

Comp 411 Notes - Wednesday, April 20 2005

Introducing System F

Scribe: Eric Cheng
(ericc@rice.edu)

We want to find a way to write the identity function once and be able to use it many times.

$$t ::= X \mid t \rightarrow t$$

What's the difference between the Hindley-Milner type system and simple types? let-polymorphism. In both of the type systems, the set of the types is just as above. We said we have polymorphism in HM, but where does quantification show up? There is usually something called a 'scheme' we use in type inference, in the form of

$$s ::= t \mid \forall X. s$$

Usually in the HM system, we have a notion of scheme. Functions such as the id function can have a schema associated with them, which is polymorphic. All the forall's have to be at the very top. You can't have nested forall's. In HM, type variables are not polymorphic. $\alpha \rightarrow \alpha$ only takes a fixed α .

For example, in the HM type system, we can write `let id x = x in (id 7, id "s")` but we cannot write `let f g = (g 7, g "s")`

If we have universal types, the set of types is then

$$t ::= X \mid t \rightarrow t \mid \forall X. t$$

which allows us to type the term as

```
let f : ( $\forall X. X \rightarrow X$ )  $\rightarrow$  (int, string)
    g : ( $\forall X. X \rightarrow X$ ) = (g 7, g "s")
```

Note that with universal types, 'forall' quantifies over all possible types. There is no way in this system we can quantify over a fixed set of types.

So, as is with all famous type constructors we've seen, what do we need to do with "forall"? Introduction and elimination.

⁰At this point, we are going past the type systems that are actually implemented in languages out there. They are not in mainstream programming languages yet. This is the most useful part of the course because we cannot learn it by just reading about some programming languages.

$$\frac{\Gamma, X \vdash e : t}{\Gamma \vdash \lambda X. e : \forall X. t} \text{ (Introduction)}$$

$$\frac{\Gamma \vdash e : \forall X. t}{\Gamma \vdash e[t'] : t[X := t']} \text{ (Elimination)}$$

The rest of the rules are the following:

$$\overline{\vdash []}$$

$$\frac{x \notin \text{dom}(\Gamma) \quad \vdash \Gamma}{\vdash \Gamma, x}$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash t}{\vdash \Gamma, x : t}$$

$$\frac{\vdash \Gamma \quad \Gamma(x) = t}{\Gamma \vdash x : t}$$

$$\frac{FV(t) \in \text{dom}(\Gamma)}{\Gamma \vdash t}$$

$$\frac{\Gamma \vdash t_1 \quad \Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x : t_1. e : t_1 \rightarrow t_2}$$

The $\Gamma \vdash t_1$ axiom makes sure $FV(t_1)$ is in Γ , which is needed in order to inductively prove that $FV(t)$ is always in Γ .

$$\frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$\frac{e \mapsto e' \quad e[t] \mapsto e'[t]}{(\lambda X. e)[t] \mapsto e[X := t]}$$

Back to the example we just saw, this is how we would write it in System F.

`let f = λg : (∀X.X → X).(g [int] 17, (g [string] "s"))`

`let f g = (g 17, g "s")`

What would happen if we evaluated under type abstraction? we will see in the next lecture.