CS4012 Topics in Functional Programming

Glenn Strong

Last year in CS3016 you learned to program using "Haskell"

Now we will build on that to explore deeper

- Programming for problem solving
  - Parallelism and Concurrency
  - IO and State management
  - Reactive functional programming
  - Using Haskell language extensions
- Type inference (What? How?? Why???)
- Beyond Haskell typing: Dependent types (featuring Agda and Idris)
- Functional programming techniques
  - Domain specific languages
  - GADT's
  - Monad transformers
  - Arrows
- Hybrid languages (Scala)

Course structure

- Three hours per week
- Two traditional lectures; one discussion hour (including in-class presentations)

Assessment

- Two programming assignments (25%)
- Readings and in-class group presentations (5%)
- Examination (70%)

Refreshing Haskell

- Haskell is a *pure* language
- Haskell is a *lazy* language
- Haskell features an advanced type system (higher-order, polymorphic, ...)

Remember the basic structure of a Haskell program; we define sets of functions:

```
sum          :: Num a => [a] -> a
sum []       = 0
sum (x:xs)   = x + sum xs
```

Functions can be defined with patterns or guards to select multiple cases. Types are inferred (and can also be supplied explicitly)

Types can be created using data declarations

```
data Tree a = Empty
            | Leaf a
            | Branch (Tree a) (Tree a)
```

We can create functions over those types

```
tmap f Empty        = Empty
tmap f (Leaf x)     = Leaf (f x)
tmap f (Branch l r) = Branch (tmap f l) (tmap f r)
```

Functions are higher-order values and can be partially applied ('curried') to produce new functions

```
threshold t v | v < t     = Leaf v
              | otherwise = Leaf t

boundTr t = tmap (threshold 0.5) t
```

The language is lazy, so we may be able to avoid long computations

```
blowup n | n <= 1    = 1
         | otherwise = blowup (n-1) * blowup (n-2)

replaceLeaves Empty t        = Empty
replaceLeaves (Leaf x) t     = Leaf x
replaceLeaves (Branch l r) t = Branch (replaceLeaves l t)
                                      (replaceLeaves r t)
```

Meaning this: `replaceLeaves Empty (blowup 1000)` does not take as long to compute as you'd think

Functions and Actions are differently typed!

```
f :: String -> Int

g :: String -> IO Int
```

When the `IO` action is run, anything could happen.

But the `IO String` value is a first-class value in the normal way. It is the way it's used (typically by the special function `main`) that causes the actions.

To see that IO actions are first class:

```
print :: Show a => a -> IO ()

printAll = map print [1..100]
```

`printAll` has type `[ IO () ]`

```
printall2= sequence_ printAll

sequence_ :: [IO a] -> IO ()
```

IO actions are combined using *monad* combinators:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```

We can define many useful instances of Monad!

```
instance Monad Tree where
   return a = Leaf a

   Nil >>= f = Nil
   Leaf a >>= f = f a
   Branch u v >>= f = Branch (u >= f ) (v >= f )
```