this case, only the `B` rule can be applied at the root position (denoted by ϵ) of the term. In order to apply the rule `B` at the root of the term, we use the command `#→ B` ϵ (typed as `#\to B \epsilon`), and the reduct `(x y` λ `y.`λ `z.z` λ `u.u)[x:=w]` is displayed exactly as one would do with paper and pencil. At this point, the command `#match-rules` returns that both the rules `App` and `Gc` can be applied at the root of the current term. Since `Gc` is a conditional rule, the user must first check if the condition is satisfied. In this case, it is not satisfied because the variable `x` has a free occurrence in the term `(x y` λ `y.`λ `z.z` λ `u.u)` and the sole option is to apply the rule `App` and we get the term `(x y)[x:=w]` `((`λ`y.`λ`z.z` λ`u. u)[x:=w])`. At this point the command `#match-rules` returns one `App`-redex at position `1`, two `Gc`-redexes at positions `1` and `2`, and a `Lam`-redex at position 2. The rule `Gc` cannot be applied at position `1` due to the side condition, but we can apply it at position `2`, since `x` does not occur in the term λ`y.`λ`z.z` λ`u. u`. The pending substitution `[x:=w]` needs to be propagated over the subterms `x` and `y`, and finally after an application of `Gc` and `Var` we get the normal form `w y` λ `y.`λ `z.z` λ `u.u`. After that, one can see that the command `#match-rules` returns an empty string because there is no redex (reducible expression) left, and no rule can be applied. The commands can be typed directly in the SUBSEXPL interactive mode, but if it is important to store the rules and the reduction then all this information must be typed in a file with extension `.se` (cf. Figure 2).

2.2 Calculi in de Bruijn notation

The so called de Bruijn notation [13] uses indexes to represent bound and free variables. In this notation, free variables are stored in a list that represents the context of the term, and its reference corresponds to its position in this list. For instance, the λ-term $\lambda xy.xv(\lambda z.yxzw)$ is represented by $\lambda\lambda 2\ 3(\lambda 2\ 3\ 1\ 5)$ in the context $[v, w]$, and as $\lambda\lambda 2\ 4(\lambda 2\ 3\ 1\ 4)$ in the context $[w, v]$. As an example of a calculus in de Bruijn notation, consider the $\lambda\sigma$-calculus [1] that is defined by the following grammar and rules:

