

# A formalized extension of the substitution lemma in Coq

Flávio L. C. de Moura

Departamento de Ciência da Computação  
Universidade de Brasília, Brasília, Brazil  
flaviomoura@unb.br

Maria Julia

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

TBD

## 1 Introduction

In this work, we are interested in formalizing an extension of the substitution lemma[1] in the Coq proof assistant. 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  $t\{x/u\}\{y/v\} = t\{y/v\}\{x/u\{y/v\}\}$ . This is a well known result already formalized several times in the context of the  $\lambda$ -calculus [?].

In the context of the  $\lambda$ -calculus with explicit substitutions its formalization is not straightforward because, in addition to the metasubstitution, there is the explicit substitution operation of the calculus.

## 2 A syntactic extension of the $\lambda$ -calculus

We consider a generic signature with the following constructors:

```
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 operation.

The notion of  $\alpha$ -equivalence is defined as follows:

```
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) →
```

$aeq (n\_abs\ x\ t1) (n\_abs\ y\ t2)$   
 $| aeq\_app : \forall\ t1\ t2\ t1'\ t2',$   
 $aeq\ t1\ t1' \rightarrow aeq\ t2\ t2' \rightarrow$   
 $aeq\ (n\_app\ t1\ t2)\ (n\_app\ t1'\ t2')$   
 $| aeq\_sub\_same : \forall\ t1\ t2\ t1'\ t2'\ x,$   
 $aeq\ t1\ t1' \rightarrow aeq\ t2\ t2' \rightarrow$   
 $aeq\ (n\_sub\ t1\ x\ t2)\ (n\_sub\ t1'\ x\ t2')$   
 $| aeq\_sub\_diff : \forall\ t1\ t2\ t1'\ t2'\ x\ y,$   
 $aeq\ t2\ t2' \rightarrow x \neq y \rightarrow x\ 'notin'\ fv\_nom\ t1' \rightarrow$   
 $aeq\ t1\ (swap\ y\ x\ t1') \rightarrow$   
 $aeq\ (n\_sub\ t1\ x\ t2)\ (n\_sub\ t1'\ y\ t2').$

where ...

**Lemma**  $aeq\_fv\_nom : \forall\ t1\ t2, t1 =_a t2 \rightarrow fv\_nom\ t1 [=] fv\_nom\ t2.$

**Lemma**  $aeq\_swap1 : \forall\ t1\ t2\ x\ y, t1 =_a t2 \rightarrow (swap\ x\ y\ t1) =_a (swap\ x\ y\ t2).$

**Lemma**  $aeq\_swap2 : \forall\ t1\ t2\ x\ y, (swap\ x\ y\ t1) =_a (swap\ x\ y\ t2) \rightarrow t1 =_a t2.$

**Corollary**  $aeq\_swap : \forall\ t1\ t2\ x\ y, t1 =_a t2 \leftrightarrow (swap\ x\ y\ t1) =_a (swap\ x\ y\ t2).$

**Lemma**  $aeq\_abs : \forall\ t\ x\ y, y\ 'notin'\ fv\_nom\ t \rightarrow (n\_abs\ y\ (swap\ x\ y\ t)) =_a (n\_abs\ x\ t).$

**Lemma**  $swap\_reduction : \forall\ t\ x\ y,$   
 $x\ 'notin'\ fv\_nom\ t \rightarrow y\ 'notin'\ fv\_nom\ t \rightarrow (swap\ x\ y\ t) =_a t.$

**Lemma**  $aeq\_swap\_swap : \forall\ t\ x\ y\ z, z\ 'notin'\ fv\_nom\ t \rightarrow x\ 'notin'\ fv\_nom\ t \rightarrow (swap\ z\ x\ (swap\ x\ y\ t)) =_a (swap\ z\ y\ t).$

**Lemma**  $aeq\_sym : \forall\ t1\ t2, t1 =_a t2 \rightarrow t2 =_a t1.$

**Lemma**  $aeq\_trans : \forall\ t1\ t2\ t3, t1 =_a t2 \rightarrow t2 =_a t3 \rightarrow t1 =_a t3.$

**Lemma**  $swap\_comp : \forall\ x\ y\ z\ t, x\ 'notin'\ fv\_nom\ t \rightarrow y\ 'notin'\ fv\_nom\ t \rightarrow$   
 $(swap\ x\ y\ (swap\ y\ z\ t)) =_a (swap\ x\ z\ t).$

**Require Import** *Setoid Morphisms.*

**Instance** *Equivalence\_aeq : Equivalence aeq.*

**Lemma**  $aeq\_sub : \forall\ t1\ t2\ x\ y, y\ 'notin'\ fv\_nom\ t1 \rightarrow (n\_sub\ (swap\ x\ y\ t1)\ y\ t2) =_a (n\_sub\ t1\ x\ t2).$

## 2.1 Capture-avoiding substitution

We need to use size to define capture avoiding substitution. Because we sometimes swap the name of the bound variable, this function is *not* structurally recursive. So, we add an extra argument to the function that decreases with each recursive call.

Fixpoint subst\_rec (n:nat) (t:n\_sexp) (u :n\_sexp) (x:atom) : n\_sexp := match n with | 0 => t | S m  
=> 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' <sup>1</sup>) in n\_abs z (subst\_rec m (swap y z t1) u x) | n\_app  
t1 t2 => n\_app (subst\_rec m t1 u x) (subst\_rec m t2 u x) | n\_sub t1 y t2 => if (x == y) then n\_sub

---

<sup>1</sup><sub>x</sub>

t1 y (subst\_rec m t2 u x) else let (z,\_) := atom\_fresh (fv\_nom u 'union' fv\_nom t 'union' <sup>2</sup>) in n\_sub (subst\_rec m (swap y z t1) u x) z (subst\_rec m t2 u x) end end.

Require Import Recdef.

