A formalized extension of the substitution lemma in Coq

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

Maria Julia
Departamento de Ciência da Computação
Universidade de Brasília, Brasília, Brazil
majuhdl@gmail.com

TBD

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 done 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 $txuyv =_\alpha tyvxuyv$.

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 straightforward 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 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[**?**, **?**, **?**];
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.

This document is built directly from a Coq script using the CoqDoc[2] tool. 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 main theorem, *i.e.* the substitution lemma, and we conclude in Section 5.

[1] https://github.com/plclub/metalib
[2] https://coq.inria.fr/refman/using/tools/coqdoc.html

Submitted to:
FROM 2023

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:

$$(\lambda_x.t)\ u \to_\beta txu \tag{2}$$

where txu 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 is done in order to avoid variable capture. In other words, txu 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 is done to avoid such capture, so in this example, one can take an α-equivalent[3] term, say $(\lambda_z.x\ z)$, and perform the β-step correctly as $(\lambda_x\lambda_y.x\ y)\ y \to_\beta \lambda_z.y\ z$. The renaming of variables in the nominal setting is done via a name-swapping, which is formally defined as follows:

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

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

$$xyt := \begin{cases} xyz, & \text{if } t = z; \\ \lambda_{xyz}.xyt_1, & \text{if } t = \lambda_z.t_1; \\ xyt_1\ xyt_2, & \text{if } t = t_1\ t_2 \end{cases} \tag{3}$$

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: $(\lambda_x\lambda_y.x\ y)\ y \to_\beta (yz(\lambda_y.x\ y))xy = (\lambda_z.x\ z)xy = \lambda_z.y\ z$. Could we have used a variable substitution instead of a swapping in the previous example? Absolutely. We could have done the reduction as $(\lambda_x\lambda_y.x\ y)\ y \to_\beta ((\lambda_y.x\ y)yz)xy = (\lambda_z.x\ z)xy = \lambda_z.y\ z$, 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:

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

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

A short example to acquaint ourselves with the Coq notation, let us show how we will write the proofs:

```
Lemma swap_var_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:

$$t ::= x \mid \lambda_x.t \mid t\ t \mid txu \tag{4}$$

where $[x := u]t$ represents a term with an operator that will be evaluated with specific rules of a 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 in 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 more granular 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 (**??**), 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 (**??**). Therefore, *n_sexp* is used to denote the set of nominal 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).
```

where *n_var* is the constructor for variables, *n_abs* for abstractions, *n_app* for applications and *n_sub* for the explicit substitution. 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 xyt, is inductively defined as in (**??**) with the additional case for the explicit substitution operator:

$$xyt := \begin{cases} xyv, & \text{if } t \text{ is the variable } v; \\ \lambda_{xyz}.xyt_1, & \text{if } t = \lambda_z.t_1; \\ xyt_1\ xyt_2, & \text{if } t = t_1\ t_2; \\ xyt_1xyzxyt_2, & \text{if } t = t_1zt_2. \end{cases}$$

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[4]. The next lemma shows that the *swap* function preserves the size of terms. It is proved by induction on the structure of the term t:

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

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

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

Equivariance is an 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).
```

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

2.2 α-equivalence

The notion of α-equivalence is defined in the usual way by the following rules:

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

$$\frac{x \neq y \qquad x \notin fv(t_2) \qquad t_1 =_\alpha yxt_2}{\lambda_x.t_1 =_\alpha \lambda_y.t_2}\ (aeq\_abs\_diff)$$

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

$$\frac{t_2 =_\alpha t_2' \qquad x \neq y \qquad x \notin fv(t_1') \qquad t_1 =_\alpha yxt_1'}{t_1 x t_2 =_\alpha t_1' y t_2'}\ (aeq\_sub\_diff)$$

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 (n_sub t1 x t2) (n_sub t1' x t2')
| 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 (n_sub t1 x t2) (n_sub t1' y t2').
```

In what follows, we use a infix notation for α-equivalence in the Coq code: 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.

