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 operation, 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 remarkably direct implementation of the α-equivalence concept. Despite this simplicity, the strategy involved in variable renaming within the proofs presents a substantial challenge, ensuring a comprehensive 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[5] in a general framework that extends the λ-calculus with an explicit substitution operator. The formalization is done in the Coq proof assistant[25] 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 [7]. 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[18, 22, 24];

[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.

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 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[15] 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 is done in order to avoid variable capture. 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 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:

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

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

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

[2] https://coq.inria.fr/refman/using/tools/coqdoc.html

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

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: $(\lambda_x\lambda_y.x\,y)\;y \to_\beta \{x := y\}((y\;z)(\lambda_y.x\;y)) = \{x := y\}(\lambda_z.x\;z) = \lambda_z.y\;z$. Could we have used a variable substitution instead of a swapping in the previous example? Absolutely. We could have done the reduction as $(\lambda_x\lambda_y.x\;y)\;y \to_\beta \{x := y\}(\{y := z\}(\lambda_y.x\;y)) = \{x := y\}(\lambda_z.x\;z) = \lambda_z.y\;z$, 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*. 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 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[20, 16].

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[1, 8, 6, 11, 21, 17, 9, 12, 19].

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 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[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.
```

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

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

Lemma *shuffle_swap* : $\forall$ *w y z t*, *w* $\neq$ *z* $\rightarrow$ *y* $\neq$ *z* $\rightarrow$ (*swap w y* (*swap y z t*)) = (*swap w z* (*swap w y t*)).

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* : $\forall$ *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* : $\forall$ *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 the term *t*:

Lemma *fv_nom_swap* : $\forall$ *z y t*, *z* ‘*notin*‘ *fv_nom t* $\rightarrow$ *y* ‘*notin*‘ *fv_nom* (*swap y z t*).

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*: $\forall$ *P* : *n_sexp* $\rightarrow$ Prop, ($\forall$ *x*, *P* (*n_var x*)) $\rightarrow$
($\forall$ *t1 z*, ($\forall$ *t2 x y*, *size t2* = *size t1* $\rightarrow$ *P* (*swap x y t2*)) $\rightarrow$ *P* (*n_abs z t1*)) $\rightarrow$
($\forall$ *t1 t2*, *P t1* $\rightarrow$ *P t2* $\rightarrow$ *P* (*n_app t1 t2*)) $\rightarrow$
($\forall$ *t1 t3 z*, *P t3* $\rightarrow$ ($\forall$ *t2 x y*, *size t2* = *size t1* $\rightarrow$ *P* (*swap x y t2*)) $\rightarrow$ *P* (*n_sub t1 z t3*)) $\rightarrow$ ($\forall$ *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* : $\forall$ *t x’ x y*, *x’* ‘*notin*‘ *fv_nom t* $\rightarrow$ *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*. 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 the Coq language. If *x’* = *z* then we have to prove that a certain element is not in a set where it was removed, and we are done by lemma *notin_remove_3*[5]. 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* then 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 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. □

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

The other direction is also true:

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 the name of bound variables. For instance, all the 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[6]. 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 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 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 express 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[5] defines it as follows:

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

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

