POPLmark 1a with Named Bound Variables

Aaron Stump
Washington University in St. Louis
`stump@cse.wustl.edu`
`http://cl.cse.wustl.edu/`

Abstract

A solution to the POPLmark challenge part 1a in Coq version 8.0 is described, where names are used for bound variables. The technical complexities associated with using names for bound variables are tamed using two main technical ideas: de Bruijn levels for free variables, and the Barendregt variable convention. The resulting solution stands at around 1250 lines of Coq (as written by a novice Coq user).

1 Introduction

The thesis of this solution to the POPLmark challenge is that programming language meta-theory can be feasibly formalized, at least in some cases, using names to represent bound variables. There are two main advantages of using names for bound variables. First, the representation of terms is more faithful to the official syntax of the language. Papers describing languages with bound variables, and tools implementing such languages, typically support an input syntax with named variables. Internally they may change representations of terms, but externally they support named variables. Hence, any formal meta-theory done with a representational technique like de Bruijn indices applies directly just to a possible internal representation of terms, not to terms as officially supported. Second, representing bound variables using names avoids the tedious technical complexities often associated with de Bruijn indices. It is perhaps believed that those complexities are to be preferred to the complexities associated with bound variables, for example those arising from the definition of capture-avoiding substitution. The present solution aims to show that at least in some cases, such complexities can be tamed, yielding an approach with similar or perhaps less technical tedium than de Bruijn indices. Note that a solution using names for bound variables is in contrast to the other solutions to the POPLmark challenge using deep embeddings of binding syntax. Those solutions instead all use de Bruijn indices, at least partially. (Solutions using deep embeddings are to be contrasted with solutions employing ideas from nominal logic, or higher-order abstract syntax.)

The rest of this writeup is organized as follows. The crucial technical ideas behind the approach are described in Section 2. A detailed description of the concepts and lemmas in the Coq solution are then described (Sections 3 and 4). Finally, Section 5 offers some thoughts on using Coq for PL meta-theory, from the point of a novice Coq user (the author began learning Coq in October, 2005). All references are to Coq version 8.0 [3].

2 Supporting Named Bound Variables

We represent bound variables by names, where we require just that equality of names is decidable. For other purposes it might be necessary to have, for example, that the set of names is infinite, but for POPLmark 1a, decidable equality is sufficient. As is well known, technical complexities with using names for bound variables arise, for example, with capture-avoiding substitution $[N/x]\ M$. When applying the substitution $[N/x]$ to a term headed by a binder binding variable y, it is necessary to rename y (safely) if it occurs free in N. Otherwise, free occurrences of y in N will in general be captured when the substitution passes beneath the binder. This kind of capture-avoiding substitution will not be straightforward to define in a language like Coq. Either simultaneous substitution must be used so that the safe renaming can take place simultaneously with the rest of the substitution; or else the recursive call to substitution will, in the case of binders, be taking placing on a term which, while of equal size, is structurally different (namely, the renamed body of the binding construct). In that case, rather sophisticated techniques will be required to pose the definition as a recursive (necessarily terminating) function in Coq, since Coq requires an argument to such functions to decrease structurally.

If substituted terms contain no free variables, on the other hand, then there is never any need to rename bound variables during substitution, and the above difficulties do not arise. Capture-avoiding substitution $[N/x]\ M$ becomes just grafting: every free occurrence of x in M is replaced by N, without any renaming. This benefit has been observed before by others, and Leroy's solution, for example, takes advantage of it. If we assume the *Barendregt variable convention*, that free and bound variables are drawn from disjoint sets, then we have the desired property, and we can use grafting in place of general capture-avoiding substitution. This is the first crucial technical idea.

The second crucial idea comes in with the subtyping rule (SA-ALL). The informal rule simply assumes that the bound variables are the same when comparing (in the subtyping relation) two universal types. If we are to use names for bound variables, then we must formalize (SA-ALL) in such a way that universal types with different bound variables may be compared. One natural way to do this is simply to replace the bound variables in the two bodies by a new common variable, before comparing the bodies. That is, the rule becomes:

$$\frac{\Gamma\ \vdash\ \mathtt{T}_1\mathtt{<:}\mathtt{S}_1 \quad \Gamma, X <: \mathtt{S}_1\ \vdash\ [\mathtt{X}/\mathtt{X}_1]\mathtt{S}_2 <: [\mathtt{X}/\mathtt{X}_2]\mathtt{T}_2}{\Gamma\ \vdash\ \forall\mathtt{X}_1\mathtt{<:}\mathtt{S}_1.\mathtt{S}_2\ \mathtt{<:}\ \forall\mathtt{X}_2\mathtt{<:}\mathtt{T}_1.\mathtt{T}_2}\ \text{SA-ALL}$$

