Formalizing a Named Explicit Substitutions Calculus in Coq

Washington L. R. de C. Segundo[1], Flávio L. C. de Moura[1⋆], Daniel L. Ventura[2]

[1]Departamento de Ciência da Computação Universidade de Brasília, Brasília, Brazil
[2]Instituto de Informática, Universidade Federal de Goiás, Goiânia, Brazil
wtonribeiro@gmail.com, flaviomoura@unb.br, daniel@inf.ufg.br

Abstract. Explicit Substitutions (ES) calculi are extensions of the λ-calculus that internalize the substitution operation, which is a meta-operation, by taking it as an ordinary operation belonging to the grammar of the ES calculus. As a formal system, ES are closer to implementations of functional languages and proof assistants, and are useful for studying properties of such systems. Nevertheless, several ES calculi do not satisfy important properties related with the simulation of the λ-calculus, such as confluence on open terms, simulation of one step β-reduction, full composition and the preservation of strong normalization (PSN). The latter, which states that λ-terms without infinite reduction sequences can only be finitely reduced in the ES calculus, is easily lost in such extensions. In a recent work, D. Kesner developed the $\lambda\mathtt{ex}$-calculus, which is the first ES calculus that satisfies all the good properties expected for an ES calculus. In this work we present a formalization of the $\lambda\mathtt{ex}$-calculus in the Coq Proof Assistant, which is fully available at `http://www.cic.unb.br/~flavio/publications.html`.

1 Introduction

An Explicit Substitutions (ES) calculus is an extension of the λ-calculus that internalizes the substitution operation by extending its grammar such as:

$$t, u ::= x \mid \lambda x.t \mid t\ u \mid t[x/u] \tag{1}$$

where, beside the variable x, abstraction $\lambda x.t$ and application $t\ u$, there is a term $t[x/u]$ denoting an explicit substitution, i.e., a pending substitution that needs to wait to be performed [4]. The free occurrences of the variable x in t are bound in both $\lambda x.t$ and $t[x/u]$, as usual. As a formal system, an ES calculus is composed by a set of rewriting rules with two parts whose aim is to simulate the λ-calculus. The first part is composed by a rule that starts the simulation of the β-reduction, and the second part is used to explicitly perform the substitution operation. For instance, the $\lambda\mathtt{x}$-calculus [18,19,24,5] is defined by the following

⋆ Corresponding author partially supported by FEMAT.

rewriting system:

$$\begin{array}{lll}
(\lambda x.t)u & \to t[x/u] & \\
x[x/u] & \to u & \\
y[x/u] & \to y & (x \neq y) \\
(t_1\ t_2)[x/u] & \to t_1[x/u]\ t_2[x/u] & \\
(\lambda y.v)[x/u] & \to \lambda y.v[x/u] & (x \neq y)
\end{array}$$

where the first rule starts the simulation of the β-reduction and the other rules are responsible for executing, by propagating through the term's structure, the pending substitution operation. In this way, the β-step $(\lambda x.(x\ y))\ z \to_\beta z\ y$ can be simulated by $(\lambda x.(x\ y))\ z \to (x\ y)[x/z] \to x[x/z]\ y[x/z] \to z\ y[x/z] \to z\ y$.

Most of the defined calculi follow one of two approaches: the first one uses names for declaring variables as presented in (1). The advantage of this approach is that its terms are more readable, but α-equivalence, i.e. the class of terms modulo renaming of bound variables, needs to be carefully manipulated. Therefore, implementations of such calculi are not straightforward. Calculi in this family include the λx-, λex- [16], λlxr- [17], λsub- [21] and λes-calculi [16]. The second approach uses *de Bruijn* indexes [8] to represent variables in such a way that each term has a unique representation and hence α-equivalence is no longer necessary. For instance the λ-term $(\lambda x.(x\ y))\ z$ is represented by $(\lambda(0\ 1))1$, where the 1 inside the λ denotes the position of the variable y in a fixed context, and the 1 outside the λ denotes the second variable in this context. Note also that the 0 is bound by the λ while the 1 inside it represents the same variable as 0 outside thus $(\lambda(0\ 1))\ 0$ represents $(\lambda x.(x\ y))\ y$. Therefore, α-equivalence is not a problem in this approach, but an algebraic manipulation of indexes is necessary while performing substitutions. A simple example of calculus in this approach is given by the λs-calculus [13] whose grammar, rules and an update function $up_k^i(b)$ are given by:

$$a, b ::= n \mid \lambda a \mid a\ b \mid a[n/b]$$

$$\begin{array}{lll}
(\lambda a)b & \to a[0/b] & \\
n[i/b] & \to \begin{cases} n-1 & \text{if } n > i \\ up_0^i(b) & \text{if } n = i \\ n & \text{if } n < i \end{cases} & up_k^i(\lambda a) \to \lambda(up_{k+1}^i(a)) \\
 & & up_k^i(n) \to \begin{cases} n+i & \text{if } n > k \\ n & \text{if } n \leq k \end{cases} \\
(a_1\ a_2)[i/b] & \to a_1[i/b]\ a_2[i/b] & up_k^i(a_1\ a_2) \to up_k^i(a_1)\ up_k^i(a_2) \\
(\lambda a)[i/b] & \to \lambda a[i+1/b] &
\end{array}$$

In this calculus, the previous example can be simulated as follows: $(\lambda(0\ 1))1 \to (0\ 1)[0/1] \to 0[0/1]\ 1[0/1] \to up_0^0(1)\ 0 \to 1\ 0$. Examples of calculi with *de Bruijn* indexes are $C\lambda\xi\phi$ [9], $\lambda\sigma$ [1], λse [14], λsusp [22] and λ_{ws} [7].

The formalization presented in this work takes advantages of both approaches: it uses *de Bruijn* indexes to represent bound variables, while free variables are represented by named variables. This framework, called *the locally nameless representation*, is already available in Coq and a more detailed explanation of its advantages can be found in [6].

2 Formalizing Explicit Substitutions Calculi

In this section we present the basic notions that will be used in our formalization of the λex-calculus[1], an ES calculus with names, in the Coq proof assistant. The notion of terms is general and could be used for any calculus having the syntax as in (1). In fact, the notion of terms given below is an extension of the formalization of the λ-calculus presented in [6]. Terms are defined over a signature of pre-terms whose set of variables is formed by the bound variables (`pterm_bvar`), indexed by natural numbers, and named free variables (`pterm_fvar`). Applications (`pterm_app`), abstractions (`pterm_abs`) and explicit substitutions (`pterm_sub`) have the standard signature.

```
Inductive pterm : Set :=
  | pterm_bvar : nat → pterm
  | pterm_fvar : var → pterm
  | pterm_app : pterm → pterm → pterm
  | pterm_abs : pterm → pterm
  | pterm_sub : pterm → pterm → pterm.
Notation t1 [t2] := (pterm_sub t u).
```

Terms of the λex-calculus are defined by an unary predicate over pre-terms. The intuition is that a term is a pre-term without dangling *de Bruijn* indexes. We extended the predicate `term` to pre-terms with explicit substitution as follows:

```
Inductive term : pterm → Prop :=
  | term_var : ∀ x,
      term (pterm_fvar x)
  | term_app : ∀ t1 t2,
      term t1 → term t2 → term (pterm_app t1 t2)
  | term_abs : ∀ L t1,
      (∀ x, x \notin L → term (t1 ^ x)) → term (pterm_abs t1)
  | term_sub : ∀ L t1 t2,
      (∀ x, x \notin L → term (t1 ^ x)) → term t2 → term (t1 [t2]).
```

where `term_sub` is the constructor that defines the conditions for (*t1* `[`*t2*`]`) to be a term. In this definition, the expression (*t1* `^` *x*) represents the term obtained from *t1* after replacing all the occurrences of the ith de Bruijn index that is in the scope of i binders for x, i.e., all the occurrences of 0 that is not in the scope of a binder, of 1 that is in the scope of one binder, of 2 that is in the scope of two binders, and so on, are replaced for x in *t1*. The condition x `\notin` L avoids the capture of free variables by the substitution operation[2]. Therefore, if (*t1* `^` *x*) and *t2* are terms then (*t1* `[`*t2*`]`) is also a term.

