Lecture 21

Lectured by Prof. Caldwell and scribed by Sunil Kothari

November 6, 2008

1 Review

We have a programming language which has constructors and destructors for functions and pairs. But we don't have numbers (and arithmetic expressions) in our language but that is easy - encode numbers as lambda terms.

We don't need pairs even but it is convenient to have pairs.

The next goal is to have types and type inference for this language. The next goal is to have a type inference algorithm for this language.

What are the types that we need to have ?

The constructs in the language are :
$\Lambda ::= x \mid MN \mid \lambda x.M \mid < M, N > \mid spread M; x, y.N$

x - Variables	Type variables
$M\ N$ - Application	
$\lambda x.M$ - Abstraction	Arrow
$< M, N >$ Pairs	Product

For the constructors we can say what the type is gonna be but for destructors we cannot say so.

So we have two languages: one involving terms and the other involving types.

So let's do it in Haskell

```
module Ttypes where
data Types = TyVar String | Arrow Types Types | Prod Types Types deriving (Eq, Show)
```

The type language is less complicated - there is no binding here

So, lets define *fv*

```
fv (TyVar x) = [x]
fv (Arrow t1 t2) = fv t1 ++ fv t2
fv (Prod t1 t2) = fv t1 ++ fv t2
```

So where are we headed ?

We will look at type checking and follow it up with type inference.

What does the typing rules look like ?

There's gonna be a notion of what type means under a context Γ. We write $\Gamma \vdash x : t$ where,
Γ is list of variable $\times$ type pairs;
M is a term; and
t is a type.

So how do we show $\lambda x.M$ has type T so it must be the type $T_1 \to T_2$. Typing rules specify this *how* aspect.

$$\frac{\Gamma \backslash x \cup \{x : \tau_1\} \ \vdash \ M : \tau_0}{\Gamma \ \vdash \ \lambda x.M \ : \ \tau_1 \to \tau_0} \quad \text{(Abs)}$$

The rule for application is:

$$\frac{\Gamma \ \vdash \ M : \tau_1 \to \tau \qquad \Gamma \ \vdash \ N : \tau_1}{\Gamma \ \vdash \ MN : \tau} \quad \text{(App)}$$

The rule for a type variable is:

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

So how does this defers from type inference ?

What we are now doing is called type checking. Here's an example:

$$\dfrac{\dfrac{\dfrac{}{y :: b, x :: b \vdash x :: b}\,\text{Axiom}}{y :: b \vdash (\lambda x.x) :: b \to b}\,\text{App} \qquad \dfrac{}{y :: b \vdash y :: b}\,\text{Axiom}}{y : b \vdash (\lambda x.x) y : b}\,\text{App}$$

There is a little bit of manipulation going on. Type inference takes care of that. We need a couple of ideas : substitutions which maps variables to types and we also need unification algorithm to solve the equations generated by these types.

1.1 Substitutions

Substitution are functions from substitutions to types. If σ is a substitution then
$\sigma\ (TyVar\ x) = \sigma(x)$
$\sigma\ (Arrow\ t_1\ t_2) = Arrow\ (\sigma\ t_1)\ (\sigma\ t_2)$
$\sigma\ (Prod\ t_1\ t_2) = Prod\ (\sigma\ t_1)\ (\sigma\ t_2)$

Given two types T_1 and T_2, we say a substitution *unifies* T_1 and T_2 iff $\sigma T_1 = \sigma T_2$.

For example, consider $a \to b$ and type c then $\sigma = [c := a \to b]$ is a unifier since

$$\begin{aligned} \sigma(a \to b) &= \sigma(a) \to \sigma(b) \\ &= a \to b \end{aligned}$$

and $\sigma c = a \to b$.

So σ unifies $a \to b$ and c.

Also $\hat{\sigma} = [b := b \to b, c := a \to (b \to b)]$ then $\hat{\sigma}(a \to b) = a \to (b \to b)$ and $\hat{\sigma}(c) = a \to b \to b$.

So $\hat{\sigma}$ is also a unifier.

Definition 1 (Most general unifier)**.** *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 $\hat{\sigma} = \overline{\sigma} \circ \sigma$.*

Example of types that are not unifiable a and $a \to a$.

Suppose there was a substitution that could unify these types, then it would be $a := a \to a$. But, then $\sigma(a) = a \to a$ and
$\sigma(a \to a) = (a \to a) \to (a \to a)$

So, there is no substitution that unifies these two.

We will use these ideas and change the rules a little bit later - generate constraints while deriving a derivation for a given type.
For example,
If $\sigma = [x := y, y := z]$, then $\sigma(x) = y$.

But consider $\sigma_1 = [x := y]$ and $\sigma_2 := [y := z]$ then $\sigma_2(\sigma_1 x) = \sigma_2 y = z$. But $\sigma(x) = y$.

One active area in programming languages is trying to reason about bindings.
We can now code what we discussed in Haskell as:

```
data Subst = S [(String, Types)] deriving (Eq,Show)

apply:: Subst -> Types -> Types
apply (S [])  t = t
apply (S ((x,t):xts)) (TyVar y) = if x == y then t else apply (S xts) (TyVar y)
```

```
apply s (Arrow t1 t2) = Arrow (apply s t1) (apply s t2)
apply s (Prod t1 t2) = Prod (apply s t1) (apply s t2)
```

We can start the code for unification as:

```
unify (TyVar x)(TyVar y) = if x == y then (S []) else (S [(x,TyVar y)])
unify (TyVar x) t = (S [(x,t)])
unify t (TyVar x) = unify (TyVar x) t
unify (Arrow t1 t2) (Arrow t3 t4) =
unify (Prod t1 t2) (Prod t3 t4) =
unify (Arrow _ _) (Prod _ _) = error "unify"
unify (Prod _ _) (Arrow _ _) = error "unify"
```

Notice we didn't do a check. This is not complete yet but we will continue next time.