A formalized extension of the substitution lemma in Coq

Maria J. D. Lima
Departamento de Ciência da Computação
Universidade de Brasília, Brasília, Brazil
majuhdl@gmail.com

Flávio L. C. de Moura
Departamento de Ciência da Computação
Universidade de Brasília, Brasília, Brazil
flaviomoura@unb.br

The substitution lemma is a renowned theorem within the realm of λ-calculus theory and concerns the interactional behaviour of the metasubstitution operation. In this work, we augment the λ-calculus's grammar with an uninterpreted explicit substitution operator, which allows the use of our framework for different calculi with explicit substitutions. Our primary contribution lies in verifying that, despite these modifications, the substitution lemma continues to remain valid. This confirmation was achieved using the Coq proof assistant. Our formalization methodology employs a nominal approach, which provides a direct implementation of the α-equivalence concept. The strategy involved in variable renaming within the proofs presents a challenge, specially on ensuring an exploration of the implications of our extension to the grammar of the λ-calculus.

1 Introduction

In this work, we present a formalization of the substitution lemma [**?**] in a general framework that extends the λ-calculus with an explicit substitution operator. The formalization is made in the Coq proof assistant [**?**] and the source code is available at:

https://github.com/flaviodemoura/lx_confl/tree/m_subst_lemma

The substitution lemma is an important result concerning the composition of the substitution operation, and is usually presented as follows in the context of the λ-calculus:

> Let t, u and v be λ-terms. If $x \notin FV(v)$ (*i.e.* x does not occur in the set of free variables of the term v) then $\{y := v\}\{x := u\}t =_\alpha \{x := \{y := v\}u\}\{y := v\}t$.

This is a well known result already formalized in the context of the λ-calculus [**?**]. Nevertheless, in the context of λ-calculi with explicit substitutions its formalization is not trivial due to the interaction between the metasubstitution and the explicit substitution operator. Our formalization is done in a nominal setting that uses the MetaLib[1] package of Coq, but no particular explicit substitution calculi is taken into account because the expected behaviour between the metasubstitution operation with the explicit substitutition constructor is the same regardless the calculus. The novel contributions of this work are twofold:

1. The formalization is modular in the sense that no particular calculi with explicit substitutions is taken into account. Therefore, we believe that this formalization could be seen as a generic framework for proving properties of these calculi that uses the substitution lemma in the nominal setting [**?**, **?**, **?**], which means it could allow the reuse and provide an easier form to understand the proof we have provided;

[1] https://github.com/plclub/metalib

Submitted to:
FROM 2023

2. A solution to a circularity problem in the proofs is given. It adds an axiom to the formalization that replaces the set equality by the syntactic equality. In this way, we are allowed to replace/rewrite sets of (free) variables by another sets of (free) variables in arbitrary contexts.

In the following section, we present the general framework and the basics of the nominal approach. In Section 3, we present our definition of metasubstitution and some of its properties. In Section 4, we present the formalization of the main theorem, *i.e.*, the substitution lemma, and we conclude in Section 5.

2 A syntactic extension of the λ-calculus

In this section, we present the framework of the formalization, which is based on a nominal approach [**?**] where variables use names. In the nominal setting, variables are represented by atoms that are structureless entities with a decidable equality:

```
Parameter eq_dec : forall x y : atom, {x = y} + {x <> y}.
```

therefore different names mean different atoms and different variables. The nominal approach is close to the usual paper and pencil notation used in λ-calculus lectures, whose grammar of terms is given by:

$$t ::= x \mid \lambda_x.t \mid t\ t \tag{1}$$

and its main rule, named β-reduction, is given by:

$$(\lambda_x.t)\ u \to_\beta \{x := u\}t \tag{2}$$

where $\{x := u\}t$ represents the term obtained from t after replacing all its free occurrences of the variable x by u in a way that renaming of bound variable may be done in order to avoid the variable capture of free variables. We call t the body of the metasubstitution, and u its argument. In other words, $\{x := u\}t$ is a metanotation for a capture free substitution. For instance, the λ-term $(\lambda_x\lambda_y.x\ y)\ y$ has both bound and free occurrences of the variable y. In order to β-reduce it one has to replace (or substitute) the free variable y for all free occurrences of the variable x in the term $(\lambda_y.x\ y)$. But a straight substitution will capture the free variable y, *i.e.* this means that the free occurrence of y before the β-reduction will become bound after the β-reduction step. A renaming of bound variables may be done to avoid such a capture, so in this example, one can take an α-equivalent[2] term, say $(\lambda_z.x\ z)$, and perform the β-step correctly as $(\lambda_x\lambda_y.x\ y)\ y \to_\beta \lambda_z.y\ z$. Renaming of variables in the nominal setting is done via a name-swapping, which is formally defined as follows:

$$(\!(x\ y)\!)z := \begin{cases} y, & \text{if } z = x; \\ x, & \text{if } z = y; \\ z, & \text{otherwise.} \end{cases}$$

This notion can be extended to λ-terms in a straightfoward way:

$$(x\ y)t := \begin{cases} (\!(x\ y)\!)z, & \text{if } t = z; \\ \lambda_{(\!(x\ y)\!)z}.(x\ y)t_1, & \text{if } t = \lambda_z.t_1; \\ (x\ y)t_1\ (x\ y)t_2, & \text{if } t = t_1\ t_2 \end{cases} \tag{3}$$

In the previous example, one could apply a swap to avoid the variable capture in a way that, a swap is applied to the body of the abstraction before applying the metasubstitution to it: $(\lambda_x\lambda_y.x\ y)\ y \to_\beta \{x :=$

[2] A formal definition of this notion will be given later in this section.

$y\}((y\ z)(\lambda_y.x\ y)) = \{x := y\}(\lambda_z.x\ z) = \lambda_z.y\ z$. Could we have used a variable substitution instead of a swapping in the previous example? Absolutely. We could have done the reduction as $(\lambda_x\lambda_y.x\ y)\ y \rightarrow_\beta \{x := y\}(\{y := z\}(\lambda_y.x\ y)) = \{x := y\}(\lambda_z.x\ z) = \lambda_z.y\ z$, but as we will shortly see, variable substitution is not stable under α-equivalence, while the swapping is stable under α-equivalence, thereby rendering it a more fitting choice when operating modulo α-equivalence.

