

System-F and the Lambda Cube

COMP3610 – Principles of Programming Languages

Ben Lippmeier

Australian National University

Semester 2, 2009

The `map` function

$$\begin{aligned} \text{map} &:: \forall a. \forall b. (a \rightarrow b) \rightarrow \text{List } a \rightarrow \text{List } b \\ \text{map} &= \lambda f. \lambda xs. \\ &\quad \mathbf{case } xs \mathbf{ of} \\ &\quad \{ Nil \quad \quad \quad \rightarrow Nil \\ &\quad ; Cons\ x\ xs \rightarrow Cons\ (f\ x)\ (map\ f\ xs)\ } \end{aligned}$$

- During compilation we want to be able to quickly reconstruct the type of an arbitrary sub expression.
- We don't want to have to perform full type inference (ie, extracting and solving type constraints) each time .. too slow!

Adding type annotations to terms

$map :: \forall a. \forall b. (a \rightarrow b) \rightarrow List\ a \rightarrow List\ b$

$map = \lambda(f : a \rightarrow b). \lambda(xx : List\ a).$

case xx **of**

$\{ Nil \rightarrow Nil$

$; Cons\ x_a\ xs_{List\ a} \rightarrow Cons\ (f_{a \rightarrow b}\ x_a)\ (map_{(a \rightarrow b) \rightarrow List\ a \rightarrow List\ b}\ f... xs...) \}$

- Annotating λ binders with their types tells us what parameters can be passed to the function.
- Annotating bound occurrences lets us locally reconstruct the types of sub-expressions. For example, the type of $(f_{a \rightarrow b}\ x_a)$ is clearly b , we don't need any more information.
- From now on we'll assume that bound occurrences of variables are annotated with their types, but won't always show them.

Inlining the definition of `map` produces a mis-typed program

```
let map =  $\lambda(f : a \rightarrow b). \lambda(xx : List\ a).$   
  case xx of  
    { Nil           $\rightarrow$  Nil  
    ; Cons x xs     $\rightarrow$  Cons (f x) (map f xs) }  
in map succ (Cons 1 (Cons 2 Nil))
```

$\xrightarrow{\text{let}}$ ($\lambda(f : a \rightarrow b). \lambda(xx : List\ a).$
 case xx **of**
 { Nil \rightarrow Nil
 ; Cons x xs \rightarrow Cons (f x) (map f xs) }) succ (Cons 1 (Cons 2 Nil))

The function *succ* has type $Int \rightarrow Int$, not $a \rightarrow b$

Type abstraction and application

let $map :: \forall a. \forall b. (a \rightarrow b) \rightarrow List\ a \rightarrow List\ b$

$map = \Lambda a. \Lambda b. \lambda(f : a \rightarrow b). \lambda(xx : List\ a).$

case xx **of**

$\{ Nil \quad \rightarrow Nil\ b$

$; Cons\ x\ xs \quad \rightarrow Cons\ b\ (f\ x)\ (map\ a\ b\ f\ xs) \}$

in $map\ Int\ Int\ succ\ (Cons\ Int\ 1\ (Cons\ Int\ 2\ (Nil\ Int)))$

- Λ = “big lambda”
- Λ behaves the same way as λ , but it binds a type parameter instead of a value parameter.
- Note that every λ corresponds to a \rightarrow in the type signature, and every Λ corresponds to a \forall .

Inlining again...

```
let map =  $\Lambda a. \Lambda b. \lambda(f : a \rightarrow b). \lambda(xx : List\ a).$   
    case xx of  
    { Nil           $\rightarrow Nil\ b$   
    ; Cons x xs     $\rightarrow Cons\ b\ (f\ x)\ (map\ a\ b\ f\ xs)$  }  
in map Int Int succ (Cons Int 1 (Cons Int 2 (Nil Int)))
```

```
 $\xrightarrow{\text{let}}$  (  $\Lambda a. \Lambda b. \lambda(f : a \rightarrow b). \lambda(xx : List\ a).$   
    case xx of  
    { Nil           $\rightarrow Nil\ b$   
    ; Cons x xs     $\rightarrow Cons\ b\ (f\ x)\ (map\ a\ b\ f\ xs)$  } )  
  
    Int Int succ (Cons Int 1 (Cons Int 2 (Nil Int)))
```

