Lecture 23

Lectured by Prof. Caldwell and scribed by Sunil Kothari

November 13, 2008

1 Review

We looked at unification, substitutions, and various substitution operators. Remember two types are *unifiable* if there exists a substitution σ such that $\sigma t_1 = \sigma t_2$.

The terms are defined as:

```
data Term = V string | Abs String Term | Ap Term Term
```

The types are defined as:

```
Type = TyVar String | Arrow Type Type
```

The goal is a function infer :: $Term \rightarrow Type$ where infer t returns the Type of term t (if there is one).

2 Type Inference

We can make it a MayBe type to handle cases when a term has no type.

```
data MayBe A = Just A | Nothing
```

We have a little proof system for the type inference.

But first lets introduce the various concepts
A sequent (a state in the proof) for the type inference $\Gamma \vdash x : t$ where,

1. Γ is called a context and is a list of string $\times$ type pairs; the context is what we know so far.
2. E is a constraint set and is a list of type $[(type, type)]$.
3. t is a term.
4. α is a type.

The constraints get synthesized from the proof by propagating back down whereas we construct the proof tree bottom up.

The rule for a type variable is:

$$\frac{}{\Gamma,\ \{\alpha = \tau\} \vdash x : \alpha} \text{ where } \ x : \tau \in \Gamma \qquad \text{(Var)}$$

Here's an example of Axiom rule:
So [x, $\alpha \to \alpha$] , $\{\alpha \to \alpha = \tau\} \vdash x : \tau$.

$$\frac{\Gamma\backslash x \cup \{x : \alpha\} \ \vdash \ M : \beta}{\Gamma\ , E \cup \{\tau = \alpha \to \beta\} \vdash \ \lambda x.M \ : \ \tau} \text{ where } \alpha \text{ and } \beta \text{ are fresh} \qquad \text{(Abs)}$$

The notation means $\Gamma\backslash x$ - Remove pairs from Γ where x is the first element.

The rule for application is:

$$\frac{\Gamma,\ E_1 \vdash \ M : \alpha \to \tau \qquad \Gamma,\ E_2 \vdash \ N : \alpha}{\Gamma,\ E_1 \cup E_2 \vdash \ MN : \tau} \text{ where } \alpha \text{ is fresh} \qquad \text{(App)}$$

So we can now do a proof

$$\dfrac{\dfrac{}{[x : \alpha], \beta = \alpha \vdash x : \beta}\text{(Var)}}{[], \{\beta = \alpha, \tau = \alpha \to \alpha\} \vdash \lambda x.x : \tau}\text{(Abs)}$$

Suppose we wanted to find the type of the term $(\lambda x.x)\ y$.

$$\dfrac{\dfrac{\dfrac{}{[y : \beta, x : \beta'], \{\tau = \beta'\} \vdash x : \tau}\text{(Var)}}{[y : \beta], \{\tau = \beta', \alpha' \to \alpha = \beta' \to \tau\} \vdash \lambda x.x : \alpha' \to \alpha}\text{(Abs)} \qquad \dfrac{}{[y : \beta], \{\alpha' = \beta\} \vdash y : \alpha'}\text{(Var)}}{[y : \beta], \{\tau = \beta', \alpha' \to \alpha = \beta' \to \tau, \alpha' = \beta\} \vdash (\lambda x.x)y : \alpha}\text{(App)}$$

The constraint set generated by collecting the constraints from the above proof tree is:
$\tau = \beta'$
$\alpha' \to \alpha = \beta' \to \tau$
$\alpha' = \beta$

The unification algorithm gives the substitution
$\alpha = \beta$
$\alpha' = \beta$
$\tau = \beta'$

This substitution when applied to the type we assumed earlier *i.e.* α gives β, which is what we expected.

Now we can type this all in Haskell. First, we start with the axiom rule.

We have a function infer which takes as argument, a context, a term, a type, and list of fresh variables generated so far. Again, the function is defined by case analysis on the term.

```
infer_type context trm typ vars  =
   case trm of
    (V x) ->
       case (lookup x context) of
         (Just t1) -> ([(typ,t1)],x:vars)
         Nothing -> error ("infer: " ++ x ++ "not in context.")
    (App m n) -> []
    (Abs x m) -> []
```

Recall that lookup has the following type:

```
*Type_inference> :t lookup
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b
```

We can test our code now (even though the code for application and abstraction returns just an empty list):

```
*Type_inference> infer_type [("x",Arrow (TyVar "a") (TyVar "a"))] (V "x")  (TyVar "a") []
([(a,(a -> a))],[])
*Type_inference> infer_type [("x",Arrow (TyVar "a") (TyVar "a"))] (V "x")  (TyVar "b") []
([(b,(a -> a))],[])
```

We can now fill up the code for abstraction and application as follows:

```
(Ap t1 t2) ->
       let a = fresh "a" ((vars_of context) ++ vars ++ (fv typ)) in
       let (e1,vars1) = infer_type context t1 (Arrow (TyVar a) typ) (a:vars) in
       let (e2,vars2) = infer_type context t2 (TyVar a) (vars1 ++ vars) in
          (e1 ++ e2, vars1 ++ vars2)

  (Abs x t1) ->
       let vars1 = ((vars_of context) ++ vars ++ (fv typ)) in
       let a = fresh "a" vars1 in
       let b = fresh "b" (a:vars1) in
       let vars2 = a:b:vars1 in
       let (e1,vars') = infer_type ((x, TyVar a):context) t1 (TyVar b) vars2 in
         ((typ, Arrow (TyVar a) (TyVar b)):e1, vars' ++ vars2)
```

We will create a helper function *infer* to pass the arguments which remains more or less the same each time.

```
infer context trm =
    let (e,_) =  infer_type context trm (fresh "a" (fvars context))  []
```

```
      in subst (unify e) trm
       where fvars [] = []
             fvars ((x,t):xts) = (fv t) ++ fvars xts
```

Now we can test our code:

```
*Type_inference> :t infer
infer :: [(String, Type)] -> Term -> ([([Char], Type)], [a])

infer [("y", TyVar "b")] (Ap (Abs "x" (V "x")) (V "y"))
aaaa
*Type_inference> infer_type [("y", TyVar "b")] (Ap (Abs "x" (V "x")) (V "y"))  (TyVar "a") []
([(b,aaaa),((aa -> a),(aaaa -> b)),(aa,b)],["aaaa","b","aa","y","aa","a"])
*Type_inference> let (e,_) = infer_type [("y", TyVar "b")] (Ap (Abs "x" (V "x")) (V "y"))
  (TyVar "a") [] in unify e
[a := aaaa,aa := aaaa,b := aaaa]
*Type_inference> let (e,_) = infer_type [("y", TyVar "b")] (Ap (Abs "x" (V "x")) (V "y"))
  (TyVar "a") [] in unify [ head e]
[b := aaaa]
*Type_inference> let (e,_) = infer_type [("y", TyVar "b")] (Ap (Abs "x" (V "x")) (V "y"))
  (TyVar "a") []
 in map2 (subst (unify [ head e])) (tail e)
[b := aaaa]
*Type_inference> infer_type [("y", TyVar "b")] (Ap (Abs "x" (V "x")) (V "y"))  (TyVar "a") []
([(b,aaaa),((aa -> a),(aaaa -> b)),(aa,b)],["aaaa","b","aa","y","aa","a"])
```