In what follows, we will adopt a mixed-notation approach, intertwining metanotation with the equivalent Coq notation. This strategy aids in elucidating the proof steps of the upcoming lemmas, enabling a clearer and more detailed comprehension of each stage in the argumentation. The corresponding Coq code for the swapping of variables, named *swap_var*, is defined as follows:

```
Definition vswap (x:atom) (y:atom) (z:atom) := if (z == x) then y else if (z == y) then x else z.
```

therefore, the swap $(\!|x\ y|\!)z$ is written in Coq as *vswap x y z*. In a short example to acquaint ourselves with the Coq notation, let us show how we will write the proofs:

```
Lemma vswap_id: ∀ x y, vswap x x y = y.
```

Proof. The proof is done by case analysis, and it is straightforward in both cases, when $x = y$ and $x \neq y$. □

2.1 An explicit substitution operator

The extension of the swap operation to terms require an additional comment because we will not work with the grammar (1), but rather, we will extend it with an explicit substitution operator:

$$t ::= x \mid \lambda_x.t \mid t\ t \mid [x := u]t \tag{4}$$

where $[x := u]t$ represents a term with an operator that will be evaluated with specific rules of a substitution calculus. The intended meaning of the explicit substitution is that it will simulate the metasubstitution. This formalization aims to be a generic framework applicable to any calculi with explicit substitutions and a named notation for variables. Therefore, we will not specify rules about how one can simulate the metasubstitution, but it is important to be aware that this is not a trivial task as one can easily lose important properties of the original λ-calculus [**?**, **?**].

Calculi with explicit substitutions are formalisms that deconstruct the metasubstitution operation into finer-grained steps, thereby functioning as an intermediary between the λ-calculus and its practical implementations. In other words, these calculi shed light on the execution models of higher-order languages. In fact, the development of a calculus with explicit substitutions faithful to the λ-calculus, in the sense of the preservation of some desired properties were the main motivation for such a long list of calculi with explicit substitutions invented in the last decades [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**].

The following inductive definition corresponds to the grammar (4), where the explicit substitution constructor, named *n_sub*, has a special notation. Instead of writing *n_sub t x u*, we will write [*x* := *u*] *t* similarly to (4). Therefore, *n_sexp* is used to denote the set of nominal lambda expressions equipped with an explicit substitution operator, which, for simplicity, we will refer to as just "terms".

```
Inductive n_sexp : Set :=
 | n_var (x:atom)
 | n_abs (x:atom) (t:n_sexp)
 | n_app (t1:n_sexp) (t2:n_sexp)
 | n_sub (t1:n_sexp) (x:atom) (t2:n_sexp).
Notation "[ x := u ] t" := (n_sub t x u) (at level 60).
```

The *size* and the set *fv_nom* of the free variables of a term are defined as usual:

```
Fixpoint size (t : n_sexp) : nat :=
  match t with
  | n_var x ⇒ 1
  | n_abs x t ⇒ 1 + size t
  | n_app t1 t2 ⇒ 1 + size t1 + size t2
  | n_sub t1 x t2 ⇒ 1 + size t1 + size t2
  end.

Fixpoint fv_nom (t : n_sexp) : atoms :=
  match t with
  | n_var x ⇒ {{x}}
  | n_abs x t1 ⇒ remove x (fv_nom t1)
  | n_app t1 t2 ⇒ fv_nom t1 `union` fv_nom t2
  | n_sub t1 x t2 ⇒ (remove x (fv_nom t1)) `union` fv_nom t2
  end.
```

The action of a permutation on a term, written $(x\ y)t$, is inductively defined as in (3) with the additional case for the explicit substitution operator:

$$(x\ y)t := \begin{cases} (\!|x\ y|\!)v, & \text{if } t \text{ is the variable } v; \\ \lambda_{(\!|x\ y|\!)z}.(x\ y)t_1, & \text{if } t = \lambda_z.t_1; \\ (x\ y)t_1\ (x\ y)t_2, & \text{if } t = t_1\ t_2; \\ [(\!|x\ y|\!)z := (x\ y)t_2](x\ y)t_1, & \text{if } t = [z := t_2]t_1. \end{cases}$$

The corresponding Coq definition is given by the following recursive function:

```
Fixpoint swap (x:atom) (y:atom) (t:n_sexp) : n_sexp :=
  match t with
  | n_var z ⇒ n_var (vswap x y z)
  | n_abs z t1 ⇒ n_abs (vswap x y z) (swap x y t1)
  | n_app t1 t2 ⇒ n_app (swap x y t1) (swap x y t2)
  | n_sub t1 z t2 ⇒ n_sub (swap x y t1) (vswap x y z) (swap x y t2)
  end.
```

The *swap* function has many interesting properties, but we will focus on the ones that are more relevant to the proofs related to the substitution lemma. Nevertheless, all lemmas can be found in the source code of the formalization[3]. The next lemma shows that the *swap* function preserves the size of terms. It is proved by induction on the structure of *t*:

```
Lemma swap_size_eq : ∀ x y t, size (swap x y t) = size t.
```

The *swap* function is involutive, which is also proved by structural induction on *t*:

```
Lemma swap_involutive : ∀ t x y, swap x y (swap x y t) = t.
```

The shuffle property given by the following lemma is also proved by structural induction on the structure of *t*:

```
Lemma shuffle_swap : ∀ w y z t, w ≠ z → y ≠ z → (swap w y (swap y z t)) = (swap w z (swap w y t)).
```

[3] https://github.com/flaviodemoura/lx_confl/tree/m_subst_lemma

Equivariance is another important property of the *swap* function. It states that a swap can uniformly be propagated over the structure of a term:

```
Lemma vswap_equivariance : ∀ v x y z w, vswap x y (vswap z w v) = vswap (vswap x y z) (vswap x y w)
(vswap x y v).

Lemma swap_equivariance : ∀ t x y z w, swap x y (swap z w t) = swap (vswap x y z) (vswap x y w) (swap
x y t).
```

If a variable, say z, is not in the set of free variables of a term t and one swaps z with another variable, say y, then y is not in the set of free variables of the term t. This is the content of the following lemma that can easily be proved using induction on the structure of t:

```
Lemma fv_nom_swap : ∀ z y t, z `notin` fv_nom t → y `notin` fv_nom (swap y z t).
```