```
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 t1 'union' {{x}} 'union' {{y}}) 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 t1 'union' {{x}} 'union' {{y}}) in
      n_sub (subst_rec_fun (swap y z t1) u x) z (subst_rec_fun t2 u x)
  end.
```

The definitions subst\_rec and subst\_rec\_fun are alpha-equivalent. Theorem subst\_rec\_fun\_equiv: forall t u x, (subst\_rec (size t) t u x) =a (subst\_rec\_fun t u x). Proof. intros t u x. functional induction (subst\_rec\_fun t u x).

- simpl. rewrite e0. apply aeq\_refl.
- simpl. rewrite e0. apply aeq\_refl.
- simpl. rewrite e0. apply aeq\_refl.
- simpl. rewrite e0. destruct (atom\_fresh (Metatheory.union (fv\_nom u) (Metatheory.union (remove y (fv\_nom t1)) (singleton x)))). admit.
- simpl. admit.
- simpl. rewrite e0. admit.
- simpl. rewrite e0.

Admitted.

Require Import EquivDec. Generalizable Variable A.

Definition equiv\_decb '{EqDec A} (x y : A) : bool := if x == y then true else false.

Definition nequiv\_decb '{EqDec A} (x y : A) : bool := negb (equiv\_decb x y).

Infix "==" := equiv\_decb (no associativity, at level 70). Infix "<>" := nequiv\_decb (no associativity, at level 70).

Parameter Inb : atom -> atoms -> bool. Definition equalb s s' := forall a, Inb

Function subst\_rec\_b (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 if (Inb y (fv\_nom u)) then let (z,\_) := atom\_fresh (fv\_nom u 'union' fv\_nom t 'union' <sup>3</sup>) in n\_abs z (subst\_rec\_b (swap y z t1) u

---

<sup>2</sup>x

<sup>3</sup>x

x) else n\_abs y (subst\_rec\_b t1 u x) | n\_app t1 t2 => n\_app (subst\_rec\_b t1 u x) (subst\_rec\_b t2 u x) | n\_sub t1 y t2 => if (x == y) then n\_sub t1 y (subst\_rec\_b t2 u x) else if (Inb y (fv\_nom u)) then let (z,\_) := atom\_fresh (fv\_nom u 'union' fv\_nom t 'union' <sup>4</sup>) in n\_sub (subst\_rec\_b (swap y z t1) u x) z (subst\_rec\_b t2 u x) else n\_sub (subst\_rec\_b t1 u x) y (subst\_rec\_b t2 u x) end. Proof.

- intros. simpl. rewrite swap\_size\_eq. auto.
- intros. simpl. lia.
- intros. simpl. lia.
- intros. simpl. lia.
- intros. simpl. lia.
- intros. simpl. rewrite swap\_size\_eq. lia.

Defined.

Our real substitution function uses the size of the size of the term as that extra argument.

**Definition** *m\_subst* (u : n\_sexp) (x:atom) (t:n\_sexp) :=

*subst\_rec\_fun* t u x.

**Notation** "[ x := u ] t" := (m\_subst u x t) (at level 60).

**Lemma** *m\_subst\_var\_eq* :  $\forall u x,$

$[x := u](n\_var\ x) = u.$

**Lemma** *m\_subst\_var\_neq* :  $\forall u x y, x \neq y \rightarrow$

$[y := u](n\_var\ x) = n\_var\ x.$

**Lemma** *fv\_nom\_remove*:  $\forall t u x y, y \text{ 'notin' } fv\_nom\ u \rightarrow y \text{ 'notin' } remove\ x\ (fv\_nom\ t) \rightarrow y \text{ 'notin' } fv\_nom\ ([x := u]\ t).$

**Search** *remove*. **Search** *remove*.

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

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

**Search** *eq\_dec*.

**Lemma** *swap\_subst\_rec\_fun*:  $\forall x\ y\ z\ t\ u, swap\ x\ y\ (subst\_rec\_fun\ t\ u\ z) = a\ subst\_rec\_fun\ (swap\ x\ y\ t)\ (swap\ x\ y\ u)\ (swap\_var\ x\ y\ z).$

**Lemma** *m\_subst\_abs* :  $\forall u\ x\ y\ t, m\_subst\ u\ x\ (n\_abs\ y\ t) = a$

**if** (x == y) **then** (n\_abs y t)

**else** **let** (z,\_) := atom\_fresh (fv\_nom u 'union' fv\_nom (n\_abs y t) 'union' {{x}}) **in**  
n\_abs z (m\_subst u x (swap y z t)).