The question naturally arises, how should we choose the new free variable X? We certainly must choose it to be sufficiently fresh: it should not occur already in the context Γ or in the types being compared. A simple way to ensure freshness is to take natural numbers as our free variables, and choose them in order. For purposes of POPLmark 1a, we may assume that our contexts contain only bounding declarations for type variables (POPLmark 2a adds type declarations for term-level variables). So the next natural number to choose for a new free variable is always just the length n of Γ, since Γ gives bounding declarations for the previous n free variables, starting with 0 for the first. In other words, we will use de Bruijn levels for free variables. Note that this is, in a sense, the complement of the approach used by solutions like Leroy's, where de Bruijn indices represent bound variables and names represent free variables. Note also that this does not conflict with the goal of supporting named variables, since free variables introduced during subtype checking are invisible to the user of the language (glossing over the issue of error reporting).

One technical advantage of requiring free variables to be chosen canonically is that we do not need a renaming lemma. Such a lemma would state that subtyping is invariant under safe renamings of free variables. It appears to be rather unpleasant to prove (although the idea of swapping free variables, propounded by nominal approaches, might help ease the difficulty).

The only "de Bruijnian motion" that arises with this approach, in the context of POPLmark 1a, is in the permutation lemma. When permuting bounding declarations for the k'th and $(k+1)$'th free variables, it will be necessary to swap the names of those variables in the rest of the context, as well as in the compared types. For POPLmark 1a, permutation is just needed to establish weakening. It then turns out that when permuting bounding declarations in the context, only one of the variables in question, namely the first one, is actually used in the rest of the subtyping judgment. The other variable corresponds to a weakening of the context, and is not used elsewhere in the judgment. So permutation need only rename free variable k to $(k+1)$, rather than swap those two variables.

3 Detailed Description of the Solution: Preliminaries

In this Section and the following one, we walk through the solution in detail. Proofs of lemmas and theorems are omitted, although theirs length in lines is indicated. (The interested reader should consult the Coq file containing the solution for the details of the proofs.) A few generic helper lemmas about natural numbers and lists, definitions of a few very minor tactics, and a few inessential Coq directives are also omitted from these Sections. The current Section presents preliminary syntactic notions with their lemmas. Section 4 presents the formalization of subtyping with its lemmas, including transitivity.

3.1 Names

As mentioned, we posit a type of names with decidable equality:

```
Parameter name : Set.
Hypothesis eq_name_dec : forall n m : name, {n = m} + {n <> m}.
```

3.2 Syntax of Types

As described above, we syntactically distinguish occurrences of bound and free variables. Uses of the former are represented using `var`, and the latter using `const`. We refer to free variables in the following as constants.

```
Inductive tp : Set :=
  var : name -> tp
| const : nat -> tp
| top : tp
| arrow : tp -> tp -> tp
| all : name -> tp -> tp -> tp.
```

One wonderful thing about Coq is its notation system, which allows us to introduce more readable notations for universal and arrow type expressions:

```
Notation "'all' n <: x . y" := (all n x y) (at level 61, no associativity).
Notation "x --> y" := (arrow x y) (at level 60, right associativity).
```

As in other solutions, it becomes necessary in some situations to reason by induction on the size of type expressions. So this must be defined.

```
Fixpoint size (T:tp) : nat :=
  match T with
  var _ => 1
| const _ => 1
| top => 1
| T1 --> T2 => 1 + size T1 + size T2
| all x <: T1. T2 => 1 + size T1 + size T2
end.

Lemma size_non_zero : forall Q : tp, size Q > 0.
[1 line]
```

3.3 Substitution

As discussed above, the Barendregt variable convention allows us to implement capture-avoiding substitution just as grafting. We may thus implement it straightforwardly as a recursive function in Coq. We rely here on our decidable equality on names.

```
Fixpoint graft_tp (Q:tp) (x:name) (T:tp) {struct T} : tp :=
  match T with
    var y => match eq_name_dec y x with
               left _ => Q
             | right _ => T
             end
  | const y => T
  | top => T
  | T1 --> T2 => (graft_tp Q x T1) --> (graft_tp Q x T2)
  | all y <: T1 . T2 =>
      let T1' := graft_tp Q x T1 in
      let T2' := match eq_name_dec y x with
                   left _ => T2
                 | right _ => graft_tp Q x T2
                 end in
      all y <: T1' . T2'
  end.
```

We have the following simple lemma, that grafting a constant into a type preserves its size.

```
Lemma size_graft_const : forall (x : nat) (y : name) (T : tp),
                         size (graft_tp (const x) y T) = size T.
[ 5 lines]
```

3.4 Replacement of Constants

As discussed above, the permutation lemma requires that we replace one constant with another. So we must define this second kind of substitution (in addition to `graft_tp` which we had above).

```
Fixpoint put_const(c' c:nat)(T:tp) {struct T} : tp :=
  match T with
    var x => T
  | top => T
  | const y => match (eq_nat_dec c y) with
                 left _ => const c'
               | right _ => T
               end
  | T1 --> T2 => (put_const c' c T1) --> (put_const c' c T2)
  | all x <: T1. T2 => all x <: (put_const c' c T1) . (put_const c' c T2)
  end.
```

