Strongly typed term representations in Coq

Nick Benton Andrew J. Kennedy
Microsoft Research Cambridge

Chung-Kil Hur
University of Cambridge

Introduction

Recently we have formalized in Coq the domain-theoretic semantics of both typed and untyped programming languages, and proved soundness and adequacy theorems [3]. In formalizing the simply-typed language, we represented terms in a *strongly-typed* fashion, ensuring the well-typedness of terms by construction. This representation led to very concise statements and straightfoward proofs, in many ways superior to the more conventional approach of defining first syntax, and then typing judgments over that syntax.

Our approach extends to richer simply-typed constructs, such as pattern matching, and we have recently completed a formalization of a polymorphic lambda calculus in the strongly-typed style. Although some of the techniques used have been known for some time in the type theory community [2, 1, 5, 4], we believe that our formalization is the first to be described for Coq that avoids the use of equality coercions.

A simply-typed language

We study a fragment of a larger CBV language, considering only variables, applications, and fixed-point recursion. Types and typing environments are defined as follows:

```
Inductive Ty := Nat | Arr (ty1 ty2 : Ty).
Definition Env := list Ty.
```

Conventionally, one would define an abstract syntax for expressions, using one's favourite method for representing binding, and then define a typing judgment over the syntax. Instead, we combine the two, *indexing* the types of variables `Var` and expressions `Exp` by the environment `E` and type `t` for which they are well-typed:

```
Inductive Var : Env → Ty → Type :=
| ZVAR : ∀ E t, Var (t::E) t
| SVAR : ∀ E t t', Var E t → Var (t'::E) t.

Inductive Exp E : Ty → Type :=
| VAR : ∀ t, Var E t → Exp E t
| FIX : ∀ t1 t2, Exp (t1 :: Arr t1 t2 :: E) t2
       → Exp E (Arr t1 t2)
| APP : ∀ t1 t2, Exp E (Arr t1 t2)
       → Exp E t1 → Exp E t2.
```

Typing rules for expressions can simply be read off the type of the appropriate constructor. An application constructor `APP` takes two expressions as argument, one of arrow type `Arr t1 t2` and the other of type `t1`. A fixed-point constructor `FIX` takes an expression typed under an environment extended with the argument of type `t1` and recursive function of type `Arr t1 t2`. In the functional programming community, `Exp` is known as a *GADT* (Generalized Algebraic Data Type).

Definitions and statements involving strongly-typed terms are beautifully concise. For example, here is part of an evaluation relation for closed expressions of type `t`:

```
Inductive Ev : ∀ t, Exp [] t → Exp [] t → Prop :=
| EvApp : ∀ t1 t2 e v v2 (e1:Exp [] (Arr t1 t2)) e2,
  Ev e1 (FIX e) →
  Ev e2 v2 →
  Ev (STmExp (doubleSub v (FIX e)) e) v →
  Ev (APP e1 e2) v.
```

This makes use of a function `STmExp` that applies a (term) substitution to an expression. It's here that our troubles begin!

Substitutions

We wish to represent *typed substitutions* that map variables typed in an environment `E` to expressions typed in an environment `E'`:

```
Definition Sub E E' := ∀ t, Var E t → Exp E' t.
```

We can then apply substitutions to expressions:

```
Fixpoint STmExp E E' t (s:Sub E E') (e:Exp E t) :=
match e with
| VAR _ v ⇒ s _ v
| FIX _ _ e ⇒ FIX (STmExp (STmL _ (STmL _ s)) e)
| APP _ _ e1 e2 ⇒ APP (STmExp s e1) (STmExp s e2)
end.
```

When going under a binder, as we have done twice with constructor `FIX`, we must *lift* the substitution to work over expressions in an extended environment:

```
Program Definition STmL E E' t
  (s:Sub E E') : Sub (t::E) (t::E') := fun t' v ⇒
match v with
| ZVAR _ _ ⇒ VAR (ZVAR _ _)
| SVAR _ _ _ v' ⇒ ShTmExp t (s _ v')
end.
```

So far, so good. (Note the use of `Program`, supporting GADT-style pattern matching. We also find `dependent destruction` invaluable when working with strongly-dependent representations such as `Exp`.) We now need a `ShTmExp` of type `Exp E t' → Exp (t::E) t'` that increments all the (term) variables in an expression. In Haskell with GADTs, we would make a (mutually-recursive) application of substitution, writing `STmExp (fun t v ⇒ SVAR t' v)`. But this is not structurally recursive, and therefore is unacceptable in Coq.

We might define `ShTmExp` directly; this is possible, and its general type is

```
∀ E E' t' t, Exp (E++E') t → Exp (E++[t']++E') t
```

But we now run into problems when proving properties of shifting, as these must be proved by induction over an expression of *arbitrary* type `Exp E t`. This involves recasting statements of the form

```
∀ E E' t (e:Exp (E++E') t), ...
```

into the form

```
∀ E0 t (e:Exp E0 t) E E', E0=E++E' → ...
```

which requires passing in a proof of the equality. Sozeau's formalization of the simply-typed lambda calculus uses this technique [6], but it requires the use of `eq_rect` cast operations and many lemmas that simply push them around.