**Search** *aeq\_swap\_swap*. **Search** *swap*. **Search** *aeq*. **Search** *n\_abs*. **Search** *remove*.

**Lemma** *m\_subst\_abs\_eq* :  $\forall u\ x\ t, [x := u](n\_abs\ x\ t) = n\_abs\ x\ t.$

**Corollary** *m\_subst\_abs\_neq* :  $\forall u\ x\ y\ z\ t, x \neq y \rightarrow z \text{ 'notin' } (fv\_nom\ u \text{ 'union' } fv\_nom\ (n\_abs\ y\ t) \text{ 'union' } {{x}}) \rightarrow [x := u](n\_abs\ y\ t) = a\ n\_abs\ z\ ([x := u](swap\ y\ z\ t)).$

**Search** *n\_abs*.

---

<sup>4</sup> x

**Lemma  $m\_subst\_abs\_diff$**  :  $\forall t u x y, x \neq y \rightarrow x \text{ 'notin' } (remove\ y\ (fv\_nom\ t)) \rightarrow [x := u](n\_abs\ y\ t) = n\_abs\ y\ t.$

**Search  $n\_abs$ .**

**Lemma  $m\_subst\_notin$**  :  $\forall t u x, x \text{ 'notin' } fv\_nom\ t \rightarrow [x := u]t = a\ t.$

**Search  $n\_sub$ . Search  $n\_sub$ .**

### 3 The substitution lemma for the metasubstitution

In the pure  $\lambda$ -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  $m\_subst\_notin\_m\_subst$** :  $\forall t u v x y, y \text{ 'notin' } fv\_nom\ t \rightarrow [y := v]([x := u]t) = [x := [y := v]u]t.$

**Lemma  $m\_subst\_lemma$** :  $\forall e1\ e2\ x\ e3\ y, x \neq y \rightarrow x \text{ 'notin' } (fv\_nom\ e3) \rightarrow ([y := e3]([x := e2]e1)) = a\ ([x := ([y := e3]e2)]([y := e3]e1)).$

We proceed by functional induction on the structure of `subst_rec_fun`, the definition of the substitution. The induction splits the proof in seven cases: two cases concern variables, the next two concern abstractions, the next case concerns the application and the last two concern the explicit substitution. The first case is about the variable. It considers that there are two variables,  $x$  and  $y$  and they differ from one another. When we rewrite the lemmas concerning equality and negation on variables substitution, we have two cases. If we only have these two variables, we can use the equality lemma to find that both sides of the proof are equal and finish it using reflexivity and in the second case assumptions are used to finish the proof. The second case is also about variables. In it, we consider a third variable,  $z$ , meaning that each variable is different from the other. In the former case, we had that  $x = y$ . To unfold the cases in this proof, we need to destruct one variable as another. We chose to do  $x == z$ . This splits the proof in two cases. In the first case, we have that  $x = z$ . To expand this case, we use the lemma  $m\_subst\_notin$  as an auxiliary lemma. It is added as an hypothesis, using the specialization tactics to match the last case in that hypothesis to the proof we want. The case of application is solved by using the auxiliary lemmas on application. First, it is rewritten so that the substitution is made inside the application, instead of on it. The same lemma is applied multiple times to make sure nothing can be replaced anymore. This leads to a case that can be solved using the standard library lemmas.

## 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](https://doi.org/10.1016/j.entcs.2007.01.018).