We need a typical lemma describing when our two kinds of substitution commute.

```
Lemma commute_graft_put_const : forall (c1 c2 c3 : nat) (x : name) (T : tp),
                                c1 <> c3 ->
                                graft_tp (const c1) x (put_const c2 c3 T) =
                                put_const c2 c3 (graft_tp (const c1) x T).
[9 lines]
```

3.5 Occurrence of Constants

In the (SA-ALL) rule, a canonical new constant is introduced. It will be necessary to ensure, when we formalize (SA-ALL) below, that this constant is not already present in the types being compared. Hence, we need the notion of a constant's occurring in a type expression, together with several lemmas relating this notion to our two kinds of substitution.

```
Fixpoint const_in (n:nat) (T : tp) {struct T} : Prop :=
  match T with
  var x => False
| const x => n = x
| top => False
| T1 --> T2 => const_in n T1 \/ const_in n T2
| all x <: T1. T2 => const_in n T1 \/ const_in n T2
end.

Lemma const_in_graft : forall (c1 c2 :nat)(x:name) (T:tp),
                       const_in c1 (graft_tp (const c2) x T) ->
                       const_in c1 T \/ c1 = c2.
[5 lines]

Lemma const_in_put_const : forall (c1 c2 c3 : nat) (T : tp),
                           const_in c1 (put_const c2 c3 T) ->
                           const_in c1 T \/ c1 = c2.
[2 lines]

Lemma put_const_not_in : forall (T : tp) (c' c : nat),
                         ~ const_in c T ->
                         put_const c' c T = T.
[12 lines]

Lemma put_const_graft : forall (c' c : nat) (x : name) (T : tp),
           ~ const_in c T ->
           put_const c' c (graft_tp (const c) x T) = graft_tp (const c') x T.
[14 lines]
```

3.6 Ground Types

Subtyping below is formalized in such a way that two types are related by the subtyping relation (in a context) only if they contain no free `var` expressions. That is, an informal subtyping judgment that has free variables must be represented in our development by a judgment with constants only (i.e., `const` expressions). The definition of what it means to be ground which we adopt may be described by the following rules, where the intention is that a type T is ground iff $\cdot \vdash_g T$ holds. The idea is to keep track of the bound variables using the list L as we descend into a type expression.

$$\frac{}{L \vdash_g \top}$$

$$\frac{}{L \vdash_g \texttt{const x}} \qquad \frac{n \in L}{L \vdash_g \texttt{var n}}$$

$$\frac{L \vdash_g \texttt{T}_1 \quad L \vdash_g \texttt{T}_2}{L \vdash_g \texttt{T}_1 \to \texttt{T}_2} \qquad \frac{L \vdash_g \texttt{T}_1 \quad L,\texttt{X} \vdash_g \texttt{T}_2}{L \vdash_g \forall\texttt{X<:T}_1\texttt{.T}_2}$$

This notion is easily formalized, and we have again several lemmas relating it to our notions of substitution.

```
Fixpoint is_ground_h(L:list name)(T:tp) {struct T} : Prop :=
  match T with
    const _ => True
  | var n => In n L
  | top => True
  | T1 --> T2 => is_ground_h L T1 /\ is_ground_h L T2
  | all x <: T1. T2 => is_ground_h L T1 /\ is_ground_h (x::L) T2
  end.

Definition is_ground (T:tp) : Prop := is_ground_h nil T.

Lemma is_ground_h_put_const : forall (T:tp)(L:list name)(c’ c:nat),
                              is_ground_h L T ->
                              is_ground_h L (put_const c’ c T).
[3 lines]

Lemma is_ground_put_const : forall (c’ c:nat)(T:tp),
                            is_ground T ->
                            is_ground (put_const c’ c T).
[2 lines]

Lemma is_ground_graft : forall (T : tp) (L : list name) (c : nat) (x : name),
                        is_ground_h L T ->
                        is_ground_h L (graft_tp (const c) x T).
[5 lines]

Lemma is_ground_h_extend_ : forall (c n:nat) (x:name) (T:tp),
                            size T <= n ->
                            forall (L1 L2:list name),
                            is_ground_h (L1 ++ L2) (graft_tp (const c) x T) ->
                            is_ground_h (L1 ++ (x :: L2)) T.
[17 lines]

Lemma is_ground_h_extend : forall (c : nat) (x:name) (T:tp),
                           forall (L1 L2:list name),
                           is_ground_h (L1 ++ L2) (graft_tp (const c) x T) ->
                           is_ground_h (L1 ++ (x :: L2)) T.
[2 lines]
```