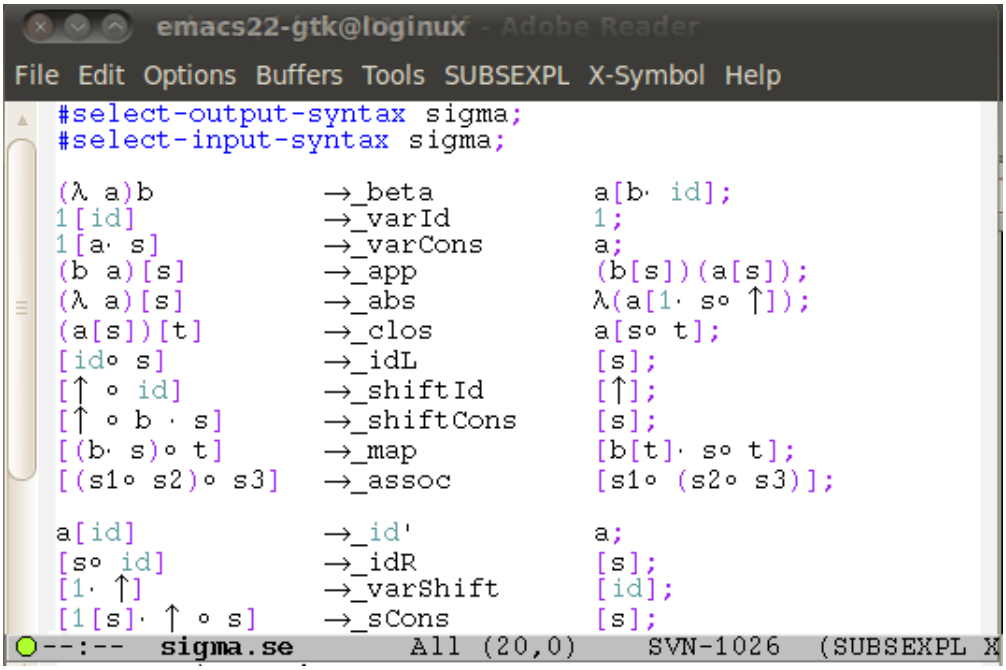

Fig. 3. The $\lambda\sigma$-calculus

TERMS $t ::= \underline{1} \mid (t\ t) \mid (\lambda t) \mid t[s]$

SUBSTITUTIONS $s ::= id \mid \uparrow \mid a.s \mid s \circ s$

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

The rewriting system of the $\lambda\sigma$-calculus is defined directly in a text file with

extension (`.se`) as shown in Figure 3. The directives `#select-output-syntax sigma` and `#select-input-syntax sigma` concerns the definition of the grammar. The former directive defines the way SUBSEXPL will output terms in the buffer, and the latter, the input grammar. This simple and friendly interaction allow us to give a much more readable presentation of Melliès's counter-example than the one presented in previous version of the system [14].

2.2.1 Melliès counter-example

In [28], Melliès proved that the $\lambda\sigma$-calculus does not preserve strong normalisation. This proof consists in building, by an adequate combination of the rules, an infinite derivation from the well-typed λ-term $\lambda((\lambda(\lambda 1)((\lambda 1)1))((\lambda 1)1))$. In Figure 4, one can see the initial steps of this infinite derivation with the very same notation that one would do on paper. This derivation corresponds to two applications of the duplication lemma as presented in [28]. The duplication lemma states that, for any λ-term t, $(\lambda 1)1 \cdot id \circ t \to^{+}_{\lambda\sigma} 1[1[t] \cdot t \circ \uparrow \circ (1[t] \cdot id)] \cdot t$. In line 7 of Figure 4, the generated term contains $(\lambda 1)1 \cdot id \circ (\lambda 1)1 \cdot id$ as subterm which is obtained after two applications of the rule `beta` followed by the rule `clos` that combines two pending substitutions into a new one. By the duplication lemma, this subterm reduces to $1[1[(\lambda 1)1 \cdot id] \cdot ((\lambda 1)1 \cdot id) \circ \uparrow \circ (1[(\lambda 1)1 \cdot id] \cdot id)] \cdot (\lambda 1)1 \cdot id$ that is a subterm of the term that is in line 25. The application of the duplication lemma is a sequence of 9 rewriting steps given by the application of the rules `map`, `app`, `idL`, `abs`, `beta`, `clos`, `map`, `varCons` and `assoc`, in this order. Following the reduction in Figure 4 one can see two successive applications of the duplication lemma: the first one starts at line 8 with an application of the rule `map` at position 12 that propagates the substitution `(λ 1) 1 · id` over a list of terms. The `app` rule applied at position 123, propagates the substitution `(λ 1) 1 · id` over an application. The rule `idL` at position 122 removes an occurrence of the substitution `id` in the left of a composition of substitutions. The rule `abs` at position 1231 generates the new `beta`-redex at position 121 that is reduced in the next step. The two pending substitutions `1·((λ 1) 1· id)∘ ↑` and `1[(λ 1) 1· id]· id` are combined by an application of the rule `clos` at position 121. The next application of `map` at position 1212 propagates the subterm `1[(λ 1) 1· id]· id` over a list of terms, and the rule `varCons` at position 12121 eliminates the substitution `id`, and finally the rule `assoc` applied at position 12122 generates a term that has `1[1[(λ 1) 1·id]· ((λ 1) 1·id)∘ ↑ ∘ 1[(λ 1) 1·id]·id]· (λ 1) 1· id` as subterm that has exactly the expected form $1[1[t] \cdot t \circ \uparrow \circ (1[t] \cdot id)] \cdot t$ for $t = (\lambda 1)1 \cdot id$. The advantage of this presentation is the latex-like notation that is presented in the subsexpl emacs mode. In this way, it is easier to follow reductions that can also be exported as a latex files.