Doing the type applications leaves a well-typed program

... ($\Lambda a. \Lambda b. \lambda(f : a \rightarrow b). \lambda(xx : List\ a).$
 case xx **of**
 { $Nil \rightarrow Nil\ b$
 ; $Cons\ x\ xs \rightarrow Cons\ b\ (f\ x)\ (map\ a\ b\ f\ xs)$ })

 $Int\ Int\ succ\ (Cons\ Int\ 1\ (Cons\ Int\ 2\ (Nil\ Int)))$

$\xrightarrow{\beta}$ ($\lambda(f : Int \rightarrow Int). \lambda(xx : List\ Int).$
 case xx **of**
 { $Nil \rightarrow Nil\ Int$
 ; $Cons\ x\ xs \rightarrow Cons\ Int\ (f\ x)\ (map\ Int\ Int\ f\ xs)$ })

 $succ\ (Cons\ Int\ 1\ (Cons\ Int\ 2\ (Nil\ Int)))$

The function $succ$ has type $Int \rightarrow Int$, and so does the function parameter...

System-F expressions

x \rightarrow (value variables)

a \rightarrow (type variables)

Types

τ ::= a (type variable)
| Int | Bool | Char (base types)
| $\tau_1 \rightarrow \tau_2$ (function type)
| $\forall a. \tau$ (quantified type)

Expressions

e ::= x
| $\lambda(x : \tau). e$ (value abstraction)
| $e_1 e_2$ (value application)
| $\Lambda a. e$ (type abstraction)
| $e \tau$ (type application)

System-F evaluation

$$(\lambda(x : \tau).e_1) e_2 \quad \xrightarrow{\beta} \quad e_1[x := e_2]$$

$$(\Lambda a. e_1) \tau \quad \xrightarrow{\beta} \quad e_1[a := \tau]$$

- We still only have one primitive operation, substitution, but now we're substitution both values and types.
- Note that in the first rule we ignore the type annotation on the function parameter.
- Some presentations use $\hat{\lambda}$ for type abstraction as well.

System-F typing rules

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x :: \tau} \text{ (Var)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 :: \tau_2}{\Gamma \vdash \lambda (x : \tau_1). e_2 :: \tau_1 \rightarrow \tau_2} \text{ (Abs)}$$

$$\frac{\Gamma, a \vdash e :: \tau \quad a \notin \text{fv}(\Gamma)}{\Gamma \vdash \Lambda a. e :: \forall a. \tau} \text{ (AbsT / Gen)}$$

$$\frac{\Gamma \vdash e_1 :: \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1 e_2 :: \tau_{12}} \text{ (App)}$$

$$\frac{\Gamma \vdash e :: \forall a. \tau_1}{\Gamma \vdash e \tau_2 :: \tau_1[a := \tau_2]} \text{ (AppT / Inst)}$$

Note: If all Λ -bound variables are unique, we can omit the arbitrariness requirement $a \notin \text{fv}(\Gamma)$. Consider $\Lambda a. \Lambda a. \dots$

Dependency

Simply typed λ -Calculus

Terms can depend on Terms: $(\lambda x.x)$ $(\lambda x.succ\ x)$

System-F

Terms can depend on Terms: $(\lambda x.x)$ $(\lambda x.succ\ x)$

and

Terms can depend on **Types**: $(\Lambda a. \lambda(x : a). x)$
 $(\Lambda a. \Lambda b. \lambda(f : a \rightarrow b). \lambda(x : a). f\ x)$

Note “X depends on Y” \equiv “Y can appear in X”

Type Functions

Consider the following Haskell type definition:

```
type Matrix a = List (List a)
```

Moving the parameter to the right of the equals..

```
type Matrix =  $\lambda a.$  List (List a)
```

