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 behavior of the metasubstitution operation. In this study, we augment the λ-calculus's grammar with an uninterpreted explicit substitution operation. 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

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[4] where variables use names. This approach contrasts with the use of De Bruijn indexes detailed in De Bruijn's landmark paper on λ-calculus notation[3]. 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}.
```

Variable renaming is done via name-swapping defined as follows:

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

and the corresponding Coq definition:

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

The next step is to extend the variable renaming operation to terms, which in our case corresponds to λ-terms augmented with an explicit substitution operation. We use *n_sexp* to denote the set of nominal expressions equipped with an explicit substitution operator, which, for simplicity, we will refer to as just "terms", and the corresponding grammar is outlined below:

```
Inductive n_sexp : Set :=
 | n_var (x:atom)
 | n_abs (x:atom) (t:n_sexp)
 | n_app (t1:n_sexp) (t2:n_sexp)
```

Submitted to:
FROM 2023

```
| n_sub (t1:n_sexp) (x:atom) (t2:n_sexp).
```

where *n_var* is the constructor for variables, *n_abs* for abstractions, *n_app* for applications and *n_sub* for the explicit substitution. Explicit substitution calculi 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[5]. The *size* of terms 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 follows:

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

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

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

The *swap* function preserves the size of terms, as stated by the following lemma: `Lemma swap_size_eq : ∀ x y t, size (swap x y t) = size t.`

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

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

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

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

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

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

```
Inductive aeq : n_sexp → n_sexp → Prop :=
| aeq_var : ∀ x, aeq (n_var x) (n_var x)
| aeq_abs_same : ∀ x t1 t2, aeq t1 t2 → aeq (n_abs x t1)(n_abs x t2)
| aeq_abs_diff : ∀ x y t1 t2, x ≠ y → x `notin` fv_nom t2 → aeq t1 (swap y x t2) → aeq (n_abs x t1)
(n_abs y t2)
| aeq_app : ∀ t1 t2 t1' t2', aeq t1 t1' → aeq t2 t2' → aeq (n_app t1 t2) (n_app t1' t2')
| aeq_sub_same : ∀ t1 t2 t1' t2' x, aeq t1 t1' → aeq t2 t2' → aeq (n_sub t1 x t2) (n_sub t1' x t2')
| aeq_sub_diff : ∀ t1 t2 t1' t2' x y, aeq t2 t2' → x ≠ y → x `notin` fv_nom t1' → aeq t1 (swap y x t1') →
aeq (n_sub t1 x t2) (n_sub t1' y t2').
```

In what follows, we use a infix notation for α-equivalence in the Coq code: we write *t =a u* instead of *aeq t u*.

The above notion defines an equivalence relation over the set *n_sexp* of nominal expressions with explicit substitutions, *i.e.* the *aeq* relation is reflexive, symmetric and transitive.

Informally, two terms are α-equivalent if they differ only by the names of the bound variables. Therefore, α-equivalent terms have the same size, and the same set of free variables:

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

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

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

There are several interesting auxiliary properties that need to be proved before achieving the substitution lemma. In what follows, we refer only to the tricky or challenging ones, but the interested reader can have a detailed look in the source files. Note that, swaps are introduced in proofs by the rules (aeq_abs_diff) and (aeq_sub_diff). As we will see, the proof steps involving these rules are trick because a naïve strategy can easily result in a proofless branch. so that one can establish the α-equivalence between two abstractions or two explicit substitutions with different binders. The following proposition states when two swaps with a common name collapse, and it is used in the transitivity proof of *aeq*:

```
Lemma aeq_swap_swap: ∀ 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).
```

2.1 The metasubstitution operation of the λ-calculus

The main operation of the λ-calculus is the β-reduction that express how to evaluate a function applied to a given argument:

$$(\lambda_x.t)\ u \to_\beta t\{x/u\}$$

In a less formal context, the concept of β-reduction means that the result of evaluating the function $(\lambda_x.t)$ with argument u is obtained by substituting u for the free ocurrences of the variable x in t. Moreover, it is a capture free substitution in the sense that no free variable becomes bound after the substitution. This operation is in the meta level because it is outside the grammar of the λ-calculus, and that's why it is called metasubstitution. As a metaoperation, its definition usually comes with a degree of informality. For instance, Barendregt[1] defines it as follows:

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

where it is assumed the so called "Barendregt's variable convention": if $t_1, t_2, \ldots, t_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

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

Definition 2.1 *Let t and u be terms, and x a variable. The result of substituting u for the free ocurrences of x in t, written $t\{x/u\}$ is defined as follows:*

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

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[1], 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 $t\{x/u\}$. 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
```