Fig. 4. The counter-example of Melliès

2.3 Teaching λ-calculus with SUBSEXPL

The system SUBSEXPL have been used in both graduate and undergraduate courses to teach the (untyped) λ-calculus. Since the notation used by SUBSEXPL is exactly the same as that presented in classes, this tool turned out to be a very useful support for students. The initial example that makes students realise how basic operations can be done in a language with a grammar as simple as $M ::= x \mid (M\ M) \mid (\lambda x.M)$ concerns the operations with the so called Church numerals that are λ-terms that codify natural numbers in the λ-calculus as follows: $C_n \equiv \lambda fx.f(f(f...(fx)))$, where the body of the abstraction has n occurrences of the parameter f. On the top of Figure 5, one can see the file `church.se` that contains the presentation of the β- and η-reductions. In the next lines of the same file, identifiers are used to codify arbitrary λ-terms: for instance, the λ-term λ `x y . y x` that represents the exponential operator for Church numerals is identified by `Aexp` by typing `Aexp \equiv \lambda x y . y x`. Similarly, the second and the third Church numerals are identified by `C2` and `C3` in the file `church.se`. As a running example, we will evaluate the expression `Aexp C2 C3` that will compute the exponential of `C2` to the power `C3`. The result after running the subsexpl top level and replacing all the identifiers by the corresponding λ-term (`#expand-macros`) is shown in the bottom of Figure 5. The positions where the defined rules (in this case, β and η) can be applied are listed by the command `#match-rules`: initially, there is only one β-redex at position 1, and there are no η-redexes. The reduction can be performed stepwise, or a normalisation strategy can be applied and the system outputs the whole reduction that ends with a normal form.

Teaching the adequacy of the λ-calculus can also benefit the use of SUBSEXPL. The use of identifiers to represent complex and long λ-terms allows a clear presentation of important notions like recursion. In fact, at the top of Figure 6, one can see a short presentation of the factorial function built from the fixpoint operator `Y` and more basic constructions like the predicate `ISZERO` that checks if its argument is the Church numeral zero, the multiplication operator `MULT` and the predecessor function `PRED`. The bottom of Figure 6 shows the tail of the rather long computation of the term `FACT C3` that corresponds to the factorial of 3: the last term of the reduction is the Church numeral 6, as expected. Such constructions can be done step by step with the students who can now follow more complex constructions and run their own examples.

3 Related Work

Non trivial rewriting reductions are usually difficult to simulate by hand. This motivated the development of several rewriting tools [10,8,25], each one with specific features to deal with (term) rewriting systems. The system SUB-