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, *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: 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 $\{x := u\}t$ 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[7], 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

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

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.

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. 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:
$\forall t'\ x\ y, |t'| = |t_1| \to \forall u\ x', x' \notin fv((x\ y)t') \to \{x' := u\}((x\ y)t') =_\alpha (x\ y)t'$. 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 hapens 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, 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_sub_eq*: ∀ *u x t1 t2*, {*x* := *u*}(*n_sub t1 x t2*) = *n_sub t1 x* ({*x* := *u*}*t2*).

Lemma *fv_nom_remove*: ∀ *t u x y*, *y* '*notin*' *fv_nom u* → *y* '*notin*' *remove x* (*fv_nom t*) →
y '*notin*' *fv_nom* ({*x* := *u*}*t*).

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 $\{x := u\}t =_\alpha \{x := u'\}t'$, 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 $\{x := u\}t =_\alpha \{x := u'\}t, \forall x, t, u, u'$; secondly, we prove that if $t =_\alpha t'$ then $\{x := u\}t =_\alpha \{x := u\}t', \forall x, t, t', u$. 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 $\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u'\}(\lambda_y.t_1)$. 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 $\lambda_{x_0}.\{x := u\}((y\ x_0)t_1) =_\alpha \lambda_{x_1}.\{x := u'\}((y\ x_1)t_1)$, 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 $\{x := u\}((y\ x_0)t_1) =_\alpha (x_0\ x_1)(\{x := u'\}((y\ x_1)t_1))$. 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:

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 $(y\ z)(\{x := u\}(\lambda_w.t_1)) =_\alpha \{(\!(y\ z)\!)x := (y\ z)u\}((y\ z)\lambda_w.t_1)$. 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 $\lambda_{(\!(y\ z)\!)w'}.(y\ z)(\{x := u\}(w\ w')t_1) =_\alpha \{(\!(y\ z)\!)x := (y\ z)u\}(\lambda_{(\!(y\ z)\!)w}.(y\ z)t_1)$, 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 $(\!(y\ z)\!)x \neq (\!(y\ z)\!)w$ and a renaming is necessary. After the renaming to a new name, say w'', such that $w'' \notin fv(\lambda_{(\!(y\ z)\!)w}.(y\ z)t_1) \cup fv((y\ z)u) \cup \{(\!(y\ z)\!)x\}$, we get the following goal $\lambda_{(\!(y\ z)\!)w'}.(y\ z)(\{x := u\}(w\ w')t_1) =_\alpha \lambda_{w''}.\{(\!(y\ z)\!)x := (y\ z)u\}((w''\ (\!(y\ z)\!)w)((y\ z)t_1))$. We consider two cases: either $w'' = (\!(y\ z)\!)w'$ or $w'' \neq (\!(y\ z)\!)w'$. In the former case, we can apply the rule *aeq_abs_same* and we get $(y\ z)(\{x := u\}((w\ w')t_1)) =_\alpha \{(\!(y\ z)\!)x := (y\ z)u\}((w''\ (\!(y\ z)\!)w)((y\ z)t_1))$ that can be proved by the induction hypothesis. When $w'' \neq (\!(y\ z)\!)w'$, the application of the rule *aeq_abs_diff* generates the goal $(w''\ (\!(y\ z)\!)w')(y\ z)(\{x := u\}(w\ w')t_1) =_\alpha \{(\!(y\ z)\!)x := (y\ z)u\}((w''\ (\!(y\ z)\!)w)((y\ z)t_1))$. 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 $\{x := u\}t =_\alpha \{x := u'\}t'$, 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.

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* ({*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* ≠ *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. □

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 operation, 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*: $\forall$ *e1 e2 x e3 y*, $x \neq y \rightarrow x$ '*notin*' (*fv_nom e3*) $\rightarrow$
({*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 case in which $y \neq z$, we start by adding and specializing the lemma *m_subst_abs_neq* to our htpothesis. Let *w* be a new name that is not in the set

