CS 457/557 Functional Programming

Lecture 5

Polymorphism; Higher-order functions

Polymorphic Length

"a" is a type variable. It is lowercase to distinguish it from types, which are uppercase.

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

- Polymorphic functions don't "look at" their polymorphic arguments.
- They use the same code now matter what the type of their polymorphic arguments.

Polymorphism

- Consider: `tag1 x = (1,x)`

```
> :type tag1
tag1 :: a -> (Int,a)
```

- Other functions have types like this; consider `(++)`

```
? :type (++)
(++) :: [a] -> [a] -> [a]
```

- What are some other polymorphic functions and their types?
  - `id ::`
  - `reverse ::`
  - `head ::`
  - `tail ::`
  - `(:) ::`

Polymorphic data structures

- Polymorphism originates from data structures that don't care what kind of data they store.

```
id :: a -> a          -- The ultimate
                      -- polymorphic function

reverse :: [a] -> [a]         -- lists
tail :: [a] -> [a]
head :: [a] -> a
(:) :: a -> [a] -> [a]

fst :: (a,b) -> a             -- tuples
swap :: (a,b) -> (b,a)
```

- How do we define new data structures with "holes" that can be polymorphic?

Maybe is polymorphic

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

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)
```

Example of its use:

```
lookup :: a -> [(a,b)] -> Maybe b
lookup k []                             = Nothing
lookup k ((k',v):rest) | k == k'        = Just v
                       | otherwise      = lookup k rest
```

Polymorphism from functions as arguments

- Another source of polymorphism comes from functions which take functions as arguments.

```
applyTwice f x = f(f x)
```

```
Main> :t applyTwice
applyTwice :: (a -> a) -> a -> a
```

- What's the type of the following useful function?

```
flip f x y = f y x
```

Polymorphism: Functions returned as values

- Consider:

```
const x = f
   where f y = x
Main> (const 3) 5
3
```

  – What's the type of `const`?

- Another Example:

```
compose f g x = f (g x)
```

  – What's the type of `compose` ?
  – Note: Prelude defines compose as an infix operator

```
(f . g) x = f (g x)
```

Abstraction Over Recursive Definitions

- Recall some definitions from previous chapters.
- Section 4.1:

```
translist []        = []
transList (p:ps)    = trans p : translist ps
```

- Section 3.1:

```
putCharList []     = []
putCharList (c:cs) = putChar c : putCharList cs
```

- There is something strongly similar about these definitions. 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.

Abstraction Yields `map`

- `trans` and `putChar` are what's different; so they should be arguments to the abstracted function.
- In other words, we would like to define a function called `map` (say) 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
```

- Given this, it is not hard to see that we can redefine `transList` and `putCharList` as:

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

`map` is Polymorphic

- The key thing about map is that it is *polymorphic*. Its most general ("principal") type is:

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

- Every use of **`map`** has a type that is an *instance* of the principal type (obtained by substituting for **`a`** and **`b`**).
- For example, since **`trans :: Vertex -> Point`**, then

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

and this use of map has type

```
map :: (Vertex -> Point) -> [Vertex] -> [Point]
```

Another Pattern: Filtering

- Consider extracting the even numbers from a list:

```
evens :: [Int] -> [Int]
evens [] = []
evens (x:xs) | even x    = x:(evens xs)
             | otherwise = evens xs
```

- Or removing the whitespace from a string:

```
nowhite :: String -> String
nowhite “” = “”
nowhite (c:cs) | not (whitesp c) = x : (nowhite cs)
               | otherwise       = nowhite cs
   where whitesp ' ' = True
         whitesp '\t' = True
         whitesp _ = False
```

Abstracting to `filter`

- Can define a common function

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []                    = []
filter p (x:xs) | p x          = x:(filter p xs)
                | otherwise = filter p xs
```

- Now can rewrite

```
evens xs = filter even xs
```

  – or just:

```
evens = filter even
```

- And

```
nowhite = filter (not . whitesp)
```

  – Recall that `(.)` represents function composition.

List comprehensions revisited

- Recall some uses of the list comprehension notation

```
putCharList cs = [putChar c | c <- cs]
evens xs = [y | y <- xs, even y]
```

- Observe that this notation incorporates both **`map`** and **`filter`**, e.g.

```
putNonWhiteChars cs =
        [putChar c | c <- cs, not (whitesp c)]
```

- Can easily define **`map`** and **`filter`** in terms of list comprehenion (try it!)
- Actually, list comprension is defined in terms of **`map`** and **`filter`** (and a few other things...)

When to Define Higher-Order Functions

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

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

and :: [Bool] -> Bool
and []       = True
and (x:xs)   = x && and xs

myminimum :: [Int] -> Int
myminimum []       = maxBound
myminimum (x:xs)   = x `min` myminimum xs
```

- Note the similarities. Also note the differences (circled), which need to become parameters to the abstracted function.

When to Define Higher-Order Functions

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

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

and :: [Bool] -> Bool
and []       = True
and (x:xs)   = x && and xs

myminimum :: [Int] -> Int
myminimum []       = maxBound
myminimum (x:xs)   = x `min` myminimum xs
```

Initial values

Combining ops

- Note the similarities. Also note the differences (circled), which need to become parameters to the abstracted function.

Abstracting to `foldr`

- This leads to:

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

- Note that **`foldr`** is also *polymorphic*:

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

  – We'll see the full power of this polymorphism shortly.

- Previous functions can now be redefined:

```
sum xs = foldr (+) 0 xs
and xs = foldr (&&) True xs
myminimum xs = foldr min maxBound xs
```

Visualizing the effect of `foldr`

- One useful way to think about what **`foldr`** does is to observe what it does on an arbitrary list written using explicit constructors:

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

- So we can think of **`foldr`** as taking a list and replacing each **`(:)`** by **`op`** and the final **`[]`** by **`init`**.

```
foldr (+) 0 (1 : (2 : (3 : [])))

       = 1 + (2 + (3 + 0))
```

- The **`r`** in **`foldr`** is because it "folds from the right".

Mystery folds

- Consider these functions:

```
mystery1 xs = foldr (*) 1 xs

mystery2 xs = foldr k 0 xs
    where k a b = b + 1

mystery3 q xs = foldr k False xs
    where k x b = q x || b

mystery4 = foldr (:) []
```

- What are their types?
- What do they do?

Two Folds are Better than One

- In addition to **foldr**, the Haskell Prelude defines another function **foldl** which "folds from the left":

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

- Exercise: define **foldl** using recursion.
- Why two folds? Often they are equivalent, but 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 more efficient than the latter (see textbook).
- In general, one or the other of **foldl** and **foldr** may be more efficient and/or lazier in any given circumstance.
- Choosing between them is non-trivial!

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`; we can redefine `reverse` as:

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

- Or just as

```
reverse = foldl (flip (:)) []
```