The standard proof strategy for the non trivial properties is induction on the structure of these terms. Nevertheless, the builtin induction principle automatically generated for the inductive definition *n_sexp* is not strong enough due to swappings. In fact, in general, the induction hypothesis in the abstraction case, for instance, refers to the body of the abstraction, while the goal involves a swap acting on the body of the abstraction. In order to circunvet this problem, we use an induction principle based on the size of terms:

```
Lemma n_sexp_induction: ∀ P : n_sexp → Prop, (∀ x, P (n_var x)) →
 (∀ t1 z, (∀ t2 x y, size t2 = size t1 → P (swap x y t2)) → P (n_abs z t1)) →
 (∀ t1 t2, P t1 → P t2 → P (n_app t1 t2)) →
 (∀ t1 t3 z, P t3 → (∀ t2 x y, size t2 = size t1 → P (swap x y t2)) → P (n_sub t1 z t3)) → (∀ t, P t).
```

We will use this induction principle to prove that if a certain variable, say x', is not in the set of free variables of a term t then the variable obtained after applying any swap to x' also is not in the set of free variables of the term obtained from t after applying the same swap to t:

```
Lemma notin_fv_nom_equivariance : ∀ t x' x y, x' `notin` fv_nom t → vswap x y x' `notin` fv_nom (swap
x y t).
```

Proof. The proof is by induction on the size of the term t. If t is a variable, say z, then we have that $x' \neq z$ by hypothesis, and we conclude by lemma *swap_neq* \footnote{\url{https://github.com/flaviodemoura/lx_confl/tree/m_subs
If t = *n_abs z t1* then we have that $x' \notin fv(t1)\backslash\{z\}$ by hypothesis. This means that either x' = z or x' is not in $fv(t1)$, *i.e.* *fv_nom t1* in Coq. If x' = z then we have to prove that a certain element is not in a set where it has been removed, and we are done by lemma *notin_remove_3*[4]. Otherwise, x' is not in $fv(t1)$, and we conclude using the induction hypothesis. The application case is straightforward from the induction hypothesis. The case of the explicit substitution, *i.e.* when t = $[z := t2]t1$ we have to prove that *vswap x y x' `notin` fv_nom (swap x y ([z := t2] t1))*. We then propagate the swap over the explicit substitution operator and, by the definition of *fv_nom*, we have to prove that both *vswap x y x' `notin` remove (vswap x y z) (fv_nom (swap x y t1))* and *vswap x y x' `notin` fv_nom (swap x y t2)*. In the former case, the hypothesis *x' `notin` remove z (fv_nom t1)* generates two cases, either x' = z or x' in not in $fv(t1)$, and we conclude with the same strategy of the abstraction case. The later case is straightforward by the induction hypothesis. □

The other direction is also true:

[4]This is a lemma from Metalib library and it states that `forall (x y : atom) (s : atoms), x = y -> y ‘notin‘ remove x s`.

```
Lemma notin_fv_nom_remove_swap: ∀ t x' x y, vswap x y x' `notin` fv_nom (swap x y t) → x' `notin` fv_nom t.
```

2.2 α-equivalence

As usual in the standard presentations of the λ-calculus, we work with terms modulo α-equivalence. This means that λ-terms are identified up to renaming of bound variables. For instance, all terms $\lambda_x.x$, $\lambda_y.y$ and $\lambda_z.z$ are seen as the same term which corresponds to the identity function. Formally, the notion of α-equivalence is defined by the following inference rules:

$$\frac{}{x =_\alpha x}\ (aeq\_var) \qquad \frac{t_1 =_\alpha t_2}{\lambda_x.t_1 =_\alpha \lambda_x.t_2}\ (aeq\_abs\_same)$$

$$\frac{x \neq y \qquad x \notin fv(t_2) \qquad t_1 =_\alpha (y\ x)t_2}{\lambda_x.t_1 =_\alpha \lambda_y.t_2}\ (aeq\_abs\_diff)$$

$$\frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{t_1\ t_2 =_\alpha t_1'\ t_2'}\ (aeq\_app) \qquad \frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{[x := t_2]t_1 =_\alpha [x := t_2']t_1'}\ (aeq\_sub\_same)$$

$$\frac{t_2 =_\alpha t_2' \qquad x \neq y \qquad x \notin fv(t_1') \qquad t_1 =_\alpha (y\ x)t_1'}{[x := t_2]t_1 =_\alpha [y := t_2']t_1'}\ (aeq\_sub\_diff)$$

Each of these rules correspond to a constructor in the *aeq* inductive definition below:

```
Inductive aeq : n_sexp → n_sexp → Prop :=
| aeq_var : ∀ x, aeq (n_var x) (n_var x)
| aeq_abs_same : ∀ x t1 t2, aeq t1 t2 → aeq (n_abs x t1)(n_abs x t2)
| aeq_abs_diff : ∀ x y t1 t2, x ≠ y → x `notin` fv_nom t2 → aeq t1 (swap y x t2) →
  aeq (n_abs x t1) (n_abs y t2)
| aeq_app : ∀ t1 t2 t1' t2', aeq t1 t1' → aeq t2 t2' → aeq (n_app t1 t2) (n_app t1' t2')
| aeq_sub_same : ∀ t1 t2 t1' t2' x, aeq t1 t1' → aeq t2 t2' → aeq ([x := t2] t1) ([x := t2'] t1')
| aeq_sub_diff : ∀ t1 t2 t1' t2' x y, aeq t2 t2' → x ≠ y → x `notin` fv_nom t1' → aeq t1 (swap y x t1') →
  aeq ([x := t2] t1) ([y := t2'] t1').
Notation "t =a u" := (aeq t u) (at level 60).
```

In what follows, we use a infix notation for α-equivalence in the Coq code. Therefore, we write *t =a u* instead of *aeq t u*. The above notion defines an equivalence relation over the set *n_sexp* of nominal expressions with explicit substitutions, *i.e.* the *aeq* relation is reflexive, symmetric and transitive. In addition, α-equivalent terms have the same size, and the same set of free variables:

```
Lemma aeq_size: ∀ t1 t2, t1 =a t2 → size t1 = size t2.
```

```
Lemma aeq_fv_nom : ∀ t1 t2, t1 =a t2 → fv_nom t1 [=] fv_nom t2.
```