Informally, two terms are α-equivalent if they differ only by the names of the bound variables. Therefore, 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 $xyt_1 =_\alpha xyt_2$. Note that this is not true for renaming substitutions: in fact, $\lambda_x.z =_\alpha \lambda_y.z$, but $(\lambda_x.z)zx = \lambda_x.x \neq_\alpha \lambda_y.x(\lambda_y.z)zx$, 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).
```

There are several 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. 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 result in a proofless branch. so that one can establish the α-equivalence between two abstractions or two explicit substitutions with different binders. The following proposition states when two swaps with a common name collapse, and it is used in the transitivity proof of *aeq*:

Lemma *aeq_swap_swap*: $\forall\ t\ x\ y\ z,\ z$ ‘*notin*‘ *fv_nom t* $\to$ *x* ‘*notin*‘ *fv_nom t* $\to$ (*swap z x* (*swap x y t*)) =*a* (*swap z y t*).

3 The metasubstitution operation of the λ-calculus

The main operation of the λ-calculus is the β-reduction that express how to evaluate a function applied to a given argument: $(\lambda_x.t)\ u \to_\beta txu$. In a less formal context, the concept of β-reduction means that the result of evaluating the function $(\lambda_x.t)$ with argument u is obtained by 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 the substitution. 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: $txu =$

$$\begin{cases} u, & \text{if } t = x; \\ y, & \text{if } t = y \text{ and } x \neq y; \\ t_1xu\ t_2xu, & \text{if } t = (t_1\ t_2)xu; \\ \lambda_y.(t_1xu), & \text{if } t = \lambda_y.t_1. \end{cases}$$

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

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 of the λ-calculus, *i.e.* the metasubstitution, a renaming is performed whenever it is propagated inside a binder. In our case, there are two binders: the abstraction and the explicit substitution.

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

$$txu = \begin{cases} u, & \text{if } t = x; \\ y, & \text{if } t = y \text{ and } x \neq y; \\ t_1xu\ t_2xu, & \text{if } t = (t_1\ t_2)xu; \\ \lambda_x.t_1, & \text{if } t = \lambda_x.t_1; \\ \lambda_z.((yzt_1)xu), & \text{if } t = \lambda_y.t_1, x \neq y \text{ and } z \notin fv(\lambda_y.t_1) \cup fv(u) \cup \{x\}; \\ t_1xt_2xu, & \text{if } t = t_1xt_2; \\ (yzt_1)xuzt_2xu, & \text{if } t = t_1yt_2, x \neq y \text{ and } z \notin fv(t_1yt_2) \cup fv(u) \cup \{x\}. \end{cases} \tag{5}$$

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[5], but it uses the size of the term in which the substitution will be performed as an extra

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

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 txu. 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.
```

The standard proof strategy for the non trivial properties is induction on the structure of the 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, refer 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).
```

The following lemma states that if $x \notin fv(t)$ then $txu =_\alpha t$. In informal proofs the conclusion of this lemma is usually stated as a syntactic equality, i.e. $txu = t$ instead of the α-equivalence, but due to the changes of the names of the bound variables when the metasubstitution is propagated inside an abstraction or inside an explicit substitution, syntactic equality does not hold here.

```
Lemma m_subst_notin: ∀ t u x, x `notin` fv_nom t → [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. One interesting case is when $t = \lambda_y.t_1$ and $x \neq y$. In this case, we have to prove that $(\lambda_y.t_1)xu =_\alpha \lambda_y.t_1$. 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:

$\forall t'\ x\ y, |t'| = |t_1| \rightarrow \forall u\ x', x' \notin fv(xyt') \rightarrow (xyt')x'u =_\alpha xyt'$.

