Proof Pearl: Substitution Revisited, Again

Gyesik Lee

Hankyong National University, Korea
gslee@hknu.ac.kr

Abstract. We revisit Allen Stoughton's 1988 paper "Substitution Revisited" and use it as our test case in exploring a nominal approach to variable binding in Coq. In our nominal approach, we use only one-sorted variable names as in pen-and-paper work. We show that it is not only feasible, but also convenient to work with the nominal approach in Coq. Furthermore, we figure out a light infrastructure which provides the base in proving a series of results about simultaneous substitution and α-congruence in the untyped λ-calculus.

1 Introduction

Stoughton (1988) demonstrated a noncanonical, but elegant and sound "nominal" approach to variable binding. Any substitution (including identity substitution) puts terms in a substitution normal form, and α-congruence is equality of substitution normal forms.

In spite of the noncanonical aspect, we find Stoughton's representation very interesting because we believe it is closely related to Barendregt's Variable Convention:

> **VARIABLE CONVENTION.** If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables. (Barendregt 1984)

This convention tacitly assume that we can rename bound variables by fresh ones when necessary, for example, in case of possible variable capture. On the other hand, in Stoughton's representation we rename bound variables whenever we do substitutions without thinking of whether variable capture occurs or not.

From the standpoint of mechanization, it is much more convenient to do reasoning without being based on tacit conventions. Indeed, as long as we stick to one-sorted variable names, there is no direct way to formalize reasoning based on such tacit assumptions like

Barendregt's Variable Convention. Such a formalization would work entirely at the level of terms quotiented by α-congruence. Pitts's Nominal Logic (Pitts 2003) and nominal techniques in Isabelle/HOL (Urban and Tasson 2005), e.g., can be regarded as methods of formalizing the spirit of Barendregt. Moreover, as Barendregt's Variable Convention leads to slick informal proofs, we find that the unrestricted substitution is adequate for formalization of reasoning in a proof assistant like Coq.

Therefore, we decided to use (Stoughton 1988) as our test case in exploring a "nominal" approach to variable binding in Coq. This approach is characterized by a close correspondence between the original reasoning on paper and ours within Coq.

To the best of our knowledge, there are two previous works where the idea of unrestricted substitution are adopted. O'Connor (2005) used it to formalize the Gödel-Rosser incompleteness theorem. But he did not use simultaneous substitution, hence could not maximize the efficiency of the unrestricted substitution, cf. O'Connor (2005, Section 2.4).

On the other hand, Garillot and Werner (2007) applied the original unrestricted substitution in a formal treatment of normalization by evaluation in type theory. There are, however, two main differences from ours. First, we use Coq's internal equality when we do reasoning about sets of free variables while it is not the case in Garillot and Werner's work. Second, we use Coq's basic terminology and techniques while they use the Ssreflect plugin. Their source code can be found at `https://github.com/huitseeker/nbe`. The file `nbe_alpha_conversion.v` contains the part about α-congruence and can be compared with ours at `http://formal.hknu.ac.kr/Stoughton/`. We find that these previous works provide an enough evidence for the adequacy of the Stoughton's unrestricted substitution in formal work.

Our primary contribution is to demonstrate that it is not only feasible, but also *convenient* to work with the nominal approach in Coq. Most of all, our formalization is very compact and close to the original reasoning on paper.[1]

[1] We even believe that it would satisfy the three criteria proposed by (Aydemir et al. 2005), i.e., *reasonable overheads, cost of entry, and transparency* in evaluating mechanization of formal metatheory. However, this goes beyond the scope of this paper.

The rest of the pearl gives explanations about our formalization in Coq. Section 2 explains our choice of the formal representation of the untyped λ-calculus, substitution, and α-congruence. In Section 3, it is demonstrated how basic, but important properties about substitution and α-congruence can be formally proved. Section 4 deals with denotational semantics of the untyped λ-calculus. Finally, Section 5 concludes.

