COSC 3015: Lecture 6

Lecture given by Prof. Caldwell and scribed by Sunil Kothari

September 16, 2008

1 Recap -HW

`data Day = Sun| Mon | Tue | Wed| Thu | Fri |Sat deriving (Eq, Ord,Enum).` By default, the enums start from 0. The *Enums* type class is as follows:

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

We can create an instance for Day type as follows:

```
instance Enum Day  where
   toEnum
     | 0 = Sun
     | 1 = Mon

    fromEnum
     | Sat = 0
     | Sun = 1
```

In Bird's book, the *toEnum* has it the other way. should be that

```
toEnum(fromEnum k) = k
fromEnum (toEnum k) = k
```

These laws cannot be checked by type checker but would require theorem proving. Since we have derived from Eq type class we can do the following

```
> Sun == Mon
> False
```

with day not declared an instance of the Eq type class

```
 > Sun == Mon
ERROR - Unresolved overloading
*** Type       : Eq Day => Bool
*** Expression : Sun == Mon
```

In general - if f is a function f:$a \rightarrow b \rightarrow c$ then ‘f‘ is an infix version of f.
For exmaple, we can define *dayAfter* and *dayBefore* functions as:

```
Main> :t dayAfter
dayAfter :: (Enum a, Enum b) => b -> a
Main> :t dayBefore
dayBefore :: (Enum a, Enum b) => b -> a
```

Type classes provide an elegant implementation of so-called ad-hoc polymorphism - where a simple name returns to many objects - which -in haskell are distinguished by their type.

```
Main> dayAfter Tue::Day
Wed
```

Another issue is whether type annotations are needed.

2 Extra

Dr. Caldwell did not do this in the class but asked me to include in the class notes. Note that this requires Haskell extensions. We can define *nextEnum* for the Day as follows:

nextEnum (e::Day) = toEnum(mod ((fromEnum e) +1) (fromEnum (maxBound::Day) + 1))
Note the type of this function is :

```
Main> :t nextEnum
nextEnum :: Enum a => Day -> a
```

This function can be evaluates as follows:

```
Main> nextEnum Sun
ERROR - Unresolved overloading
*** Type       : Enum a => a
*** Expression : nextEnum Sun

Main> nextEnum Sun::Day
Mon
```

We can define a generic version of it as follows:

```
nextEnum1 e = toEnum( mod ((fromEnum e) +1)  (fromEnum (k maxBound e) + 1))
                  where  k:: a-> a -> a
                         k x y = x

Main> nextEnum1 Sun::Day
Mon
```

Here, we don't need a particular type. So using it we can define another function which enumerates to a different type. The function *nexEnum'* increments the integer value associated with element e1 of an enumerated type then maps that element to the associated element of enumerated type of e2 modulo the size of the type of e2. The element e2 is ANY element of the target type, it does not matter which one. e2 is used to coerce maxBound to the proper type.

```
nextEnum' e1 e2 =
  k (toEnum ((fromEnum e1 + 1) `mod` (fromEnum max + 1))) e2
    where k :: a -> a -> a
          k x y = x
          max = k maxBound e2
```

Here's the interaction with Haskell interpreter for the above function:

```
Main> :t nextEnum'
nextEnum' :: (Enum a, Enum b, Bounded a) => b -> a -> a
Main> nextEnum' Wed True
False
Main> nextEnum' Wed False
False
Main> nextEnum' Tue False
True
Main> nextEnum' Tue True
True
Main>
```

The function *nextEnum* defined above can then be defined as :
nextEnum e = nextEnum' e e
We can use the new function as:

```
 Main> nextEnum Tue
Wed
Main> nextEnum Wed
Thu
Main>
```