CSE399: Advanced Programming

Handout 3

Polymorphism

The Length Function is Polymorphic

```
length        :: [a] -> Int
length []      = 0
length (x:xs) = 1 + length xs
```

The “a” in the type of `length` is a placeholder that can be replaced with any type when `length` is applied.

```
length [1,2,3]          ⇒  3
length ["a","b","c"]    ⇒  3
length [[1],[],[2,3]]   ⇒  3
```

Many of Haskell's predefined functions are polymorphic

```
(++) :: [a] -> [a] -> [a]
id   :: a -> a
head :: [a] -> a
tail :: [a] -> [a]
[]   :: [a]                -- interesting!
```

Quick check: what is the type of `tag1`?

```
tag1 x = (1,x)
```

Polymorphic Data Structures

Polymorphic functions — functions that can operate on any type of data — are often associated with **polymorphic data structures** — structures that can contain any type of data.

The previous examples involved lists and tuples. In particular, here are the types of the list and tuple constructors:

```
(:) :: a -> [a] -> [a]
(,) :: a -> b -> (a,b)
```

Note the way that the tupling operator is identified, which generalizes to `(,,)`, `(,,,)`, etc. When we write `(1,2,3,4)`, we really mean `(1,(2,(3,4)))`.

We can also define new polymorphic data structures...

A User-Defined Polymorphic Data Structure

The type variable a on the left-hand-side of the = tells Haskell that Maybe is a polymorphic data type:

```
data Maybe a = Nothing | Just a
```

Note the types of the constructors:

```
Nothing :: Maybe a
Just :: a -> Maybe a
```

Thus:

```
Just 3        :: Maybe Int
Just "x"      :: Maybe String
Just (3,True) :: Maybe (Int,Bool)
Just (Just 1) :: Maybe (Maybe Int)
```

Maybe May Be Useful

The most common use of Maybe is with a function that "may" return a useful value, but may also fail.

For example, the division operator `div` in Haskell will cause a run-time error if its second argument is zero. Thus we may wish to define a safe division function, as follows:

```
safeDivide :: Int -> Int -> Maybe Int
safeDivide x 0 = Nothing
safeDivide x y = Just (x `div` y)
```

Higher-Order Functions

Abstraction Over Recursive Definitions

Recall from Section 4.1:

```
transList          :: [Vertex] -> [Point]
transList []        = []
transList (p:ps)    = trans p : transList ps
```

(where `trans` converts ordinary cartesian coordinates into screen coordinates).

Also, from Chapter 3:

```
putCharList        :: [Char] -> [IO ()]
putCharList []      = []
putCharList (c:cs) = putChar c : putCharList cs
```

These definitions are very similar. Indeed, the only thing different about them (besides the variable names) is the function `trans` vs. the function `putChar`.

We can use the abstraction principle to take advantage of this regularity.

Abstraction Yields `map`

Since `trans` and `putChar` are the differing elements, they should be arguments to the abstraction. In other words, we would like to define a function — let's call it `map` — such that `map trans` behaves like `transList` and `map putChar` behaves like `putCharList`.

No problem:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Now it is easy to redefine `transList` and `putCharList` in terms of `map`:

```
transList   xs = map trans   xs
putCharList cs = map putChar cs
```

`map` is Polymorphic

The great thing about `map` is that it is polymorphic. Its most general (or principal) type is:

```
map :: (a->b) -> [a] -> [b]
```

Whatever type is instantiated for the type variable `a` must be the same at both instances of `a`, and similarly for `b`.

For example, since `trans :: Vertex -> Point`, we have

```
map trans :: [Vertex] -> [Point]
```

and since `putChar :: Char -> IO ()`,

```
map putChar :: [Char] -> [IO ()]
```

Digression: Arithmetic Sequences

Haskell provides a convenient special syntax for lists of numbers obeying simple rules:

```
[1 .. 6]              ⇒   [1,2,3,4,5,6]
[1,3 .. 9]            ⇒   [1,3,5,7,9]
[5,4 .. 1]            ⇒   [5,4,3,2,1]
[2.4, 2.1 .. 0.3]     ⇒   [2.4, 2.1, 3.8, 3.5, etc.]
```

Another Example of Map

```
circles :: [Shape]
circles = map circle [2.4, 2.1 .. 0.3]
```

Now let's draw them...

