A formalized extension of the substitution lemma in Coq

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

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

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

TBD In this work, we present a formalization of an extension of the substitution lemma[1] with an explicit substitution operator in the Coq proof assistant[6]. The substitution lemma is an important result concerning the composition of the substitution operation, and is usually presented as follows: if x does not occur in the set of free variables of the term v then $t\{x/u\}\{y/v\} =_\alpha t\{y/v\}\{x/u\{y/v\}\}$. This is a well known result already formalized several times in the context of the λ-calculus [2].

In the context of the λ-calculus with explicit substitutions its formalization is not straightforward because, in addition to the metasubstitution operation, there is the explicit substitution operator. Our formalization is done in a nominal setting that uses the MetaLib 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.

- This paper is written from a Coq script file.
- include [2]
- repository

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}.
```

Submitted to:
FROM 2023

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

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

\noindent 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)
 | 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 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 defined 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 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).
```

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 replacing all free ocurrences of the variable x by u in t. 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.

We define the metasubstitution as a recursive function 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. 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 $t\{x/u\}$.

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

```
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).
```

```
    Search remove. Search remove.
```

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

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

```
Lemma aeq_abs_notin: ∀ t1 t2 x y, x ≠ y → n_abs x t1 =a n_abs y t2 → x `notin` fv_nom t2.
```

```
Lemma aeq_sub_notin: ∀ t1 t1' t2 t2' x y, x ≠ y → n_sub t1 x t2 =a n_sub t1' y t2' → x `notin` fv_nom
t1'.
```

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

```
Corollary aeq_m_subst_eq: ∀ t t' u u' x, t =a t' → u =a u' → ([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: ∀ 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 : ∀ u x t, [x := u](n_abs x t) = n_abs x t.
```

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

The following lemmas 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.

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

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

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

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*: ∀ *e1 e2 x e3 y*, *x* ≠ *y* → *x* ‘*notin*‘ (*fv_nom e3*) → ([*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 λ*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* ⟨ *z* := *e12* ⟩. 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.