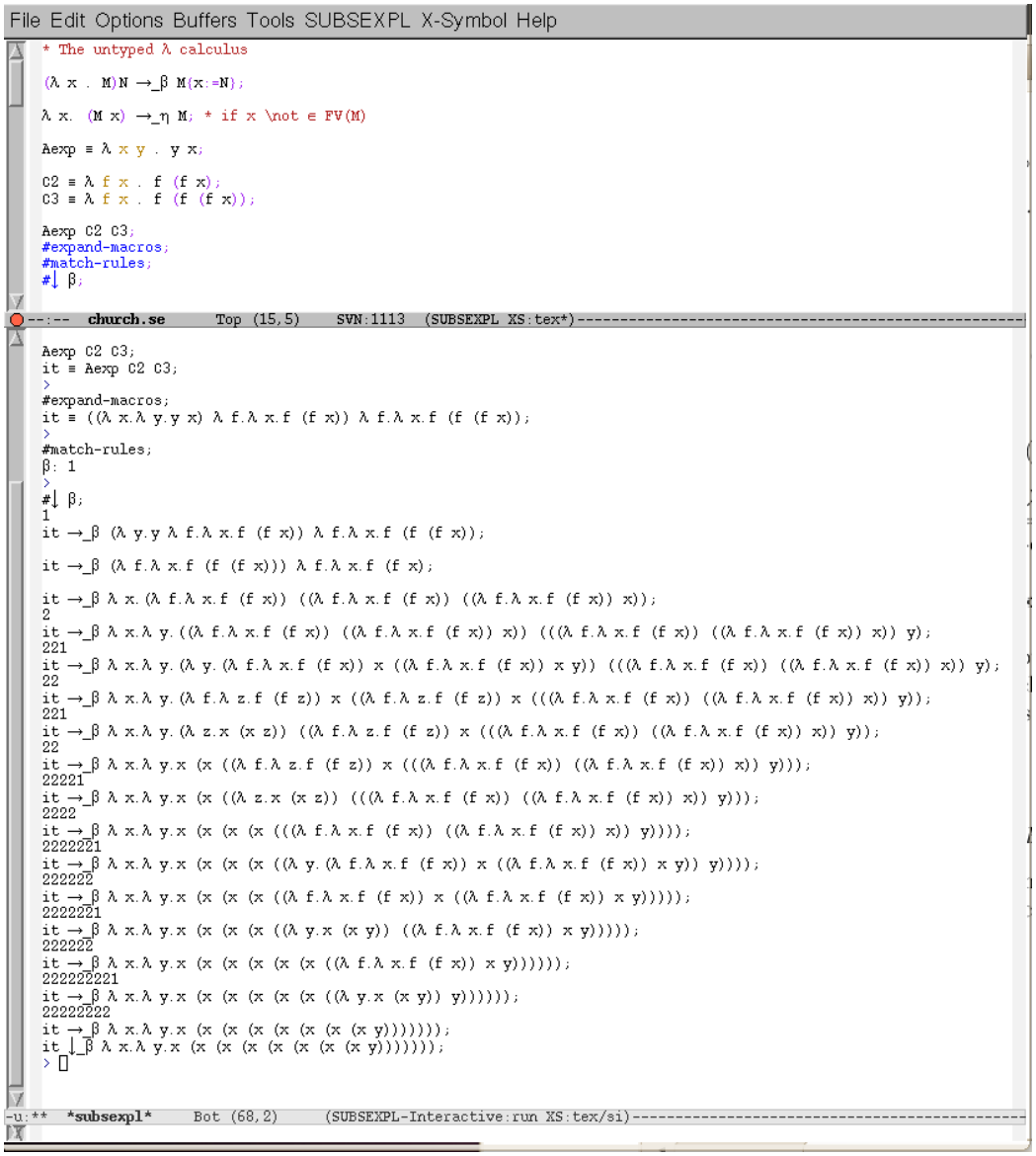

Fig. 5. Church Numerals

SEXPL has some similarities with these systems on what concerns the simulation of rewriting systems; systems like CIME focus on the proofs of confluence and termination, while SUBSEXPL is focused on the simulation and comparison of explicit substitution calculi. In particular, SUBSEXPL intends to fill the lack of tools for teaching λ-calculus and its extensions with explicit substitutions. The presented version is very successful in this task since the notation used is the very same that is used on paper.

The use of SUBSEXPL for teaching λ-calculus and explicit substitutions calculi is simplified by the emacs interface augmented with the x-symbol package which makes the output very close to usual writing; it also allows going