3.7 Contexts

The contexts used in subtyping judgments are formalized as Coq lists of bounding declarations. We introduce some notation to allow contexts to grow to the right, thus mirroring the syntax used in the paper typing rules. It seems (to the novice Coq user) that Coq does not consistently maintain this notation during developments, and some lemmas stated below slip back to the original Coq notations for lists.

```
Inductive decl : Set :=  bound : tp -> decl.

Definition ctxt := list decl.

Notation "G ~:: d" := (cons d G) (at level 62, left associativity):list_scope.
Notation "G1 ~++ G2" := (G2 ++ G1) (at level 62, left associativity):list_scope.
```

3.8 Occurrence of Constants in Contexts

We homomorphically extend the notion of occurrence of constants in type expressions to contexts.

```
Fixpoint ctxt_const_in (n:nat) (G:ctxt) {struct G} : Prop :=
  match G with
    G' ~:: (bound T) => const_in n T \/ ctxt_const_in n G'
  | nil => False
  end.
```

3.9 Replacement of Constants in Contexts

We homomorphically extend the notion of replacement of constants in type expressions to contexts, and record a simple lemma.

```
Fixpoint ctxt_put_const (c' c:nat)(G:ctxt) {struct G} : ctxt :=
  match G with
    G' ~:: (bound T) => (ctxt_put_const c' c G') ~:: (bound (put_const c' c T))
  | nil => nil
  end.

Lemma length_ctxt_put_const : forall (c' c : nat) (G : ctxt),
                              length (ctxt_put_const c' c G) = length G.
[4 lines]
```

3.10 Ground Contexts

We homomorphically extend the notion of being a ground type to contexts. We have some simple lemmas relating the new notion to the notion of being a ground type and to replacement of constants in types.

```
Fixpoint ctxt_is_ground(G:ctxt) : Prop :=
  match G with
    G' ~:: (bound T) => is_ground T /\ ctxt_is_ground G'
  | nil => True
  end.

Lemma ctxt_is_ground_put_const_append: forall (G2 G1:ctxt)(c' c:nat),
                                 ctxt_is_ground (G2 ++ G1) ->
                                 ctxt_is_ground ((ctxt_put_const c' c G2)++G1).
[5 lines]

Lemma ctxt_is_ground_permute : forall (G2 G1 : ctxt) (U2 U1 : tp),
                          ctxt_is_ground (G2 ++ bound U2 :: bound U1 :: G1) ->
                          ctxt_is_ground (G2 ++ bound U1 :: bound U2 :: G1).
[4 lines]

Lemma ctxt_is_ground_change : forall (G1 G2 : ctxt) (P Q : tp),
                              ctxt_is_ground (G1 ~:: bound Q ~++ G2) ->
                              is_ground P ->
                              ctxt_is_ground (G1 ~:: bound P ~++ G2).
[2 lines]

Lemma ctxt_is_ground_split : forall G1 G2 : ctxt,
                              ctxt_is_ground (G1 ~++ G2) ->
                              ctxt_is_ground G1 /\ ctxt_is_ground G2.
[2 lines]
```

3.11 Looking up Constants in Contexts

The subtyping rule (SA-Trans-TVar) relies on being able to look up bounding declarations for constants in contexts. Given a natural number k and a context Γ, the operation of looking up the k'th bounding declaration in Γ is a partial function: if k is greater than or equal to the length of Γ, there is no bounding declaration to return. Hence, we encode this operation as a relation. One of the lemmas below states that this relation is functional. Other lemmas record how looking up constants works for the concatenation of contexts or for contexts obtained by replacing constants.

```
Inductive lookup : nat -> ctxt -> tp -> Prop :=
  lookup_s : forall (T:tp) (G:ctxt), lookup 0 (G ~:: (bound T)) T
| lookup_n : forall (n:nat) (T T':tp) (G:ctxt),
                    lookup n G T ->
                    lookup (S n) (G ~:: (bound T')) T.

Lemma lookup_skip : forall (G2 G1 : ctxt) (U : tp),
                    lookup (length G2) (G2 ++ bound U :: G1) U.
[4 lines]

Lemma lookup_skip1 : forall (G2 G1 : ctxt) (k : nat) (U : tp),
                     lookup k G1 U ->
                     lookup (length G2 + k) (G2 ++ G1) U.
[5 lines]

Lemma lookup_skip2 : forall (G2 G1 : ctxt) (k : nat) (U : tp),
                     lookup (length G2 + k) (G2 ++ G1) U ->
                     lookup k G1 U.
[5 lines]

Lemma lookup_start1 : forall (G2 G1 : ctxt) (x : nat) (U : tp),
                      x < length G2 ->
                      lookup x (G2 ++ G1) U ->
                      lookup x G2 U.
[9 lines]

Lemma lookup_start2 : forall (G2 G1 : ctxt) (x : nat) (U : tp),
                      x < length G2 ->
                      lookup x G2 U ->
                      lookup x (G2 ++ G1) U.
[8 lines]

Lemma lookup_functional : forall (G : ctxt) (U1 U2 : tp) (n : nat),
                          lookup n G U1 ->
                          lookup n G U2 ->
                          U1 = U2.
[8 lines]

Lemma lookup_greater : forall (G2 G1 : ctxt) (c n : nat) (U : tp),
                       lookup n (G2 ++ G1) U ->
                       n < length G2 ->
                       ~ ctxt_const_in c G2 ->
                       ~ const_in c U.
[7 lines]

Lemma lookup_ctxt_put_const : forall (G : ctxt) (c' c x : nat) (U : tp),
                           lookup x G U ->
                           lookup x (ctxt_put_const c' c G) (put_const c' c U).
[8 lines]
```