2.1 The λex-calculus

The λex-calculus [15] was the first ES calculus that simultaneously satisfies all the desired properties for an ES calculus: termination, confluence on open terms,

[1] The full formalization is available at `www.cic.unb.br/~flavio/publications.html`
[2] The details can be found in [6]

simulation of one step β-reduction, full composition and preservation of strong normalization (PSN). It is defined by the following rewriting system

$$\begin{aligned}
(\lambda x.t)\ u &\to_{\mathtt{B}} t[x/u] \\
x[x/u] &\to_{\mathtt{Var}} u \\
t[x/u] &\to_{\mathtt{Gc}} t, \text{ if } x \notin \mathtt{fv}(t) \\
(t\ v)[x/u] &\to_{\mathtt{App}} t[x/u]\ v[x/u] \\
(\lambda y.t)[x/u] &\to_{\mathtt{Lamb}} \lambda y.t[x/u] \\
(t[y/v])[x/u] &\to_{\mathtt{Comp}} t[x/u][y/v[x/u]], x \in \mathtt{fv}(v)
\end{aligned} \tag{2}$$

and an equation for permutation of independent substitutions

$$t[y/v][x/u] =_{\mathtt{C}} t[x/u][y/v], \text{ if } x \notin \mathtt{fv}(v) \text{ and } y \notin \mathtt{fv}(u) \tag{3}$$

where the rewriting relation generated by the rules in (2) is denoted by `Bx`, and by `x` if one considers all the rules but `B`. The equivalence relation generated by the conversions α and `C` is denoted by `e`, and the reduction relation generated by each rewriting relation modulo `e` specifies rewriting of `e`-equivalence classes as follows:

$$\begin{aligned}
t \to_{\mathtt{ex}} t' &\text{ iff } \exists s, s' \text{ such that } t =_{\mathtt{e}} s \to_{\mathtt{x}} s' =_{\mathtt{e}} t' \\
t \to_{\lambda\mathtt{ex}} t' &\text{ iff } \exists s, s' \text{ such that } t =_{\mathtt{e}} s \to_{\mathtt{Bx}} s' =_{\mathtt{e}} t'
\end{aligned} \tag{4}$$

The formalization of α-equivalence is unnecessary due to the *locally nameless representation*, as explained in the introduction. Nevertheless, the formalization of the equivalence of terms modulo the equation (3) is not straightforward and constitutes an important contribution of this work. The reduction relations in (4) allow a, possible empty, permutation of substitutions before and/or after the rewriting step, and its formalization is done in several steps as explained below. First, we inductively define the permutation of independent substitutions:

```
Inductive eqc : pterm → pterm → Prop :=
  | eqc_rf: ∀ u, eqc u u
  | eqc_rx: ∀ t u v, term u → term v → eqc (t[u][v]) ((& t)[v][u]) .
```

The constructor `eqc_rf` defines the reflexivity of **eqc**, i.e., of equation (3), and `eqc_rx` permutes two independent substitutions. The independence of the substitutions `[`u`]` and `[`v`]` in t`[`u`][`v`]` is assured by the conditions (**term** u) and (**term** v) because no dangling bound variable is allowed to occur in terms. As a consequence, we have that **eqc** is also symmetric and transitive, and hence **eqc** defines an equivalence relation. The permutation of independent substitutions done by the constructor `eqc_rx` requires an adjustment of the indexes of the term t that is replaced by (& t := `bswap_rec` 0 t), which is obtained from t after replacing each occurrence of the index bound to the substitution `[`u`]` by the one bound to the substitution `[`v`]` and vice-versa. This swap of indexes is formally defined by the idempotent recursive function `bswap_rec`.

