Type Reconstruction

CAS 706

Stephen Forrest

7 March 2006

Type Variables and Substitutions

Many of the variants of λ-calculi studied already have featured unspecified and *uninterpreted* base types. Each such type is generally assumed to represent a specific concrete type whose details we don't care about.

We would like to formalize this idea, and regard these uninterpreted types as *type variables*, upon which we may perform *type substitutions* to obtain more concrete types.

Type Substitutions (I)

A *type substitution* is a finite map from *type variables* to *types or other type variables*. E.g. $[X \mapsto \mathsf{Nat},\ Y \mapsto U]$.

A substitution σ is defined in the expected way against types:

$$
\begin{array}{lcl}
\sigma(X) & = & \begin{cases} \mathsf{T} & \text{if } (X \mapsto T) \in \sigma \\ \mathsf{X} & \text{if } X \notin \mathrm{dom}(\sigma) \end{cases} \\
\sigma(\mathsf{Nat}) & = & \mathsf{Nat} \\
\sigma(\mathsf{Bool}) & = & \mathsf{Bool} \\
\sigma(S \to T) & = & \sigma(S) \to \sigma(T)
\end{array}
$$

As with expression substitution, the action is applied simultaneously, so the substitution $[X \mapsto Y,\ Y \mapsto X]$ (if valid at all) would swap the types X and Y.

Type Substitutions (II)

Observe that, unlike expression substitution, we needn't have any fear of accidental "type variable capture", since there isn't (yet) any binding context for type variables.

We define σ on a term t by simply applying σ to all type annotations appearing in t.

Similarly, we define σ on a context $\Gamma = (x_1 : T_1, \ldots, x_2 : T_2)$:

$$\sigma(\Gamma) = (x_1 : \sigma(T_1), \ldots, x_2 : \sigma(T_2))$$

We can also define compositions of substitutions $\sigma \circ \gamma$; they behave as one would expect, and $(\sigma \circ \gamma) S = \sigma(\gamma S)$.

Preservation of Typing under Substitution

We need to make sure that type substitution doesn't break the well-typedness of our expression!

Fortunately, this is easy to prove. If a term which is a value is well-typed with respect to the type variable X, then it must be well-typed for any subsituted type also. The result follows from induction on typing derivations.

Therefore if σ is a subsitution and $\Gamma \vdash \mathsf{t} : T$, then $\sigma\Gamma \vdash \sigma\mathsf{t} : \sigma T$.