2 Definitions

We assume a denumerable and decidable set `var` of variable names. A set is decidable when the equality of two elements are decidable. For simplicity, we use `nat`, the type of natural numbers, as `var`. In the rest of the paper, we use the following notations.

```
Notation " x @ L " := (In x L) (at level 70).
Notation " x # L " := (~ In x L) (at level 70).
Notation " L \ { x } " :=
  (remove eq_nat_dec x L) (at level 59).
```

Terms and free variables. Untyped terms are defined as usual:

```
Inductive term : Set :=
  Var : var -> term
| App : term -> term -> term
| Lam : var -> term -> term.
```

The set `FV M` of free variables in a term `M` is defined recursively and realized by Coq's standard (finite) lists of variables. A variable `x` is free in a term `M` iff `(x @ FV M)`.

```
Fixpoint FV (M : term) : list var :=
  match M with
    | Var x     => x :: nil
    | App M0 M1 => (FV M0) ++ (FV M1)
    | Lam x M'  => (FV M') \ {x}
  end.
```

Unrestricted simultaneous substitution. `(var -> term)` denotes the set of simultaneous substitutions. The identity substitution `iota` is defined by `iota x = Var x`.

```
Definition substitution := var -> term.
Notation iota := (fun x : var => Var x).
```

Defining the set of substitutions as a function type is another difference from Garillot and Werner's formalization where they used finite lists of pairs of variables and terms. Whether to use functions or finite lists to represent simultaneous substitutions is an important choice factor for implementation. We also started first using finite lists, then found it a little bit too much when equality of substitutions is involved. We could save a lot after we took the function type instead. Moreover, it made our formalization closer to the reasoning on paper. Note then that functional extensionality is necessary to deal with functional equality.

As already mentioned, the unrestricted substitution always renames all the bound variables in order to avoid variable capture. For that we first need to know which variables occur during substitution. Then we use three operations to choose fresh ones. First, `fv_coll` collects all the free variables from the codomain of a substitution function with respect to a finite list. Second, `old` uses `fv_coll` to collect free variables which come up during a substitution. Note that `old` is the dual function of Stoughton's *new*. It builds cofinite sets of variables which do not come up during a substitution. Finally, given a finite set `L` of variables, `choice` chooses a fresh variable which does not belong to `L`. We take the least one bigger than any element from `L`. In fact, it does not matter how to choose a fresh one, cf. `Corollary C3_11` on page 9.

```
Fixpoint fv_coll (L : list var)(rho : substitution) :
  list var :=
  match L with
    | nil     => nil
    | x :: l0 => FV (rho x) ++ fv_coll l0 rho
  end.

Definition old (x : var)(M : term)(rho : substitution) :
  list var := fv_coll (FV M \ {x}) rho.

Definition choice (L : list var) : var :=
  S (fold_right max 0 L).
```

These three operations guarantee the freshness of the new variables in the definition of unrestricted simultaneous substitution. We also define compositions of simultaneous substitutions.

```
Fixpoint subst (M : term)(rho : substitution) : term :=
  match M with
    | Var x     => rho x
    | App M0 M1 => App (subst M0 rho) (subst M1 rho)
    | Lam x M'  => let y := choice (old x M' rho) in
                   Lam y (subst M' (rho [[Var y => x]]))
  end.

Notation " M {{ rho }} " := (subst M rho) (at level 55).

Definition comp (eta rho : substitution) :=
  fun x => (rho x) {{eta}}.
```

Here `rho [[Var y => x]]` stands for the substitution which differs from `rho` only by the value at `x`. That is, it renames every occurrence of the variable `Var x` by `Var y`.

This formal version of the unrestricted substitution shows clearly the importance of simultaneous substitution. If we had used a standard, single substitution, then we needed to first rename `Var x` by `Var y`, then apply the substitution. This, however, means that we could not have structural recursion. See O'Connor (2005, Section 2.4) for further detail.

The following lemma is most crucial in the whole development. It reduces properties about substitution and free variables to the level of collecting free variables. So we do not need worry about the complex mechanism of substitution. Furthermore, we use Coq's internal equality instead of any kind of set-theoretic equalities or setoid equalities.

```
Lemma fv_coll_subst :
  forall (M : term) (rho : substitution),
    FV (M {{rho}}) = fv_coll (FV M) rho.
```

α-congruence. Stoughton defined the α-congruence relation, $=_\alpha$, as the least equivalence relation over `term` such that

(μ) `M N` $=_\alpha$ `M' N'` if `M` $=_\alpha$ `M'` and `N` $=_\alpha$ `N'`; and
(α) `Lam x M` $=_\alpha$ `Lam y N` if either
  (i) `x` $=$ `y` and `M` $=_\alpha$ `N`, or
  (ii) `y` $\notin$ `FV M` and `M {{iota [[Var y => x]]}}` $=_\alpha$ `N`.

and showed that it can be characterized by a structural induction. Here we take this structural characterization of α-congruence as our definition. We find it mathematically more natural to use a structural characterization.