The key point of the nominal approach is that the swap operation is stable under α-equivalence in the sense that, $t_1 =_\alpha t_2$ if, and only if $(x\ y)t_1 =_\alpha (x\ y)t_2, \forall t_1, t_2, x, y$. Note that this is not true for renaming substitutions: in fact, $\lambda_x.z =_\alpha \lambda_y.z$, but $\{z := x\}(\lambda_x.z) = \lambda_x.x \neq_\alpha \{z := x\}\lambda_y.x(\lambda_y.z)$, assuming that $x \neq y$. This stability result is formalized as follows:

```
Corollary aeq_swap: ∀ t1 t2 x y, t1 =a t2 ↔ (swap x y t1) =a (swap x y t2).
```

When both variables in a swap does not occur free in a term, it eventually rename bound variables only, *i.e.* the action of this swap results in a term that is α-equivalent to the original term. This is the content of the followin lemma:

```
Lemma swap_reduction: ∀ t x y, x `notin` fv_nom t → y `notin` fv_nom t → (swap x y t) =a t.
```

There are several other interesting auxiliary properties that need to be proved before achieving the substitution lemma. In what follows, we refer only to the tricky or challenging ones, but the interested reader can have a detailed look in the source files [5]. Note that, swaps are introduced in proofs by the rules *aeq_abs_diff* and *aeq_sub_diff*. As we will see, the proof steps involving these rules are trick because a naïve strategy can easily get blocked in a branch without proof. We conclude this section, with a lemma that gives the conditions for two swaps with a common variable to be merged:

```
Lemma aeq_swap_swap: ∀ t x y z, z `notin` fv_nom t → x `notin` fv_nom t → (swap z x (swap x y t)) =a
(swap z y t).
```

Proof. Initially, observe the similarity of the left hand side (LHS) of the α-equation with the lemma *shuffle_swap*. In order to use it, we need to have that both $z \neq y$ and $x \neq y$. If $z = y$ then the right hand side (RHS) reduces to t because the swap is trivial, and the LHS also reduces to t since swap is involutive. When $z \neq y$ then we proceed by comparing x and y. If $x == y$ then both sides of the α-equation reduces to *swap z y t*, and we are done. Finally, when $x \neq y$, we can apply the lemma *shuffle_swap* and use lemma *aeq_swap* to reduce the current goal to *swap z x t =a t*, and we conclude by lemma *swap_reduction* since both z and x are not in the set of free variables of the term t. □

3 The metasubstitution operation of the λ-calculus

The main operation of the λ-calculus is the β-reduction that expresses how to evaluate a function, say $(\lambda_x.t)$, applied to an argument u: $(\lambda_x.t)\ u \to_\beta \{x := u\}t$, where $\{x := u\}t$ is called a β-contractum and represents the result of the evaluation of the function $(\lambda_x.t)$ with argument u. In other words, $\{x := u\}t$ is the result of substituting u for the free ocurrences of the variable x in t. Moreover, it is a capture free substitution in the sense that no free variable becomes bound after a β-reduction. This operation is in the meta level because it is outside the grammar of the λ-calculus, and that's why it is called metasubstitution. As a metaoperation, its definition usually comes with a degree of informality. For instance, Barendregt [**?**] defines it as follows:

$$\{x := u\}t = \begin{cases} u, & \text{if } t = x; \\ y, & \text{if } t = y \text{ and } x \neq y; \\ \{x := u\}t_1\ \{x := u\}t_2, & \text{if } t = \{x := u\}(t_1\ t_2); \\ \lambda_y.(\{x := u\}t_1), & \text{if } t = \lambda_y.t_1. \end{cases}$$

where it is assumed the so called "Barendregt's variable convention":

[5] https://github.com/flaviodemoura/lx_confl/tree/m_subst_lemma

> If $t_1, t_2, \ldots, t_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

This means that we are assumming that both $x \neq y$ and $y \notin fv(u)$ in the case $t = \lambda_y.t_1$. This approach is very convenient in informal proofs because it avoids having to rename bound variables. In order to formalize the capture free substitution, *i.e.* the metasubstitution, there exists different possible approaches. In our case, we perform a renaming of bound variables whenever the metasubstitution is propagated inside a binder. In our case, there are two binders: abstractions and explicit substitutions.

Let t and u be terms, and x a variable. The result of substituting u for the free ocurrences of x in t, written $\{x := u\}t$ is defined as follows:

$$\{x := u\}t = \begin{cases} u, & \text{if } t = x; \\ y, & \text{if } t = y \text{ and } x \neq y; \\ \{x := u\}t_1\ \{x := u\}t_2, & \text{if } t = \{x := u\}(t_1\ t_2); \\ \lambda_x.t_1, & \text{if } t = \lambda_x.t_1; \\ \lambda_z.(\{x := u\}((y\ z)t_1)), & \text{if } t = \lambda_y.t_1, x \neq y \text{ and } z \notin fv(t) \cup fv(u) \cup \{x\}; \\ [x := \{x := u\}t_2]t_1, & \text{if } t = [x := t_2]t_1; \\ [z := \{x := u\}t_2]\{x := u\}((y\ z)t_1), & \text{if } t = [y := t_2]t_1, x \neq y \text{ and } z \notin fv(t) \cup fv(u) \cup \{x\}. \end{cases} \tag{5}$$

and the corresponding Coq code is as follows:

```
Function subst_rec_fun (t:n_sexp) (u :n_sexp) (x:atom) {measure size t} : n_sexp :=
  match t with
  | n_var y ⇒ if (x == y) then u else t
  | n_abs y t1 ⇒ if (x == y) then t else let (z,_) :=
    atom_fresh (fv_nom u ‘union‘ fv_nom t ‘union‘ {{x}}) in n_abs z (subst_rec_fun (swap y z t1) u x)
  | n_app t1 t2 ⇒ n_app (subst_rec_fun t1 u x) (subst_rec_fun t2 u x)
  | n_sub t1 y t2 ⇒ if (x == y) then n_sub t1 y (subst_rec_fun t2 u x) else let (z,_) :=
    atom_fresh (fv_nom u ‘union‘ fv_nom t ‘union‘ {{x}}) in
    n_sub (subst_rec_fun (swap y z t1) u x) z (subst_rec_fun t2 u x)
end.
Definition m_subst (u : n_sexp) (x:atom) (t:n_sexp) :=
  subst_rec_fun t u x.
Notation "{ x := u } t" := (m_subst u x t) (at level 60).
```

Note that this function is not structurally recursive due to the swaps in the recursive calls. A structurally recursive version of the function *subst_rec_fun* can be found in the file *nominal.v* of the *Metalib* library[6], but it uses the size of the term in which the substitution will be performed as an extra argument that decreases with each recursive call. We write [*x*:=*u*]*t* instead of *subst_rec_fun t u x* in the Coq code to represent $\{x := u\}t$.