Parametric Polymorphism and Type Reconstruction

One powerful tool that types variables offer us is *parametric polymorphism*: we can use type variables to generalize code that would otherwise be type-specific, without introducing the complexities of subtyping. An example from Haskell is `map`:

```
map ::  (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

(The type variables `a` and `b` allow mapping over lists of any type.)

Parametric polymorphism requires that every substitution be well-typed. A different but related question is, given a term t with type variables and context Γ, does there exist a type substitution σ and type T such that $\sigma\Gamma \vdash \sigma\mathsf{t} : \sigma T$?

The process of finding such valid instantiations of type variables is called *type reconstruction*.

Type Reconstruction

We will briefly formalize the notion of a valid instantiation.

Let t be a term with associated context Γ. We say that a *solution* for (Γ, t) is a pair (σ, T) such that $\sigma\Gamma \vdash \sigma\mathsf{t} : \sigma T$.

Notice that, just as with satisfying assignments to Boolean variables, this need not be unique. For example, take $\Gamma = \mathsf{a} : X, \mathsf{b} : Y$ and $\mathsf{t} = \mathsf{b}\ \mathsf{a}$. Then both of the following are solutions:

$$([Y \mapsto X \to Z], Z),\ ([X \mapsto \mathsf{Bool}, Y \mapsto \mathsf{Bool} \to \mathsf{Bool}], \mathsf{Bool})$$

Constraint-Based Typing

To help us towards solving (Γ, t), we would like to compute a set of *constraints* that must be satisfied by any solution.

Instead of type-checking the term, we'll simply record its constraints, and resolve them later, generating fresh type variables on the fly.

A *constraint set* C is a set of equations $\{S_i = T_i : i \in 1, \ldots, n\}$. A substitution σ *unifies* the equation $S = T$ if $\sigma S = \sigma T$; also, σ *unifies the constraint set* C if it unifies every equation in C.

The *constraint typing relation* $\Gamma \vdash t : T \mid_{\mathcal{X}} C$ is defined by the rules in the following page.

Constraint-Based Typing Rules

CT-Var:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \,|_{\emptyset}\, \{\}}$$

CT-Abs:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \,|_{\mathcal{X}}\, C}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2 \,|_{\mathcal{X}}\, C}$$

CT-App:

$$\Gamma \vdash t_1 : T \,|_{\mathcal{X}_1}\, C_1,$$
$$\Gamma \vdash t_2 : T \,|_{\mathcal{X}_2}\, C_2,$$
$$\mathcal{X}_1 \cap \mathcal{X}_2 =$$
$$\mathcal{X}_1 \cap FV(T_2) = \mathcal{X}_2 \cap FV(T_1) = \emptyset,$$
$$X \notin \mathcal{X}_1, \mathcal{X}_2, T_1, T_2, C_1, C_2, \Gamma, t_1, t_2$$
$$\frac{C' = C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}}{\Gamma \vdash t_1\ t_2 : X |_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{X\}}\, C'}$$ **(CT-IF)**

CT-Zero:

$$\Gamma \vdash 0 : \mathsf{Nat} \,|_{\emptyset}\, \{\}$$

CT-Succ, CT-Pred, CT-IsZero:

$$\frac{\Gamma \vdash t_1 : T \,|_{\mathcal{X}}\, C,\ C' = C \cup \{T = \mathsf{Nat}\}}{\Gamma \vdash \mathsf{succ}\ t_1 : \mathsf{Nat} \,|_{\mathcal{X}}\, C'}$$
$$\frac{\Gamma \vdash t_1 : T \,|_{\mathcal{X}}\, C,\ C' = C \cup \{T = \mathsf{Nat}\}}{\Gamma \vdash \mathsf{pred}\ t_1 : \mathsf{Nat} \,|_{\mathcal{X}}\, C'}$$
$$\frac{\Gamma \vdash t_2 : T \,|_{\mathcal{X}}\, C,\ C' = C \cup \{T = \mathsf{Nat}\}}{\Gamma \vdash \mathsf{iszero}\ t_1 : \mathsf{Bool} \,|_{\mathcal{X}}\, C'}$$

CT-True, CT-False:

$$\Gamma \vdash \mathsf{true} : \mathsf{Bool} \,|_{\emptyset}\, \{\}$$
$$\Gamma \vdash \mathsf{false} : \mathsf{Bool} \,|_{\emptyset}\, \{\}$$

Constraint-Based Typing

This algorithm and the constraint typing relation together motivate a new definition:

Suppose that $\Gamma \vdash t : T \mid_{\mathcal{X}} C$. We say that a *solution* for the problem $(\Gamma, \mathsf{t}, S, C)$ is a pair (σ, T) such that σ satisfies the constraints C and $\sigma S = T$.

How does this relate to our definition of "solution"s to the problem (Γ, t) from before? One is "existential" and independent of our constraint machinery; the other is algorithmic.

It is analogous to a theorem and its proof from logic, and as in logic there are both soundness and completeness results.

Constraint-Based Typing

Soundness of Constraint Typing:

Suppose that $\Gamma \vdash t : T \mid_{\mathcal{X}} C$. If (σ, T) is a solution for $(\Gamma, \mathsf{t}, S, C)$, then it is a solution for (Γ, t).

Write $\sigma \setminus \mathcal{X}$ for the substitution which is not defined for each variable in $\mathcal{X}$, but is otherwise identical to σ.

Completeness of Constraint Typing:

Suppose that $\Gamma \vdash t : T \mid_{\mathcal{X}} C$. If (σ, T) is a solution for (Γ, t) and the domain of σ is disjoint from $\mathcal{X}$, then there is some σ' with $\sigma' \setminus \mathcal{X} = \sigma$ such that (σ', T) is a solution for $(\Gamma, \mathsf{t}, S, C)$.

Unification

We've seen an algorithm for computing constraints: we would now like to solve them. As we have seen, we have no reason to believe there is a unique solution, and in general there is not.

Partial order on substitutions:
We want the most general solution possible. With that in mind, we define a partial order on substitutions. We say that σ is *less specific* than σ', and write $\sigma \sqsubseteq \sigma'$, if there exists γ such that $\sigma' = \gamma \circ \sigma$.

(The intuitive idea here makes sense: the fewer substitutions σ makes, the more general it is, since each substituted variable acts to "specialize" a type.)

Let C be a constraint set. Let σ be the infimum (greatest lower bound) of all the substitutions which satisfy C, with respect to $\sqsubseteq$. If σ itself satisfies C, then it is called a *principal unifier* of C.

Unify algorithm

The algorithm $unify$ always halts. It either returns a principal unifier of C or fails, for non-unifiable constraint sets.

$$
\begin{array}{ll}
unify(C) = & \text{if } C = \emptyset \text{ then } [] \\
& \text{else let } \{S = T\} \cup C' = C \text{ in} \\
& \quad \text{if } S = T \text{ then} \\
& \qquad unify(C') \\
& \quad \text{else if } S = X \text{ and } X \notin FV(T) \text{ then} \\
& \qquad unify([X \mapsto T]\; C') \circ [X \mapsto T] \\
& \quad \text{else if } T = X \text{ and } X \notin FV(S) \text{ then} \\
& \qquad unify([X \mapsto S]\; C') \circ [X \mapsto S] \\
& \quad \text{else if } S = S_1 \to S_2 \text{ and } T = T_1 \to T_2 \text{ then} \\
& \qquad unify(C' \cup \{S_1 = T_1, S_2 = T_2\}) \\
& \quad \text{else} \\
& \qquad fail
\end{array}
$$

Principal Types

Earlier, when discussing parametric polymorphism, the idea of a most general type which is still well-typed was mentioned.

With the tools we've built thus far, we can be precise about the meaning of this. Given a constraint problem $(\Gamma, \mathsf{t}, S, C)$, a *principal solution* for this problem is a pair (σ, T), where σ is smaller than any other solution according to $\sqsubseteq$.

The type T above is then called a *principal type*.

From results about unification, it follows that if there is any solution to $(\Gamma, \mathsf{t}, S, C)$, there is a principal one.

Exercise

Find a principal type for

$$\lambda x : X \,.\, \lambda y : Y \,.\, \lambda z : Z \,.\, (x\ z)\ (y\ z)$$

Let's look at the arguments:

y accepts a z, so $Y = Z \to B$ for some new type B.
x also accepts a z, so $X = Z \to D$ for some new type D.
$(x\ z)$, of type D, accepts $(y\ z)$, of type B, so $D = B \to C$ for some new type C.
For simplicity, rename Z to A. Then we have:
$x : (A \to B \to C)$, $y : (A \to B)$, and $z : A$.

(It can be shown this is the most general solution.)

Reference

Pierce, Benjamin C., *Types and Programming Languages*, The MIT Press, Cambridge (Massachusetts), London (England), 2002.