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]

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 instead of De Bruijn indexes[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:

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

Submitted to:
FROM 2023

\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 given by the following inductive grammar:

```
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. Explicit substitution calculi are formalisms that decompose the metasubstitution operation into more atomic steps behaving as a bridge between the λ-calculus and its implementations. In other words, explicit substitution calculi "allow a better understanding of the execution models of higher-order languages"[5]. 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 key point of this approach is that the swap opearation ...

```
Lemma shuffle_swap' : ∀ w y n z,
    w ≠ z → y ≠ z →
    (swap w y (swap y z n)) = (swap z w (swap y w n)).

Lemma swap_var_equivariance : ∀ v x y z w,
    swap_var x y (swap_var z w v) =
    swap_var (swap_var x y z) (swap_var x y w) (swap_var x y v).
```

The notion of α-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 : ∀ 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_same_abs: ∀ x t1 t2, n_abs x t1 =a n_abs x t2 → t1 =a t2.

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

Lemma aeq_same_sub: ∀ x t1 t1' t2 t2', (n_sub t1 x t2) =a (n_sub t1' x t2') → t1 =a t1' ∧ t2 =a t2'.

Lemma aeq_diff_sub: ∀ x y t1 t1' t2 t2', (n_sub t1 x t2) =a (n_sub t1' y t2') → t1 =a (swap x y t1') ∧ t2
=a t2'.

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.

```
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 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 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.

[1] x
[2] x

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) | 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.
```

The behaviour of free variables in a metasubstitution.

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

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

```
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_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.
```

[3] x

[4] x

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

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

Corollary *aeq_m_subst_eq*: $\forall$ *t t' u u' x*, *t =a t'* $\to$ *u =a u'* $\to$ ([*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.

Lemma *m_subst_abs*: $\forall$ *t1 u x y*, [*x* := *u*](*n_abs y t1*) =*a*
 if (*x* == *y*) then (*n_abs y t1*)
 else let (*z*,_) := *atom_fresh* (*fv_nom u* '*union*' *fv_nom* (*n_abs y t1*) '*union*' {{*x*}}) in *n_abs z* (*subst_rec_fun* (*swap y z t1*) *u x*).

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*, $x \neq y \to$ *z* '*notin*' *fv_nom u* '*union*' *fv_nom* (*n_abs y t*) '*union*' {{*x*}} $\to$ [*x* := *u*](*n_abs y t*) =*a n_abs z* ([*x* := *u*](*swap y z t*)).

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_sub*: $\forall$ *t1 t2 u x y*, [*x* := *u*](*n_sub t1 y t2*) =*a*
 if (*x* == *y*) then (*n_sub t1 y* ([*x* := *u*]*t2*))
 else let (*z*,_) := *atom_fresh* (*fv_nom u* '*union*' *fv_nom* (*n_sub t1 y t2*) '*union*' {{*x*}}) in
 n_sub ([*x* := *u*](*swap y z t1*)) *z* ([*x* := *u*]*t2*).

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*, $x \neq y \to$ *z* '*notin*' *fv_nom u* '*union*' *fv_nom* (*n_sub t1 y t2*) '*union*' {{*x*}} $\to$ [*x* := *u*](*n_sub t1 y t2*) =*a n_sub* ([*x* := *u*](*swap y z t1*)) *z* ([*x* := *u*]*t2*).

Lemma *m_subst_lemma*: $\forall$ *e1 e2 x e3 y*, $x \neq y \to$ *x* '*notin*' (*fv_nom e3*) $\to$
 ([*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* $\langle$ *z* := *e12* $\rangle$.

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.