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 $\lambda_z.(zyt_1)xu =_\alpha \lambda_y.t_1$. Since $z \notin fv(\lambda_y.t_1) = fv(t_1)\backslash\{y\}$, there are two cases:

1. $z = y$: In this case, we have to prove that $\lambda_z.(zzt_1)xu =_\alpha \lambda_z.t_1$. By the rule *aeq_abs_same* we get $(zzt_1)xu =_\alpha t_1$, 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 zzt_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 $(zyt_1)xu =_\alpha zyt_1$ which holds by the induction hypothesis, since $|zyt_1| = |t_1|$ and $x \notin fv(zyt_1)$ 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. □

We will now prove some stability results for the metasubstitution w.r.t. α-equivalence. More precisely, we will prove that if $t =_\alpha t'$ and $u =_\alpha u'$ then $txu =_\alpha t'xu'$, where x is any variable and t, t', u and u' are any *n_sexp* terms. This proof is split in two steps: firstly, we prove that if $u =_\alpha u'$ then $txu =_\alpha txu', \forall x, t, u, u'$; secondly, we prove that if $t =_\alpha t'$ then $txu =_\alpha t'xu, \forall x, t, t', u$. These two steps are then combined through the transitivity of the α-equivalence relation. Nevertheless, this task were not straighforward. Let's follow the steps of our first trial.

Lemma *aeq_m_subst_in_trial*: ∀ *t u u' x*, *u =a u'* → ([*x* := *u*] *t*) =*a* ([*x* := *u'*] *t*).

Proof. The proof is done by induction on the size of the term *t*. The interesting case is when *t* is an abstraction, *i.e.* $t = \lambda_y.t_1$. We need to prove that $(\lambda_y.t_1)xu =_\alpha (\lambda_y.t_1)xu'$. If $x = y$ then the result is trivial. Suppose $x \neq y$. The metasubstitution will be propagated inside the abstraction on each side of the α-equation, after generating a new name for each side. The new goal is then $\lambda_{x_0}.(yx_0t_1)xu =_\alpha \lambda_{x_1}.(yx_1t_1)xu'$, where $x_0 \notin fv(\lambda_y.t_1) \cup fv(u) \cup \{x\}$ and $x_1 \notin fv(\lambda_y.t_1) \cup fv(u') \cup \{x\}$. The variables x_0 and x_1 are either the same or different. In the former case the result is trivial because $u =_\alpha u'$. In the latter case, $x_0 \neq x_1$ and we need to prove that $(yx_0t_1)xu =_\alpha x_0x_1((yx_1t_1)xu')$. Therefore, we need to propagate the swap over the metasubstitution before been able to apply the induction hypothesis. The propagation of the swap over the metasubstitution is stated by the following lemma:

Let t, u be terms, and x, y, z variables. Then $yz(txu) =_\alpha (yzt)yzxyzu$.

whose corresponding Coq version is given by:

Lemma *swap_m_subst*: ∀ *t u x y z*, *swap y z* ([*x* := *u*]*t*) =*a* ([(*vswap y z x*) := (*swap y z u*)] (*swap y z t*)).

