

A Formalization of an extension of the substitution lemma in Coq

Flávio L. C. de Moura

Maria Julia

April 13, 2023

1 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: $\forall e1\ e2\ x\ e3\ y, x \neq y \rightarrow x \text{ 'notin' } (fv_nom\ e3) \rightarrow ([y := e3]([x := e2]e1)) =_a ([x := ([y := e3]e2)]([y := e3]e1)).$

Proof.

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. `intros e1 e2 x. functional induction (subst_rec_fun e1 e2 x).`

The first case is about the variable. It considers that there are two variables, x and y and they differ from one another. `- intros e3 y XY IH. rewrite m_subst_var_eq. rewrite m_subst_var_neq.`

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. `+ rewrite m_subst_var_eq. apply aeq_refl.`

`+ assumption.`

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 $x = y$. `- intros e3 z XY IH. rewrite m_subst_var_neq.`

To unfold the cases in this proof, we need to destruct one variable as another. We chose to do $x == z$. `+ destruct (y == z) eqn:Hyz.`

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 rest of the cases are finished using the variable substitution's negation of equality, the variable substitution's equality or the standard library lemmas. `× subst. rewrite m_subst_var_eq. pose proof m_subst_notin. specialize (H e3 ([z := e3] u) x). apply aeq_sym. apply H; assumption.`

```

    × rewrite m_subst_var_neq.
    ** rewrite m_subst_var_neq.
      *** apply aeq_refl.
      *** auto.
    ** auto.
  + auto.
- intros e3 z XY IH. destruct (x == z) eqn:Hxz. contradiction. rewrite m_subst_abs_eq.
  pose proof m_subst_notin. specialize (H ([z := e3] n_abs x t1) ([z := e3] u) x).
  apply aeq_sym. apply H. pose proof m_subst_abs_diff. specialize (H0 t1 e3 z x).
rewrite H0.
  admit. auto. admit.
- admit.

```

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.

```

- intros e3 z XY IH. rewrite
m_subst_app. rewrite m_subst_app. rewrite m_subst_app. rewrite m_subst_app. auto.
- intros e3 z XY IH. admit.
- admit.

```

Admitted.

Lemma m_subst_lemma: $\forall e1\ e2\ e3\ x\ y, x \neq y \rightarrow x \text{ 'notin' } (fv_nom\ e3) \rightarrow$
 $aeq\ (m_subst\ e3\ y\ (m_subst\ e2\ x\ e1))\ (m_subst\ (m_subst\ e3\ y\ e2)\ x\ (m_subst\ e3\ y\ e1)).$

Proof.

Admitted.

*)

Inductive betax : $n_sexp \rightarrow n_sexp \rightarrow \text{Prop} :=$
 $| \text{step_betax} : \forall (e1\ e2 : n_sexp) (x : atom),$
 $\text{betax}\ (n_app\ (n_abs\ x\ e1)\ e2)\ (n_sub\ e1\ x\ e2).$

Fixpoint f_pix (t : n_sexp) : $n_sexp :=$
 $\text{match } t \text{ with}$
 $| (n_sub\ (n_var\ x)\ y\ e) \Rightarrow \text{if } x == y \text{ then } e \text{ else } (n_var\ x)$
 $| (n_sub\ (n_abs\ x\ e1)\ y\ e2) \Rightarrow$
 $\text{let } (z, -) :=$
 $\text{atom_fresh } (fv_nom\ (n_abs\ x\ e1)\ \text{'union' } fv_nom\ e2\ \text{'union' } \{\{y\}\}) \text{ in}$
 $(n_abs\ z\ (n_sub\ (swap\ x\ z\ e1)\ y\ e2))$
 $| (n_sub\ (n_app\ e1\ e2)\ y\ e3) \Rightarrow (n_app\ (n_sub\ e1\ y\ e3)\ (n_sub\ e2\ y\ e3))$
 $| _ \Rightarrow t$
 end.

Inductive pix : $n_sexp \rightarrow n_sexp \rightarrow \text{Prop} :=$
 $| \text{one_step} : \forall t, \text{pix } t\ (f_pix\ t).$

Inductive betapi : $n_sexp \rightarrow n_sexp \rightarrow \text{Prop} :=$
 $| b_rule : \forall t\ u, \text{betax } t\ u \rightarrow \text{betapi } t\ u$
 $| x_rule : \forall t\ u, \text{pix } t\ u \rightarrow \text{betapi } t\ u.$

