CS611 Lecture 28 Type Unification 11 November 2005

Scribe: Kevin Markman and Ryan Peterson Lecturer: Andrew Myers

Last lecture we saw whether it would be possible to unify types, but we never showed how to actually see the result. We change our definition of unify from last lecture to be:

$$\mathsf{unify}(T = \tau, E) = \mathsf{unify}(E\{\tau/T\}) \circ \{T \mapsto \tau\}$$

1 Type Checker

We can code this up our type inference in pseudo-SML:

```
datatype type = Int | Bool | Arrow of type * type | TypeVar of type option ref

fun tcheck(Γ, e):type =
   case e of
   Num n ⇒ Int
    | Var x ⇒ Γ(x)
    | App(e0, e1) ⇒
     let
        t0 = tcheck(Γ, e0)
        t1 = tcheck(Γ, e1)
        T2 = freshType()
     in
        (unify(t0, Arrow(t1, T2)); T2)
    |  Lambda(x, e) ⇒
     let
        T1 = freshType()
        t2 = tcheck(extend(Γ, x, T1), e)
     in
        Arrow(T1, t2)
```

Now we have to define our freshType function:

```
fun freshType () =
   TypeVar(ref NONE)
```

Next we have to resolve our types:

```
fun resolve(t:type):type =
   case t of
      TypeVar(r as ref(SOME t')) ⇒
         let t'' = resolve(t') in (r := t''; t'')
    | _ ⇒ t
```

Next we have to unify the types:

```
fun unify(t1, t2) =
   case(resolve(t1), resolve(t2)) of
      (Int, Int) ⇒ ()
    | (Bool, Bool) ⇒ ()
    | (Arrow(t1, t2), Arrow(t3, t4)) ⇒ (unify(t1, t3); unify(t2, t4))
    | (TypeVar(r as ref NONE), t2) ⇒
         if not_in(r, t2) then r := t2 else raise Fail "Error"
    | (t1, TypeVar(r as ref NONE)) ⇒
         if not_in(r, t1) then r := t1 else raise Fail "Error"
    | _ ⇒ raise Fail "Error"
```

2 Type Schemas

The problem with the above code is that we do not quite have as much polymorphism as we expect to have. Consider, for example, binding the identity function to a variable and then applying it to an Int and then to a Bool. The type checker encounters the Int first and says that the function is of type Int→Int and then it gives us an error when we try and use the identity function on the bool parameter. We have to do something special for this, so we go use *type schemas* and *type variables* instead. Type schemas are patterns for types that can be instantiated to create actual types.

$$\forall X.\ X \to X$$

Similarly for multivariate expressions:

$$\forall X, Y.\ X \to Y$$

To do this we have to modify Γ. It is now a mapping from variables to type schemas:

$$\Gamma = x_1 : \sigma_1, x_2 : \sigma_2, \ldots$$

We now have to modify our tcheck function in order to accommodate for our new Γ. We change the Var case in tcheck to be:

```
| Var x ⇒ let s = Γ(x) in instantiate(s)
```

We also have to provide an extra case for the Let statement:

```
| Let(x, e1, e2) ⇒
    let t1 = tcheck(Γ, e1) in
      tcheck(extend(Γ, x, generic(t1, Γ)), e2)
```

Now we have to show what our schema is and the definitions for generic and instantiate:

schema = Typepar list * type

generic(τ, Γ):Typepar = FTV(τ) $\setminus$ FTV(Γ)

instantiate(s:schema):type = replace all Typepar of s with fresh TypeVar

Note that schemas are only created inside Let expressions. For this reason, this approach is called *let-polymorphism*. In theory, it can cause the type checker to run in exponential time, but in practice this is not a problem.

3 Polymorphic Lambda-Calculus

We can add support for type schemas to the λ-calculus. The resulting language is called the *polymorphic λ-calculus*, or the *second-order λ-calculus* because the supported type operations are much like the regular expressions, but at a higher level. In this new language,

$$e ::= \cdots \mid \Lambda X.e \mid e[\tau]$$

where the new expressions are type abstraction and type application. The supported types are now

$$\tau ::= b \mid \tau_1 \to \tau_2 \mid X \mid \forall X.\ \tau.$$

The addition to the operational semantics should look familiar:

$$\frac{}{(\Lambda X.e)\tau \to e\{\tau/X\}}$$

This just gives the rule for instantiating type schema. Since this only affects the types, it can be performed at compile time. In order to write the new typing rules, we need a notion of well-formed types. We introduce a new context Δ that maps type variable names to their *kinds* (for now, there is only one kind: type).

$$\Delta = X_1 :: \text{type}, X_2 :: \text{type}, \ldots$$

where we use double colons to remind ourselves that this is not just a typing context; we're at a higher level than the types we've seen before (remember, second order!). Now we can formalize new higher-order rules for determining the legal types:

$$\frac{}{\Delta, X :: \text{type} \vdash X :: \text{type}} \qquad \frac{\Delta \vdash \tau_1 :: \text{type} \quad \Delta \vdash \tau_2 :: \text{type}}{\Delta \vdash \tau_1 \to \tau_2 :: \text{type}} \qquad \frac{\Delta, X :: \text{type} \vdash \tau :: \text{type}}{\Delta \vdash \forall X.\tau :: \text{type}}$$

The form of the typing judgement is modified to use the new typing context Δ:

$$\Delta; \Gamma \vdash e : \tau$$

where we will only try to construct such judgements when the context Γ contains well-formed types:

$$\forall \tau \in \Gamma.\ \Delta \vdash \tau :: \text{type}$$

or as a shorthand,

$$\Delta \vdash \Gamma.$$

Here are the language's new typing rules, taking Δ into account:

$$\frac{}{\Delta; \Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Delta; \Gamma \vdash e_0 : \tau_1 \to \tau \quad \Delta; \Gamma \vdash e_1 : \tau_1}{\Delta; \Gamma \vdash e_0\ e_1 : \tau} \qquad \frac{\Delta; \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau :: \text{type}}{\Delta; \Gamma \vdash \lambda x.e : \tau \to \tau'}$$

$$\frac{\Delta; \Gamma \vdash e : \forall X.\tau' \quad \Delta \vdash \tau}{\Delta; \Gamma \vdash e[\tau] : \tau'\{\tau/X\}} \qquad \frac{\Delta, X :: \text{type}; \Gamma \vdash e : \tau \quad X \notin \Delta}{\Delta; \Gamma \vdash \Lambda X.e : \forall X.\tau}$$

To finish up, here are a few properties of the polymorphic λ-calculus:

1. it's possible to give a type schema for $\lambda x.\ x\ x$, but not for Ω, which has a recursive type,
2. it can only implement primitive recursive functions,
3. it is still strongly normalizing (not obvious, but not proved here),
4. it is still not Turing Complete,
5. type inference is undecidable, so the programmer must provide types.