Proof. The proof is by induction on the size of the term *t*. The interesting case is the abstraction, where we need to prove that $yz((\lambda_w.t_1)xu) =_\alpha (yz\lambda_w.t_1)yzxyzu$. On the left hand side, we can propagate the metasubstitution over the abstraction in the case that $x \neq w$ (the other is straighforward) and the new goal after the propagation of the swap over the abstraction is $\lambda_{yzw'}.yz(ww't_1xu) =_\alpha (\lambda_{yzw}.yzt_1)yzxyzu$, where $w' \notin fv(\lambda_w.t_1) \cup fv(u) \cup \{x\}$. Now we propagate the metasubstitution over the abstraction in the right hand side term. Since $x \neq w$, we get $yzx \neq yzw$ and a renaming is necessary. After the renaming to a new name, say w'', such that $w'' \notin fv(\lambda_{yzw}.yzt_1) \cup fv(yzu) \cup \{yzx\}$, we get the following goal $\lambda_{yzw'}.yz(ww't_1xu) =_\alpha \lambda_{w''}.(w''yzw(yzt_1))yzxyzu$. We consider two cases: either $w'' = yzw'$ or $w'' \neq yzw'$. In the former case, we can apply the rule *aeq_abs_same* and we get $yz((ww't_1)xu) =_\alpha (w''yzw(yzt_1))yzxyzu$ that can be proved by the induction hypothesis. When $w'' \neq yzw'$, the application of the rule *aeq_abs_diff* generates the goal $w''yzw'yz(ww't_1xu) =_\alpha (w''yzw(yzt_1))yzxyzu$. We can use the induction hypothesis to propagate the swap inside the metasubstitution, and then we get an α-equality with metasubstitution as main operation on both sides, and whose correspondent components are α-equivalent. In a more abstract way, we have to prove an α-equality of the form $txu =_\alpha t'xu'$, where $t =_\alpha t'$ and $u =_\alpha u'$. The problem is that we cannot rewrite α-equalities inside metasubstitution unless we prove some special lemmas stating the compatibilities between them using the `Equations` library or something similar. Alternatively, if we decide to analise the metasubtitution componentwise, *i.e.* as stated in a lemma similar to *aeq_m_subst_in_trial*, we get a circular proof problem because both *aeq_m_subst_in_trial* and *swap_m_subst* depend on each other to be proved. We will present a solution that do not use any additional library, but it adds the following axiom to the formalization:

Axiom *Eq_implies_equality*: ∀ *s s'*: *atoms*, *s* [=] *s'* → *s* = *s'*.

This axiom transform a set equality into a syntactic equality. This will allow us to rewrite sets of atoms in a more flexible way. To show how it works, we will start proving the lemma *aeq_m_subst_in* without the need of the lemma *swap_m_subst*:

Lemma *aeq_m_subst_in*: ∀ *t u u' x*, *u* =*a u'* → ([*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. We have by hypothesis that $u =_\alpha u'$ therefore both u and u' have the same set of free variables by lemma *aeq_fv_nom*. With the axiom *Eq_implies_equality*, we can replace the set $fv(u)$ by $fv(u')$, or vice-versa, in such a way that instead of generating two new names for the propagation of the metasusbstitutions inside the abstractions, we need just one new name and there is no more the case where the binders of the abstractions were different names. The case of the explicit substitution is similar, and with this strategy we avoid the rules *aeq_abs_diff* and *aeq_sub_diff* that introduce swappings. □

The next lemma, named *aeq_m_subst_out* will benefit the strategy used in the previous proof, but it is not straightfoward. In the proof below, we will mostly use Coq notation, instead of the metanotation of the previous proof. We believe that at this point of the work, even the readers not familiar with Coq, can easily understand the Coq code interleaved with metanotation. Lemma *aeq_m_subst_out*: ∀ *t t' u x*, *t* =*a t'* → ([*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* ≠ *y0* 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* ≠ *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* ≠ *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* ≠ *x*, *y0* ≠ *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 applycation 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*: ∀ *t t' u u' x*, *t* =*a t'* → *u* =*a u'* → ([*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*: ∀ *x y z t u*, *swap x y* (*subst_rec_fun t u z*) =*a subst_rec_fun* (*swap x y t*) (*swap x y u*) (*vswap x y z*).

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 as *t1*. Firstly, we compare the variables *y'* and *z* to check whether we should propagate the metasubstitution inside the abstraction of the LHS. When *y'* = *z* the metasubstitution is erased according to the definition (3) 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 $fv(xy\lambda'_y.t_1) \cup fv(xyu) \cup \{xyz\}$. 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 $fv(xy\lambda'_y.t_1) \cup fv(xyu) \cup \{xyz\}$. 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* ≠ (*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 of the metasubstitution, 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 necessary. Let *x0* be a new name not in the set $fv(t1y't2) \cup fv(u) \cup \{z\}$, and *x1*, a new name not in the set $fv(xyt1xyy'xyt2) \cup fv(xyu) \cup \{xyz\}$. After the propagation step, we have the goal *n_sub* (*swap x y* ([*z* := *u*](*swap y' x0 t1*))) (*vswap x y x0*) (*swap x y* ([*z* := *u*]*t2*)) =*a n_sub* ([(*vswap x y z*) := (*swap x y u*)](*swap* (*vswap x y y'*) *x1* (*swap x y t1*))) *x1* ([(*vswap x y*

z) := (*swap x y u*)](*swap x y t2*)). 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*, each component of the explicit substitution is proved by the induction hypothesis. If *x1* ≠ *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 of the metasubstitution, therefore we can apply lemma *aeq_m_subst_eq* and each generated case is proved by routine manipulation of swaps. □

