# 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 insterested 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 $t\{x/u\}\{y/v\} =_\alpha t\{y/v\}\{x/u\{y/v\}\}$. This is a well known result already formalized several times in the context of the $\lambda$-calculus [2].

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) →
```

Submitted to:
FROM 2023

```
    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').
```

where ...

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

Lemma aeq_swap1: ∀ t1 t2 x y, t1 =a t2 → (swap x y t1) =a (swap x y t2).

Lemma aeq_swap2: ∀ t1 t2 x y, (swap x y t1) =a (swap x y t2) → t1 =a t2.

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

Lemma aeq_abs: ∀ t x y, y `notin` fv_nom t → (n_abs y (swap x y t)) =a (n_abs x t).

Lemma swap_reduction: ∀ t x y,
    x `notin` fv_nom t → y `notin` fv_nom t → (swap x y t) =a t.

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).

Lemma aeq_sym: ∀ t1 t2, t1 =a t2 → t2 =a t1.

Lemma aeq_trans: ∀ t1 t2 t3, t1 =a t2 → t2 =a t3 → t1 =a t3.

Require Import Setoid Morphisms.

Instance Equivalence_aeq: Equivalence aeq.

Lemma aeq_sub: ∀ t1 t2 x y, y `notin` fv_nom t1 → (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` [1]) 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 t1 y (subst_rec m t2 u x) else let (z,_) := atom_fresh (fv_nom u `union` fv_nom t `union` [2]) in n_sub (subst_rec m (swap y z t1) u x) z (subst_rec m t2 u x) end end.

---

[1] x

[2] x

```
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 "==b" := equiv_decb (no associativity, at level 70). Infix "<>b" := 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' [3]) in n_abs z (subst_rec_b (swap y z t1) u 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)

[3] 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‘ [4]) 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 : ∀ u x,
    [x := u](n_var x) = u.
```

```
Lemma m_subst_var_neq : ∀ u x y, x ≠ y →
    [y := u](n_var x) = n_var x.
```

```
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.
```

```
Lemma m_subst_app: ∀ t1 t2 u x, [x := u](n_app t1 t2) = n_app ([x := u]t1) ([x := u]t2).
```

```
Lemma aeq_m_subst: ∀ t t’ u u’ x, t =a t’ → u =a u’ → ([x := u] t) =a ([x := u’] t’).
  Search eq_dec.
```

```
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).
Lemma m_subst_abs : ∀ 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 : ∀ u x t, [x := u](n_abs x t) = n_abs x t.
```

```
Corollary m_subst_abs_neq : ∀ u x y z t, 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)).
  Search n_abs.
```

```
Lemma m_subst_abs_diff : ∀ t u x y, x ≠ y → x ‘notin‘ (remove y (fv_nom t)) → [x := u](n_abs y t) =
n_abs y t.
```

[4] x

Search *n_abs*.

Lemma *m_subst_notin* : $\forall$ *t u x*, *x* ‘*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* ‘*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*, $x \neq y \rightarrow x$ ‘*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 aplication, 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.