COSC 3015: Midterm Review

Lecture given by Prof. Caldwell and scribed by Sunil Kothari

October 14, 2008

1 Review

We will have a midterm exam on Thursday. The exam will be closed book.

HW1 was a warm up and we looked at Qsort and had you analyze some code. Something like that might not be a bad question.

HW3 was like beginning of the real functional programming.

1.1 Lambda terms - Higher Order functions

$\backslash x - > e$ where e is a haskell expression
$f\ x\ y = e$

is same as
$\backslash f\ x \to \backslash y \to e$

is same as
$f = \backslash x \to \backslash y \to e$

For example, $fxy = x + y$
$f\ x = \backslash y \to x + y$

is same as
$f = \backslash x \to \backslash y \to x + y$
Given,
$plusc\ x\ y = x + y$

```
>:t plusc 5
Num a => a -> a
```

1.2 HW3 - curry and uncurry

$curry :: ((a, b) \to c) \to (a \to (b \to c))$
$curry\ f\ x\ y = f\ (x, y)$

$uncurry :: (a \to b \to c) \to ((a, b) \to c)$
$uncurry\ f\ (x, y) = f\ x\ y$

1.3 HW4- Extensionality

Extensionality - equality between functions. There is no general algorithm to decide when two functions are equal. It's equivalent to halting problem but you can look at individual programs and see if it halts. So same is the case with equality on functions.
If $f, g : a \to b$ then
$f = g \overset{\text{def}}{=} \forall x : a.fx = gx$

Function composition
$f \circ g = f\ (g\ x)$

So what is the type of function composition
$(\circ) :: (b \to c) \to (a \to b) \to a \to c$

In the HW, we had to prove that $curry\ plus = plusc$

$curryplus :: Int \to (Int \to Int)$
$plusc :: Int \to (Int \to Int)$

Q: How do we show that the two are equal ?
A: Use extensionality.

Proof. Choose arb. $x \in Int$ and show
$curry\ plus\ x = plusc\ x$

Choose an arb. $y \in Int$ and show
$curry\ plus\ x\ y = plusc\ x\ y$

On the left, $curry\ plusc\ x\ y = plus\ (x, y) = x + y$
On the right, $plusc\ x\ y = plus\ (x, y) = x + y$

□

Note that we had to apply extensionality twice since we have a function of two arguments. If we apply extensionality once, we are still showing two functions are equal. Note that we cannot choose arbitrarily any y but something different from x.
And in the HW I asked you to write flip.
$flip :: (a \to b \to c) \to b \to a \to c$
$flip\ x\ y = f\ y\ x$

We can also think this as operation on function
$flip\ f = \backslash x \rightarrow \backslash y \rightarrow f\ y\ x$

Sometimes you will see an example when you asked Haskell about a type flip would show up.

1.4 HW5

Remember, *toEnum* and *fromEnum* are consistently wrong in the book.

```
class Enum a where
  toEnum:: Int -> a
  fromEnum:: a -> int
```

We have this property $toEnum \circ fromEnum = \backslash x \rightarrow x$

This is an expectation whenever you instantiate Enum class.

And then the HW asked you to write Enum for booleans

```
data MyBool = TT | FF deriving Show

instance Enum MyBool where
  toEnum 0 = FF
  toEnum 1 = TT
  fromEnum TT = 1
  fromEnum FF = 0
```

1.5 HW6 - Show functions and instantiating a type class

It had to do with show functions. Given the date data type, write a a show function to display dates in a fancy form. So we had
$data\ Date = DMY(Int, Int, Int)$
The show function was supposed to do something like this:

```
>DMY(2,1,2008)
  2nd January 2008
```

The whole point was that you can overload the show function in the way you like. This is a powerful thing. The type class is an elegant implementation of ad-hoc polymorphism. The interpreter/compiler figures at compile-time which function to call based on the types. Java generics is designed by the same person and works similarly.

1.6 HW7 - Induction (on Nat)

$data\ Nat = Zero \mid Succ\ Nat$
$\quad (+) :: Nat \rightarrow Nat \rightarrow Nat$
$m + zero = m$
$m + (Succ\ k) = Succ\ (m + k)$

To show that some property P holds for all finite nats (i.e. not involving $\perp$):

- Show P is true for Zero
- Assuming P is true for K, show P is true for (Succ k)

To show for all Nats

- Show $P(\perp)$

Q: What type of induction is that ?
A: It is really ordinarily mathematical induction but in fact is structural induction - more general in the same way.

There will be probably a question on induction principle.

1.7 HW8

In Haskell, lists are defined as
$data\ [a] = [] \mid (::)\ a\ [a]$

Or alternatively,
$data\ List\ a = Nil \mid Cons\ a\ (List\ a)$

In the HW the lists are defined as :
$data\ Liste\ a = Nil \mid Snoc\ (Liste\ a)\ a$

It might be interesting to see what is induction principle for Lists formed from snoc ?
$convert :: (Listea) \rightarrow [a]$

1.8 HW9 - Writing list functions

Given type and desired behavior, define the function.

1.9 HW 10 - Modeling finite functions as lists

Implement finite functions using lists and instantiate the Eq type class for this data type so that $f == g$ returns true iff f and g are functional, and they really are equal.

For finite functions, we can do this kind of thing but not for infinite functions. If we use equality on Lists, then the finite function equality would be slightly different.
$FinFun[(1,2)(2,1)] == FinFun[(2,1),(1,2)]$

1.10 HW 11

Lists are defined as:
$data\ List\ a = Nil \mid Cons\ a\ (List\ a)$

Structural Induction on lists
$(P(Nil) \wedge \quad \forall x : a, \forall m : List\ a.\ P(m) \Rightarrow P(Cons\ a\ m))$
$\Rightarrow \forall l : List\ a.\ P(l)$

This can also be written as:

1. $P(Nil)$.
2. Assuming $P(m)$ show $P(x :: m)$ for arb. x and m.

A binary tree is defined as:
$data\ Btree\ a = Leaf\ a \mid Fork\ (Btree\ a)\ (Btree\ a)$

1.11 HW 12

$data\ Btree\ a = Leaf\ a \mid Fork\ (Btree\ a)\ (Btree\ a)$

$(\forall x : a, P(Leaf\ x) \wedge \forall t1, t2 : (Btree\ a), P(t1) \wedge P(t2) \Rightarrow P(Fork\ t1\ t2))$
$\Rightarrow \forall t : (Btree\ a), P(t)$

1.12 HW13

$data\ (Ord\ a) => Stree\ a = Null|Fork\ (Stree\ a)\ a\ (Stree\ a)$

So what does the induction principle look like ?
$(P(Null) \wedge \forall x : a, \forall t1, t2 : (Stree\ a), P(t1) \wedge P(t2) \Rightarrow P(Fork\ t1\ x\ t2))$
$\Rightarrow \forall t : Stree\ a, P(t)$.

So what about map ?
$map\ f\ Null = Null$
$map\ f\ (fork\ t1\ x\ t2) = Fork\ (map\ f\ t1)\ (f\ x)\ (map\ f\ t2)$

fold is given as:
$fold\ f\ g\ Null = g$
$fold\ f\ g\ (fold\ t1\ x\ t2) = g\ (fold\ f\ g\ t1)(fold\ f\ g\ t2)$

For *Stree* the fold function is
$fold\ f\ g\ (Leaf\ x) = f\ x$
$fold\ f\ g\ (Fork\ t1\ t2) = g\ (fold\ f\ g\ t1)\ (fold\ f\ g\ t2)$