3.12 Types with Bounded Constants

To ensure that derivable subtyping judgments `G |- S <::  T` are well-formed, it is necessary to ensure that the constants used in `S` and `T` all have bounding declarations in `G`. Given our approach to the introduction of constants (see Section 2 above), it suffices to ensure that all the constants in `S` and `T` are (strictly) bounded above by the length of `G`. This property of type expressions, that all their constants are bounded above, is formalized by the following relation. The lemmas relate the new concept to our notions of substitution, and the notion of occurrence of a constant in a type.

```
Fixpoint consts_bounded (n:nat) (T:tp) {struct T} : Prop :=
  match T with
  var x => True
| const x => x < n
| top => True
| T1 --> T2 => consts_bounded n T1 /\ consts_bounded n T2
| all x <: T1. T2 => consts_bounded n T1 /\ consts_bounded n T2
end.

Lemma consts_bounded_strengthen : forall (Q : tp) (m : nat),
                                  consts_bounded (S m) Q ->
                                  ~ const_in m Q ->
                                  consts_bounded m Q.
[2 lines]

Lemma consts_bounded_graft : forall (Q : tp) (n :nat) (x : name),
                             consts_bounded (S n) Q ->
                             consts_bounded (S n) (graft_tp (const n) x Q).
[9 lines]

Lemma consts_bounded_graft2 : forall (Q : tp) (n :nat) (x : name),
                              consts_bounded (S n) (graft_tp (const n) x Q) ->
                              consts_bounded (S n) Q.
[5 lines]

Lemma consts_bounded_step : forall (Q : tp) (n : nat) (x : name),
                            ~ const_in n Q ->
                            consts_bounded (S n) (graft_tp (const n) x Q) ->
                            consts_bounded n Q.
[7 lines]

Lemma consts_bounded_weaken : forall (T : tp) (x y : nat),
                              consts_bounded x T ->
                              x <= y ->
                              consts_bounded y T.
[3 lines]

Lemma consts_bounded_put_const : forall (c1 c2 c3 : nat) (T : tp),
                                 consts_bounded c1 T ->
                                 c2 < c1 ->
                                 consts_bounded c1 (put_const c2 c3 T).
[2 lines]

Lemma consts_bounded_not_in : forall (U : tp) (n : nat),
                              consts_bounded n U ->
                              ~ const_in n U.
[3 lines]
```

3.13 Contexts with Bounded Constants

We extend the notion of types with bounded constants to contexts. The stated lemmas, some rather technical, relate the new notion to our notions of substitution, occurrence of constants in types, boundedness of constants in types, and looking up constants in contexts.

```
Fixpoint ctxt_consts_bounded (G:ctxt) : Prop :=
  match G with
  G' ~:: (bound T) => consts_bounded (length G') T /\ ctxt_consts_bounded G'
| nil => True
end.

Lemma ctxt_consts_bounded_put_const : forall (G2 G1 : ctxt) (c' c : nat),
                            c' < length G1 ->
                            ctxt_consts_bounded (G2 ++ G1) ->
                            ctxt_consts_bounded (ctxt_put_const c' c G2 ++ G1).
[5 lines]

Lemma permute_ctxt_consts_bounded : forall (G2 G1:ctxt) (U1 U2:tp),
                    ctxt_consts_bounded (G2 ++ bound U2 :: bound U1 :: G1) ->
                    ~ const_in (length G1) U2 ->
                    ctxt_consts_bounded (G2 ++ bound U1 :: bound U2 :: G1).
[9 lines]

Lemma ctxt_consts_bounded_not_in1 : forall (G2 G1 : ctxt)(n : nat)(U2 U1 : tp),
                     ctxt_consts_bounded (G2 ++ bound U2 :: bound U1 :: G1) ->
                     n >= length G1 ->
                     ~ const_in n U1.
[10 lines]

Lemma ctxt_consts_bounded_not_in2 : forall (x : nat) (G : ctxt) (U : tp),
                                    lookup x G U ->
                                    ctxt_consts_bounded G ->
                                    forall y : nat,
                                    y >= length G ->
                                    ~ const_in y U.
```

```
[10 lines]

Lemma lookup_const_not_in : forall (G2 G1 : ctxt) (x : nat) (U : tp),
                            x < length G1 ->
                            lookup (length G1 - 1 - x) G1 U ->
                            ctxt_consts_bounded (G2 ++ G1) ->
                            ~ const_in (length G1) U.
[8 lines]

Lemma ctxt_consts_bounded_weaken : forall G2 G1 : ctxt,
                                     ctxt_consts_bounded (G2 ++ G1) ->
                                     ctxt_consts_bounded G1.
[4 lines]

Lemma ctxt_consts_bounded_change : forall (G1 G2 : ctxt) (P Q : tp),
                                   ctxt_consts_bounded (G1 ~:: bound Q ~++ G2) ->
                                   consts_bounded (length G1) P ->
                                   ctxt_consts_bounded (G1 ~:: bound P ~++ G2).
[2 lines]
```