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

```
  | n_abs y t1 ⇒ if (x == y) then t else let (z,_) :=
    atom_fresh (fv_nom u `union` fv_nom t `union` {{x}}) in n_abs z (subst_rec_fun (swap y z t1) u x)
  | n_app t1 t2 ⇒ n_app (subst_rec_fun t1 u x) (subst_rec_fun t2 u x)
  | n_sub t1 y t2 ⇒ if (x == y) then n_sub t1 y (subst_rec_fun t2 u x) else let (z,_) :=
    atom_fresh (fv_nom u `union` fv_nom t `union` {{x}}) in
    n_sub (subst_rec_fun (swap y z t1) u x) z (subst_rec_fun t2 u x)
end.
```

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

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

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

```
Lemma m_subst_notin: ∀ t u x, x `notin` fv_nom t → [x := u]t =a t.
```

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

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

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

1. $z = y$: In this case, we have to prove that $\lambda_z.((z\ z)t_1)\{x/u\} =_\alpha \lambda_z.t_1$. By the rule *aeq_abs_same* we get $((z\ z)t_1)\{x/u\} =_\alpha t_1$, 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 $((z\ y)t_1)\{x/u\} =_\alpha (z\ y)t_1$ 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. □

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 $t\{x/u\} =_\alpha t'\{x/u'\}$, 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 $t\{x/u\} =_\alpha t\{x/u'\}, \forall x, t, u, u'$; secondly, we prove that if $t =_\alpha t'$ then $t\{x/u\} =_\alpha t'\{x/u\}, \forall x, t, t', u$. 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*: $\forall$ *t u u' x*, *u =a u'* $\rightarrow$ ([*x* := *u*] *t*) =*a* ([*x* := *u'*] *t*).

Proof. The proof is done by induction on the size of the term *t*. The interesting case is when *t* is an abstraction, *i.e.* $t = \lambda_y.t_1$. We need to prove that $(\lambda_y.t_1)\{x/u\} =_\alpha (\lambda_y.t_1)\{x/u'\}$. 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 $\lambda_{x_0}.((y\ x_0)t_1)\{x/u\} =_\alpha \lambda_{x_1}.((y\ x_1)t_1)\{x/u'\}$, 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 $((y\ x_0)t_1)\{x/u\} =_\alpha (x_0\ x_1)(((y\ x_1)t_1)\{x/u'\})$. 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 2.2 *Let t, u be terms, and x, y, z variables. Then* $(y\ z)(t\{x/u\}) =_\alpha ((y\ z)t)\{(\!(y\ z)\!)x/(y\ z)u\}$.

whose corresponding Coq version is given by:

Lemma *swap_m_subst*: $\forall$ *t u x y z*, *swap y z* ([*x* := *u*]*t*) =*a* ([(*swap_var 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 $(y\ z)((\lambda_w.t_1)\{x/u\}) =_\alpha ((y\ z)\lambda_w.t_1)\{(\!(y\ z)\!)x/(y\ z)u\}$. 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 $\lambda_{(\!(y\ z)\!)w'}.(y\ z)((w\ w')t_1\{x/u\}) =_\alpha (\lambda_{(\!(y\ z)\!)w}.(y\ z)t_1)\{(\!(y\ z)\!)x/(y\ z)u\}$, where $w' \notin fv(\lambda_w.t_1) \cup fv(u) \cup \{x\}$. Now we can 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 $\lambda_{(\!(y\ z)\!)w'}.(y\ z)((w\ w')t_1\{x/u\}) =_\alpha \lambda_{w''}.((w''\ (\!(y\ z)\!)w)((y\ z)t_1))\{(\!(y\ z)\!)x/(y\ z)u\}$. 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 $(y\ z)(((w\ w')t_1)\{x/u\}) =_\alpha ((w''\ (\!(y\ z)\!)w)((y\ z)t_1))\{(\!(y\ z)\!)x/(y\ z)u\}$. Axiom *Eq_implies_equality*: $\forall$ *s s'*: *atoms*, *s* [=] *s'* $\rightarrow$ *s* = *s'*.

Lemma *aeq_m_subst_in*: $\forall$ *t u u' x*, *u =a u'* $\rightarrow$ ([*x* := *u*] *t*) =*a* ([*x* := *u'*] *t*).

Lemma *aeq_abs_notin*: $\forall$ *t1 t2 x y*, *x* $\neq$ *y* $\rightarrow$ *n_abs x t1 =a n_abs y t2* $\rightarrow$ *x* ‘*notin*‘ *fv_nom t2*.

Lemma *aeq_sub_notin*: $\forall$ *t1 t1' t2 t2' x y*, *x* $\neq$ *y* $\rightarrow$ *n_sub t1 x t2 =a n_sub t1' y t2'* $\rightarrow$ *x* ‘*notin*‘ *fv_nom t1'*.