Digression: zipping

Another useful higher-order function:

```
zip :: [a] -> [b] -> [(a,b)]

zip (a:as) (b:bs) = (a,b) : zip as bs
zip _ _           = []
```

For example:

```
  zip [1,2,3] [True,False,False]
⇒ [(1,True), (2,False), (3,False)]
```

Quick check: What does `zip [1..3] [1..5]` yield?

Coloring Our Circles

```
colCircles :: [(Color,Shape)]
colCircles = zip [Red,Blue,Green,
                  Cyan,Red,Magenta,
                  Yellow,White]
                 circles
```

Drawing Colored Shapes

```
drawShapes :: Window -> [(Color,Shape)] -> IO ()

drawShapes w css =
  sequence_ (map aux css)
  where aux (c,s) =
          drawInWindow w
            (withColor c
               (shapeToGraphic s))
```

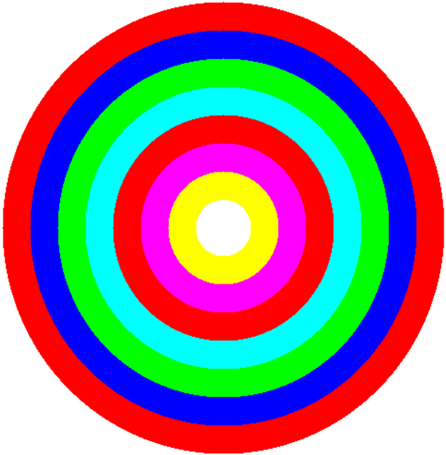
Recall from Chapter 3 that `sequence_` takes a list of `IO()` actions and returns an `IO()` action that performs all the actions in the list in sequence.

The Main Action

```
g = do w <- openWindow "Bulls eye" (600,600)
       drawShapes w colCircles
       k <- getKey w
       closeWindow w

main = runGraphics g
```

The Result



When to Define Higher-Order Functions

Recognizing repeating patterns is the key, as we did for map. As another example, consider:

```
listSum []      = 0
listSum (x:xs)  = x + listSum  xs

listProd []     = 1
listProd (x:xs) = x * listProd xs
```

Note the similarities. Also note the differences (`0` vs. `1` and `+` vs. `*`): it is these that will become parameters to the abstracted function.

Fold

Abstracting out the differences (`op` and `init`) leaves this common part:

```
fold op init []      = init
fold op init (x:xs) = x `op` fold op init xs
```

We recover `listSum` and `listProd` by instantiating `fold` with the appropriate parameters:

```
listSum  xs = fold (+) 0 xs
listProd xs = fold (*) 1 xs
```

Note that fold is polymorphic:

```
fold :: (a -> b -> b) -> b -> [a] -> b
```

Two Folds Are Better Than One

The `fold` function is predefined in Haskell, but it is actually called `foldr`, because it "folds from the right." That is:

```
   foldr op init (x1 : x2 : ... : xn : [])
⇒ x1 `op` (x2 `op` (...(xn `op` init)...))
```

There is another predefined function `foldl` that "folds from the left":

```
   foldl op init (x1 : x2 : ... : xn : [])
⇒ (...((init `op` x1) `op` x2)...) `op` xn
```

Two Folds Are Better Than One

Why two folds? Because sometimes using one can be more efficient than the other. For example:

```
foldr (++) [] [x,y,z]  ⇒  x ++ (y ++ z)
foldl (++) [] [x,y,z]  ⇒  (x ++ y) ++ z
```

The former is considerably more efficient than the latter (as discussed in the book); but this is not always the case — sometimes `foldl` is more efficient than `foldr`. Choose wisely!

Another Application of Fold

We have seen the function `sequence_`, which takes a list of actions of type `IO()` and produces a single action of type `IO()`.

We can define `sequence_` in terms of `>>` and `foldl` as follows:

```
sequence_ :: [IO ()] -> IO ()
sequence_ acts = foldl (>>) (return ()) acts
```

Reversing a List

Obvious but inefficient (why?):

```
reverse []      = []
reverse (x::xs) = reverse xs ++ [x]
```

Much better (why?):

```
reverse xs = rev [] xs
  where rev acc  []     = acc
        rev acc (x:xs) = rev (x:acc) xs
```

This looks a lot like `foldl`. Indeed, we can redefine `reverse` as:

```
reverse xs = foldl revOp [] xs
             where revOp a b = b : a
```