A Deep Embedding of Parametric Polymorphism in Coq

Robert Atkey
LFCS, School of Informatics, University of Edinburgh
bob.atkey@ed.ac.uk

1 Introduction

We describe a deep embedding of System F inside Coq, along with a denotational semantics that supports reasoning using Reynolds parametricity [4]. The denotations of System F types are given exactly by objects of sort Set in Coq, and the relations used to formalise Reynolds parametricity are Coq predicates with values in Prop. A key feature of the model is the extensive use of dependent types to maintain agreement between the parts of the model. We use type dependency to represent well-formedness of types and terms, and to keep track of the relation between the denotations of types and the parametricity relations between them.

We have used an extension of this formalisation in other research [1], where we proved that the parametric polymorphic representation of Higher-Order Abstract Syntax is adequate.

More information, and the formalisation itself, are available from: http://homepages.inf.ed.ac.uk/ratkey/parametricity/.

2 Preparing the Metatheory

In order to use Coq Sets as denotations of System F types, we require impredicativity. The denotation of the type $\forall\alpha.\tau$ quantifies over all denotations of types (i.e. Sets). By default, Coq's type theory is predicative for Set (although it is impredicative in the type of propositions, Prop), so one cannot construct a new object of sort Set by quantifying over all objects of sort Set. Fortunately, Coq supports a command line option `-impredicative-set` that allows us to proceed.

We also require three axioms to be added to Coq's theory. The first of these is proof irrelevance, which states that all proofs of a given proposition are equal:

$$\forall P : \mathsf{Prop}.\ \forall p_1, p_2 : P.\ p_1 = p_2.$$

We also require extensionality for functions, which states that two functions are equal if they are equal for all inputs:

$$\forall A : \mathsf{Type}, B : A \to \mathsf{Type}, f, g : (\forall a.Ba).(\forall x.\ fx = gx) \to f = g$$

Extensionality for functions allows our denotational model to support the η-equality rules of System F. We also require propositional extensionality, which will allow us to treat equivalent propositions as equal:

$$\forall P, Q : \mathsf{Prop},\ (P \leftrightarrow Q) \to P = Q$$

These axioms allow us to define data with embedded proofs that are equal if their computational contents are equal, which will aid us in proving equalities between denotations of System F types.

We informally justify our use of these axioms, plus impredicativity, by the existence of models of CIC in intuitionistic set theory.

3 Syntax and Semantics of Types

We represent the syntax of System F types using de Bruijn indicies for the bound type variables. Defining the syntax and the well-formedness condition at the same time makes the rest of the development easier.

```
Inductive variable : nat → Set :=
| var :  ∀g i, i < g → variable g.

Inductive type : nat → Set :=
| ty_var  :  ∀g, variable g → type g
| ty_arr  :  ∀g, type g → type g → type g
| ty_univ :  ∀g, type (S g) → type g.
```

System F types are interpreted as functions from type environments to Coq Sets. Type environments are themselves vectors of Sets:

```
Inductive ty_environment : nat → Type :=
| ty_nil  : ty_environment 0
| ty_cons :  ∀i, Set → ty_environment i →
                       ty_environment (S i).
```

The denotation of the parametric universal type quantifier depends on the relational interpretation of types, so we must mutually define the relational interpretation with the underlying denotation of types. The relational interpretation will be a function from relation environments, which are parameterised by a pair of type environments:

```
Inductive rel_environment :
    ∀n, ty_environment n → ty_environment n → Type :=
| rel_nil  : rel_environment ty_nil ty_nil
| rel_cons :  ∀(n:nat) (A B:Set)
              (env1 env2:ty_environment n),
  (A → B → Prop) →
  rel_environment env1 env2 →
  rel_environment (A;:env1) (B;:env2).
```

The definition of the denotations of types and derived relations between them is given by induction on the structure of the type:

```
Definition ty_sem_rel (g:nat) (t:type g) :
  { ty_sem : ty_environment g → Set
  &  ∀(e1 e2 : ty_environment g),
       rel_environment e1 e2 →
       ty_sem e1 → ty_sem e2 → Prop }.
```

