COSC 3015: Lecture 2

Lecture given by Prof. Caldwell and scribed by Sunil Kothari

September 16, 2008

1 Recap

We talked about functional Programming vs. Logic programming vs. Imperative Programming. The first two categories are *declarative*.

```
qsort [] = []
qsort (h:hs) = qsort smaller ++ [h] ++ qsort larger
    where smaller = [a | a<- hs, a <= h]
          larger = [b | b <- hs, b > h]
```

This is more declarative than what you would do in Java or C++. This is a very natural description of the problem. The functional languages like ML, Haskell are *strongly typed languages*. They have *type inference*. Type inference works well with the functional programming languages. For example,

```
>:t qsort
 Ord a => [a] -> [a]
```

where,

1. `a` is a type variable
2. `[a]` is the type of lists containing elements of type `a`.

We can run `qsort` as:

```
> qsort (reverse [1 ..10])
> [1,2,3,4,5,6,7,8,9,10]

> qsort (reverse ['a'..'z'])
> "abcdefghijklmnopqrstuvwxyz"
```

2 Interaction with the programming environment

⋮

3 Functions in Haskell

Q:What's an example of things that cannot be compared ?
A:Functions.

3.1 Notation for functions

Notation that allows you to write down a function but you don't have to give a name to a function. Haskell expression or functions: *add1 x = x + 1* ≡ `(\x -> x + 1)` where,

- (x+1) is the body of a function

We cannot print a function (in a resonable way) but we can get its type. but we can apply the function so `(\ x -> x+ 1) 5)` returns 6 in Haskell. So, how does a function gets evaluated ?

$$\begin{array}{rcl} (\lambda x.x+1)5 & = & (x+1)[x := 5]) \\ & \hookrightarrow & (x[x := 5] + 1[x := 5]) \\ & \hookrightarrow & (5+1) \\ & \hookrightarrow & 6 \end{array}$$

where, *x[x:=5]* is a substitution operation.
map is a higher-order function. Its type is `map :: (a -> b) -> [a] -> [b]`

How do we know which ways the arrows associate ? arrows associate to the right so $a \to b \to c$ is $a \to (b \to c)$

map is defined as :

$$\begin{array}{lcl} map\ f\ [] & = & [] \\ map\ f\ (h : hs) & = & f\ h : map\ f\ hs \end{array}$$

In Haskell, we can use *map* as:

```
Main> ( \x-> x + 1) [1..10]
[2,3,4,5,6,7,8,9,10,11]

Main> :t (\x -> x + 1)
\x -> x + 1 :: Num a => a + 1

Main> :t infinity
infinity ::a
```

So, let's back up on what is a function.

3.2 Cartesian Product

Remeber $A \times B = \{< a, b >| \ a \in A \wedge b \in B\}$
$\{1, 2, 3\} \times \{a, b\} = \{< 1, a >, < 1, b >, < 2, a >, < 2, b >, < 3, a >, < 3, b >\}$
$\{a, b\} \times \{1, 2, 3\} = \{< a, 1 >, < a, 2 >, < a, 3 >, < b, 1 >, < b, 2 >, < b, 3 >\}$

In Haskell, (a,b) - is a pair if a::A and b::B (A,B) - is the cartesian product - if A and B are types

```
Main> (1,"abc")
(1,"abc")::(Integer,[Char])
Main>
```

R is a binary relation on A and B if $R \subseteq A \times B$.
For example, $A = \{1, 2, 3\}$
$R \subseteq A \times A$
$R = \{< 1, 1 >, < 1, 2 >, < 1, 3 >\}$

The proeprty of being a function, called functionality for $f \subseteq A \times B$, is

$$\forall x : A, \forall y, z : B.(f(x) = y \wedge f(x) = z) \Rightarrow y = z$$

A function is *total* if

$$\forall x : A, \exists y : B.f(x) = y$$

```
Main> :t ( \x ->   if x <0 then infinity else -x )
(\x ->   if x <0 then infinity else -x)::(Num a, Ord a) => a -> a
Main> (\x ->   if x <0 then infinity else -x)(-5)
{Interrupted!}
```

When a system gives a type it is saying that if the function halts it will have the type but if it does not halt then it might be undefined but still it has a type.

So, when we write $f :: a \to b$ as a Haskell type- we mean that f is a function (not necessarily total) from domain a to range b.

Here's an example of a function from any type to any other type.

```
Main> :t (\x -> infinity)
 \x -> infinity :: a -> b
```

A lot of Haskell-related topics are on YouTube - some are Google tech talks.