4 Detailed Description of the Solution: Subtyping

We are finally in a position to formalize the subtyping relation, and prove the lemmas about it leading up to transitivity. We again make use of Coq's support for user-defined notations to introduce pleasant notation for subtyping judgments. Some conditions are sprinkled into the encodings of the rules to ensure well-formedness. Particularly in the axioms (`sa_top` and `sa_refl_tvar`), we need to enforce a number of syntactic properties. In `sa_trans_tvar`, we look variables up from the left of the context (which grows to the right). This is why we must subtract x from the length minus 1 before we look up the constant. Since the subtraction provided by the Coq library for natural numbers is cut-off subtraction, we must also require `x` to be bounded by the length of the context `G`. In `sa_all`, we graft in the new constant, which is canonically chosen to be the length of the context `G`, before comparing the bodies (see the discussion in Section 2 above). We must require that this constant is not already present in the types being compared. We then have some simple lemmas about subtyping, which show that the well-formedness conditions we are trying to enforce on derivable subtyping judgments have indeed been enforced.

4.1 Subtyping

```
Reserved Notation "G |- Q <:: T" (at level 70, no associativity).

Inductive subtp : ctxt -> tp -> tp -> Prop :=
  sa_top : forall (G : ctxt) (Q : tp),
           consts_bounded (length G) Q ->
           ctxt_consts_bounded G ->
           is_ground Q ->
           ctxt_is_ground G ->
           G |- Q <:: top
| sa_refl_tvar : forall (G : ctxt) (x:nat),
                 ctxt_consts_bounded G ->
                 ctxt_is_ground G ->
                 x < length G ->
                 G |- (const x) <:: (const x)
```

```
| sa_trans_tvar : forall (G : ctxt) (x:nat) (T U : tp),
                  lookup (length G - 1 - x) G U ->
                  x < length G ->
                  G |- U <:: T ->
                  G |- (const x) <:: T
| sa_arrow : forall (G:ctxt) (Q1 Q2 T1 T2:tp),
             G |- T1 <:: Q1 ->
             G |- Q2 <:: T2 ->
             G |- (Q1 --> Q2) <:: (T1 --> T2)
| sa_all : forall (G:ctxt) (x1 x2 : name) (Q1 Q2 T1 T2 : tp),
           G |- T1 <:: Q1 ->
           ~ const_in (length G) Q2 ->
           ~ const_in (length G) T2 ->
           (G ~:: (bound T1)) |- (graft_tp (const (length G)) x1 Q2)
                                 <:: (graft_tp (const (length G)) x2 T2) ->
           G |- (all x1 <: Q1 . Q2) <:: (all x2 <: T1 . T2)

where "G |- Q <:: T" := (subtp G Q T).

Lemma subtp_ground : forall (G : ctxt) (S T : tp),
                     G |- S <:: T ->
                     is_ground S /\ is_ground T.
[6 lines]

Lemma subtp_ground1 : forall (G : ctxt) (S T : tp),
                      G |- S <:: T ->
                      is_ground S.
[1 line]

Lemma subtp_ground2 : forall (G : ctxt) (S T : tp),
                      G |- S <:: T ->
                      is_ground T.
[1 line]

Lemma subtp_consts_bounded : forall (G : ctxt) (S T : tp),
                    G |- S <:: T ->
                    consts_bounded (length G) S /\ consts_bounded (length G) T.
[3 lines]

Lemma subtp_consts_bounded1 : forall (G : ctxt) (S T : tp),
                              G |- S <:: T ->
                              consts_bounded (length G) S.
[1 line]

Lemma subtp_consts_bounded2 : forall (G : ctxt) (S T : tp),
                              G |- S <:: T ->
                              consts_bounded (length G) T.
[1 line]

Lemma subtp_ctxt_consts_bounded : forall (G : ctxt) (T U : tp),
                                  G |- T <:: U ->
```

```
                                        ctxt_consts_bounded G.
[1 line]

Lemma subtp_ctxt_is_ground : forall (G : ctxt) (T U : tp),
                             G |- T <:: U ->
                             ctxt_is_ground G.
[1 line]
```

