A Formalized Extension of the Substitution Lemma in Coq

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

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

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

1 Introduction

In this work, we present a formalization of the substitution lemma [5] in a general framework that extends the λ-calculus with an explicit substitution operator using the Coq proof assistant [24]. The source code is publicly available at

https://flaviomoura.info/files/msubst.v

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, $x \neq y$ and $x \notin FV(v)$, where $FV(v)$ is the set of free variables of v. Then $\{y := v\}\{x := u\}t = \{x := \{y := v\}u\}\{y := v\}t$.

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 trivial due to the interaction between the metasubstitution and the explicit substitution operator. Our formalization is done in a nominal setting that uses the MetaLib[1] package of Coq, but no particular explicit substitution calculi is taken into account because the expected behaviour between the metasubstitution operation with the explicit substitutition constructor is the same regardless the calculus. The formalization was done with Coq (platform) version 8.15.2, which already comes with the Metalib package. The novel contributions of this work are twofold:

1. The formalization is modular in the sense that no particular calculi with explicit substitutions is taken into account. Therefore, we believe that this formalization could be seen as a generic framework for proving properties of these calculi that uses the substitution lemma in the nominal setting [16, 20, 21];

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

Submitted to:
FROM 2023

© M. J. D. Lima & F. L. C. de Moura
This work is licensed under the
Creative Commons Attribution License.

2. A solution to a circularity problem in the proofs is given. It adds an axiom to the formalization that allow a rewrite step inside a let expression. Such a rewrite step is problematic and does not seem to have a trivial solution.

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 [12] 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, whose grammar of terms is given by:

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

where x represents a variable which is taken from an enumerable set, $\lambda_x.t$ is an abstraction, and $t\ t$ is an application. The abstraction is the only binding operator: in the expression $\lambda_x.t$, x binds in t, called the scope of the abstraction. This means that all free occurrence of x in t is bound in $\lambda_x.t$. A variable that is not in the scope of an abstraction is free. A variable in a term is either bound or free, but note that a varible can occur both bound and free in a term, as in $(\lambda_y.y)\ y$.

The main rule of the λ-calculus, named β-reduction, is given by:

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

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

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

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

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

[2] 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 modulo α-equivalence, while the swapping is, thereby rendering it a more fitting choice when operating with α-classes.

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 *vswap*, 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*. As a short example to acquaint ourselves with the Coq notation, let us show how we will write the proofs:

`Lemma` *vswap_id*: $\forall$ *x y*, *vswap x x y* = *y*.
Proof. The proof is by case analysis, and it is straightforward in both cases, when $x = y$ and $x \neq y$. □

2.1 An explicit substitution operator

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

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

where $[x := u]t$ represents a term with an operator that will be evaluated with specific rules of a substitution calculus. The intended meaning of the explicit substitution is that it will simulate the metasubstitution. This formalization aims to be a generic framework applicable to any calculi with explicit substitutions using a named notation for variables. Therefore, we will not specify rules about how one can simulate the metasubstitution, but it is important to be aware that this is not a trivial task as one can easily lose important properties of the original λ-calculus [18, 14].

Calculi with explicit substitutions are formalisms that deconstruct the metasubstitution operation into finer-grained steps, thereby functioning as an intermediary between the λ-calculus and its practical implementations. In other words, these calculi shed light on the execution models of higher-order languages. In fact, the development of a calculus with explicit substitutions faithful to the λ-calculus, in the sense of the preservation of some desired properties were the main motivation for such a long list of calculi with explicit substitutions invented in the last decades [1, 23, 6, 10, 19, 15, 8, 11, 17].

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). Accordingly, *n_sexp* denotes the set of nominal λ-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).
```

The *size* of a term, also written as $|t|$, and the set *fv_nom* of the free variables of a term are defined as usual:

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

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

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

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

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

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

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

```
Lemma swap_neq: ∀ x y z w, z ≠ w → vswap x y z ≠ vswap x y w.
```

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

```
Lemma swap_symmetric : ∀ t x y, swap x y t = swap y x t.
```

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

[3] https://flaviomoura.info/files/msubst.v

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

Lemma *swap_equivariance* : $\forall\ t\ x\ y\ z\ w,\ swap\ x\ y\ (swap\ z\ w\ t) = swap\ (vswap\ x\ y\ z)\ (vswap\ x\ y\ w)\ (swap\ x\ y\ t)$.

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 used so far is induction on the structure of terms. Nevertheless, the builtin induction principle automatically generated in Coq for the inductive definition *n_sexp* is not strong enough due to swappings:

```
forall P :n_sexp -> Prop,
 (forall x:atom, P(n_var x)) ->
 (forall (x:atom) (t:n_sexp), P t -> P(n_abs x t)) ->
 (forall t1:n_sexp, P t1 -> forall t2:n_sexp, P t2 -> P(n_app t1 t2)) ->
 (forall t1:n_sexp, P t1 -> forall (x:atom) (t2:n_sexp), P t2 -> P([x:=t2]t1)) ->
       forall t:n_sexp, P t
```

In fact, in general, the induction hypothesis in the abstraction case (resp. explicit substitution case) refers to the body *t* of the abstraction (resp. *t1* of the explicit substitution), while the goal involves a swap acting on the body of the abstraction (resp. explicit substitution). In order to circunvet this problem, we defined a customized induction principle based on the size of terms:

Lemma *n_sexp_induction*: $\forall\ P : n\_sexp \rightarrow \mathtt{Prop},\ (\forall\ x,\ P\ (n\_var\ x)) \rightarrow$
$(\forall\ t1\ z,\ (\forall\ t2\ x\ y,\ size\ t2 = size\ t1 \rightarrow P\ (swap\ x\ y\ t2)) \rightarrow P\ (n\_abs\ z\ t1)) \rightarrow$
$(\forall\ t1\ t2,\ P\ t1 \rightarrow P\ t2 \rightarrow P\ (n\_app\ t1\ t2)) \rightarrow$
$(\forall\ t1\ t3\ z,\ P\ t3 \rightarrow (\forall\ t2\ x\ y,\ size\ t2 = size\ t1 \rightarrow P\ (swap\ x\ y\ t2)) \rightarrow P\ (n\_sub\ t1\ z\ t3)) \rightarrow (\forall\ t,\ P\ t)$.

which states that in order to conclude that a certain property *P* holds for all terms, we need to prove that:

1. *P* must hold for any variable;
2. If *P* holds for the term $(x\ y)t_2$, where t_1 and t_2 have the same size, then it also holds for the abstraction $\lambda_z.t_1, \forall x, y, z, t_1$ and t_2;
3. If *P* holds for the terms t_1 and t_2 the it also holds for the application $t_1\ t_2$;
4. If *P* holds for the term t_3 and for the term $(x\ y)t_2$, where t_1 and t_2 have the same size, then it also holds for the explicit substitution $[z := t_3]t_1, \forall x, y, z, t_1, t_2$ and t_3.

The following lemma is a first example of the use of the *n_sexp_induction* principle:

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. Note that in the paper and pencil notation, this lemma states that:

If $x' \notin fv\_nom(t)$ then $(\!(x\ y)\!)x' \notin fv\_nom((x\ y)t)$.

The proof is by induction on the size of the term *t*.

1. If *t* is a variable, say *z*, then $x' \neq z$ by hypothesis, and we need to prove that $(\!(x\ y)\!)x' \neq (\!(x\ y)\!)z$. We conclude by lemma *swap_neq*.

2. If is an abstraction, say $t = \lambda_z.t_1$, then we have by induction hypothesis that if $x' \notin (x\ y)t_2$ then $((x_0\ y_0))x' \notin (x_0\ y_0)(x\ y)t_2$ for any term t_2 with the same size as t_1, and any variables x, y, x_0 and y_0. At this point is important to notice that an structural induction would generate an induction hypothesis with t_1 only, which is not strong enough to prove the goal $((x\ y))x' \notin fv\_nom((x\ y)\lambda_z.t_1)$ that has $(x\ y)t_1$ (and not t_1 alone!) after the propagation of the swap. In addition, we have by hypothesis that $x' \notin fv\_nom(t_1)\backslash\{z\}$. This means that either $x' = z$ or $x' \notin fv\_nom(t_1)$, and there are two subcases:
   (a) If $x' = z$ then the goal is $((x\ y))z \notin fv\_nom((x\ y)\lambda_z.t_1) \Leftrightarrow ((x\ y))z \notin fv\_nom(\lambda_{((x\ y))z}.(x\ y)t_1) \Leftrightarrow ((x\ y))z \notin fv\_nom((x\ y)t_1)\backslash\{((x\ y))z\}$ we are done by lemma *notin_remove_3*.[4]
   (b) Otherwise, $x' \notin fv\_nom(t_1)$, and we conclude using the induction hypothesis taking $x_0 = x$, $y_0 = y$ and the universally quantified variables x and y of the internal swap as the same variable (it does not matter which one).
3. The application case is straightforward from the induction hypothesis.
4. The case of the explicit substitution, *i.e.* when $t = [z := t_2]t_1$, we have to prove that $((x\ y))x' \notin fv\_nom((x\ y)([z := t_2]t_1))$. We then propagate the swap over the explicit substitution operator and show, by the definition of fv_nom, we have to prove that both $((x\ y))x' \notin (fv\_nom((x\ y)t_1))\backslash\{((x\ y))z\}$ and $((x\ y))x' \notin fv\_nom((x\ y)t_2)$.
   (a) In the former case, the hypothesis $x' \notin fv\_nom(t_1)\backslash\{z\}$ generates two subcases, either $x' = z$ or $x' \notin fv\_nom(t_1)$, and we conclude with the same strategy of the abstraction case.
   (b) The later case is straightforward by the induction hypothesis. □

The other direction is also true, but we skip the proof that is also by induction on the size of term t:

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

2.2 α-equivalence

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

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

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

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

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

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

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').
```