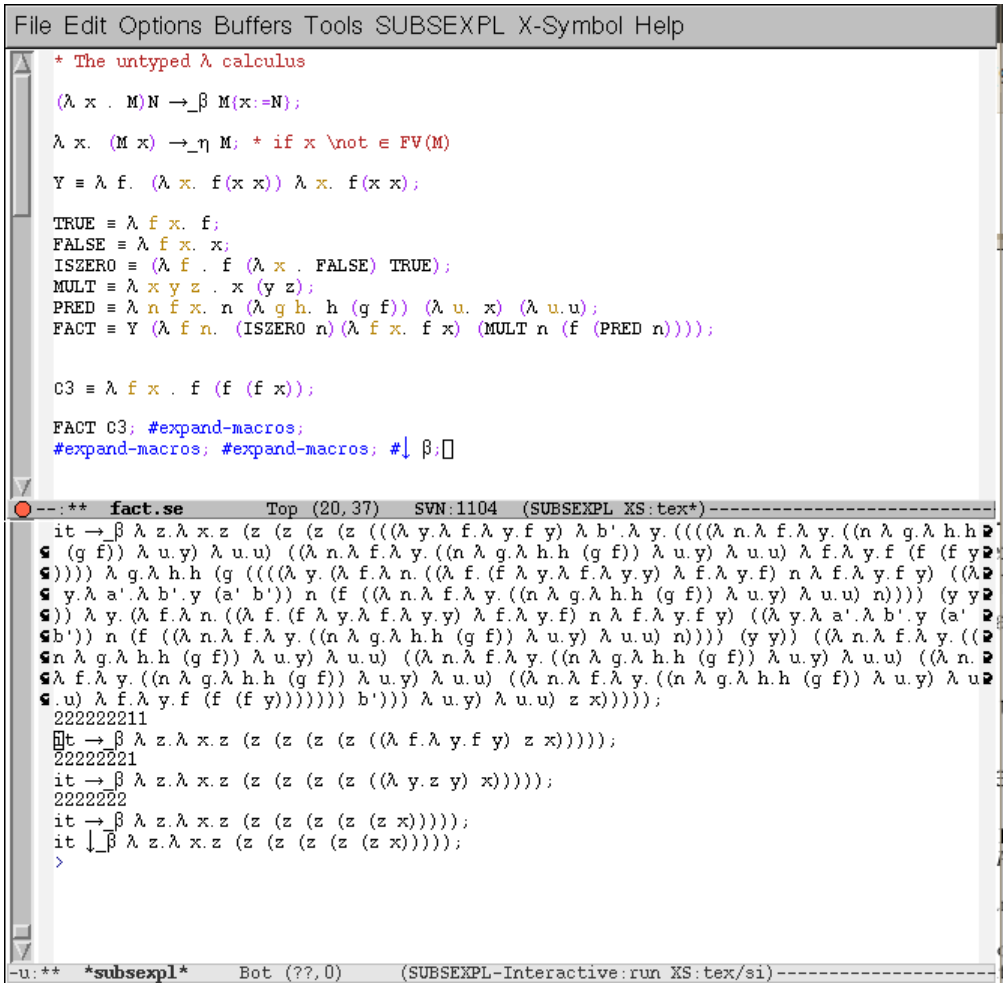

Fig. 6. Towards the expressive power of the λ-calculus

forth and back in the reductions, or to save the reductions in files and also allows to export the latex code. There exists some tools and nice experiments in this direction; for example, in [19], Huet shows how a functional programming language can be used to assist with the understanding of non-trivial properties of the λ-calculus such as the Böhm's theorem.

4 Conclusions and Future Work

We presented an extension of the SUBSEXPL system which is an OCaml implementation for comparing, simulating and studying λ-calculus and explicit substitutions calculi [24]. The source code is freely available at the SUBSEXPL web page: `http://www.cic.unb.br/~flavio/subsexpl/index.html`.

We showed how SUBSEXPL has been used for both educational and research purposes with calculi using either variables as names or de Bruijn in-

dices. In this extension, the user can easily define new calculi of explicit substitutions, simulate reductions and normalisations, export latex code, among other facilities. The notation presented in the Emacs buffer is a latex-like notation and the user can use alias in order to represent complex λ-terms. This facility is particularly important for teaching the theory of the λ-calculus, basic properties of variations of the λ-calculus as well as elaborated computational operations such as iteration and recursion implemented in variations of the λ-calculus in several styles of explicit substitutions.

Several real specifications and implementations of calculi of explicit substitutions in modern systems use a new hybrid approach known as *locally nameless*] [4] in which bound variables are represented by de Bruijn indexes, and free variables by names. This approach benefits the unitary representation of classes of α-equivalent terms without the need of context for free variables. In its future versions, SUBSEXPL will allow the definition of calculi using the locally nameless approach.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *Journal of Functional 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. *Annals of Pure and Applied Logic*, 134:5–41, 2005.