The parallel contextual closure of **eqc** is denoted by the permutation of independent substitutions in any position of a term. The parallel contextual closure

generalizes the standard contextual closure by allowing applications of a reduction in parallel positions:

```
(∀ x, x \notin L → p_contextual_closure eqc (t^x) (t'^x)) →
    p_contextual_closure eqc u u' →
    p_contextual_closure eqc (t[u]) (t'[u'])
```

So, in order to take advantage of rewriting facilities of Coq, we need to show the compatibility of λex-contexts w.r.t. the equivalence relation **eqc**. This means, for example, that two term applications that are **eqc** equivalent result in corresponding arguments are **eqc** equivalent. The same happens to abstractions and explicit substitutions **eqc** equivalent terms. In this way, one can directly replace equivalent λex-terms using the `rewrite` tactic of Coq [26]. For this purpose, and since (**p_contextual_closure eqc**) is no longer transitive, we defined the transitive parallel contextual closure of **eqc**, denoted by =e.

Finally, the reduction relation in (4) is parametrized by a relation R as follows: `∃ t' u', (t =e t')/\(contextual_closure R t' u')/\(u' =e u)`, where R will be instantiated either with ex or lex.

The rewriting rules in (2) are formalized via binary predicates over the set of pre-terms. Therefore, the rule B, here called **rule_b**, has type **pterm** → **pterm** → `Prop`, where the first argument corresponds to the left hand side of the rule, and the second argument corresponds to its right hand side:

```
Inductive rule_b : pterm → pterm → Prop :=
    reg_rule_b : ∀ (t u:pterm), body t → term u →
      rule_b (pterm_app(pterm_abs t) u) (t[u]).
Notation "t →_B u" := (rule_b t u).
```

The system x, called `sys_x`, is given by the remaining rules in (2). The rules `Lamb` and `Comp` need to take into account the correct manipulation of bound variables in t while propagating the substitution over a binder operator. In particular, the conditions body $(t[u])$ and $\neg$**term** u in reg_rule_comp assure that the substitutions $[u]$ and $[v]$ are not independent.

```
Inductive sys_x : pterm → pterm → Prop :=
| reg_rule_var : ∀ t, term t → sys_x (pterm_bvar 0 [t]) t
| reg_rule_gc : ∀ t u, term t → term u → sys_x (t[u]) t
| reg_rule_app : ∀ t1 t2 u, body t1 → body t2 → term u →
  sys_x ((pterm_app t1 t2)[u]) (pterm_app (t1[u]) (t2[u]))
| reg_rule_lamb : ∀ t u, body (pterm_abs t) → term u →
  sys_x ((pterm_abs t)[u]) (pterm_abs ((& t)[u]))
| reg_rule_comp : ∀ t u v, body (t[u]) → ¬ term u → term v →
  sys_x (t[u][v]) (((& t)[v])[ u[ v ] ]).

Notation "t →_x u" := (sys_x t u).
```

The result of the (implicit) substitution, replacing all occurrences of index 0 (at the root level), of term t by a term u is written `(t ^^ u)`.

The first non-trivial property that is proved is called *full composition*: it states that the explicit substitution simulates the implicit substitution of the λ-calculus and nothing else:

Lemma full_comp: ∀ t u, body t → **term** u → t[u] -->ex+ (t ^^ u).

Proof. The proof is done by induction on t as in [15], but the induction principle need to be adapted to the locally nameless notation because t is not a term in this notation. In fact, t is a body, and hence the induction is on the body structure. Apart from this, everything else runs as in the original proof.

The simulation of one step β-reduction is another important property of ES calculi. It states that one step β-reduction in the λ-calculus can be simulated in the ES calculus in a finite number of steps:

Lemma sim_beta_reduction : ∀ t t', (t -->Beta t') → (t -->lex* t').

Proof. The proof is done by induction on t -->Beta t'. For the base case one has to show that pterm_app (pterm_abs t) u -->lex* (t ^^ u), which after an application of **rule_b** reduces to t[u] -->ex+ (t ^^ u), and we conclude by lemma full_comp. The inductive cases are straightforward.