Furthermore, defining α-congruence this way is more appropriate for a formal development in Coq where inductive reasoning is the most powerful tool. It provides not only a structural induction principle, but also reduces the number of cases to be considered.

```
Inductive a_cong : term -> term -> Prop :=
| al_var : forall (x : var), (Var x) Eq (Var x)
| al_app : forall (M N M0 N0 : term),
  M Eq M0 -> N Eq N0 -> (App M N) Eq (App M0 N0)
| al_lam_eq : forall (x y : var) (M N : term),
  x = y -> M Eq N -> (Lam x M) Eq (Lam y N)
| al_lam_notin : forall (x y : var) (M N : term),
  y # FV M -> (M {{iota [[Var y => x]]}}) Eq N ->
  (Lam x M) Eq (Lam y N)

  where " M 'Eq' N " := (a_cong M N).
```

We also need to talk about the α-congruence of substitutions.

```
Definition subst_cong (rho eta : substitution)
  (X : list var) :=
  forall x, x @ X -> (rho x) Eq (eta x).
```

3 Substitution and α-congruence

Stoughton (1988) implicitely used the fact that the α-congruence relation is an equivalence relation. Here we will make it explicit where the reflexivity, symmetry, and transitivity of the relation of α-congruence are necessary. And they will be proved when it is possible. The reflexivity is a basic one needed everywhere.

```
Lemma a_cong_refl (M : term) : M Eq M.
```

Remark. In this and next section we use the same numbering for lemmas, theorems, and corollaries as in the original paper. For example, `L3_1_ii` corresponds to Lemma 3.1.(ii) in (Stoughton 1988).

As already mentioned just before `Lemma fv_coll_subst`, properties about substitution and free variables can be reduced to those about collecting free variables. For instance, in order to prove

```
Lemma L3_1_ii : forall (M : term)(x : var)
  (rho : substitution),
  x @ FV (M {{rho}}) <->
  exists y :var,  y @ FV M /\ x @ FV (rho y).
```

we just need to show the following which is much more convenient to deal with. It can be proved by a simple induction on lists.

```
Lemma L3_1_ii' : forall (L : list var)(x : var)
  (rho : substitution),
  x @ fv_coll L rho <->
  exists y : var, y @ L /\ x @ FV (rho y).
```

Except for Lemma 3.1.(vi) in Stoughton (1988), most of the properties can be formally proved almost in the same way as on paper.

```
Lemma L3_1_vi : forall (M : term), M {{iota}} Eq M.
```

The original short proof of `Lemma L3_1_vi` is based on the fact that the α-congruence relation is symmetric. Note however that we do not have the symmetry yet. In our formalization, `Lemma L3_1_vi` cannot be proved until we have the following syntactic Substitution Lemma.

```
Theorem T3_2 : forall (M : term)(rho eta : substitution),
  (M {{rho}}) {{eta}} = M {{comp eta rho}}.
```

And this Substitution Lemma is necessary to prove `Theorem T3_5` which says that applying a substitution to congruent terms yields equal, not just α-congruent, terms.

```
Theorem T3_5 : forall (M N : term),
  M Eq N -> forall rho, M {{rho}} = N {{rho}}.
```

An immediate consequence is that identity substitutions normalize terms with respect to α-congruence.

```
Corollary C3_6_i : forall (M N : term),
  M Eq N <-> M {{iota}} = N {{iota}}.
```

However, in order to prove this corollary from `Theorem T3_5`, we first need to prove that the relation of α-congruence is an equivalence relation. Surprisingly, the symmetry and transitivity can be proved only after we have `Theorem T3_5`. Therefore, in our formalization, `Lemma L3_1_vi` is necessary to prove the symmetry and transitivity of the α-congruence relation, not the other way around.