4.2 Permutation for Subtyping

The most difficult lemma in the whole development is permutation for subtyping. Essentially, the lemma says the following. Suppose a subtyping judgment is derivable where the context has (adjacent) bounding declarations for the k'th and $(k+1)$'th constants. Suppose that constant k is not used in the bound for constant $(k+1)$, and suppose that constant $(k+1)$ is not used anywhere. Then another subtyping judgment is derivable, where the context has the bounding declarations for k and $(k+1)$ swapped, and constant k is replaced in all relevant places by constant $(k+1)$. The relevant places are the rest of the context following the $(k+1)$'th bounding declaration, and in the types being compared. The permutation lemma is stated in a somewhat involved manner, partly for readability in 80 columns, and partly to accomodate the fact that Coq's induction tactic does not (it seems) work adequately when inducting on an object of an indexed inductive type, where the indices used are syntactically constrained. That is, an induction on the structure of a subtyping judgment with a context of a particular form (namely, with two adjacent bounding declarations) does not appear to be handled adequately by Coq's induction tactic. We can work around this by inducting on an unconstrained judgment, and explicitly adding the syntactic constraint that the context must be of the appropriate form (this is the premise beginning `G = G1  ::  ...`). Not surprisingly, most of the effort in this long and tedious proof goes into the cases for `sa_trans_tvar` and `sa_all`.

```
Lemma permutation_subtp : forall (G:ctxt)(Q T U1 U2:tp),
                          G |- Q <:: T ->
                          forall (G1 G2:ctxt),
                          G = G1 ~:: bound U1 ~:: bound U2 ~++ G2 ->
                          ~ const_in (S (length G1)) Q ->
                          ~ const_in (S (length G1)) T ->
                          ~ const_in (length G1) U2 ->
                          ~ ctxt_const_in (S (length G1)) G2 ->
                          let G2' := ctxt_put_const (S (length G1))
                                       (length G1) G2 in
                          let G' := G1 ~:: bound U2 ~:: bound U1 ~++ G2' in
                          G' |- put_const (S (length G1)) (length G1) Q
                            <:: put_const (S (length G1)) (length G1) T.
[209 lines]
```

4.3 Weakening for Subtyping

With the permutation lemma proved, we can now derive weakening. We first show that weakening with a single bounding declaration is derivable, and then extend this to weakening with multiple bounding declarations. Recall that weakening is needed in the proof of narrowing (which is the crucial lemma for establishing transitivity).

```
Lemma weaken_subtp : forall (U:tp) (G:ctxt) (T Q : tp),
                     is_ground U ->
                     G |- T <:: Q ->
                     consts_bounded (length G) U ->
```

```
                    G ~:: (bound U) |- T <:: Q.
[54 lines]

Lemma multi_weaken_subtp : forall (T Q : tp) (G2:ctxt),
                           G2 |- T <:: Q ->
                           forall (G1:ctxt),
                           ctxt_is_ground G1 ->
                           ctxt_consts_bounded (G2 ~++ G1) ->
                           G2 ~++ G1 |- T <:: Q.
[4 lines]
```

4.4 Transitivity of Subtyping

Armed with our weakening lemma, we may now prove transitivity of subtyping (closely following the paper proof). We benefit here from Vouillon's solution, which we follow closely in the high-level structure of these proofs (the details of the proofs are, naturally, completely different).

```
Let trans_stmt(Q:tp) :=
  (forall (G:ctxt) (S T : tp),
      G |- S <:: Q ->
      G |- Q <:: T ->
      G |- S <:: T).

Let narrow_stmt(Q:tp) :=
  (forall (G:ctxt) (M N : tp),
                  G |- M <:: N ->
                  forall (G1 G2 : ctxt) (P : tp),
                  G = G1 ~:: bound Q ~++ G2 ->
                  G1 |- P <:: Q ->
                  G1 ~:: bound P ~++ G2 |- M <:: N).

Lemma transitivity_case: forall Q:tp,
                         (forall Q':tp, size Q' < size Q -> trans_stmt(Q')) ->
                         (forall Q':tp, size Q' < size Q -> narrow_stmt(Q')) ->
                         trans_stmt(Q).
[58 lines]

Lemma narrow_case: forall Q : tp,
                  (forall Q' : tp, size Q' = size Q -> trans_stmt Q') ->
                  narrow_stmt(Q).
[104 lines]

Lemma both_at_once : forall (n : nat) (Q : tp),
                     size Q <= n ->
                     trans_stmt Q /\ narrow_stmt Q.
[10 lines]

Theorem transitivity_subtp : forall (G : ctxt) (S Q T : tp),
                             G |- S <:: Q ->
                             G |- Q <:: T ->
                             G |- S <:: T.
[2 lines]
```

5 Reflections on Using Coq for PL Meta-Theory

As mentioned above, the author is a novice Coq user (2.5 months using Coq). Hence, while there is a danger that any reflections offered may miss points familiar to more experienced Coq users, there is the hope that they may shed a little light on how new users intent on formalizing PL meta-theory experience Coq.