2.2 Towards a Formalization of PSN for the λex-calculus

In this subsection we present the steps towards a formalization of the PSN property for the λex-calculus, which is a desirable property for ES, but it can be easily lost. In fact, there are many examples in the literature of ES calculi that do not enjoy PSN, such as the $\lambda\sigma$- [20] and the λse-calculus [10]. The first step is to formalize a many-step strategy for x-terms that preserves λex-normal forms, and which is used to characterize the strongly normalizing λex-terms.

```
Inductive many_step : pterm → pterm → Prop :=
| p_var : ∀ (x : var) (t t' : pterm) (lu lv : list pterm),
  ((NF lex) %% lu) → many_step t t' →
  many_step ((pterm_app ((pterm_fvar x) // lu) t) // lv)
            ((pterm_app ((pterm_fvar x) // lu) t') // lv)
| p_abs : ∀ L (t t' : pterm),
  (∀ x, x \notin L → many_step (t ^ x) (t' ^ x)) →
  many_step (pterm_abs t) (pterm_abs t')
| p_B : ∀ (t u : pterm) (lu : list pterm),
  many_step ((pterm_app (pterm_abs t) u) // lu) (t[u] // lu)
| p_subst1 : ∀ (t u : pterm) (lu : list pterm),
  SN lex u →
  many_step (t[u] // lu) ((t ^^ u) // lu)
| p_subst2 : ∀ (t u u' : pterm) (lu : list pterm),
  (¬ SN lex u) → many_step u u' →
  many_step (t[u] // lu) (t[u'] // lu) .

Notation "t ⇝ t'" := (many_step t t').
```

The strategy above can be simulated in the λex-calculus as stated by the following lemma. And so we proved that it is deterministic, fact that is only stated in [15]. Both proofs are done by induction followed by inversion on the

strategy $\rightsquigarrow$:

```
Lemma perp_proposition : ∀ t t', term t → (t ⇝ t') → (t -->lex+ t').
Lemma det_many_step : ∀ t u v, term t → ((t ⇝ u) ∧ (t ⇝ v) → u = v).
```

The PSN proof for the λex-calculus relies on the fact that strong normalization can be inductively defined. The so called IE property establishes the conditions for a term to be strongly normalizing in the λex-calculus. A term is strongly normalizing (SN) if every reduction sequence starting from it is finite. Therefore, the SN predicate inductively characterizes when a term t is strongly normalizing w.r.t. a reduction relation R by assuring the existence of a natural number n that bounds the maximum size of any reduction from t.

```
Inductive SN_ind
 (n : nat) (R : pterm → pterm → Prop) (t : pterm) : Prop :=
 | SN_intro : (∀ t', R t t' → ∃ k, k < n ∧ SN_ind k R t') → SN_ind n R t.
Definition SN R t := ∃ n, SN_ind n R t.
```

The PSN property for the λex-calculus is proved under the hypothesis that IE holds. We are currently working on the formalization of the IE property, but it is not straightforward, as shown in [15].

```
Hypothesis IE_property : ∀ t u lv, body t → term u →
SN lex u → SN lex ((t ^^ u) // lv) → SN lex ((t[u]) // lv).
```

A perpetual strategy gives either an infinity reduction sequence of a term when it exists, or a reduction sequence leading to some normal form. The theorem perpetuality states that the **many_step** strategy is perpetual by relating it with the SN predicate.

```
Theorem perpetuality : ∀ t t', term t → (t ⇝ t') → SN lex t' → SN lex t.
```

Proof. The proof is by induction on the strategy **many_step**. The IE property is used in the base case for the rule p_subst1.