In normal mathemetical notation the denotations and relational interpretations of types are given by the following clauses:

$$\begin{aligned}
\mathcal{T}[\![\alpha]\!]\gamma &= \gamma(\alpha) \\
\mathcal{T}[\![\tau_1 \to \tau_2]\!]\gamma &= \mathcal{T}[\![\tau_1]\!]\gamma \to \mathcal{T}[\![\tau_2]\!]\gamma \\
\mathcal{T}[\![\forall\alpha.\tau]\!]\gamma &= \{\, x : \forall A : \mathsf{Set}.\ \mathcal{T}[\![\tau]\!](\gamma[\alpha \mapsto A]) \\
&\qquad \mid \forall A_1, A_2, R : \mathrm{Rel}(A_1, A_2). \\
&\qquad\quad \mathcal{R}[\![\tau]\!](\Delta_\gamma[\alpha \mapsto R])\ (x\,A_1)\ (x\,A_2)\ \} \\
\mathcal{R}[\![\alpha]\!]\rho\ x\ y &= \rho(\alpha)\ x\ y \\
\mathcal{R}[\![\tau_1 \to \tau_2]\!]\rho\ f\ g &= \forall x : \mathcal{T}[\![\tau_1]\!]\gamma_1, y : \mathcal{T}[\![\tau_1]\!]\gamma_1. \\
&\qquad \mathcal{R}[\![\tau_1]\!]\rho\ x\ y \to \mathcal{R}[\![\tau_2]\!]\rho\ (f\,x)\ (g\,y)
\end{aligned}$$

$$\mathcal{R}[\![\forall\alpha.\tau]\!]\rho\ x\ y \quad = \quad \forall A_1, A_2, R : \mathrm{Rel}(A_1, A_2).\ \mathcal{R}[\![\tau]\!](\rho[\alpha \mapsto R])\ (x\ A_1)\ (y\ A_2)$$

Note that the clause for $\mathcal{T}[\![\forall\alpha.\tau]\!]$ restricts to Set-indexed values that are parametric. The notation Δ_γ denotes the diagonal relation for the type environment γ; in Coq this is written `diagonal` γ.

We define aliases for the projection functions from this definition for type denotations and relational interpretations:

```
Definition ty_sem (g:nat)
   (e:ty_environment g) (t:type g) : Set.
Definition ty_rel (g:nat) (e1 e2:ty_environment g)
   (re:rel_environment e1 e2) (t:type g) :
   ty_sem e1 t → ty_sem e2 t → Prop.
```

A key property of the relational interpretation of types is the identity extension lemma:

```
Lemma rel_diagonal : ∀g (e:ty_environment g) ty x y,
   ty_rel (diagonal e) ty x y ↔ x = y.
```

4 Type Shifting and Substitution

The definition of the semantics of terms requires the shifting and substitution on of types. In the case where we only want to shift or substitute the first variable we have the following operations on terms:

```
Definition shift1 : ∀g, type g → type (S g).
Definition subst1 : ∀g, type (S g) → type g → type g.
```

Note that, by our representation of types, the definitions of shifting and substitution automatically preserve well-formedness. As one would expect, defining these operations actually requires defining them for arbitrarily deep contexts.

Semantically, shifting a type into a larger context should not change its denotation. Informally, this is obvious since types τ and `shift1` τ are syntactically the same. However, to Coq they have different (Coq) types, so we need to prove that the denotations of shifted types are equal:

```
Lemma shift1_equal : ∀g (ty : type g) e A,
  ty_sem e ty = ty_sem (A;:e) (shift1 ty).
```

Again, this lemma is a consequence of the analogous lemma for arbitrary contexts. Also, we must prove equality of type denotations in tandem with equality of relational interpretations under shifting. At the empty context level, this is the following lemma:

```
Lemma rel_shift1_equal : ∀g (e1 e2:ty_environment g)
  (re : rel_environment e1 e2) ty
  (A1 A2 : Set) (R : A1 → A2 → Prop) t1 t2 t1' t2',
  JMeq t1 t1' → JMeq t2 t2' →
  ty_rel re ty t1 t2 = ty_rel (R;;re) (shift1 ty) t1' t2'.
```

We have used *John Major* equality [3], `JMeq`, to express the equality between the elements `t1`,`t2` of the denotation of the unshifted type and the elements `t1'`,`t2'` of the shifted type.