The following lemma states that if $x \notin fv(t)$ then $\{x := u\}t =_\alpha t$. In informal proofs the conclusion of this lemma is usually stated as a syntactic equality, i.e. $\{x := u\}t = t$ instead of the α-equivalence, but the function *subst_rec_fun* renames bound variables whenever the metasubstitution is propagated inside an abstraction or an explicit substitution, even in the case that the metasubstitution has no effect in a subterm. That's why the syntactic equality does not hold here.

[6] https://github.com/plclub/metalib

Lemma *m_subst_notin*: $\forall\ t\ u\ x, x$ ‘*notin*‘ *fv_nom t* $\rightarrow \{x := u\}t =_a t$.

Proof. The proof is done by induction on the size of the term t using the *n_sexp_induction* principle. The interesting cases are the abstraction and the explicit substituion. We focus in the abstraction case, *i.e.* when $t = \lambda_y.t_1$ and $x \neq y$. In this case, we have to prove that $\{x := u\}(\lambda_y.t_1) =_\alpha \lambda_y.t_1$. The induction hypothesis express the fact that every term with the same size as the body of the abstraction t_1 satisfies the property to be proven:

$\forall t'\ x\ y, |t'| = |t_1| \rightarrow \forall u\ x', x' \notin fv((x\ y)t') \rightarrow \{x' := u\}((x\ y)t') =_\alpha (x\ y)t'$. Therefore, according to the function *subst_rec_fun*, the variable y will be renamed to a new name, say z, such that $z \notin fv(\lambda_y.t_1) \cup fv(u) \cup \{x\}$, and we have to prove that $\{x := u\}\lambda_z.((z\ y)t_1) =_\alpha \lambda_y.t_1$. Since $z \notin fv(\lambda_y.t_1) = fv(t_1)\backslash\{y\}$, there are two cases, either $z = y$ or $z \in fv(t_1)$:

1. $z = y$: In this case, we have to prove that $\{x := u\}\lambda_z.((z\ z)t_1) =_\alpha \lambda_z.t_1$. By the rule *aeq_abs_same* we get $\{x := u\}((z\ z)t_1) =_\alpha t_1$, but in order to apply the induction hypothesis the body of the metasubstitution and the term in the right hand side need to be the same and both need to be a swap. For this reason, we use the transitivity of α-equivalence with $(z\ z)t_1$ as intermediate term. The first subcase is proved by the induction hypothesis, and the second one is proved by the reflexivity of α-equivalence.
2. $z \neq y$: In this case, $x \notin fv(t)$ and we can apply the rule *aeq_abs_diff*. The new goal is $\{x := u\}((z\ y)t_1) =_\alpha (z\ y)t_1$ which holds by the induction hypothesis, since $|(z\ y)t_1| = |t_1|$ and $x \notin fv((z\ y)t_1)$ because $x \neq z$, $x \neq y$ and $x \notin fv(t)$.

The explicit substitution case is also interesting, but it follows a similar strategy used in the abstraction case for t_1. For t_2 the result follows from the induction hypothesis. □

The following lemmas concern the expected behaviour of the metasubstitution. For instance, the next two lemmas show what happens when the variable in the meta-substitution is equal to the one in the abstraction and in the explicit substitution. The proofs were straightforward from the definition of the meta-substitution, where each case is respectively each equivalent one in the definition.

Lemma *m_subst_abs_eq*: $\forall\ u\ x\ t$, $\{x := u\}$(*n_abs x t*) = *n_abs x t*.

Lemma *m_subst_sub_eq*: $\forall\ u\ x\ t1\ t2$, $\{x := u\}$(*n_sub t1 x t2*) = *n_sub t1 x* ($\{x := u\}$*t2*).

The next lemma, named *aeq_m_subst_out* will benefit the strategy used in the previous proof, but it is not straightfoward.

Lemma *aeq_m_subst_out*: $\forall\ t\ t'\ u\ x$, *t* =*a t'* $\rightarrow$ ($\{x := u\}t$) =*a* ($\{x := u\}t'$).

Proof. The proof is by induction on the size of the term *t*. The interesting case is the abstraction. The are two cases based on the definition of the α-equivalence relation: either the binders have the same name or they are different. In the former case, we have to prove ([*x* := *u*] *n_abs y t1*) =*a* ([*x* := *u*] *n_abs y t2*) assumming that *t1* =*a t2*. In both sides of the α-equation, the metasubstitutition need to be propagated over the abstraction, and according to our definition of metasubstitution, one name will be generated for each propagation. The new name to be generate for the term [*x* := *u*] (*n_abs y t1*) (LHS) is such that it is not in the set $fv(\lambda_y.t_1) \cup fv(u) \cup \{x\}$, while the new name to be generated for the term [*x* := *u*] (*n_abs y t2*) (RHS) is such that it is not in the set $fv(\lambda_y.t_2) \cup fv(u) \cup \{x\}$. Since *t1* =*a t2*, by lemma *aeq_fv_nom* the set of free variables of *t1* is equal to the set of free variables of *t2*, and therefore, we can generate just one new name for both propagations of the metasubstitution. If this new name is *x0* then the new goal to be proved is *n_abs x0* (*subst_rec_fun* (*swap y x0 t1*) *u x*) =*a n_abs x0* (*subst_rec_fun* (*swap y x0 t2*) *u x*), which can be proved by the induction hypothesis. If *y* = *x* then *x* $\neq$ *y0*, where y0 is the new