Example:

$$\begin{array}{l} \textit{Matrix Char} \xrightarrow{\text{let}} (\lambda a. \textit{List} (\textit{List} a)) \textit{Char} \\ \xrightarrow{\beta} \textit{List} (\textit{List Char}) \end{array}$$

Kinds

What about:

$$\begin{aligned} \textit{Matrix List} &\xrightarrow{\text{let}} (\lambda a. \textit{List} (\textit{List} a)) \textit{List} \\ &\xrightarrow{\beta} \textit{List} (\textit{List} \textit{List}) \\ &\quad ??? \end{aligned}$$

No values exist that have type *List*

A list has to be a list of *something*

Give *Int* the kind $*$ “*Int* is a (simple) type”

Give *List* the kind $* \rightarrow *$ “*List* takes a type and produces a type”

Now *List Char* has kind $*$

List (List List) is mis-kinded

Type functions again

type $Matrix = \lambda(a : *). List (List a)$

System- F_ω

Terms can depend on Terms: $(\lambda x.x)$ $(\lambda x.succ\ x)$

and

Terms can depend on **Types**: $(\Lambda a. \lambda(x : a). x)$
 $(\Lambda a. \Lambda b. \lambda(f : a \rightarrow b). \lambda(x : a). f\ x)$

and

Types can depend on **Types**: $(\lambda(a : *). List (List a))$ $Char$
 $(\lambda(f : * \rightarrow *). \lambda(a : *). f\ a)$

What about Types depending on Terms?

Add a natural number to the type of lists that records how long the list is.

The *Cons* function adds an element to the front of the list, so the resulting list is one element longer than the initial one.

$$\text{Cons} :: \forall(a : *). \forall(n : \text{Nat}). a \rightarrow \text{List } n \ a \rightarrow \text{List } (n + 1) \ a$$

The *append* function joins two lists together, so the length of the resulting list is the sum of the lengths of the first two.

$$\text{append} :: \forall(m : \text{Nat}). \forall(n : \text{Nat}). \text{List } m \ a \rightarrow \text{List } n \ a \rightarrow \text{List } (m + n) \ a$$

Your value, your type.. what's the difference?

The *replicate* function produces a list containing some number of elements.

$$\text{replicate} :: \forall (a : *). \forall (n : \text{Nat}). a \rightarrow \text{List } n a$$

What is the type of the following expression:

$$\text{replicate } \text{Int } (\text{fac } 10) 5$$

Factorial 10 is 3628800, so

$$(\text{replicate } \text{Int } (\text{fac } 10) 5) :: \text{List } 3628800 \text{ Int}$$

To infer the type of $(\text{replicate } \text{Int } (\text{fac } 10) 5)$, the type inferencer has to compute the value of $(\text{fac } 10)$!!!

A is for Amnesia (what is A for?)

- “It is non-linearity and *amnesia* which make a type system work”.
(Moller-Neergaard + Mairson, 2004)
- The type system must be capable of forgetting the details of the program..
- ... otherwise checking for type correctness degenerates to running the program and seeing if it fails.
- Systems which permit types to depend on terms are usually called “dependently typed systems”.
- They tend to be used as core languages for theorem provers, not as general purpose languages.
- Type inference for these systems is not easy.

The Calculus of Constructions (λC)

Terms can depend on Terms: $(\lambda x.x)$ $(\lambda x.succ\ x)$

and

Terms can depend on Types: $(\Lambda a. \lambda(x : a). x)$
 $(\Lambda a. \Lambda b. \lambda(f : a \rightarrow b). \lambda(x : a). f\ x)$

and

Types can depend on Types: $(\lambda(a : *). List\ (List\ a))$
 $(\lambda(f : * \rightarrow *). \lambda(a : *). f\ a)$

and

Types can depend on Terms: $\forall(a : *). \forall(n : Nat). a \rightarrow List\ n\ a$

Note: Presentations of λC can use Π instead of \forall and λ instead of Λ , but we're just trying to get a feel for it at the moment..

The Lambda Cube (Henk Barendregt, 1991)