The transitivity is one of the most difficult properties to show in our formalization. It needs an auxiliary lemma `a_cong_trans_iota` which says that the transitivity holds with respect to the normalized terms.

```
Lemma a_cong_sym (M N : term) :
  M Eq N -> N Eq M.

Lemma a_cong_trans_iota : forall (M N : term),
  M {{iota}} Eq N -> M Eq N.

Lemma a_cong_trans : forall (M N P : term),
  M Eq N -> N Eq P -> M Eq P.
```

We close this section by listing some other important properties. First, α-equivalent terms contain the same free variables:

```
Lemma L3_1_iii : forall (M N : term),
  M Eq N -> FV M = FV N.
```

Second, substitution and alpha-congruence are compatible:

```
Corollary C3_8 : forall (M N : term)
  (rho eta : substitution),
  M Eq N ->
  subst_cong rho eta (FV M) ->
  M {{rho}} Eq N {{eta}}.
```

Third, in the definition of `Lam x M {{rho}}`, the choice of fresh variables can be arbitrary up to α-congruence as long as it does not belong to `old x M rho`:

```
Corollary C3_11 : forall (M : term)(x y : var)
  (rho : substitution),
  y # old x M rho ->
  (Lam x M) {{rho}} Eq Lam y (M {{rho [[Var y => x]]}}).
```

4 Substitution and denotational semantics

This section consists of a proof of the semantic analogue of the syntactic Substitution Lemma in a denotational semantics. The denotational semantics is based on a complete partial order (cpo):

- `E`, a cpo of *expression values*, is a nontrivial solution to the isomorphism equation `E` $\cong$ `(E -> E)`;
- `In : (E -> E) -> E` and `Out : E -> (E -> E)` are continuous functions such that `Out` $\circ$ `In` $=$ $\mathtt{id}_{\mathtt{(E\ ->\ E)}}$ and `In` $\circ$ `Out` $=$ $\mathtt{id}_{\mathtt{E}}$.

Here we just assume that such a cpo `E` with two continuous functions `In` and `Out` are given.

```
Parameter E : Type.
Parameter In : (E -> E) -> E.
Parameter Out : E -> (E -> E).
Axiom OutIn : forall (f : E -> E), Out (In f) = f.
Axiom InOut : forall (x : E), In (Out x) = x.
```

We now define a denotational semantics for the untyped λ-calculus by a structural recursion. We let `U` stand for the cpo of environments f`var -> E`, ordered componentwise.

```
Fixpoint DS (M : term) : U -> E :=
  match M with
    | Var x => fun tau => tau x
    | App M1 M2 =>
      fun tau => (Out (DS M1 tau)) (DS M2 tau)
    | Lam x M'  =>
      fun tau => In (fun e => DS M' (tau [[e => x]]))
  end.
```

Then the semantic analogue of the syntactic Substitution Lemma holds. The expression `e_comp tau rho` denotes the composition of an environment `tau` and a substitution `rho`.

```
Definition e_comp (tau : U) (rho : substitution) : U :=
  fun x => DS (rho x) tau.

Theorem T4_2 : forall (M : term)(tau : U)
  (rho : substitution),
  DS (M {{rho}}) tau = DS M (e_comp tau rho).
```

Finally, we can show that α-equivalent terms are equal in the denotational semantics.

```
Corollary C4_5 : forall (M N : term),
  M Eq N -> DS M = DS N.
```

5 Conclusion

Our proof pearl introduces a formalization of "Substitution Revisited" (Stoughton 1988) in Coq. The formalization is done as close as possible to the original reasoning on paper and show that it is not only feasible, but also convenient to work with the nominal approach in Coq:

- We apply nominal approach with one-sorted variable names to variable binding in Coq.
- We use Coq's internal equality instead of any kind of set-theoretic equalities or setoid equalities.
- We employ only Coq's basic terminology and techniques.

Our future work is to extend this work by applying the unrestricted simultaneous substitution to more serious tasks. We hope that this work encourages others to get more interested in nominal approach in Coq.

Availability The sources which this paper is based on are available from http://formal.hknu.ac.kr/Stoughton/.

References

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.

François Garillot and Benjamin Werner. Simple Types in Type Theory: Deep and Shallow Encodings. In *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2007.

Russell O'Connor. Essential incompleteness of arithmetic verified by coq. In *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2005.

Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

Allen Stoughton. Substitution Revisited. *Theor. Comput. Sci.*, 59: 317–325, 1988.

Christian Urban and Christine Tasson. Nominal Techniques in Isabelle/HOL. In *CADE*, volume 3632 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 2005.