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

[4] B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In George C. Necula and Philip Wadler, editors, *POPL*, pages 3–15. ACM, 2008.

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

[6] D. Basin. Verification of combinational logic in Nuprl. *Lecture Notes in Computer Science*, 408:333–357, July 1989.

[7] R. Bloo and K. H. Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *Computer Science in the Netherlands*, November 1995.

[8] R. Bündgen, C. Sinz, and J. Walter. Redux 1.5: New facets of rewriting. In Ganzinger [17], pages 412–415.

[9] R. Constable et al. *Implementing Mathematics with the NUPRL Development System.* Prentice-Hall, 1986.

[10] E. Contejean and C. Marché. CiME: Completion Modulo E. In Ganzinger [17], pages 416–419. System Description available at `http://cime.lri.fr/`.

[11] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.2.* INRIA-Rocquencourt, 2008.

[12] R. David and B. Guillaume. A lambda-calculus with explicit weakening and explicit substitution. *Mathematical Structures in Computer Science*, 11(1):169–206, 2001.

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

[14] F.L.C. de Moura, M. Ayala-Rincón, and F. Kamareddine. SUBSEXPL: A Framework for Simulating and Comparing Explicit Substitutions Calculi. *Journal of Applied and Non-classical Logics*, 16(1-2):119–150, 2006.

[15] F.L.C. de Moura, M. Ayala-Rincón, and F. Kamareddine. Higher-Order Unification: A structural relation between Huet's method and the one based on explicit substitutions. *Journal of Applied Logic*, 6(1):72–108, 2008.

[16] F.L.C. de Moura, F. Kamareddine, and M. Ayala-Rincón. Second order matching via explicit substitutions. In F. Baader and A. Voronkov, editors, *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04)*, volume 3452 of *LNAI*, pages 433–448. Springer-Verlag, 2005.

[17] H. Ganzinger, editor. *Rewriting Techniques and Applications, 7th International Conference, RTA-96, New Brunswick, NJ, USA, July 27-30, 1996, Proceedings*, volume 1103 of *Lecture Notes in Computer Science*. Springer, 1996.

[18] T. Hardin and J.-J. Lévy. A Confluent Calculus of Substitutions. *France-Japan Artificial Intelligence and Computer Science Symposium*, December 1989.

[19] G. Huet. An analysis of böhm's theorem. *TCS*, 121:145–167, 1993.

[20] S. P. Jones, editor. *Haskell 98 Language and Libraries - The Revised Report*. Cambridge University Press, 2003.

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

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

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

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

[25] N. Kawaguchi, T. Sakabe, and Y. Inagaki. Terse: A visual environment for supporting analysis, verification and transformation of term rewriting systems. In Martin Wirsing and Maurice Nivat, editors, *AMAST*, volume 1101 of *Lecture Notes in Computer Science*, pages 571–574. Springer, 1996.

[26] D. Kesner. A Theory of Explicit Substitutions with Safe and Full Composition. *Logical Methods in Computer Science*, 5(3:1):1–29, 2009.

[27] R. Lins. A new formula for the execution of a categorical combinators. In *8th Conference on Automated Deduction (CADE)*, volume 230 of *LNCS*, pages 89–98. Springer-Verlag, 1986.

[28] P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate in Proceedings of TLCA'95. *LNCS*, 902, 1995.

[29] A. Mendelzon, A. Ríos, and B. Ziliani. Swapping: a natural bridge between named and indexed explicit substitution calculi. In *5th International Workshop on Higher-Order Rewriting - HOR 2010*, pages 41–46, 2010.

[30] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1991.

[31] 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.

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

[33] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[34] K. H. Rose. Explicit cyclic substitutions. In Michal Rusinowitch and Jean-Luc Remy, editors, *CTRS*, volume 656 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 1992.

[35] X. Leroy *et. al.* The Objective Caml system - release 3.11. Technical report, INRIA, 2008.