$fv(\lambda'_z.e_1) \cup fv(e3) \cup fv(e2) \cup \{x\} \cup \{y\}$, we use this to apply the transitivity of the α-equivalency using ([*y* := *e3*] *n_abs w* ([*x* := *e2*] *swap z w e1*)). In the first case, we have the α-equivalency of the LHS and our new term, used in the transitivity. The use of *aeq_m_subst_out* is enought to turn this into a trivial proof since the result is the application of the lemma *m_subst_abs_neq*. The second case consists of removing the abstractions from the metasubstitutions to use the induction hypothesis on the remaining goal. We start on the RHS by removing the abstraction from inside the metasubstitution in the subterm ([*y* := *e3*] *n_abs z e1*). To do this we use the lemma *m_subst_abs_neq* and the transitivity of the α-equivalency using as a middle term ([*x* := [*y* := *e3*] *e2*] *n_abs w* ([*y* := *e3*] *swap z w e1*)). As a first case, we have to prove that the LHS is α-equivalent to the new middle term. Let *w'* be a new name that is not in the set $fv(\lambda'_y.e_1) \cup fv(e3) \cup fv(e2) \cup \{x\} \cup \{y\} \cup \{w\}.Then, acomparisonismadebetween[z]and[w']$. For *z* == *w'*, we remove the abstractions from inside the metasubstitutions using the same strategy as previously, using the lemma *m_subst_abs_neq*. Then, we use the lemma *aeq_abs_same* to eliminate the abstractions from each side of the α-equivalency since given that if the variables are the same, the terms have to be α-equivatent. Then we use the lemma *swap_subst_rec_fun* to insert the swaps from outside of the inner metasubstitution to inside them. After that, the induction hypothesis is applied to the goal. Then, all we have are intermediate results that are proved trivially by applying previous lemmas on *fv_nom* and *not_eq*. For the second case in the transitivity of the α- equivalency, we have the α-equivalency of our new term ([*y* := *e3*] *n_abs w* ([*x* := *e2*] *swap z w e1*)) and the RHS of our initial α-equivalency. First, we compare *w* and *z*. For *w* == *z*, first we apply *m_subst_abs_neq* propagate the internal metasubstitution over the abstraction taking the new names that have already been defined. Then we use the lemma *aeq_abs_same* to remove the outter abstractions, since they have the same variable. Then we use the lemma *swap_subst_rec_fun* to insert the swaps from outside of the inner metasubstitution to inside them. After that, the transitivity of the α-equivalency is applied to ([*y* := *e3*] [*x* := *e2*] (*swap w1 z e1*)). We ude the lemma *aeq_m_subst_out* to simplify the α-equivalency to the terms inside the metasubstitutions. The remaining steps of the proof are done by unfolding and solving *vswap* comparisons and applying *aeq_m_subst_eq* to simplify the terms. It was also necessary to use lemmas that regard *fv_nom* and *not_eq*, but the proofs were fairly trivial. For the first case in the transivity we have the α-equivalency of the LHS and the new term. In the explicit substitution case, the initial goal is ([*y* := *e3*] ([*x* := *e2*] *n_sub e1_1 z e1_2*)) =*a* ([*x* := [*y* := *e3*] *e2*] ([*y* := *e3*] *n_sub e1_1 z e1_2*)), 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 that has the same size as *e1_1* 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_sub e1_1 x e1_2*) is reduced to *n_sub e1_1 x* ([*x* := *e2*] *e1_2*) by applying the lemma in m_subst_sub_eq and the LHS is reduced to ([*y* := *e3*] *n_sub e1_1 x* ([*x* := *e2*] *e1_2*)). Let *w* be a new name that is not in the set $fv([x := e1_2]e1_1) \cup fv(e2) \cup fv(e3) \cup \{x\} \cup \{y\}$. We use this new variable to propagate the explicit substition to outside of the metasubstitution. We then move on to the RHS of the α-equivalency. Let *w'* be a new name that is not in the set $fv([x := e1_2]e1_1) \cup fv(e2) \cup fv(e3) \cup \{x\} \cup \{w\} \cup \{y\}$. Then, the lemma *m_subst_abs_neq* is applied again to propagate the metasubstitution to inside the explicit substitution. The next step is to do the same thing for the RHS. We start by applying the transitivity in the α-eqeuivalency with the term generated by applying *m_subst_sub_neq* into ([*y* := *e3*] *n_sub e1_1 x e1_2*) to propagate the inner metasubstitution to inside the explicit substitution. The same is done for the outter metasubstitution but in this case the transitivity is not necessary, all we have to do is rewrite our new term with the metasubstitutions propagated to inside the explicit substitution. Then, the lemma *aeq_sub_diff* e applied to remove the explicit substitution from both sides of the α-equivalency. Since the variables are different, this lemma is used instead of the one for same variables used previously. The induction hypothesis that states that the two sets of the metasubstitutions to any expression, given the same inducton hypothesis that we had on the star of this proof, are α-equivalent. The rest of the proof is done using the lemmas regarding *fv_nom*, *swap* and $\neq$ in a trivial manner. For the other case of this transitivity, we have the middle term and the RHS. The solution is simple, using the *aeq_m_subst_out* lemma, which simplifies the goal into the application of *m_subst_sub_neq* in ([*y* := *e3*] *n_sub e1_1 x e1_2*). The rest of the proof is done using the lemmas regarding *fv_nom*, *swap* and $\neq$ in a trivial manner. Now we are in the second case of the explicit substitution, the one in which $z \neq x$. We start by comparing *y* and *z*. The first case is the one in which $y = z$. Let *w* be a new name that is not in the set $fv([z := e1_2]e1_1) \cup fv(e3) \cup fv(e2) \cup \{x\}$. This new atom is used in applying the lemma *m_subst_sub_neq*, in th LHS, to propagate the inner metasubstitution to inside the explicit substitution. The transitivity of the α-equivalency is used with the propagation as the middle term. The first case of this trasitivity is simply our previous hypothesis generated using the *m_subst_sub_neq* lemma and proving it is trivial. In this case, in which $z \neq x$, we start by comparing *z* and *w*. In the $z = w$ case, we use *m_subst_sub_eq* in the LHS and *m_subst_sub_neq* in the RHS to propagate the metasubstitutions to inside the explicit substitutions. We then apply *aeq_sub_same* to simplify the α-equivalency to the one of the terms of the explicit substitutions. We then apply the *aeq_m_subst_in* lemma to simplify the α-equivalency into the one if the terms inside the metasubstitutions. The rest of the proof is done trivially using the lemmas for *notin*, *fv_nom*, *swap* and $\neq$. In the case $z \neq w$, we propagate the metasubstitution on both sides to inside the explicit substitution and use the *abs_subs_same* lemma to make the α-equivalency about the terms of the α-equicalency on each side of it. We then apply lemmas on the *swap* so that it is possible to apply the induction hypothesis that states taht the same pattern of metasubstitutions as the one in our original assumption in valid for an expression, given the same hypothesis valid for this case and the fact that this term has to have the same size as *e1_1*. We then use *aeq_m_subst_out* to simplifies each side of the α-equivalency into the internal terms of each metasubstitution. The rest of the proof is done trivially using the lemmas for *fv_nom*, *swap*, *notin* and $\neq$. We are now in the case which $w \neq z$. First, let *w* be a new name that is not in the set $fv([z := e1_2]e1_1) \cup fv(e3) \cup fv(e2) \cup \{x\} \cup \{y\}$. We start with the LHS by simplifing ([*x* := *e2*] *n_sub e1_1 z e1_2*) into *n_sub* ([*x* := *e2*] *swap z w e1_1*) *w* ([*x* := *e2*] *e1_2*) using the *m_subst_sub_neq* lemma. We have to use the transitivity of the α-equivalency to do this. It generates two cases. The first case is proved by simplifying the α-equivalency into the one for the terms inside the metasubstituions in each side. The rest is done trivially using the lemmas for *notin*, We do the same thing dor the RHS, propagating the metasubstitution to inside the explicit substitution by applying the *m_subst_sub_neq* lemma and the α-equivalency, using ([*x* := [*y* := *e3*] *e2*] *n_sub* ([*y* := *e3*] *swap z w e1_1*) *w* ([*y* := *e3*] *e1_2*)). For the case in which we have the LHS and our new term, let *w'* be a new name that is not in the set $fv([y := e1_2]e1_1) \cup fv(e3) \cup fv(e2) \cup \{x\} \cup \{w\} \cup \{y\}$. Then, we compare this new element with *z*. We start the case *z* = *w'* by applying the *m_subst_sub_neq* lemma on both sides of the α-equivalency to propagate the metasubstitution to inside the explicit substitution. Then, we apply the *aeq_sub_same* lemma to remove the explicit substitutions and are left with only the internal terms of them. We then apply the transitivity of the α-equivalency using as middle term (*subst_rec_fun* (*subst_rec_fun* (*swap w' w e1_1*) *e2 x*) *e3 y*). The *m_subst* has been unfolded into *subst_rec_fun* so that we can apply to it the lemma *swap_subst_rec_fun*. Doing so we were able to propagate the swaps to inside the metasubstitutions. By using *aeq_m_subst_out*, all we had left to do was solve the other two cases using *fv_nom*, *swap* and $\neq$ lemmas, which was trivial. The other case if also similar to this. In the case in which *z* = *w'*, we first compare *w* and *z*. The first case is done quite similarly to the rest of the proof. The goal is to propagate the metasubstitution to inside the explicit substitution and use the *aeq_sub_same* lemma to remove the substitutions and set as goal the α-equivalency of the equivalent terms in the explicit substitutions. The second case also follows from a similar path to solution. □

