Lecture 18

Functions with varible number of arguments.

Consider the printf function in C. The number of arguments it take depends on the format string that is provided to it. Can one have similar functions in Haskell ?

Let us analyse the type of `printf` in various expressions. In the expression `printf "Hello"`, `printf` should have type `String -> IO ()` where as in an expression like `printf "Hello %s" "world"`, it should have the type `String -> String -> IO()`. This chamelon like behaviour is what we need to define printf.

In this lecture we show how to hack Haskell's type class mechanism to simulate such variable argument functions. The stuff we will do is beyond Haskell 98 and hence we need to enable certain extensison. We do it via the following compiler pragmas. The very first set of lines in the source code if they are comments that start with {`-#` and end with `#-`} are treated as compiler pragmas. In this case we want to allow the extensions FlexibleInstances OverlappingInstances and InvoherentInstances. One can also give these extensions at compile time; to enable the extension Foo use the flag -XFoo at compile time.

Ofcourse it is difficult to remember all the extensions that you need for this particular code to work. The easiest way to know what extensions are required is to go ahead and use it in your file. GHC will warn you with an appropriate error message if it expects an extension.

```
> {-# LANGUAGE FlexibleInstances        #-}
> {-# LANGUAGE OverlappingInstances     #-}
> {-# LANGUAGE IncoherentInstances      #-}
```

```
>
> module Printf where
```

Recall that an expression using printf would look something like the one below.

```
printf fmt e1 e2 ... em
```

We want our type system to infer `IO ()` for this expression. The type of `printf` in such a case should be `String -> t1 -> ... -> tm -> IO ()` where `ti` is the type of `ei`. The main idea is to define a type classes say `Printf` whose instances are precisely those types that are of the form `t1 -> ... -> tm -> IO ()`. As a base case we will define an instance for `IO ()`. We will then inductively define it for other types.

The definition of the type class `Printf` and therefore `printf` will be easier if we first declare a data type for formating.

```
>
> data Format = L String -- ^ A literal string
>             | S        -- ^ %s
>             | G        -- ^ %g for instances of Show
>
>
```

We would rather work with the more convenient `[Format]` instead of the format string. Writing a function to convert from format string to `[Format]` is not too difficult.

Now for the actual definition of the `Printf` class.

```
> class Printf s where
>       printH :: IO [Format] -> s
>
```

The member function `printH` of the class `Printf` is a helper function that will lead us to definition of `printf`. Intutively it takes as argument an IO action which prints whatever arguments it has seen so far and returns the rest of the formating action required to carry out s.

We will define the `Printf` instances for each of the type `t1 -> ... tm -> IO ()` inductively. The base case is when there are no arguments.

```
> instance Printf (IO ()) where
>       printH fmtIO  = do fmt <- printLit fmtIO
>                          if null fmt then return ()
>                             else fail "Too few arguments provided"
>
```

Now the inductive instance declaration.

```
> instance (Printf s, Show a) => Printf (a -> s) where
>          printH fmtIO a = printH action
>                 where action = do rfmt <- printLit fmtIO
>                                   case rfmt of
>                                        []     -> fail "Too many argument"
>                                        (_:xs) -> do putStr $ show a; return xs
```

We give a specialised instance for string which depending on whether the formating character is %s or %g uses putStr or print.

```
>
> instance Printf s => Printf (String -> s) where
>          printH fmtIO s = printH action
>                 where action = do rfmt <- printLit fmtIO
>                                   case rfmt of
>                                        [] -> fail "Too many arguments"
>                                        (G:xs) -> do putStr $ show s; return xs
>                                        (S:xs) -> do putStr s; return xs
>
```

What is remaining is to define printLit

```
>
> printLit fmtIO = do fmt <- fmtIO
>                     let (lits,rest) = span isLit fmt
>                         in do sequence_ [putStr x | L x <- lits]
>                               return rest
> isLit (L _) = True
> isLit  _    = False
>
```

We can now define printf

```
> printf :: Printf s => String -> s
> printf = printH . format
>
> format :: String -> IO [Format]
> format ""          = return []
> format "%"         = fail "incomplete format string"
> format ('%':x:xs) = do fmts <- format xs
>                        case x of
>                             's' -> return (S:fmts)
>                             'g' -> return (G:fmts)
>                             '%' -> return (L "%" :fmts)
>                             _   -> fail ("bad formating character %" ++ [x])
> format zs = let (lit,xs) = span (/='%') zs
>             in do fmts <- format xs
>                   return (L lit:fmts)
>
```