Inductive ctx ($R : n_sexp \rightarrow n_sexp \rightarrow \mathbf{Prop}$): $n_sexp \rightarrow n_sexp \rightarrow \mathbf{Prop} :=$
 $| \text{step_aeq} : \forall e1\ e2, \text{aeq } e1\ e2 \rightarrow ctx\ R\ e1\ e2$
 $| \text{step_redex} : \forall (e1\ e2\ e3\ e4 : n_sexp), \text{aeq } e1\ e2 \rightarrow R\ e2\ e3 \rightarrow \text{aeq } e3\ e4 \rightarrow ctx\ R\ e1\ e4$
 $| \text{step_abs_in} : \forall (e\ e' : n_sexp) (x : \text{atom}), ctx\ R\ e\ e' \rightarrow ctx\ R\ (n_abs\ x\ e)\ (n_abs\ x\ e')$
 $| \text{step_app_left} : \forall (e1\ e1'\ e2 : n_sexp), ctx\ R\ e1\ e1' \rightarrow ctx\ R\ (n_app\ e1\ e2)\ (n_app\ e1'\ e2)$
 $| \text{step_app_right} : \forall (e1\ e2\ e2' : n_sexp), ctx\ R\ e2\ e2' \rightarrow ctx\ R\ (n_app\ e1\ e2)\ (n_app\ e1\ e2')$
 $| \text{step_sub_left} : \forall (e1\ e1'\ e2 : n_sexp) (x : \text{atom}), ctx\ R\ e1\ e1' \rightarrow ctx\ R\ (n_sub\ e1\ x\ e2)\ (n_sub\ e1'\ x\ e2)$
 $| \text{step_sub_right} : \forall (e1\ e2\ e2' : n_sexp) (x : \text{atom}), ctx\ R\ e2\ e2' \rightarrow ctx\ R\ (n_sub\ e1\ x\ e2)\ (n_sub\ e1\ x\ e2')$.

Definition $lx\ t\ u := ctx\ betapi\ t\ u$.

Lemma $step_abs_eq$: $\forall (e1\ e2 : n_sexp) (y : \text{atom}), \exists (z : \text{atom}) (e : n_sexp), \text{refltrans_aeq } (ctx\ pix)\ (n_sub\ (n_abs\ y\ e1)\ y\ e2)\ (n_abs\ z\ e) \wedge (n_abs\ z\ e =_a n_abs\ y\ e1)$.

Proof.

induction $e1$ using $n_sexp_size_induction$. generalize dependent H . case $e1$.
- intros $x\ IH\ e2\ y$. pose proof eq_dec . specialize $(H\ x\ y)$. destruct H .
+ subst.

Admitted.

Lemma $step_redex_R$: $\forall (R : n_sexp \rightarrow n_sexp \rightarrow \mathbf{Prop})\ e1\ e2,$
 $R\ e1\ e2 \rightarrow ctx\ R\ e1\ e2$.

Proof.

intros. pose proof $step_redex$. specialize $(H0\ R\ e1\ e1\ e2\ e2)$.
apply $H0$.
- apply aeq_refl .
- assumption.
- apply aeq_refl .

Qed.

1.1 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 $subst_rec\ (n : \text{nat})\ (t : n_sexp)\ (u : n_sexp)\ (x : \text{atom}) : n_sexp :=$ match n with — $0 =_j$ t — $S\ m =_j$ match t with — $n_var\ y =_j$ if $(x == y)$ then u else t — $n_abs\ y\ t1 =_j$ if $(x == y)$ then t else let $(z, -) := \text{atom_fresh}\ (\text{fv_nom}\ u\ \text{'union' } \text{fv_nom}\ t\ \text{'union' } ^1)$ in $n_abs\ z\ (\text{subst_rec}\ m\ (\text{swap}\ y\ z\ t1)\ u\ x)$ — $n_app\ t1\ t2 =_j$ $n_app\ (\text{subst_rec}\ m\ t1\ u\ x)\ (\text{subst_rec}\ m\ t2\ u\ x)$ — $n_sub\ t1\ y\ t2 =_j$ if $(x == y)$ then $n_sub\ t1\ y\ (\text{subst_rec}\ m\ t2\ u\ x)$ else let $(z, -) := \text{atom_fresh}\ (\text{fv_nom}\ u\ \text{'union' } \text{fv_nom}\ t\ \text{'union' } ^2)$ in $n_sub\ (\text{subst_rec}\ m\ (\text{swap}\ y\ z\ t1)\ u\ x)\ z\ (\text{subst_rec}\ m\ t2\ u\ x)$ end end.

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

Definition $m_subst\ (u : n_sexp)\ (x : \text{atom})\ (t : n_sexp) := \text{subst_rec}\ (\text{size}\ t)\ t\ u\ x$. Notation “ $x := u\ t$ ” := $(m_subst\ u\ x\ t)$ (at level 60).

¹ x

² x

Lemma `m_subst_var_eq` : forall u x, $x := u(n_var\ x) = u$. Proof. intros. unfold `m_subst`. simpl. rewrite `eq_dec_refl`. reflexivity. Qed.