I would first like to say that I found Coq to be an extremely well designed, robust, and highly usable tool. The error messages are great, and the libraries are well documented and intuitive. The coordination of the Vernacular command language, the Gallina type theory (i.e., the Calculus of Inductive Constructions), and the tactic language is truly a masterpiece of simplicity and good engineering. Once I learned a few methodological points, I felt I was never fighting with Coq, but rather just with the proofs themselves.

These methodological points are summarized next in the form of some suggestions on using Coq for PL meta-theory. I did not find these suggestions in the excellent book by Bertot and Castéran, which was otherwise a superb guide for the novice Coq user (and served well as a textbook for an Automated Theorem Proving class at the graduate level that I taught Fall 2005) [2]. Perhaps more experienced Coq users will have further thoughts or even corrections on some of these points, or would find some of them obvious. I hazard to offer them anyway, since they would have saved me several wrong turns in earlier versions of the development if I had known them in advance.

Do not mix external and internal verification. External verification is an approach where programs are written (e.g., recursive functions in Coq) and properties proved about them. Internal verification is an approach where programs are written intertwined with their proofs: functions take proofs of their preconditions as additional inputs, and emit proofs of their postconditions as additional outputs (see [1] for a good introduction). In Coq at least, it seems to be dangerous to mix these styles. One should not write proof-manipulating functions and then try to prove properties of them. Such an approach can quickly lead to a need for proof irrelevance, for example, to reason that different calls to a function return the same values. This happened at an earlier stage in my development, where I tried to implement the operation of looking up a constant in a context as a recursive function, where a proof that the constant was less than the length of the context (assuring that the lookup would succeed) was required as an additional argument. Proving that calls to `lookup` were equal then required proof irrelevance for less-than-or-equal judgments, which seemed overly complex for this setting.

Use `eq_nat_dec` to compare natural numbers in functions you will reason about. The function `eq_nat_dec`, provided in the Coq library, compares natural numbers and returns either a proof that they are equal or a proof that they are not equal. Lemmas about functions that use `eq_nat_dec` can rely on the existence of these proofs. Otherwise, a little function (returning just `bool`) must be written to compare natural numbers, and then helper lemmas about this function must be proved and used to reason about the code.

Implement partial functions as relations, instead of as functions returning optional values. To implement a partial function in Coq with some return type `A`, one could write a function returning `option A`. Such functions can become tedious to reason about, however. From a hypothesis that a call to the function returns a particular value, there appears not to be a nice way to use a tactic like `inversion` to deduce things about either the arguments or the return value. In contrast, if a partial function has been encoded as a relation (i.e., as an inductive type), then the `inversion` tactic supports very natural reasoning from such a hypothesis.

To perform induction on a term in a syntactically constrained inductive type, consider the term in an unconstrained type and add the constraint as an additional premise. The `induction` tactic is wonderful, except that it seems not adequately to handle the case where the instance of the inductive type family is syntactically constrained. For example, using `induction` on a term with a type like `G1 ++ G2 |- S <:: T` does not work as expected; the syntactic constraint that the context is a concatenation of two subcontexts seems to get lost. The workaround for this, as mentioned above in the discussion of the permutation lemma, is to phrase the induction in terms of a suitably generic type, here `G |- S <:: T`, and then add the syntactic constraint `G = G1 ++ G2` as an additional premise to the lemma being proved.

It is worthwhile to organize inductive proofs so that parametric premises are left parametric in induction hypotheses. It simplifies inductive proofs with many parameters if these parameters are pulled into the Coq context before beginning the induction. Then they do not clutter induction hypotheses, and in many cases tactics like `tauto` can discharge proof obligations fully automatically (where `tauto` would fail if an induction hypothesis had unnecessarily quantified variables). Organizing inductive proofs this way sometimes takes some extra thought, but the effort is worthwhile[1].

References

[1] Thorsten Altenkirch. Integrated verification in Type Theory. Lecture notes for a course at ESSLLI 96, Prague, 1996. Available from the author's website.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* Springer Verlag, 2004.

[3] The Coq Development Team. The Coq Proof Assistant Reference Manual, Version V8.0, 2004. http://coq.inria.fr.

[1]Thanks to students in the fall 2005 class CSE 545, "Introduction to Automated Theorem Proving," at Washington University in St. Louis for helpful participation learning Coq and working on POPLmark. This research was supported in part by NSF contract CCF-0448275, "CAREER: Semantic Programming."