variable generated by applying the definition of the meta-substitution, and the metasubstitution $[x := u]$ has no effect on the LHS, but it can be propagated on the RHS, *i.e.* over the abstraction (*n_abs y0 t2*) but it also has no effect in *t2* because x does not occur free in *t2*. If $y \neq x$ then the metasubstitution can be propagated over the abstraction of the LHS, and similarly we compare x with *y0* to see what happens in the RHS. When $y0 = x$ then the metasubstitution has no effect on the abstraction of the RHS. On the LHS the metasubstitution is propagated since $y \neq x$ but, as in the previous case, it has no effect in *t1* because *y0* does not occur free in *t1*. Now we have to prove that $[x := u](n\_abs\ y\ t1) =_a [x := u](n\_abs\ y0\ t2)$, when $y \neq x$, $y0 \neq x$ and $n\_abs\ y\ t1 =_a n\_abs\ y0\ t2$. Since the set of free variables of *n_abs y t1* is equal to the set of free variables of *n_abs y0 t2*, we can as in the previous case generate only one new name, say *x0* that fulfill the condition to propagate the metasubstitution on both sides of the α-equation. More precisely, $x_0 \notin fv(\lambda_y.t1) \cup fv(u) \cup \{x\}$, and the goal to be proved is $n\_abs\ x0\ ([x := u](swap\ y\ x0\ t1)) =_a n\_abs\ x0\ ([x := u](swap\ y0\ x0\ t2))$. As just one new name was generated, there is no case where the binders of the abstractions are different. Remember that abstractions with different binders were the cause of the circularity problem in the proofs because the application of the rule *aeq_abs_diff* introduces a new swap that will be outside the metasubstitution in this case, whose solution would require a lemma of the form of *swap_m_subst*. Therefore, after an application of the rule *aeq_abs_same*, we conclude with the induction hypothesis. The explicit substitution operation is also interesting, but the proof strategy is similar to the one used in the abstraction case. □

As a corollary, one can join the lemmas *aeq_m_subst_in* and *aeq_m_subst_out* as follows:

`Corollary` *aeq_m_subst_eq*: $\forall\ t\ t'\ u\ u'\ x, t =_a t' \rightarrow u =_a u' \rightarrow (\{x := u\}t) =_a (\{x := u'\}t')$.

Now, we show how to propagated a swap inside metasubstitutions using the decomposition of the metasubstitution provided by the corollary *aeq_m_subst_eq*.

`Lemma` *swap_subst_rec_fun*: $\forall\ x\ y\ z\ t\ u, swap\ x\ y\ (\{z := u\}t) =_a (\{(vswap\ x\ y\ z) := (swap\ x\ y\ u)\}(swap\ x\ y\ t))$.

Proof. Firstly, we compare x and y, since the case $x = y$ is trivial. The proof proceeds by induction on the size of the term t, assuming that $x \neq y$. The tricky cases are the abstraction and explicit substitution. In the abstraction case, *i.e.* when $t = \lambda_{y'}.t_1$ then we must prove that $swap\ x\ y\ (\{z := u\}(n\_abs\ y'\ t1)) =_a \{(vswap\ x\ y\ z) := (swap\ x\ y\ u)\}(swap\ x\ y\ (n\_abs\ y'\ t1))$, and the induction hypothesis states that a swap can be propagated inside a metasubstitution whose body is a term with the same size of *t1*. Firstly, we compare the variables *y'* and z to check whether, according to the definition of the metasubstitution, we should propagate the metasubstitution inside the abstraction of the LHS. When $y' = z$ the metasubstitution is erased according to the definition (5) on both sides of the α-equation and we are done. When $y' \neq z$ then the metasubstitutions on both sides of the α-equation need to be propagated inside the corresponding abstractions. In order to do so, a new name need to be created. Note that in this case, it is not possible to create a unique name for both sides because the name of the LHS cannot belong to the set $fv(\lambda'_y.t_1) \cup fv(u) \cup \{z\}$, while the name of the RHS cannot belong to the set $fv((x\ y)\lambda'_y.t_1) \cup fv((x\ y)u) \cup \{(\!|x\ y|\!)z\}$. Let *x0* be a new name that is not in the set $fv(\lambda'_y.t_1) \cup fv(u) \cup \{z\}$, and *x1* a new name that is not in the set $fv((x\ y)\lambda'_y.t_1) \cup fv((x\ y)u) \cup \{(\!|x\ y|\!)z\}$. After renaming and propagating the metasubstitutions inside the abstractions, the current goal is $n\_abs\ (vswap\ x\ y\ x0)\ (swap\ x\ y\ (\{z := u\}(swap\ y'\ x0\ t1))) =_a n\_abs\ x1\ (\{(vswap\ x\ y\ z) := (swap\ x\ y\ u)\}(swap\ (vswap\ x\ y\ y')\ x1\ (swap\ x\ y\ t1)))$. We proceed by comparing *x1* with $(vswap\ x\ y\ x0)$. If $x1 = (vswap\ x\ y\ x0)$ then we use the induction hypothesis to propagate the swap inside the metasubstitution in the LHS and the current goal is $\{(vswap\ x\ y\ z) := (swap\ x\ y\ u)\}(swap\ x\ y\ (swap\ y'\ x0\ t1)) =_a \{(vswap\ x\ y\ z) := (swap\ x\ y\ u)\}(swap\ (vswap\ x\ y\ y')\ (vswap\ x\ y\ x0)\ (swap\ x\ y\ t1))$ that is proved by the swap equivariance

lemma *swap_equivariance*. If *x1* $\neq$ (*vswap x y x0*) then by the rule *aeq_abs_diff* we have to prove that the variable *vswap x y x0* is not in the set of free variables of the term {(*vswap x y z*) := (*swap x y u*)}(*swap* (*vswap x y y'*) *x1* (*swap x y t1*)) and that *swap x y* ({*z* := *u*}(*swap y' x0 t1*)) =*a* *swap x1* (*vswap x y x0*) ({(*vswap x y z*) := (*swap x y u*)}(*swap* (*vswap x y y'*) *x1* (*swap x y t1*))). The former condition is routine. The later condition is proved using the induction hypothesis twice to propagate the swaps inside the metasubstitutions on each side of the α-equality. This swap has no effect on the variable *z* of the metasubstitution because *x1* is different from *vswap x y z*, and *x0* is different from *z*. Therefore we can apply lemma *aeq_m_subst_eq*, and each generated case is proved by routine manipulation of swaps. The case of the explicit substitution follows a similar strategy of the abstraction. The initial goal is to prove that *swap x y* ({*z* := *u*}(*n_sub t1 y' t2*)) =*a* {(*vswap x y z*) := (*swap x y u*)}(*swap x y* (*n_sub t1 y' t2*)) and we start comparing the variables *y'* and *z*. When *y'* = *z*, the metasubstitution has no effect on the body of the metasubstitution but it can still be propagated to the term *t2*. Therefore, this case is proved using the induction hypothesis over *t2*. When *y'* $\neq$ *z*, then the metasubstitutions are propagated on both sides of the α-equation. Analogously to the abstraction case, one new name for each propagation is created. Let *x0* be a new name not in the set $fv([y' := t2]t1) \cup fv(u) \cup \{z\}$, and *x1*, a new name not in the set $fv([(\!(x\ y)\!)y' := (x\ y)t2](x\ y)t1) \cup fv((x\ y)u) \cup \{(\!(x\ y)\!)z\}$. After the propagation step, we have the goal [(*vswap x y x0*) := (*swap x y* ({*z* := *u*}*t2*))](*swap x y* ({*z* := *u*}(*swap y' x0 t1*))) =*a* [*x1* := ({(*vswap x y z*) := (*swap x y u*)}(*swap x y t2*))]({(*vswap x y z*) := (*swap x y u*)}(*swap* (*vswap x y y'*) *x1* (*swap x y t1*))). We proceed by comparing *x1* and (*swap x y x0*). If *x1* = *vswap x y x0* then after an application of the rule *aeq_sub_same*, we are done by the induction hypothesis for both the body and the argument of the explicit substitution. If *x1* $\neq$ *vswap x y x0* then we apply the rule *aeq_sub_diff* to decompose the explicit substitution in its components. The second component is straightforward by the induction hypothesis. The first component follows the strategy used in the abstraction case. The current goal, obtained after the application of the rule *aeq_sub_diff* is *swap x y* ({*z* := *u*}(*swap y' x0 t1*)) =*a* *swap x1* (*vswap x y x0*) ({(*vswap x y z*) := (*swap x y u*)}(*swap* (*vswap x y y'*) *x1* (*swap x y t1*))). The induction hypothesis is used twice to propagate the swap on both the LHS and RHS of the α-equality. This swap has no effect on the variable *z* of the metasubstitution, therefore we can apply lemma *aeq_m_subst_eq*, and each generated case is proved by routine manipulation of swaps. □