An inductive characterization of strongly normalizing terms is very convenient because it simplifies proofs. The predicate **ISN** characterizes the set of strongly normalizing terms in the λex-calculus. This claim is proved by the lemma ISN_prop which states the equivalence between **ISN** and SN lex .

```
Inductive ISN : pterm → Prop :=
| isn_var : ∀ (x : var) (lu : list pterm),
  (∀ u, (In u lu) → ISN u) → ISN ((pterm_fvar x) // lu)
| isn_NF : ∀ (u : pterm),
  NF lex u → ISN u
| isn_app : ∀ (u v : pterm) (lu : list pterm),
  ISN (u[v] // lu) → ISN ((pterm_app (pterm_abs u) v) // lu)
| isn_subs : ∀ (u v : pterm) (lu : list pterm),
  ISN ((u ^^ v) // lu) → ISN v → ISN (u[v] // lu)
| isn_abs : ∀ L (u : pterm),
```

$(\forall\ x,\ x\ \texttt{\textbackslash notin}\ L \to \textbf{ISN}\ (u\ \texttt{\^{}}\ x)) \to \textbf{ISN}\ (\mathsf{pterm\_abs}\ u)$.

The preservation of strong normalization for the $\lambda\mathtt{ex}$-calculus is proved by showing that every λ-term without infinite reduction sequences do not have infinite reduction sequences in the $\lambda\mathtt{ex}$-calculus, i.e., if a λ-term t is strongly normalizing then it is also strongly normalizing in the $\lambda\mathtt{ex}$-calculus.

```
Lemma ISN_prop : ∀ t, term t → (ISN t ↔ SN lex t).
Theorem PSN : ∀ t, Lterm t → SN Beta t → SN lex t.
```

Proof. The proof is done by induction on `SN_Beta t`, defined as follows [27]:

```
Inductive SN_Beta : pterm → Prop :=
| sn_beta_var : ∀ (x : var) (lu : list pterm),
  (∀ u, (In u lu) → SN_Beta u) → SN_Beta ((pterm_fvar x) // lu)
| sn_beta_abs : ∀ L (u : pterm),
  (∀ x, x \notin L → SN_Beta (u ^ x)) → SN_Beta (pterm_abs u)
| sn_beta_meta_sub : ∀ (t u : pterm) (lv : list pterm),
  SN_Beta u → SN_Beta ((t ^^ u) // lv) →
  SN_Beta ((pterm_app (pterm_abs t) u) // lv).
```

The full formalization follows [15] and is divided in 5 Coq files that sum up about 4000 lines of code, excluding the framework provided by [6].

Related work There is a formalization of the $\lambda\sigma$-calculus proving its confluence [25]. In [12], termination of both the $\lambda\sigma$- and λs-calculus are formalized in ALF [11]. Nevertheless, both the $\lambda\sigma$- and the λs-calculus uses *de Bruijn* indexes and do not have equations, and hence its formalization do not need a framework for dealing with equivalence classes of terms.

The framework in [6] was used in the formalization of the full composition property for the untyped structural λ-calculus [23].

3 Conclusion and Future Work

Formalization of ES calculi are important because they form the theoretical basis for the implementation of functional languages and proof assistants. In this way, ES calculi allow, as a formal framework, not only the study of properties and further improvements of the implemented systems, but also the study of the λ-calculus itself [3].

In this work, we presented a formalization of the $\lambda\mathtt{ex}$-calculus in Coq using a framework based on locally nameless representation [6]. In this formalization we proved that the $\lambda\mathtt{ex}$-calculus satisfies a number of interesting properties such as full composition, simulation of one step β-reduction and that the **many_step** strategy, which can be simulated in the $\lambda\mathtt{ex}$-calculus, is used to characterize strongly normalizing terms. Assuming the IE property, we proved that the $\lambda\mathtt{ex}$-calculus enjoys the PSN property.

As future work, we will formalize the IE property and metaconfluence, i.e., confluence on open terms for the $\lambda\mathtt{ex}$-calculus. This will provide a complete formalization of the $\lambda\mathtt{ex}$-calculus. We will investigate whether our formalization could be adapted to formalize other calculi that uses names instead of *de Bruijn* indexes, such as the family of ES calculi that acts at a distance [21,2].