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[1, 2, 3, 4, 10, 14].

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 [13] to prove properties of calculi with explicit substitutions such as confluence[22, 23, 18]. In

References

[1] M. Abadi, L. Cardelli, P.-L. Curien & J.-J. Lévy (1991): *Explicit Substitutions*. *Journal of Functional Programming* 1(4), pp. 375–416.

[2] B. Accattoli (2012): *An Abstract Factorization Theorem for Explicit Substitutions*. In: *RTA*, pp. 6–21.

[3] M. Ayala-Rincón, F.C. de Moura & F. Kamareddine (2002): *Comparing Calculi of Explicit Substitutions with Eta-reduction*. In R.J.G.B. de Queiroz, editor: *Proceedings Ninth Workshop on Logic, Language, Information and Computation (WoLLIC 2002)*, *ENTCS* 67, Elsevier Science Publishers, pp. 77–96.

[4] M. Ayala-Rincón, F. L. C. de Moura & F. Kamareddine (2005): *Comparing and Implementing Calculi of Explicit Substitutions with Eta-Reduction*. *Annals of Pure and Applied Logic* 134, pp. 5–41.

[5] H. P. Barendregt (1984): *The Lambda Calculus : Its Syntax and Semantics (Revised Edition)*. North Holland.

[6] Z.-el-A. Benaissa, D. Briaud, P. Lescanne & J. Rouyer-Degli (1996): *$\lambda\upsilon$, a Calculus of Explicit Substitutions Which Preserves Strong Normalization*. *JFP* 6(5), pp. 699–722.

[7] Stefan Berghofer & Christian Urban (2007): *A Head-to-Head Comparison of de Bruijn Indices and Names*. *Electronic Notes in Theoretical Computer Science* 174(5), pp. 53–67, doi:10.1016/j.entcs.2007.01.018.

[8] R. Bloo & K. Rose (1995): *Preservation of Strong Normalisation in Named Lambda Calculi with Explicit Substitution and Garbage Collection*. In: *CSN-95: COMPUTER SCIENCE IN THE NETHERLANDS*, pp. 62–72.

[9] Roel Bloo & Herman Geuvers (1999): *Explicit Substitution: On the Edge of Strong Normalization*. *Theoretical Computer Science* 211(1-2), pp. 375–395, doi:10.1016/s0304-3975(97)00183-7.

[10] E. Bonelli (2001): *Perpetuality in a Named Lambda Calculus With Explicit Substitutions*. *Mathematical Structures in Computer Science* 11(1), pp. 47–90, doi:10.1017/s0960129500003248.

[11] Pierre-Louis Curien, Thérèse Hardin & Jean-Jacques Lévy (1996): *Confluence Properties of Weak and Strong Calculi of Explicit Substitutions*. *Journal of the ACM* 43(2), pp. 362–397, doi:10.1145/226643.226675.

[12] R. David & B. Guillaume (2001): *A Lambda-Calculus with Explicit Weakening and Explicit Substitution*. *Mathematical Structures in Computer Science* 11(1), pp. 169–206.

[13] Flávio L. C. de Moura & Leandro O. Rezende (2021): *A Formalization of the (Compositional) z Property*. In: *Fifth Workshop on Formal Mathematics for Mathematicians*.

[14] Ken-etsu Fujita & Koji Nakazawa (2016): *Church-Rosser Theorem and Compositional Z-Property*.

[15] M. Gabbay & A. Pitts (1999): *A New Approach to Abstract Syntax Involving Binders*. In: *14th Symposium on Logic in Computer Science (LICS'99)*, IEEE, Washington - Brussels - Tokyo, pp. 214–224.

[16] B. Guillaume (2000): *The λS_e-Calculus Does Not Preserve Strong Normalization*. *J. of Func. Programming* 10(4), pp. 321–325.

[17] Fairouz Kamareddine & Alejandro Ríos (1997): *Extending a Lambda-Calculus with Explicit Substitution Which Preserves Strong Normalisation Into a Confluent Calculus on Open Terms*. *J. Funct. Program.* 7(4), pp. 395–420.

[18] D. Kesner (2008): *Perpetuality for Full and Safe Composition (in a Constructive Setting)*. In: *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, pp. 311–322, doi:10.1007/978-3-540-70583-3_26.

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

[20] P.-A. Melliès (1995): *Typed λ-Calculi with Explicit Substitutions May Not Terminate in Proceedings of TLCA'95*. *LNCS* 902.

[21] C. A. Muñoz (1996): *Confluence and Preservation of Strong Normalisation in an Explicit Substitutions Calculus*. In: *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pp. 440–447, doi:10.1109/LICS.1996.561460.

[22] Koji Nakazawa & Ken-etsu Fujita (2016): *Compositional Z: Confluence Proofs for Permutative Conversion*. *Studia Logica* 104(6), pp. 1205–1224, doi:10.1007/s11225-016-9673-0.

[23] Koji Nakazawa & Ken-etsu Fujita (2017): *Z for Call-by-Value*. In: *6th International Workshop on Cofluence (IWC 2017)*, pp. 57–61.

[24] Koji Nakazawa, Ken-etsu Fujita & Yuta Imagawa (2023): *Z Property for the Shuffling Calculus*. *Mathematical Structures in Computer Science*, pp. 1–13, doi:10.1017/S0960129522000408.

[25] The Coq Development Team (2021): *The Coq Proof Assistant*. Zenodo, doi:10.5281/ZENODO.5704840.