Likewise, we state the semantic effect of substitution:

```
Lemma subst1_equal : ∀g ty1 ty2 (e:ty_environment g),
  ty_sem (ty_sem e ty2;:e) ty1 =
  ty_sem e (subst1 ty1 ty2).
Lemma rel_subst1_equal :
   ∀g ty1 ty2 (e1 e2 : ty_environment g)
    (re : rel_environment e1 e2) t1 t2 t1' t2',
  JMeq t1 t1' → JMeq t2 t2' →
  ty_rel (ty_rel re ty1;;re) ty2 t1' t2' =
  ty_rel re (subst1 ty2 ty1) t1 t2.
```

5 Semantics of Terms

As with types we formalise terms using de Bruijn indicies, defining well-typed terms as an inductive family:

```
Inductive term : ∀g, context g → type g → Set := ...
```

where `context g` is a list of System F types well-formed in a type environment of size `g`.

Our main result is that all well-typed terms can be understood as functions from the denotations of contexts (with the evident extension of type denotations to contexts) to the denotation of the result type. Moreover, this function preserves all relations:

```
Definition term_sem : ∀g ctxt ty,
  term ctxt ty →
  { x : ∀(e : ty_environment g),
                  context_sem e ctxt → ty_sem e ty
    | ∀(e1 e2 : ty_environment g)
      (re : rel_environment e1 e2)
      (g1:context_sem e1 ctxt) (g2:context_sem e2 ctxt),
     context_rel ctxt re g1 g2 →
     ty_rel re ty (x e1 g1) (x e2 g2) }.
```

The definition/proof of this result relies on the identity extension lemma above, and the semantic properties of shifting and substitution from the previous section.

6 Proofs using Parametricity

The objective of this formalisation is to carry out mechanised proofs of the correctness of parametric polymorphic representations of datatypes. For example, it is not too hard to prove that $\mathcal{T}[\![\forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha]\!]\emptyset \cong \mathsf{nat}$, and that $\mathcal{T}[\![\forall\alpha.\alpha \to (\sigma \to \alpha \to \alpha) \to \alpha]\!]\emptyset \cong \mathsf{list}(\mathcal{T}[\![\sigma]\!]\emptyset)$ for all closed types σ. The proofs all proceed much as they would on paper, functions are defined in both directions, and parametricity is used to prove that they form an isomorphism. It often turns out to be easier to use tactics to define the witnessing functions.

As we mentioned in the introduction, this model has also been extended to parametricity over Kripke relations, and has been used to prove that the denotation of the type $\forall\alpha.((\alpha \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha$ is isomorphic to the Set of all closed de Bruijn terms [1]. Thus, we have shown that the representation of HOAS using polymorphism is adequate in this model.

Obviously, there is much further one could go here. In the future we intend to formalise the general theorem for parametric representations of inductive and co-inductive datatypes over representable functors, and also to extend the parametric representations of HOAS to arbitrary binding algebras. Also we wish to extend the formalisation to include parametricity for type operators (i.e. System F_ω) and also to formalisations of domain theory [2], so that parametricity results for programming languages may be formalised.

References

[1] Robert Atkey. Syntax for free: Representing syntax with binding using parametricity. In *Typed Lambda Calculi and Applications (TLCA)*, volume 5608 of *LNCS*, pages 35–49. Springer, 2009. To appear.

[2] Nick Benton, Andrew Kennedy, and Carsten Varming. Some Domain Theory and Denotational Semantics in Coq. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, volume 5674 of *LNCS*. Springer, 2009.

[3] Conor McBride. Elimination with a motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *TYPES*, volume 2277 of *LNCS*, pages 197–216. Springer, 2000.

[4] John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, pages 513–523, 1983.