The lemma *swap_subst_rec_fun* is essential to prove the following results:

Lemma *m_subst_abs_neq*: $\forall$ *t u x y z*, *x* $\neq$ *y* $\rightarrow$ *z* ‘*notin*‘ *fv_nom u* ‘*union*‘ *fv_nom* (*n_abs y t*) ‘*union*‘ {{*x*}} $\rightarrow$ {*x* := *u*}(*n_abs y t*) =*a* *n_abs z* ({*x* := *u*}(*swap y z t*)).

Lemma *m_subst_sub_neq* : $\forall$ *t1 t2 u x y z*, *x* $\neq$ *y* $\rightarrow$ *z* ‘*notin*‘ *fv_nom u* ‘*union*‘ *fv_nom* ([*y* := *t2*]*t1*) ‘*union*‘ {{*x*}} $\rightarrow$ {*x* := *u*}([*y* := *t2*]*t1*) =*a* ([*z* := ({*x* := *u*}*t2*)]({*x* := *u*}(*swap y z t1*))).

In fact, the need of the lemma *swap_subst_rec_fun* in the proofs of the two previous lemmas is justified because when the α-equation involves abstractions with different binders, or explicit substitutions with different binders, the rules *aeq_abs_diff* and *aeq_sub_diff* introduce swaps that are outside the metasubstitutions.

4 The substitution lemma

In the pure λ-calculus, the substitution lemma is probably the first non trivial property. In our framework, we have defined two different substitution operators, namely, the metasubstitution denoted by {*x*:=*u*}*t* and the explicit substitution, written as [*x*:=*u*]*t*. In what follows, we present the main steps of our proof of the substitution lemma for *n_sexp* terms, *i.e.* for nominal terms with explicit substitutions.

Lemma *m_subst_lemma*: ∀ *t1 t2 t3 x y*, $x \neq y \rightarrow x$ ‘*notin*‘ (*fv_nom t3*) →
$(\{y := t3\}(\{x := t2\}t1)) =_a (\{x := (\{y := t3\}t2)\}(\{y := t3\}t1))$.

Proof. The proof is by induction on the size of *t1*. The interesting cases are the abstraction and the explicit substitution. We focus on the former, whose initial goal is

$(\{y := t3\}\ (\{x := t2\}\ n\_abs\ z\ t11)) =_a (\{x := \{y := t3\}\ t2\}\ (\{y := t3\}\ n\_abs\ z\ t11))$