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 (proofs in the source file[5]). 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 do not occur free in a term, it eventually renames only bound variables, *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 following 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 file. 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. Before commenting this proof, we state the lemma with the pencil and paper (meta)notation:

[5] https://flaviomoura.info/files/msubst.v

If $z \notin fv\_nom(t)$ and $x \notin fv\_nom(t)$ then $(z\,x)(x\,y)t =_\alpha (z\,y)t$.

Initially, observe the similarity of the left hand side (LHS) of the α-equation with the lemma *shuffle_swap*:

$\forall w\ y\ z\ t, w \neq z \to y \neq z \to (w\ y)((y\ z)t) = (w\ z)((w\ y)t)$

In order to use it, we need to have that both $z \neq y$ and $x \neq y$. We start comparing z and y:

1. If $z = y$ then the right hand side (RHS) reduces to t because the swap is trivial, and the LHS also reduces to t since swap is involutive.

2. When $z \neq y$ then we proceed by comparing x and y:

   (a) If $x = y$ then both sides of the α-equation reduces to $(z\,y)t$, and we are done.

   (b) Finally, when $x \neq y$, we can apply the lemma $shuffle\_swap$, and use lemma *aeq_swap* to reduce the current goal to $(z\,x)t =_\alpha t$, 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

As presented in Section 2, the main operation of the λ-calculus is the β-reduction (2) that expresses how to evaluate a function applied to an argument. The β-contractum $\{x := u\}t$ represents a capture free in the sense that no free variable becomes bound by the application of the metasubstitution. This operation is in the meta level because it is outside the grammar of the λ-calculus (and hence its name). In [5], Barendregt defines it as follows:

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

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

> 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 are different possible approaches. In our case, we perform a renaming of bound variables whenever the metasubstitution is propagated inside a binder. In our case, there are two binders: abstractions and explicit substitutions.

