COSC 3015: Lecture 1

Lectured given by Prof. Caldwell and scribed by Sunil Kothari

August 26, 2008

1 Books and Programming Environments

1.1 Books for functional programming

Haskell is a lazy language. Chris Okasaki looked at imperative data structures in a functional setting. For example, binomial trees, splay trees, red-black trees, etc.

1.2 Higher-order Perl by Mark Jason Dominus

Lisp has higher-order functions. That means, functions are first class members. You have C++ pointers which pint to a piece of code. Java has lambda expressions now. There are links to the author's lecture on the web. Perl6 is being implemented in Haskell.

1.3 Programming Environments

There are two implementations:

- Hugs - Haskell Implementation, choose the editor you want to use
- GHCi - Glasgow Haskell Compiler
- Emacs - Text editor

Dr. Caldwell uses emacs - as a text editor. We will use interpreter even though Haskell programs can be compiled. Let's go for Hugs then.

2 What is functional programming ?

It's defined by other programming paradigms.

2.1 Imperative Programming

- Fortran, C , C++, Ada, Pascal, Java, Basic, Perl.
- Inherent in this model of programming is the state based computation; command based.
- A computation is a sequence of memory states.
  $S_0 \rightarrow S_1 \rightarrow S_2 - > ...$

Object-oriented programming can also be looked as imperative programming. It's a kind of data abstraction mechanism built upon imperative (or functional) programming language.
- *e.g.* OCaml - Object oriented Caml a functional PL.

2.2 Logic Programming

Prolog - a variant is Lambda Prolog (functional logical hybrid). A *program* is a logical description of a problem to be solved. *Computation* is a form of search for a solution. You have to write description of a problem in the form of *Horn clauses*. It's a kind of a restricted logic.

2.3 Functional Programming

Functional programming languages are expression based. You write an expression describing the desired solution – and computation is the process of evaluating the expression. Here's a Haskell program for Quick sort.

2.3.1 Lists in Haskell

There are two constructors.

- [] - empty list
- $h : hs$ - cons - that sticks the value created by h onto the left hand of the list hs.
  For example,

  The usage is as follows:

- $1 : [] \hookrightarrow [1]$
- $1 : (2 : []) \hookrightarrow [1, 2]$
- $2 : (1 : []) \hookrightarrow [2, 1]$
- $"xy" : ("zzy"' : []) \hookrightarrow ["xy", "zzy"]$

2.3.2 Append

Append (++)
$[1,2] + +[3,4] \hookrightarrow [1,2,3,4]$

2.4 Quick Sort

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

$$\begin{aligned} qsort[2,1] &= qsort(2:[1]) \\ &\hookrightarrow (qsort[1] + +[2] + +qsort[]) \\ &\hookrightarrow (qsort[] + +[1] + +qsort[]) + +[2] + +[]) \\ &\hookrightarrow (qsort[] + +[1] + +[]) + +[2]) \\ &\hookrightarrow [1] + +[2] \\ &\hookrightarrow [1,2] \end{aligned}$$

Note that List comprehensions are Haskell macros.

2.5 Typed vs. untyped

Functional programming languages can be categorized as:

- Typed - Haskell, OCaml, ML, F# (Supports type inference).
- Untyped - Lisp, Scheme

2.6 Type Inference

$(\backslash x- > x + 5)$ is a Haskell expression for the function that adds 5 to the argument x. It can also be written as $add5x = x + 5$. The compiler figures out the type of the expression if it has a type.

```
>:t add 5
> (Num a) => a -> a
```

The Num above is a type class.

The type for `qsort` is

```
>:t qsort
>(Ord a) => [a] -> [a].
```

Here `Ord` is a type class.