assuming that $x \neq y$ and x ‘*notin*‘ *fv_nom t3*. The induction hypothesis generated by this case states that the lemma holds for any term of the size of *t11*, *i.e.* any term with the same size of the body of the abstraction. We start comparing z with x aiming to apply the definition of the metasubstitution on the LHS of the goal. When $z = x$, the subterm $\{x := t2\}(n\_abs\ x\ t11)$ reduces to $(n\_abs\ x\ t11)$ by lemma *m_subst_abs_eq*, and then the LHS reduces to $\{y := t3\}(n\_abs\ x\ t11)$. The RHS $\{x := \{y := t3\}\ t2\}\ (\{y := t3\}\ n\_abs\ x\ t11)$ also reduces to it because x does not occur free neither in $(n\_abs\ x\ t11)$ nor in *t3*, and we are done. When $z \neq x$, then we compare y with z. When $y = z$ then the subterm $\{y := t3\}(n\_abs\ z\ t11)$ reduces to $(n\_abs\ z\ t11)$, by applying the lemma *m_subst_abs_neq*. On the LHS $\{z := t3\}\ (\{x := t2\}\ n\_abs\ z\ t11)$, we propagate the internal metasubstitution over the abstraction taking a fresh name w as a new binder. The variable w is taken such that it is not in the set $fv(\lambda_z.t_{11}) \cup fv(t3) \cup fv(t2) \cup \{x\}$. The resulting terms are α-equivalent, and although the strategy is similar to the one used in the lemmas *aeq_m_subst_in*, *aeq_m_subst_out* and *swap_subst_rec_fun* the proof requires much more steps. We proceed by transitivity of the α-equivalency using $(\{z := t3\}\ n\_abs\ w\ (\{x := t2\}\ swap\ z\ w\ t11))$ as an intermediate term. In the first subcase, we need to prove that $(\{x := t2\}\ n\_abs\ z\ t11) =_a n\_abs\ w\ (\{x := t2\}\ swap\ z\ w\ t11)$ that is proved by lemma *m_subst_abs_neq*. In the other subcase, we need to prove that $(\{z := t3\}\ n\_abs\ w\ (\{x := t2\}\ swap\ z\ w\ t11)) =_a (\{x := \{z := t3\}\ t2\}\ n\_abs\ z\ t11)$, and we start comparing z and w. Note that the fresh variable w can be equal to some bound variable, that's why it needs to be compared with z. When $z = w$, we need to prove that $(\{w := t3\}\ n\_abs\ w\ (\{x := t2\}\ t11)) =_a (\{x := \{w := t3\}\ t2\}\ n\_abs\ w\ t11)$, which is α-equivalent to $(\{w := t3\}\ n\_abs\ w\ (\{x := t2\}\ t11)) =_a (\{x := t2\}\ n\_abs\ w\ t11)$, since w does not occur in the free variables of *t2*. We conclude with lemma *m_subst_abs_neq*. When $z \neq w$, then we need to prove that $(\{z := t3\}\ n\_abs\ w\ (\{x := t2\}\ swap\ z\ w\ t11)) =_a (\{x := \{z := t3\}\ t2\}\ n\_abs\ z\ t11)$. Since we also have that $x \neq z$ then we can propagate the metasubstitution over the abstraction on both LHS and RHS of this α-equation using the same fresh name w. This provides a great simplification on the size of the proof because there is no need to analyse the case when fresh names are different. In the case in which $y \neq z$, the goal is $(\{y := t3\}\ (\{x := t2\}\ n\_abs\ z\ t11)) =_a (\{x := \{y := t3\}\ t2\}\ (\{y := t3\}\ n\_abs\ z\ t11))$. Similarly to the previous case, we pick a fresh name w that is not in the set $fv(\lambda_z.t_{11}) \cup fv(t_3) \cup fv(t_2) \cup \{x\} \cup \{y\}$, and since we also have that $x \neq z$, then we can propagate all metasubstitutions inside the abstractions (LHS and RHS) and we conclude by the induction hypothesis. □ In the explicit substitution case, the initial goal is $(\{y := t3\}\ (\{x := t2\}\ ([z := t12]\ t11))) =_a (\{x := \{y := t3\}\ t2\}\ (\{y := t3\}\ ([z := t12]\ t11)))$, and we start comparing x and z. When $z = x$, the LHS $(\{y := t3\}\ (\{x := t2\}\ ([z := t12]\ t11)))$ reduces to $([x := \{x := t2\}\ t12]\ t11)$, but differently to the abstraction case, the external metasubstitution of the RHS cannot be ignored because x may occur free in *t12*, and it will therefore be propagated over the explicit substitution. We then need a fresh name, say w, that is not in the set $fv(t_3) \cup fv(t_2) \cup fv([x := t_{12}]t_{11}) \cup \{y\}$. We use lemma *m_subst_sub_neq* to perform the propagation. We proceed by comparing x and w because if they are equal the external metasubstitution of the RHS can be removed as in the abstraction case. The current goal is $(\{y := t3\}\ ([w := \{w := t2\}\ t12]\ t11)) =_a ([w := \{w := \{y := t3\}\ t2\}\ (\{y := t3\}\ t12)]\ (\{y := t3\}\ t11))$, and the next step is to propagate the external metasubstitution of the LHS without the need of a new name. As the same name w is used on both sides, we can proceed with *aeq_sub_same*. The first subcase is trivial. And the second is proved by the induction hypothesis for *t12*. When $x \neq w$, then we can propagate the external metasubstitutions on both sides of the current goal $(\{y := t3\}\ ([x := \{x := t2\}\ t12]\ t11)) =_a (\{x := \{y := t3\}\ t2\}\ ([w := \{y := t3\}$

t12] ({*y* := *t3*} *swap x w t11*))). We use two different instances of *m_subst_sub_neq*, and on both cases we use the fresh name *w* that was already created. Again, since we have used the same fresh name *w* on both sides of the α-equation, we proceed with *aeq_sub_same*. In the first subcase, we need to prove that ({*y* := *t3*} *swap x w t11*) =*a* ({*x* := {*y* := *t3*} *t2*} ({*y* := *t3*} *swap x w t11*)), and we conclude with *m_subst_notin*, since *x* does not occur free in ({*y* := *t3*} *swap x w t11*). The second subcase is proved by the induction hypothesis on *t12*. When $z \neq x$, then we take a fresh name *w* such that it is not in the set $fv(t_3) \cup fv(t_2) \cup fv([z := t_{12}]t_{11}) \cup \{x\} \cup \{y\}$. The current goal is ({*y* := *t3*} ({*x* := *t2*} ([*z* := *t12*] *t11*))) =*a* ({*x* := {*y* := *t3*} *t2*} ({*y* := *t3*} ([*z* := *t12*] *t11*))) and we start propagating the internal metasubstitution. Let's start with the LHS. After the propagation, we get the following goal ({*y* := *t3*} ([*w* := {*x* := *t2*} *t12*] ({*x* := *t2*} *swap z w t11*))) =*a* ({*x* := {*y* := *t3*} *t2*} ({*y* := *t3*} ([*z* := *t12*] *t11*))). We now compare *y* and *z*, and propagate the internal metasubstitution of the RHS. When $y = z$, we have the goal ({*z* := *t3*} ([*w* := {*x* := *t2*} *t12*] ({*x* := *t2*} *swap z w t11*))) =*a* ({*x* := {*z* := *t3*} *t2*} ([*z* := {*z* := *t3*} *t12*] *t11*)). The next step is to propagate the external metasubstitutions on both sides of the current goal. To do so, we will use the same fresh name *w* on both propagations. When $y \neq z$, we again propagate all the metasubstitutions, one in the LHS and two in the RHS, using the same fresh name *w* for all of them.

5 Conclusion and Future work

In this work, we presented a formalization of the substitution lemma in a framework that extends the λ-calculus with an explicit substitution operator. Calculi with explicit substitutions are important frameworks to study properties of the λ-calculus and have been extensively studied in the last decades [**?**, **?**, **?**, **?**, **?**, **?**].

The formalization is modular in the sense that the explicit substitution operator is generic and could be instantiated with any calculi with explicit substitutions in a nominal setting. The main contribution of this work, besides the formalization itself, is the solution to a circular proof problem. Several auxiliary (minor) results were not included in this document, but they are numerous and can be found in the source file of the formalization that is available in a GitHub repository (`https://github.com/flaviodemoura/lx_confl/tree/m_subst_lemma`).

As future work, we plan to integrate this formalization with another one related to the Z property [**?**] to prove confluence of calculi with explicit substitutions [**?**, **?**], as well as other properties in the nominal framework [**?**].

References