COSC 3015: Lecture 3

Lecture given by Prof. Caldwell and scribed by Sunil Kothari

September 16, 2008

1 Recap

Last time we talked about functions and what they are. In Haskell, you can write programs which fail to terminate. We wrote: `:t infinity = infinity a` One of the features of Haskell program is *polymorphism* - an expression can have many types. So , infinity has all types. Often, we want to show functions have similar property.

2 Functions

Q:How to tell if functions are equal ? A:Extensionality (pointwise equality)

Definition 1 *Extensionality If* $f, g \in A \to B$*,*
$f = g$ *if and only if* $\forall x : A.f(x) = g(x)$

This is introduced in discrete maths (COSC 2030). For any type we have three things:

1. constructors
2. destructors
3. type notation

For functions, the constructor is $\backslash x \to b$ is a function with one argument (x here) and defined by the expression b. The destructor for a function is function application - write name of the function next to the argument. For example, $(\backslash x \to b)a$. The type notation is $a \to b$. The notation in Haskell is $\backslash x- > x :: a- > a$.

2.1 Tuples

The constructor for tuples is "()" brackets. The destructors for tuples are *fst,snd* and are defined as follows:

$$fst(a, b) = a$$
$$snd(a, b) = b$$

The type notation is (a, b). In math, you use cartesian product of the form $(a \times b)$.

2.2 Bool

The constructors for *Bool* are $True, False$. There are no destructors. And the type notation is *Bool*.

2.3 List

The constructor for list is [] - Empty list, and "::" - called cons. The destructors are *head* and *tail* and they have the types: `head:: a list -> a`
`tail:: a list -> a`

2.4 Historical Note

About 1930, a guy named Alonzo Church invented a notation for a calculus of functions. This was called the *lambda calculus.*

$\Lambda ::=$	x	$\mid \backslash x.M$	$\mid MN$
	varibales	abstraction	application

where M, N are lambda terms constructed by the above grammar.

Alan Turing - student of Church - showed that Turing Machine are equivalent to lambda calculus. Church's thesis says

> Everything that can in principle be computed can be computed by lambda- term

Haskell Curry was working on logical systems called combinatory logic - turn out closely related to lambda calculus. Why was everyone working on it ? Around 1920's people were wondering about what can be computed.

Curry noted that a propositional logic formula is valid (true) if and only if the type associated with the formula is "inhabited" by the λ-term. This is called the *Curry-Howard isomorphism.*

So what's the translation ?

$$\begin{array}{lcl} \overline{A \Rightarrow B} & = & \overline{A} \rightarrow \overline{B} \\ \overline{A \wedge B} & = & \overline{(A, B)} \\ \overline{A} & = & \text{A} \end{array}$$

2.5 Currying and Uncurrying

Curry noticed the following

$$((A \wedge B) \Rightarrow C) \Rightarrow (A \Rightarrow (B \Rightarrow C)) \text{ (Currying)}$$
$$(A \wedge (B \Rightarrow C)) \Rightarrow ((A \wedge B) \Rightarrow C)) \text{ (uncurrying)}$$

We can do the translation as follows:

$$\begin{aligned} &\overline{((A \wedge B \Rightarrow C) \Rightarrow (A \Rightarrow (B \Rightarrow C))} \\ = &\overline{((A \wedge B) \Rightarrow C)} \Rightarrow \overline{(A \to (B \Rightarrow C))} \\ = &\vdots \\ = &((A, B) \to C) \to (A \to (B \to C)) \end{aligned}$$

What is the function that has this type ? Approach - start labelling this arguments $curry :: ((A, B) \to C) \to (A \to (B \to C))$
`:t curry f A ->(B ->C)`
Suppose we are given $f :: (A, B) \to C$, $x :: A$, $y :: B$. How can we get something of type C ?

$$curry\ f\ x\ y = f\ (x, y)$$

To do this in Haskell, do the following

1. create a file curry.hs
2. put the definition of the curry function in to the file
3. do ¿:t curry

This will give the following output `:t Main.curry ((a,b)-> c -> a -> b ->c`
Note that arrow associates to the right by default so, $a \to b \to c$ means $a \to (b \to c)$.
This is like looking at a type and figuring out the function. But you should be wondering how does the compiler looks at a function and figures out the type ? There is a type inference algorithm that looks at the context and figures out the type.
So, what is uncurry ? It has the type $uncurry :: (a \to b \to c) \to (a, b) \to c$. We can define uncurry as:

$$uncurry\ f\ p = f\ (fst\ p)\ (snd\ p)$$

We can define *add* as: `add x y = x + y`

```
>:t add
add:: Num a => (a,a) -> a

>:t curry add
curry add:: Num a => a -> a ->a
```

Suppose Haskell only has Int's, then

```
>t: add
add:: (Int,Int) -> Int
>t: curry add
 curry add: Int -> Int -> Int
 >t: addc
 addc:: Int -> Int -> Int
```

We can define curry in a new way as :

$$curry f = \backslash x \rightarrow \backslash y \rightarrow f\ (x, y)\ uncurry f = \backslash p \rightarrow f\ (fst\ p)\ (snd\ p)$$
$$uncurry f = \backslash (x, y) \rightarrow f\ x\ y$$

We can also do the following in Haskell

```
addc 5:: Int -> Int
addc 5 5 :: Int
```