Renamings

We instead borrow a trick from [1, 5], and observe that substitutions that are just shifts are instances of the simpler notion of *renaming*, a map from variables to variables:

```
Definition Ren E E' := ∀ t, Var E t → Var E' t.
```

It's easy to define lift for renamings, without running into issues with recursion:

```
Program Definition RTmL E E' t
  (r : Ren E E') : Ren (t::E) (t::E') := fun t' v ⇒
match v with
| ZVAR _ _ ⇒ ZVAR _ _
| SVAR _ _ _ v' ⇒ SVAR _ (r _ v')
end.
```

Applying a renaming to an expression is straightforward:

```
Fixpoint RTmExp E E' t (r:Ren E E') (e:Exp E t) :=
match e with
| VAR _ v ⇒ VAR (r _ v)
| FIX _ _ e ⇒ FIX (RTmExp (RTmL _ (RTmL _ r)) e)
| APP _ _ e1 e2 ⇒ APP (RTmExp r e1) (RTmExp r e2)
end.
```

We can now define `ShTmExp` by applying a trivial renaming:

```
Definition ShTmExp E t t' : Exp E t → Exp (t'::E) t
  := RTmExp (fun _ v ⇒ SVAR _ v).
```

The downside is that we must define things twice: once for renaming, and once for substitutions. And we now have *four* notions of composition! To prove standard lemmas about composition of substitutions, we prove the equivalent lemmas for all four notions of composition, *in order*, with each building on the previous. Somewhat informally, these go as follows:

```
RTmL t (r o r') = RTmL t r o RTmL t r'
RTmExp (r o r') e = RTmExp r (RTmExp r' e)
STmL t (s o r) = STmL t s o RTmL t r
STmExp (s o r) e = STmExp s (RTmExp r e)
STmL t (r o s) = RTmL t s o RTmL t r
STmExp (r o s) e = RTmExp r (STmExp s e)
STmL t (s o s') = STmL t s o STmL t s'
STmExp (s o s') e = STmExp s (STmExp s' e)
```

We actually save a little on copy-and-paste proving by observing the commonality between `RTmExp`/`RTmL` and `STmExp`/`STmL`, writing instead an appropriately-parameterized function `travExp` [5].

In summary, we use renaming to boot-strap substitution, and similarly boot-strap lemmas concerning their composition.

Extensions

It's straightforward to model other simply-typed language features. Here, for example, are SML-style patterns:

```
Inductive Pat : Env → Ty → Type :=
| PPAIR : ∀ E1 E2 t1 t2,
  Pat E1 t1 → Pat E2 t2 → Pat (E1++E2) (Prod t1 t2)
| PVAR : ∀ t, Pat [t] ty
| PWILD : ∀ t, Pat [] ty
| PFAIL : ∀ t, Pat [] ty
| PAS  : ∀ E t, Pat E t → Pat (t::E) t
| PINL : ∀ E t1 t2, Pat E t1 → Pat E (Sum t1 t2)
| PINR : ∀ E t1 t2, Pat E t2 → Pat E (Sum t1 t2).

Inductive Exp E : Ty → Type :=
| LETPAT : ∀ E' t1 t2,
  Exp E t1 → Pat E' t1 → Exp (E'++E) t2 → Exp E t2
...
```

The extension to System F uses the renamings + substitutions idea for both types and terms. There is a mild combinatorial explosion in the number of forms of application and composition (e.g. the action of a type substitution on a term renaming) but, fortunately, not *all* combinations show up in establishing the lemmas that clients need. `Ty` and `Env` are now indexed by the number of type variables in the environment and the definition of terms remains elegant, with ∀-intro and elim forms looking like this:

```
Inductive Exp n (E:Env n) : Ty n → Type :=
| TLAM : ∀ t,
  Exp (S n) (ShTmE E) t → Exp E (All t)
| TAPP : ∀ t t',
  Exp E (All t) → Exp E (STyT (singleSubT t') t)
...
```

However, this time we currently still need *some* coercions. The action of a type substitution on an expression looks like:

```
Program Fixpoint STyExp u w (sub: SubT u w) (env: Env u)
                       ty (te: Exp env ty)
                : Exp (STyE sub env) (STyT sub ty) :=
  match te with
    | LAM _ _ e ⇒ LAM (STyExp sub e)
    | TLAM _ e ⇒ TLAM (iso (iso1 _ _ _)
                           (STyExp (STyL sub) e))
    | TAPP _ f t ⇒ iso (iso2 _ _ _ _)
                      (TAPP (STyExp sub f) (STyT sub t))
...
```

where, for example,

```
Lemma iso1: ∀ u w (sub: SubT u w) (env: Env u)
                                    (ty: Ty (S u)),
  (Exp (STyE (STyL sub) (ShTyE env))
       (STyT (STyL sub) ty)) =
  (Exp (ShTyE (STyE sub env))
       (STyT (STyL sub) ty)).
```

is a proof that the type Coq infers for the result of applying the lifted type substitution to the body of a type abstraction is equal to that required to give the `TLAM` case of `STyExp` the declared type, and `iso` has type `∀ A B: Type, A=B → A → B`. We tame these coercions by absorbing them into heterogeneous equality via

```
Lemma iso_JMeq: ∀ (A B:Type) (pf: A = B) (a: A),
  JMeq (iso pf a) a.
Lemma iso_eq_JMeq: ∀ (A B:Type) (a:A) (b:B),
  JMeq a b <→ (exists pf, b = iso pf a).
```

and using lemmas that various of our constructions are congruences with respect to `JMeq` in their dependently-typed arguments, which is much easier than explicitly working with coercions.

References

[1] R. Adams. Formalized metatheory with terms represented by an indexed family of types. In *Types for Proofs and Programs*, volume 3839 of *LNCS*, 2006.

[2] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, volume 1683 of *LNCS*, 1999.

[3] N. Benton, A. J. Kennedy, and C. Varming. Some domain theory and denotational semantics in Coq. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, 2009.

[4] H. Goguen and J. McKinna. Candidates for substitution. Technical Report ECS-LFCS-97-358, University of Edinburgh, 1997.

[5] C. McBride. Type-preserving renaming and substitution. Draft, 2005.

[6] M. Sozeau. A dependently-typed formalization of simply-typed lambda-calculus: substitution, denotation, normalization. Draft, 2007.