A Formalization of an extension of the substitution lemma in Coq

Flávio L. C. de Moura Maria Julia

April 10, 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: ∀ 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)).

Lemma m_subst_lemma: ∀ e1 e2 e3 x y, x ≠ y → x `notin` (fv_nom e3) →
  aeq (m_subst e3 y (m_subst e2 x e1)) (m_subst (m_subst e3 y e2) x (m_subst e3 y e1)).

Fixpoint f_pix (t: n_sexp): n_sexp :=
  match t with
  | (n_sub (n_var x) y e) ⇒ if x == y then e else (n_var x)
  | (n_sub (n_abs x e1) y e2) ⇒
      let (z,_) :=
        atom_fresh (fv_nom (n_abs x e1) `union` fv_nom e2 `union` {{y}}) in
      (n_abs z (n_sub (swap x z e1) y e2))
  | (n_sub (n_app e1 e2) y e3) ⇒ (n_app (n_sub e1 y e3) (n_sub e2 y e3))
  | _ ⇒ t
  end.

Inductive pix : n_sexp → n_sexp → Prop :=
| one_step : ∀ t, pix t (f_pix t).

Inductive betapi: n_sexp → n_sexp → Prop :=
| b_rule : ∀ t u, betax t u → betapi t u
| x_rule : ∀ t u, pix t u → betapi t u.

Inductive ctx (R : n_sexp → n_sexp → Prop): n_sexp → n_sexp → Prop :=
 | step_aeq: ∀ e1 e2, aeq e1 e2 → ctx R e1 e2
 | step_redex: ∀ (e1 e2 e3 e4: n_sexp), aeq e1 e2 → R e2 e3 → aeq e3 e4 → ctx R e1 e4
 | step_abs_in: ∀ (e e': n_sexp) (x: atom), ctx R e e' → ctx R (n_abs x e) (n_abs x e')
```

| *step_app_left*: ∀ (*e1 e1' e2*: *n_sexp*) , *ctx R e1 e1'* → *ctx R* (*n_app e1 e2*) (*n_app e1' e2*)
| *step_app_right*: ∀ (*e1 e2 e2'*: *n_sexp*) , *ctx R e2 e2'* → *ctx R* (*n_app e1 e2*) (*n_app e1 e2'*)
| *step_sub_left*: ∀ (*e1 e1' e2*: *n_sexp*) (*x* : *atom*) , *ctx R e1 e1'* → *ctx R* (*n_sub e1 x e2*) (*n_sub e1' x e2*)
| *step_sub_right*: ∀ (*e1 e2 e2'*: *n_sexp*) (*x*:*atom*), *ctx R e2 e2'* → *ctx R* (*n_sub e1 x e2*) (*n_sub e1 x e2'*).

Definition *lx t u* := *ctx betapi t u*.

Lemma *step_abs_eq*: ∀ (*e1 e2*: *n_sexp*) (*y*: *atom*), ∃ (*z*: *atom*) (*e*: *n_sexp*), *refltrans_aeq* (*ctx pix*) (*n_sub* (*n_abs y e1*) *y e2*) (*n_abs z e*) ∧ (*n_abs z e* =*a n_abs y e1*).

Lemma *step_redex_R* : ∀ (*R* : *n_sexp* → *n_sexp* → Prop) *e1 e2*,
R e1 e2 → *ctx R e1 e2*.

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: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.

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 (size t) 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. Proof. intros. unfold m_subst. simpl. rewrite eq_dec_refl. reflexivity. Qed.

Lemma m_subst_var_neq : forall u x y, x ¡¿ y -¿ *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).

[1] x
[2] x
[3] x

- - 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 ¡¿ y -¿ 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*(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 ‘notin‘ fv_nom t -¿ *x* := *u*t = 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.
- 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* := *e2*e1)) =a (*x* := ([*y* := *e3*]*e2*)(*y* := *e3*e1)). Proof.

induction e1 using n_sexp_size_induction.

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

[4] x

- 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