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

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

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.
```

Note that this function is not structurally recursive due to the swaps in the recursive calls, and that's why we need to provide the size of the term t as the measure parameter. Alternatively, a structurally recursive version of the function *subst_rec_fun* can be found in the file *nominal.v* of the *Metalib* library[6]. It has the size of the term as an explicit parameter in which the substitution will be performed, and hence one has to deal with the size of the term in each recursive call. We write $\{x{:=}u\}t$ instead of *subst_rec_fun t u x*, and refer to it just as "metasubstitution".

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

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

Proof. The proof is done by induction on the size of the term t using *n_sexp_induction* defined above. The interesting cases are the abstraction and the explicit substituion. We focus in the abstraction case, *i.e.* when $t = \lambda_y.t_1$, where the goal to be proven is $\{x := u\}(\lambda_y.t_1) =_\alpha \lambda_y.t_1$. We consider two cases:

1. If $x = y$ the result is trivial because both LHS and RHS are equal to $\lambda_y.t_1$
2. If $x \neq y$, we have to prove that $\lambda_z.\{x := u\}(y\ z)t_1 =_\alpha \lambda_y.t_1$, where z is a fresh name not in the set $fv\_nom(u) \cup fv\_nom(\lambda_y.t_1) \cup \{x\}$. The induction hypothesis express the fact that every term with the same size as the body t_1 of the abstraction satisfies the property to be proven: $\forall t', |t'| = |t_1| \rightarrow \forall u\ x'\ x_0\ y_0, x' \notin fv((x_0\ y_0)t') \rightarrow \{x' := u\}((x_0\ y_0)t') =_\alpha (x\ y)t'$. Therefore, according to the definition of the metasubstitution (function [subst_rec_fun]), the variable y will be renamed to z, and the metasubstitution is propagated inside the abstraction resulting in the following goal:

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

$\lambda_z.\{x := u\}((z\ y)t_1) =_\alpha \lambda_y.t_1$. Since $z \notin fv\_nom(\lambda_y.t_1) = fv\_nom(t_1)\backslash\{y\}$, there are two cases to consider, either $z = y$ or $z \in fv(t_1)$:

(a) $z = y$: In this case, we are done by the induction hypothesis taking $x_0 = y_0 = y$, for instance.

(b) $z \neq y$: In this case, we can apply the rule *aeq_abs_diff*, resulting in the goal $\{x := u\}((y\ z)t_1) =_\alpha (y\ z)t_1$ which holds by the induction hypothesis, since $|(z\ y)t_1| = |t_1|$ and $x \notin fv\_nom((y\ z)t_1)$ because $x \neq z$, $x \neq y$ and $x \notin fv\_nom(t_1)$.

The explicit substitution case is also interesting, *i.e.* if $t = [y := t_2]t_1$, 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 when the metasubstitution's variable is equal to the abstraction's variable. Their proofs are straightforward from the definition *subst_rec_fun*. The corresponding version when the metasubstitution's variable is different from the abstraction's variable will be presented later.

`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*).

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 a variable and t, t', u and u' are terms. This proof is split in two cases: 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 cases are then combined through the transitivity of the α-equivalence relation. Nevertheless, this task was 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 term t, and we will focus on the abstraction case, *i.e.* $t = \lambda_y.t_1$. The goal in this case is $\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u'\}(\lambda_y.t_1)$.

1. If $x = y$ then the result is trivial by lemma *m_subst_abs_eq*.
