Lecture 22

Lectured by Prof. Caldwell and scribed by Sunil Kothari

November 11, 2008

1 Review

Remember two types are *unifiable* if there exists a substitution σ such that $\sigma t_1 = \sigma t_2$.

The types are defined by the following grammar:

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

The HW was to extend the unification algorithm to product types. Last time, we couldn't get the unification completely done. The goal is to use unification to solves equations to infer a type for a term.

2 Substitution

$unify :: type \to type \to Substitution$

```
data Substitution = S [(String, Type)]

subst (S []) t = t
subst (S (x,t1):xts) (TyVar y) = if x == y then t1 else subst (S xts) (TyVar y)
subst s (Arrow t1 t2) = Arrow (subst s t1) (subst s t2)
```

The *subst* is a simultaneous substitution or parallel substitution.
We need a couple of operators: $subst\_subst, +\!+\!+$.

1. *subst_subst*

   ```
   subst_subst s1 (S s2) = S (map_snd (subst s1) s2)
    where  map_snd f [] = []
           map_snd f ((x,t):xts) = (x,f t): map_snd f xts
   ```

   Our goal is that the substitution should behave as a function.

2. The composition of substitution ($s1{+}{+}{+}s2$) is defined as:
   $subst\ (s1{+}{+}{+}s2)\ t = subst\ s1\ (subst\ s2\ t)$

   The mathematical notion for composition is:
   $(f.g)\ x = f\ (g\ x)$

```
(S s2) +++ (S s1) =
     let (S s1') = subst_subst (S s2)(S s1) in
```

   But doing like this we have a problem. Suppose $s2 = [y := t1]$ and $s1 = [x := Vy]$
   then $subst\_subst\ s2\ s1 = [x := t1]$

   But when we apply this subst $subst\ (s2{+}{+}{+}s1)\ (Arrow\ (V\ y)\ \ (V\ x)) \rightsquigarrow Arrow\ t1\ t1$.

   So we change to

```
(S s2) +++ (S s1) =
     let (S s1') = subst_subst (S s2)(S s1) in
       S (functionalize (s1' ++ s2))
   where functionalize [] = []
         functionalize ((x,t):xts) = (x,t):functionalize (filter ((/= x).fst)  xts)
```

   The complicated piece of code $(filter((/=x).fst)xts)$ is : apply first to each element of xts and if it's not equal to x keep it.

   But this is a $O(n^2)$ algorithm - but we are not concerned since substitutions don't become very big for the terms that we consider here.

3 Unification Algorithm

1. A variable (say x) unifies with any term t1 as long as $x \notin FV(t)$. i.e. make a substitution $[(x,t)]$
2. $t1$ and $t2$ unify if $t1$ and $t2$ have the same constructor and their corresponding subterms unify.

The code in the HW unifies a list of term-term pairs.

```
unify (Var x) t = if x `elem` (fv t) then error "unify:occurs check" else
                  case t of
                   Var y -> if x == y then S [] else S[(x,t)]
```

Just a bit of history, unification was invented by Robinson in 1965 for theorem proving.

What happens with occurs_check ? If we could unify the types a and $a \to a$ by the substitution $S[(a, a \to a)]$ then we loose the property that substitution makes the types equal.
subst $s\ a = a \to a \neq$
subst $s\ (a \to a) = (a \to a) \to (a \to a)$

```
unify t (Var x) = unify (Var x) t
unify (Arrow t1 t2) (Arrow t3 t4) = let s = unify t1 t3 in let
                                        let s2 = unify (subst s1 t2) (subst s1 t4) in
                                           s2.
```

It turns out that we want to have a unification algorithm for a *list* of equations.

```
unify\_list [] = S []
unify\_list ((t,t')::tts) = let s = unify t t' in
                            s +++ unify_list (map2 (subst s) tts)
                             where map2 [] = []
                                   map2 ((x,y):(xys)) = (f x, f y):map2 f xys
```

Note that this unification algorithm returns most general unifier. Here's an example:
$unify\ a\ b \to b = S[(a, b \to b)]$

Another substitution could be $s' = S[(a, c \to c), (b, c)]$
Then $s'a = c \to c$ and

$$\begin{aligned} s'(b \to b) &= (s'b) \to (s'b) \\ &= c \to c \end{aligned}$$

A substitution σ is a most general unifier of T_1 and T_2 if for every $\hat{\sigma}$ that unifies T_1 and T_2 there exists a substitution $\overline{\sigma}$ such that $\hat{\sigma} = \overline{\sigma} \circ \sigma$.

Definition 1 (Most general unifier)**.** *σ is a most general unifier for t_1 and t_2 iff for all unifiers σ', there exists σ such that $\sigma' = \hat{\sigma}\ +\!+\ \sigma$.*

The unification algorithm discussed above has linear complexity if we represent terms as DAGs but the time complexity is exponential if the terms are represented as strings.

Why are we doing this ? Remember we had to do some reasoning and do some manipulation but with type inference algorithms we do not have to do those manipulations.

4 Type Inference

If $\{\} \vdash M : \alpha$ yields σ then there is a proof in the type system that $\{\} \vdash M : \sigma(\alpha)$.

The first type inference algorithm was given by Milner in 1978 but Hindley and Seldin came up with the ideas in 1965. Mitchell Wand (1986) gave an algorithm where substitutions are not built as the proof tree is constructed. Instead the equations are generated and substitution is then generated from those equations. Here's the proof rules for the Wand's type system
We write $\Gamma \vdash x : t$ where,

1. Γ is called a context and is a list of variable $\times$ type pairs;
2. E is a constraint set and is a list of type $[(type, type)]$.
3. t is a term; and
4. α is a type.

The terms are given by the following grammar
$Term ::= V\ String \mid Abs\ String\ Term \mid Ap\ Term\ Term$

The rule for a type variable is:

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

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

The rule for application is:

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

Next time we will look at a recursive algorithm which will generate this constraint set on which we apply the unification algorithm to give a type for the term.