The following lemmas state, respectively, what hapens when the variable in the meta-substitution is equal or different from the one in the abstraction. When it is equal, the meta-substitution is irrelevant. When they are different, we take a new variable that does not occur freely in the substituted term in the meta-substitution nor in the abstraction and is not the variable in the meta-substitution, and the abstraction of this new variable using the meta-substitution of the swap of the former variable in the meta-substitution is alpha-equivalent to the original meta-substitution of the abstraction. The proofs were straightforward from the definition of the meta-substitution, each case being respectively each one in the definition.

Lemma *m_subst_abs_eq* : ∀ *u x t*, [*x* := *u*](*n_abs x t*) = *n_abs x t*.

Lemma *m_subst_abs_neq* : ∀ *t u x y z*, *x* ≠ *y* → *z* ‘*notin*‘ *fv_nom u* ‘*union*‘ *fv_nom* (*n_abs y t*) ‘*union*‘ {{*x*}} → [*x* := *u*](*n_abs y t*) =*a n_abs z* ([*x* := *u*](*swap y z t*)).

The following lemmas are the equivalent version of the two previous lemmas for the case of the explicit substitution. They state, respectively, what hapens when the variable in the meta-substitution is equal or different from the one in the explicit substitution. When it is equal, the meta-substitution is irrelevant on *t1*, but it is applied to *e2*. When they are different, we take a new variable that does not occur freely in the substituted term in the meta-substitution nor in the substitution and is not the variable in the meta-substitution, and the explicit substitution of this new variable using the meta-substitution of the swap of the former variable in the meta-substitution in *e11* and the application of the original meta_substitution in *e12* is alpha-equivalent to the original meta-substitution of the explicit substitution. The proofs are straighforward from the definition of the meta-substitution, each case being respectively each one in the definition.

Lemma *m_subst_sub_eq* : ∀ *u x t1 t2*, [*x* := *u*](*n_sub t1 x t2*) = *n_sub t1 x* ([*x* := *u*] *t2*).

Lemma *m_subst_sub_neq* : ∀ *t1 t2 u x y z*, *x* ≠ *y* → *z* ‘*notin*‘ *fv_nom u* ‘*union*‘ *fv_nom* (*n_sub t1 y t2*) ‘*union*‘ {{*x*}} → [*x* := *u*](*n_sub t1 y t2*) =*a n_sub* ([*x* := *u*](*swap y z t1*)) *z* ([*x* := *u*]*t2*).

4 The substitution lemma for the metasubstitution

In the pure λ-calculus, the substitution lemma is probably the first non trivial property. In our framework, we have defined two different substitution operation, namely, the metasubstitution denoted by [*x*:=*u*]*t* and the explicit substitution that has *n_sub* as a constructor. In what follows, we present the main steps of our proof of the substitution lemma for the metasubstitution operation:

Lemma *m_subst_lemma*: ∀ *e1 e2 x e3 y*, $x \neq y \rightarrow x$ ‘*notin*‘ (*fv_nom e3*) → ([*y* := *e3*]([*x* := *e2*]*e1*)) =*a* ([*x* := ([*y* := *e3*]*e2*)]([*y* := *e3*]*e1*)).

