COSC 3015: Lecture 4

Lecture given by Prof. Caldwell and scribed by Sunil Kothari

September 16, 2008

1 Recap

Somebody asked about what's going on ? So here is the response. Recall functions from dom a to codomain b: $a \rightarrow b$ where, a and b are type variables and they can stand for any type. Suppose we instantiate b with the type $Int \rightarrow String$ Then if $f :: a \rightarrow (Int \rightarrow String)$ so f maps things of type a to functions of type $Int \rightarrow String$. That means if $x :: a$ then
$fx :: Int \rightarrow String$

The result of applying the function f to the argument x. Here's a concrete example:

Example 1 *$f\ k = \backslash\ m- > m + k$ Here's another equivalent definition $f\ k\ m = m + k$ So the type of f is $Num\ a \Rightarrow a \rightarrow (a \rightarrow a)$ What does f look like as a set of pairs ?*

$f\ 0 = \backslash m \rightarrow m + 0$
$f\ 1 = \backslash m \rightarrow m + 1$
$f\ (-1) = \backslash m \rightarrow m + (-1)$

In the last HW notes f was called plus.

In Haskell, the default is curried.

```
>:t plus (x,y) = x + y
  Num a => (a,a) -> a
>:t  plusc x y = x + y
  Num a => a -> (a -> a)
```

input	output
< 0	$,<< 0, 0 >, < 1, 1 >< -1, -1 >, \cdots >>$
< 1	$,<< 0, 1 >, < 1, 2 >< -1, 0 >, \cdots >>$
$\vdots$	

```
>:t plusc 5
   Int -> Int\\
>:t plus 5
  ! splat - type error
```

2 Curry and Uncurry

So *curry* can be defined as :
$curryf = \backslash x \rightarrow \backslash y \rightarrow f\ (x, y)$ Here's an alternative definition $curryf\ x\ y = f\ (x, y)$ so the type of f is $((a, b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c$.

Remember arrow associates to the right. so, $a \rightarrow b \rightarrow c$ means $a \rightarrow (b \rightarrow c)$. Also, function application associates to the left. So - $f\ x\ y$ means $(f\ x)\ y$ and not $f\ (x\ y)$

By default, type checker is gonna get rid of as many paranthesis as it can. Normally, pairs are written as catesian product but in Haskell both type and terms are represented by the same notation i.e. depending on the context (a,b) might be a product type or a pair of terms.

And *uncurry* could be written as: $uncurry\ f = \backslash p \rightarrow f\ (fst\ p)\ (snd\ p)$ ¿:t fst Here's an anternative definition $uncurry\ f = \backslash (x, y) \rightarrow f\ x\ y$ Here's another definition $uncurry\ f\ (x, y) = f\ x\ y$

So the type of uncurry is $(a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c$. Again *curry* and *uncurry* are fundamental notions. By default, things are written in curried form.

3 Function composition

Consider three types A, B, C with $f : A \rightarrow B, g : B \rightarrow C$ so *function composition* is defined as $(g \circ f)(x) = g(fx)$ The type of $g \circ f$ is $a \rightarrow c$.

To make it syntactic valid Haskell we can write backquotes around it. For example '$\circ$'. In Haskell, we can write is as $g.f$. So $g.f = \backslash x- > g(fx)$. so - "." is a special infix operator denoting function composition.

There is a Haskell notation for turning infix operator into a prefix operator. If $\oplus$ is an arbitrary binary argument taking a and b to c - then $(\oplus) :: a \rightarrow b \rightarrow c$.

So, what is the type of function composition $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

Example 2 $add1x = x + 1\ times2x = 2 * x\ add1.times2 \rightsquigarrow \backslash y- > 2 * y + 1$ $times2.add1 \rightsquigarrow \backslash y- > 2 * (y + 1)$

How can we calculate this ? what is the type of $add1.times2$? It's $Int \rightarrow Int$. So - choose an arbitrary int (say x) and then calculate with $(add1.times2)x \rightsquigarrow add1(times2x) \rightsquigarrow add1(2 * x) \rightsquigarrow (2 * x) + 1$

Now, consider *show*. One of the types of *show* is:: $Int \rightarrow String$. Consider $length :: [a] \rightarrow Int$. So, what is *length.show* - calculates the length of the string representation of its argument.

The type of *length.show* is `(Show a) => a -> Int`

3.1 Reasoning about functions

Consider the identity function $idx = x$.

$x + 0 = x$ 0 is the right identity for +
$1 * x = x$ 1 is the left idenenity for $*$

id is the left and right identity for the function composition. So, we are saying $f.id = id.f = f$

Recall that if $f, g :: a \to b$ then $f = g$ iff $\forall x : a.f\ x = g\ x$. So that the type of $f.id$ and $id.f$ is $a \to b$

To show $f = f.id$ show $\forall x : a.f\ x = f.id(x)$ Choose arb. $x :: a$ and show $fx = f.id(x)$.

Starting on the right $(f.id)\ x \quad = \quad f\ (id\ x)$
$\qquad = \quad f\ x$

also, $(id.f)\ x \quad = \quad id\ (f\ x)$
$\qquad = \quad f\ x$

What good is knowing what the identity of an operation is ?
Consider sum that sums all elemnets of the list

$sum[] = 0$
$sumh :: t = h + sumt$

$prod[] = 1$
$prodh :: t = h * (prodt)$

$compose[] = id$
$compose(h : t) = h.composet$

So $compose[f1, f2, f3] = f1.(f2.(f3.id)) = f1.f2.f3$