2. If $x \neq y$ then we need two fresh names in order to propagate the metasubstitution inside the abstractions on each side of the α-equation. Let x_0 be a fresh name not in the set $fv\_nom(u) \cup fv\_nom(\lambda_y.t_1) \cup \{x\}$, and x_1 be a fresh name not in the set $fv\_nom(u') \cup fv\_nom(\lambda_y.t_1) \cup \{x\}$. After propagating the metasubstitution we need to prove $\lambda_{x_0}.\{x := u\}((y\ x_0)t_1) =_\alpha \lambda_{x_1}.\{x := u'\}((y\ x_1)t_1)$, and we proceed by comparing x_0 and x_1:
   (a) If $x_0 = x_1$ then we are done by the induction hypothesis.
   (b) Otherwise, we need to apply the rule *aeq_abs_diff* and the goal is $\{x := u\}((y\ x_0)t_1) =_\alpha (x_0\ x_1)(\{x := u'\}((y\ x_1)t_1))$. But in order to proceed we need to know how to propagate the swap inside the metasubstitution, which is the content of 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. We write the statement of the lemma in metanotation before starting the proof:

$$\forall t\ u\ x\ y\ z, (y\ z)(\{x := u\}t) =_\alpha \{(\!(y\ z)\!)x := (y\ z)u\}(y\ z)t$$

The proof is by induction on the size of the term t, and again we will focus only on the abstraction case, *i.e.* when $t = \lambda_w.t_1$. The goal in this case is $(y\ z)(\{x := u\}(\lambda_w.t_1)) =_\alpha \{(\!(y\ z)\!)x := (y\ z)u\}((y\ z)\lambda_w.t_1)$, and we proceed by comparing x and w.

1. If $x = w$ the α-equality is trivial.
2. If $x \neq w$ then we need a fresh name, say w_0, to be able to propagate the metasubstitution inside the abstraction on the LHS of the α-equation. The variable w_0 is taken such that it is not in the set $fv\_nom(u) \cup fv\_nom(\lambda_w.t_1) \cup \{x\}$, and we get the goal $\lambda_{(\!(y\ z)\!)w_0}.(y\ z)(\{x := u\}(w\ w_0)t_1) =_\alpha \{(\!(y\ z)\!)x := (y\ z)u\}(\lambda_{(\!(y\ z)\!)w}.(y\ z)t_1)$. Now we propagate the metasubstitution over the abstraction in the RHS of the goal. Since $x \neq w$ implies $(\!(y\ z)\!)x \neq (\!(y\ z)\!)w$, we need another fresh name, say w_1, not in the set $fv\_nom((y\ z)u) \cup fv\_nom(\lambda_{(\!(y\ z)\!)w}.(y\ z)t_1) \cup \{(\!(y\ z)\!)x\}$, and after the propagation we need to prove that $\lambda_{(\!(y\ z)\!)w_0}.(y\ z)(\{x := u\}(w\ w_0)t_1) =_\alpha \lambda_{w_1}.\{(\!(y\ z)\!)x := (y\ z)u\}((w_1\ (\!(y\ z)\!)w)((y\ z)t_1))$. We consider two cases: either $w_1 = (\!(y\ z)\!)w_0$ or $w_1 \neq (\!(y\ z)\!)w_0$. In the former case, we apply the rule *aeq_abs_same* and we are done by the induction hypothesis. When $w_1 \neq (\!(y\ z)\!)w_0$, the application of the rule *aeq_abs_diff* generates the goal

$$(w_1\ (\!(y\ z)\!)w_0)(y\ z)(\{x := u\}(w\ w_0)t_1) =_\alpha \{(\!(y\ z)\!)x := (y\ z)u\}((w_1\ (\!(y\ z)\!)w)((y\ z)t_1)) \quad (6)$$

   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, whose corresponding 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 $u =_\alpha u'$, but this is exactly what we were trying to prove in the previous lemma.

Therefore, we are in a circular problem because both *aeq_m_subst_in_trial* and *swap_m_subst* depend on each other to be proved!

Our solution to this problem consists in taking advantage of the fact that α-equivalent terms have the same set of free variables (see lemma *aeq_fv_nom*), and noting that the external swap in the LHS of (6) was generated by the application of the rule *aeq_abs_diff* because the abstractions have different bindings. Let's go back to the proof of lemma *aeq_m_subst_in*: `Lemma` *aeq_m_subst_in*: $\forall$ *t u u' x*, *u* =*a u'* $\rightarrow$ ({$x := u$}*t*) =*a* ({$x := u'$}*t*).

Proof. We go directly to the abstraction case. When $t = \lambda_y.t_1$, the goal is $\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u'\}(\lambda_y.t_1)$. If $x \neq y$ then the fresh name needed for the LHS must not belong to the set $fv\_nom(u) \cup fv\_nom(\lambda_y.t_1) \cup \{x\}$, while the fresh name for the RHS must not belong to $fv\_nom(u') \cup fv\_nom(\lambda_y.t_1) \cup \{x\}$. These sets differ only by the subsets $fv\_nom(u)$ and $fv\_nom(u')$. Nevertheless, these subsets are equal because u and u' are α-equivalent (see lemma *aeq_fv_nom*). Concretely, the current goal is as follows:

```
(let (z, _) := atom_fresh (union (fv_nom u) (union (fv_nom (n_abs y t1))
       (singleton x))) in n_abs z (subst_rec_fun (swap y z t1) u x)) =a
(let (z, _) := atom_fresh (union (fv_nom u') (union (fv_nom (n_abs y t1))
         (singleton x))) in n_abs z (subst_rec_fun (swap y z t1) u' x))
```

where the sets $fv\_nom(u)$ and $fv\_nom(u')$ appear in different *let* expressions, each one is responsible for generating one fresh name. But since these sets are equal, if one could replace $fv\_nom(u)$ by $fv\_nom(u')$ (or vice-versa) then only one fresh name is generated after evaluating the *atom_fresh* function. Nevertheless, the only way that we managed to do such replacement was by adding the following axiom:

```
Axiom Eq_implies_equality: forall t1 t2, t1 =a t2 -> fv_nom t1 = fv_nom t2.
```

This axiom is similar to lemma *aeq_fv_nom* where the set equality [=] was replaced by the syntactic (Leibniz) equality =. Now, we can generate just one fresh name and propagate the metasubstitution on both sides of the goal, and we are done by the induction hypothesis. The case of the explicit substitution is similar, and with this strategy we avoid both 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*: $\forall\ t\ t'\ u\ x,\ t =a\ t' \rightarrow (\{x := u\}t) =a (\{x := u\}t')$.

Proof. The proof is by induction on the size of the term t. Note that induction on the hypothesis $t =a\ t'$ does not work due to a similar problem involving swaps that appears when structural induction on t is used. The abstraction and the explicit substitution are the interesting cases.

In the abstraction case, we need to prove that $\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u\}t'$, where $\lambda_y.t_1 =_\alpha t'$ by hypothesis. Therefore, t' must be an abstraction, and according to our definition of α-equivalence there are two possible subcases:

1. In the first subcase, $t' = \lambda_y.t_2$, where $t_1 =_\alpha t_2$, and hence the current goal is $\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u\}(\lambda_y.t_2)$. We proceed by comparing x and y:
   (a) If $x = y$ then, we are done by using twice lemma *m_subst_abs_eq*.
   (b) When $x \neq y$, then we need to propagate the metasubstitution on both sides of the goal. On the LHS, we need a fresh name that is not in the set $fv(u) \cup fv(\lambda_y.t_1) \cup \{x\}$, while for the RHS, the fresh name cannot belong to the set $fv(u) \cup fv(\lambda_y.t_2) \cup \{x\}$. From the hypothesis $t_1 =_\alpha t_2$, we know, by lemma *aeq_fv_nom*, that the sets $fv\_nom(t_1)$ and $fv\_nom(t_2)$ are equal. Therefore, we can take just one fresh name, say z, and propagate both metasubstitutions over abstractions with the same binding, and we conclude with the induction hypothesis.
2. In the second subcase, $t' = \lambda_{y_0}.t_2$, where $t_1 =_\alpha (y_0\ y)t_2$ and $y \neq y_0$. The current goal is
   $$\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u\}(\lambda_{y_0}.t_2)$$
   and we proceed by comparing x and y:
   (a) If $x = y$ then the goal simplifies to $\lambda_y.t_1 =_\alpha \{x := u\}(\lambda_{y_0}.t_2)$ by lemma *m_subst_abs_eq*, and we pick a fresh name x, that is not in the set $fv\_nom(u) \cup fv\_nom(\lambda_{y_0}.t_2) \cup \{y\}$, and propagate the metasubstitution on the RHS of the goal, resulting in the new goal $\lambda_y.t_1 =_\alpha \lambda_x.\{y := u\}((y_0\ x)t_2)$. Note that the metasubstitution on the RHS has no effect in the term $(y_0\ x)t_2$ because $y \neq y_0$, $y \neq x$ and y does not occur free in t_2 and we conclude by hypothesis.
   (b) If $x \neq y$ then we proceed by comparing x and y_0 on the RHS, and the proof, when $x = y_0$, is analogous to the previous subcase. When both $x \neq y$ and $x \neq y_0$ then we need to propagate the metasubstitution on both sides of the goal $\{x := u\}(\lambda_y.t_1) =_\alpha \{x := u\}(\lambda_{y_0}.t_2)$. We have that $\lambda_y.t_1 =_\alpha \lambda_{y_0}.t_2$ and hence the sets $fv\_nom(\lambda_y.t_1)$ and $fv\_nom(\lambda_{y_0}.t_2)$ are equal. Therefore, only one fresh name, say x_0, that is not in the set $x_0 \notin fv\_nom(u) \cup fv\_nom(\lambda_{y_0}.t_2) \cup \{x\}$ is enough to fulfill the conditions for propagating the metasubstitutions on both sides of the goal, and we are done by the induction hypothesis.
3. The explicit substitution operation is also interesting, but we will not comment because we are running out of space. □

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

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

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

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

Proof. Firstly, we write the lemma in metanotation: $\forall x\,y\,z\,t\,u, (x\,y)\{z := u\}t =_\alpha \{(\!(x\,y)\!)z := (x\,y)u\}(x\,y)t$. Next, we compare x and y, since the case $x = y$ is trivial. When $x \neq y$, the proof proceeds by induction on the size of the term t. The tricky cases are the abstraction and explicit substitution, and we comment just the former case. If $t = \lambda_{y'}.t_1$ then we must prove that $(x\ y)\{z := u\}(\lambda_{y'}.t_1) =_\alpha \{(\!(x\ y)\!)z := (x\ y)u\}(x\ y)(\lambda y'.t_1)$. Firstly, we compare the variables y' and z according to the definition of the metasubstitution:

1. When $y' = z$ the metasubstitution is erased according to the definition (5) on both sides of the goal and we are done.
2. When $y' \neq z$ then the metasubstitutions on both sides of the goal 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 two sets are different. In fact, in the LHS the fresh name cannot belong to the set $fv\_nom(\lambda'_y.t_1) \cup fv\_nom(u) \cup \{z\}$, while the name of the RHS cannot belong to the set $fv\_nom((x\ y)\lambda'_y.t_1) \cup fv\_nom((x\ y)u) \cup \{(\!(x\ y)\!)z\}$. Let x_0 be a fresh name that is not in the set $fv\_nom(\lambda'_y.t_1) \cup fv\_nom(u) \cup \{z\}$, and x_1 a fresh name that is not in the set $fv\_nom((x\ y)\lambda'_y.t_1) \cup fv\_nom((x\ y)u) \cup \{(\!(x\ y)\!)z\}$. After the propagation of the metasubstitutions, we have to prove that $\lambda_{(\!(x\ y)\!)x0}.((x\ y)(\{z := u\}((y'\ x_0)t_1)) =_\alpha \lambda_{x_1}.(\{(\!(x\ y)\!)z := (x\ y)u\}(((\!(x\ y)\!)y')\ x_1)((x\ y)t_1))$. We proceed by comparing x_1 with $(\!(x\ y)\!)x_0$.
   (a) If $x_1 = (\!(x\ y)\!)x_0$ then we use the induction hypothesis to propagate the swap inside the metasubstitution in the LHS, and we get the goal $\{(\!(x\ y)\!)z := (x\ y)u\}((x\ y)((y'\ x_0)t_1)) =_\alpha \{(\!(x\ y)\!)z := (x\ y)u\}((((\!(x\ y)\!)y')\ ((\!(x\ y)\!)x_0))((x\ y)t_1))$ that is proved by the swap equivariance lemma *swap_equivariance*.
   (b) If $x_1 \neq (\!(x\ y)\!)x_0$ then by the rule *aeq_abs_diff* we have to prove that the variable $(\!(x\ y)\!)x_0$ is not in the set of free variables of the term $\{(\!(x\ y)\!)z := (x\ y)u\}(((\!(x\ y)\!)y'\ x_1)(x\ y)t_1)$ and that $(x\ y)(\{z := u\}((y'\ x_0)t_1)) =_\alpha (x_1\ ((\!(x\ y)\!)x_0))(\{(\!(x\ y)\!)z := (x\ y)u\}((((\!(x\ y)\!)y')\ x_1)((x\ y)t_1))$. 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 x_1 is different from $(\!(x\ y)\!)z$, and x_0 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 following two lemmas toghether with lemmas *m_subst_abs_eq* and *m_subst_sub_eq* are essential in simplifying the propagations of metasubstitution. They are presented here because they depend on lemma *swap_subst_rec_fun*.

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

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

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

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

Proof. The proof is by induction on the size of *t1*. The interesting cases are the abstraction and the explicit substitution. We focus on the former, *i.e.* $t1 = \lambda_z.t_1'$, whose initial goal is

$\{y := t_3\}(\{x := t_2\}(\lambda_z.t_1')) =_\alpha \{x := \{y := t_3\}t_2\}(\{y := t_3\}(\lambda_z.t_1'))$

assuming that $x \neq y$ and $x \notin fv\_nom(t_3)$. The induction hypothesis generated by this case states that the lemma holds for any term of the size of t_1', *i.e.* any term with the same size of the body of the abstraction. We start comparing z with x aiming to apply the definition of the metasubstitution on the LHS of the goal.

1. When $z = x$, the subterm $\{x := t_2\}\lambda_x.t_1'$ reduces to $\lambda_x.t_1'$ by lemma *m_subst_abs_eq*, and then the LHS reduces to $\{y := t_3\}\lambda_x.t_1'$. The RHS $\{x := \{y := t_3\}t_2\}\{y := t_3\}\lambda_x.t_1'$ also reduces to it because x does not occur free neither in $\lambda_x.t_1'$ nor in t_3, and we are done.
2. When $z \neq x$, then we compare y with z.
   (a) When $y = z$, the subterm $\{y := t_3\}(\lambda_z.t_1')$ can be simplified to $\lambda_z.t_1'$, by lemma *m_subst_abs_eq*. On the LHS, we propagate the internal metasubstitution over the abstraction taking a fresh name w not in the set $fv\_nom(\lambda_z.t_1') \cup fv\_nom(t_3) \cup fv\_nom(t_2) \cup \{x\}$, where the goal is $\{z := t_3\}(\lambda_w.(\{x := t_2\}(z\ w)t_1')) =_\alpha \{x := \{z := t_3\}t_2\}(\lambda_z.t_1')$. We proceed by comparing z and w:
      i. If $z = w$ then the current goal simplifies to
      $\{w := t_3\}(\lambda_w.(\{x := t_2\}t_1')) =_\alpha \{x := \{w := t_3\}t_2\}(\lambda_w.t_1')$
      We can propagate the metasubstitution on the RHS and there is no need for a fresh name since the variable w fullfil the condition required by lemma *m_subst_abs_neq*. We conclude with lemmas *aeq_m_subst_in* and *m_subst_notin*.
      ii. If $z \neq w$ then we can propagate the metasubstitutions on both sides of the goal taking w as the fresh name that fullfil the conditions of lemma *m_subst_abs_neq*. We proceed with *aeq_abs_same*, and conclude by the induction hypothesis.
   (b) If $y \neq z$ then we follow a similar strategy that avoids unnecessary generation of fresh names. In this way, we take a fresh w that is not in the set $fv\_nom(t_3) \cup fv\_nom(t_2) \cup fv\_nom(\lambda_z.t_1') \cup \{x\} \cup \{y\}$, and propagate the metasubstitution inside the abstraction resulting in the goal $\lambda_w.(\{y := t_3\}(\{x := t_2\}(z\ w)t_1')) =_\alpha \lambda_w.(\{x := \{y := t_3\}t_2\}(\{y := t_3\}(z\ w)t_1')$. We conclude by the induction hypothesis. □