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

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'*).

The following lemma states that a swap can be propagated inside the metasubstitution resulting in an α-equivalent term.

Lemma *swap_subst_rec_fun*: $\forall$ *x y z t u*, *swap x y* (*subst_rec_fun t u z*) =*a subst_rec_fun* (*swap x y t*) (*swap x y u*) (*swap_var x y z*).

Firstly, we compare x and y which gives a trivial case when they are the same. In this way, we can assume in the rest of the proof that x and y are different from each other. The proof proceeds by induction on the size of the term t. The tricky case is the abstraction and substitution cases.

The following lemmas state, respectively, what hapens when the variable in the meta-substitution is equal or different from the one in the abstraction. When it is equal, the meta-substitution is irrelevant. When they are different, we take a new variable that does not occur freely in the substituted term in the meta-substitution nor in the abstraction and is not the variable in the meta-substitution, and the abstraction of this new variable using the meta-substitution of the swap of the former variable in the meta-substitution is alpha-equivalent to the original meta-substitution of the abstraction. The proofs were made using the definition of the meta-substitution, each case being respectively each one in the definition. Lemma $m\_subst\_abs\_eq : \forall\ u\ x\ t, [x := u](n\_abs\ x\ t) = n\_abs\ x\ t$.

Lemma $m\_subst\_abs\_neq : \forall\ t\ u\ x\ y\ z, x \neq y \rightarrow z\ `notin`\ fv\_nom\ u\ `union`\ fv\_nom\ (n\_abs\ y\ t)\ `union`\ \{\{x\}\} \rightarrow [x := u](n\_abs\ y\ t) =a\ n\_abs\ z\ ([x := u](swap\ y\ z\ t))$.

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

Lemma $m\_subst\_sub\_eq : \forall\ u\ x\ t1\ t2, [x := u](n\_sub\ t1\ x\ t2) = n\_sub\ t1\ x\ ([x := u]\ t2)$.

Lemma $m\_subst\_sub\_neq : \forall\ t1\ t2\ u\ x\ y\ z, x \neq y \rightarrow z\ `notin`\ fv\_nom\ u\ `union`\ fv\_nom\ (n\_sub\ t1\ y\ t2)\ `union`\ \{\{x\}\} \rightarrow [x := u](n\_sub\ t1\ y\ t2) =a\ n\_sub\ ([x := u](swap\ y\ z\ t1))\ z\ ([x := u]t2)$.

3 The substitution lemma for the metasubstitution

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

Lemma m_subst_lemma: $\forall\ e1\ e2\ x\ e3\ y, x \neq y \rightarrow x\ `notin`\ (fv\_nom\ e3) \rightarrow ([y := e3]([x := e2]e1)) =a ([x := ([y := e3]e2)]([y := e3]e1))$.

We procced by case analisys on the structure of $e1$. The cases in between square brackets below mean that in the first case, $e1$ is a variable named z, in the second case $e1$ is an abstraction of the form $\lambda z.e11$, in the third case $e1$ is an application of the form $(e11\ e12)$, and finally in the fourth case $e1$ is an explicit substitution of the form $e11\ \langle\ z := e12\ \rangle$. The variable case was proved using the auxiliary lemmas on the equality and inequality of the meta-substitution applied to variables. It was also necessary to compare the variable in the meta-substitution and the variable one in each case of this proof. In the abstraction case, we used a similar approach, comparing the variable in the meta substitution and the one in the abstraction. When usion the using the auxiliary lemmas on the equality and inequality of the meta-substitution applied to abstractions, it was necessary to create new variables in each use of

the inequality. This is due to the atempt of removing the abstraction from inside the meta-substitution. The case of the application is quite simple to solve. It consisted of applying the auxiliary lemma of removing the application from inside the meta-substitution. In the explicit substitution case, we used the same approach used in the abstraction for the left side and the same as the application for the right side of the substitution. It consisted of comparing the variable in the meta substitution and the one in the substitution. We used the auxiliary lemmas on the equality and inequality of the meta-substitution applied to explicit substitutions and it was necessary to create new variables in each use of the inequality. This is due to the atempt of removing the explicit substitution from inside the meta-substitution. When tis removal was made, the proof consisted in proving a similar case for the abstraction in the left side of the explicit substitution and the one similar to the application was used for the right part of it.

References

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

[2] 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.

[3] N. G. de Bruijn (1972): *Lambda Calculus Notation With Nameless Dummies, a Tool for Automatic Formula Manipulation, With Application To the Church-Rosser Theorem*. *Indagationes Mathematicae (Proceedings)* 75(5), pp. 381–392, doi:10.1016/1385-7258(72)90034-0.

[4] 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.

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

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