Proof. The proof is by induction on the size of the term *e1*. The interesting cases are the abstraction and the explicit substitution. In the abstraction case, the initial goal is ([*y* := *e3*] ([*x* := *e2*] *n_abs z e1*)) =*a* ([*x* := [*y* := *e3*] *e2*] ([*y* := *e3*] *n_abs z e1*)), assuming that $x \neq y$ and *x* ‘*notin*‘ *fv_nom e3*. The induction hypothesis generated by this case states that the lemma can be propagated to a swap inside a metasubstitution that is applied to a term thath has the same size as *e1* and the same hypothesis for the main lemma also apply to this case. 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* := *e2*](*n_abs x e1*) is reduced to (*n_abs x e1*) by applying the lemma in m_subst_abs_eq and then the LHS reduces to ([*y* := *e3*] *n_abs x e1*). The RHS reduces to ([*y* := *e3*] *n_abs x e1*) because *x* does not occur free neither in (*n_abs x e1*) nor in *e3*, which has also been proved. When *y* = *z* then the subterm ([*y* := *e3*] *n_abs z e1*) reduces to (*n_abs z e1*), by applying the lemma m_subst_abs_neq which also generates certain hypothesis that help prove this fact. On the LHS, we propagate the internal metasubstitution over the abstraction taking a fresh name for the binder that is necessary because of the hypothesys generated using the lemma m_subst_abs_neq as explained previously. Let *w* be a new name that is not in the set $fv(\lambda'_z.e_1) \cup fv(e3) \cup fv(e2) \cup \{x\}$. 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. First, we apply the transitivity of the α-equivalency using (*z* := *e3* n_abs w (*x* := *e2* swap z w e1)) as intermediate term. The first case is the α-equivalency of LHS with this new term, wich is solved by applying aeq_m_subst_out trivially. The other case is the α-equivalency of RHS with this new term. In this case it was necessary to compare other two pairs of variables. Initially, we compared *z* and *w*. When *z* = *w* then in the LHS the metasubstitution is removed using the lemma *m_subst_abs_eq* and the subterm ([*w* := *e2*] *swap w w e1*) is reduced to ([*w* := *e2*]) *e1*, remaining only *n_abs w* ([*x* := *e2*] *e1*). On the RHS, we propagate the internal metasubstitution over the abstraction taking the *w* as the new variable for this substitution. Then, the lemma *aeq_abs_same* is used to remove the abstractions on both sides of the α-equivalency and because they have the same variable, all we have to do is assure that the expressions are α-equivalent. By applying the lemma *aeq_m_subst_in*, we can see that they are. In the last part of this case, we propagate the internal metasubstitution over the abstraction taking a fresh name for the binder that is necessary because of the hypothesys generated using the lemma m_subst_abs_neq as explained previously. Let *w'* be a new name that is not in the set $fv(\lambda'_z.e_1) \cup fv(e3) \cup fv(e2) \cup \{x\}$. We then proceed by comparing *w* and *w'*. When *w* == *w'*, the proof if given by removing the abstractions from inside the metasubstitutions using the lemma *m_subst_abs_neq* and the new variable *w'*. Then we remove them altogether using the lemma *aeq_abs_same*, given that the variables in the abstractions are the same. The subterm *swap w' w'* ([*x* := *e2*] *swap z w' e1*) gets simplified to ([*x* := *e2*] *swap z w' e1*), in the LHS. What remain is simmilar to what we have in our induction hypothesys, which is applied to our conclusion and the rest of the proof is solved trivially. The case in which $w \neq w'$ is solved similarly. In the explicit substitution case, we used the same approach used in the abstraction for the left side and the same as the application for the right side of the substitution. It consisted of comparing the variable in the meta substitution and the one in the substitution. We used the auxiliary lemmas on the equality and inequality of the meta-substitution applied to explicit substitutions and it was necessary to create new variables in each use of the inequality. This is due to the atempt of removing the explicit substitution from inside the meta-substitution. When tis removal was made, the proof consisted in proving a similar case for the abstraction in the left side of the explicit substitution and the one similar to the application was used for the right part of it. □

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 from the following url 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 properties of calculi with explicit substitutions such as confluence[**?**, **?**, **?**].