4 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, 9].

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. Despite the fact that our definition of metasubstitution, called *subst_rec_fun*, performs a renaming with a fresh name whenever it is propagated inside a binding structure (either an abstraction or an explicit substitution in our case), we showed how to avoid unnecessary generation of fresh names that could result in a circular problem in the proofs. 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 publicly available at https://flaviomoura.info/files/msubst.v

As future work, we intend to get rid of the axiom *Eq_implies_equality*. The natural candidate for this would be the use of generalized rewriting, *i.e.* setoid rewriting, but it not clear whether generalized rewriting allows a rewrite step in a let expression. Another possibility is the implementation of the metasubstitution using recursors [22, 13]. In addition, we plan to integrate this formalization with another one related to the Z property[7] to prove confluence of calculi with explicit substitutions [20, 21], as well as other properties in the nominal framework [16].

References

[1] M. Abadi, L. Cardelli, P.-L. Curien & J.-J. Lévy (1991): *Explicit Substitutions*. *Journal of Functional Programming* 1(4), pp. 375–416, doi:10.1017/S0956796800000186.

[2] Beniamino Accattoli (2012): *An Abstract Factorization Theorem for Explicit Substitutions*, p. 16 pages. doi:10.4230/LIPICS.RTA.2012.6.

[3] Mauricio Ayala-Rincón, Flávio L.C. De Moura & Fairouz Kamareddine (2002): *Comparing Calculi of Explicit Substitutions with Eta-reduction*. *Electronic Notes in Theoretical Computer Science* 67, pp. 76–95, doi:10.1016/S1571-0661(04)80542-5.

[4] Mauricio Ayala-Rincón, Flávio L.C. De Moura & Fairouz Kamareddine (2005): *Comparing and Implementing Calculi of Explicit Substitutions with Eta-Reduction*. *Annals of Pure and Applied Logic* 134(1), pp. 5–41, doi:10.1016/j.apal.2004.06.009.

[5] H. P. Barendregt (1984): *The Lambda Calculus: Its Syntax and Semantics*, rev. ed edition. *Studies in Logic and the Foundations of Mathematics* v. 103, North-Holland ; Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co, Amsterdam ; New York : New York, N.Y.

[6] Zine-El-Abidine Benaissa, Daniel Briaud, Pierre Lescanne & Jocelyne Rouyer-Degli (1996): *$\Lambda\nu$, a Calculus of Explicit Substitutions Which Preserves Strong Normalisation*. *Journal of Functional Programming* 6(5), pp. 699–722, doi:10.1017/S0956796800001945.

[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] 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.

[9] 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.

[7] https://cicm-conference.org/2021/cicm.php?event=fmm&menu=general

[10] 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.

[11] R. David & B. Guillaume (2001): *A Lambda-Calculus with Explicit Weakening and Explicit Substitution*. *Mathematical Structures in Computer Science* 11(1), pp. 169–206, doi:10.1017/S0960129500003224.

[12] Murdoch J. Gabbay & Andrew M. Pitts (2002): *A New Approach to Abstract Syntax with Variable Binding*. *Formal Aspects of Computing* 13(3-5), pp. 341–363, doi:10.1007/s001650200016.

[13] Lorenzo Gheri & Andrei Popescu (2020): *A Formalized General Theory of Syntax with Bindings: Extended Version*. *Journal of Automated Reasoning* 64(4), pp. 641–675, doi:10.1007/s10817-019-09522-2.

[14] Bruno Guillaume (2000): *The λ s $_{e}$ -Calculus Does Not Preserve Strong Normalisation*. *Journal of Functional Programming* 10(4), pp. 321–325, doi:10.1017/S0956796800003695.

[15] Fairouz Kamareddine & Alejandro Ríos (1997): *Extending a λ-Calculus with Explicit Substitution Which Preserves Strong Normalisation into a Confluent Calculus on Open Terms*. *Journal of Functional Programming* 7(4), pp. 395–420, doi:10.1017/S0956796897002785.

[16] 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.

[17] Delia Kesner (2009): *A Theory of Explicit Substitutions with Safe and Full Composition*. *Logical Methods in Computer Science* Volume 5, Issue 3, p. 816, doi:10.2168/LMCS-5(3:1)2009.

[18] Paul-André Mellies (1995): *Typed λ-Calculi with Explicit Substitutions May Not Terminate*. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Mariangiola Dezani-Ciancaglini & Gordon Plotkin, editors: *Typed Lambda Calculi and Applications*, 902, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 328–334, doi:10.1007/BFb0014062.

[19] 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.

[20] 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.

[21] 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.

[22] Andrei Popescu (2023): *Nominal Recursors as Epi-Recursors*. arXiv:2301.00894.

[23] K. H. Rose, R. Bloo & F. Lang (2011): *On Explicit Substitution With Names*. *J Autom Reasoning* 49(2), pp. 275–300, doi:10.1007/s10817-011-9222-5.

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