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. B. Accattoli and D. Kesner. The structural lambda-calculus. In *19th EACSL Annual Conference on Computer Science and Logic (CSL)*, volume 6247 of *LNCS*, pages 381–395. Springer-Verlag, 2010.
3. B. Accattoli and D. Kesner. The permutative lambda calculus. In Nikolaj Bjørner and Andrei Voronkov, editors, *LPAR*, volume 7180 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2012.
4. M. Bezem, J. W. Klop, and R. de Vrijer, editors. *Term Rewriting Seminar – Terese*. Cambridge University Press, 2003.
5. R. Bloo and K. Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *CSN-95: COMPUTER SCIENCE IN THE NETHERLANDS*, pages 62–72, 1995.
6. A. Charguéraud. The Locally Nameless Representation. *Journal of Automated Reasoning*, pages 1–46, 2011.
7. R. David and B. Guillaume. A lambda-calculus with explicit weakening and explicit substitution. *Mathematical Structures in Computer Science*, 11(1):169–206, 2001.
8. N. G. de Bruijn. Lambda-Calculus Notation with Namefree Formulas Involving Symbols that Represent Reference Transforming Mappings. *Indag. Mat.*, 40:348–356, 1978.
9. N. G. de Bruijn. A namefree lambda calculus with facilities for internal definition of expressions and segments. Technical Report T.H.-Report 78-WSK-03, Technische Hogeschool Eindhoven, Nederland, 1978.
10. B. Guillaume. The λs_e-calculus Does Not Preserve Strong Normalization. *J. of Func. Programming*, 10(4):321–325, 2000.
11. M. Hanus. The alf system. In *PLILP*, pages 423–424, 1991.
12. F. Kamareddine and H. Qiao. Formalizing strong normalization proofs of explicit substitution calculi in alf. *Journal of Automated Reasoning*, 30(1):59–98, 2003.
13. 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.
14. F. Kamareddine and A. Ríos. Extending a λ-calculus with Explicit Substitution which Preserves Strong Normalisation into a Confluent Calculus on Open Terms. *Journal of Functional Programming*, 7:395–420, 1997.
15. D. Kesner. Perpetuality for full and safe composition (in a constructive setting). In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 311–322. Springer, 2008.
16. D. Kesner. A Theory of Explicit Substitutions with Safe and Full Composition. *Logical Methods in Computer Science*, 5(3:1):1–29, 2009.
17. D. Kesner and S. Lengrand. Resource operators for the λ-calculus. *Information and Computation*, 205:419–473, April 2007.

18. R. Lins. A new formula for the execution of categorical combinators. *8th Conference on Automated Deduction (CADE)*, volume 230 of LNCS:89–98, 1986.
19. R. Lins. Partial categorical multi-combinators and church rosser theorems. Technical Report 7/92, Computing Laboratory, University Kent at Canterbury, May 1992.
20. P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. In *Proceedings of TLCA'95*, volume 902 of *LNCS*. Springer-Verlag, 1995.
21. R. Milner. Local bigraphs and confluence: Two conjectures: (extended abstract). *ENTCS*, 175(3):65–73, 2007.
22. G. Nadathur and D. S. Wilson. A Notation for Lambda Terms: A Generalization of Environments. *TCS*, 198:49–98, 1998.
23. Fabien Renaud. *Les ressources explicites vues par la théorie de la réécriture*. PhD thesis, 2011.
24. K. Rose. Explicit cyclic substitutions. In Michaël Rusinowitch and Jean-Luc Rémy, editors, *3rd International Workshop on Conditional Term Rewriting Systems (CTRS)*, volume 656, pages 36–50. Springer-Verlag, 1992.
25. A. Saïbi. Formalization of a lambda-Calculus with Explicit Substitutions in Coq. In *TYPES '94: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 183–202, London, UK, 1995. Springer-Verlag.
26. M. Sozeau. A new look at generalized rewriting in type theory. *J. Formalized Reasoning*, 2(1):41–62, 2009.
27. F. van Raamsdonk. *Confluence and Normalization for Higher-Order Rewriting*. PhD thesis, Amsterdam University, Netherlands, 1996.