Lemma `m_subst_var_neq` : forall u x y, $x \neq y \rightarrow y := u(n_var\ x) = n_var\ x$. Proof. intros. unfold `m_subst`. simpl. destruct (y == x) eqn:Hxy.

- subst. contradiction.
- reflexivity.

Qed.

Lemma `m_subst_abs` : forall u x y t, `m_subst` u x (`n_abs` y t) = if (x == y) then (`n_abs` y t) else let (z,-) := `atom_fresh` (fv_nom u 'union' fv_nom (`n_abs` y t) 'union' ³) in `n_abs` z (`m_subst` u x (swap y z t)). Proof. intros. case (x == y).

- intros. unfold `m_subst`. rewrite e. simpl. case (y == y).
 - - trivial.
 - - unfold not. intros. assert (y = y). { reflexivity. } contradiction.
- intros. unfold `m_subst`. simpl. case (x == y).
 - - intros. contradiction.
 - - intros. pose proof `AtomSetImpl.union_1`. assert (forall z, size t = size (swap y z t)). { intros. case (y == z).
 - * intros. rewrite e. rewrite `swap_id`. reflexivity.
 - * intros. rewrite `swap_size_eq`. reflexivity.
 - } destruct (atom_fresh (Metatheory.union (fv_nom u) (Metatheory.union (remove y (fv_nom t)) (singleton x)))). specialize (H0 x0). rewrite H0. reflexivity.

Qed.

Corollary `m_subst_abs_eq` : forall u x t, $x := u(n_abs\ x\ t) = n_abs\ x\ t$. Proof. intros u x t. pose proof `m_subst_abs`. specialize (H u x x t). rewrite `eq_dec_refl` in H. assumption. Qed.

Corollary `m_subst_abs_neq` : forall u x y t, $x \neq y \rightarrow$ let (z,-) := `atom_fresh` (fv_nom u 'union' fv_nom (`n_abs` y t) 'union' ⁴) in $x := u(n_abs\ y\ t) = n_abs\ z\ (x := u(\text{swap}\ y\ z\ t))$. Proof. intros u x y t H. pose proof `m_subst_abs`. specialize (H0 u x y t). destruct (x == y) eqn:Hx.

- subst. contradiction.
- destruct (atom_fresh (Metatheory.union (fv_nom u) (Metatheory.union (fv_nom (`n_abs` y t)) (singleton x)))). assumption.

Qed.

Lemma `m_subst_notin` : forall t u x, $x \text{ 'notin' } fv_nom\ t \rightarrow x := ut = t$. Proof. induction t.

- intros u x' H. unfold `m_subst`. simpl in *. apply `notin_singleton_1` in H. destruct (x' == x) eqn:Hx. + subst. contradiction. + reflexivity.

³x

⁴x

- intros u x' H. simpl in *.

•

•

intros. unfold m_subst. simpl. destruct (y == x) eqn:Hxy.

- subst. contradiction.
- reflexivity.

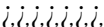
Qed.

Lemma m_subst_lemma: forall e1 e2 e3 x y, x != y -> x 'notin' (fv_nom e3) -> (y := e3(x := e2e1)) =a (x := ([y := e3]e2)(y := e3e1)). Proof.

induction e1 using n_sexp_size_induction.

generalize dependent e1. intro e1; case e1 as z | z e11 | e11 e12 | e11 z e12.

- intros IH e2 e3 x y Hneq Hfv. destruct (x == z) eqn:Hxz. + subst. rewrite (m_subst_var_neq e3 z y). * repeat rewrite m_subst_var_eq. apply aeq_refl. * assumption. + rewrite m_subst_var_neq. * subst. apply aeq_sym. pose proof subst_fresh_eq. change (subst_rec (size e3) e3 (subst_rec (size e2) e2 e3 z) x) with (m_subst (m_subst e3 z e2) x e3). apply H. assumption. * apply aeq_sym. change (subst_rec (size (n_var z)) (n_var z) (subst_rec (size e2) e2 e3 y) x) with (m_subst (m_subst e3 y e2) x (n_var z)). apply subst_fresh_eq. simpl. apply notin_singleton_2. intro H. subst. contradiction.
- intros IH e2 e3 x y Hneq Hfv. unfold m_subst at 2 3. simpl. destruct (x == z) eqn:Hxz. + subst. change (subst_rec (size (m_subst e3 y (n_abs z e11))) (m_subst e3 y (n_abs z e11)) (m_subst e3 y e2) z) with (m_subst (m_subst e3 y e2) z (m_subst e3 y (n_abs z e11))). rewrite subst_abs_eq. +

Admitted.  52cf4c422428638712e894346e04a71a1e69b53f