

Patterns of Recursion

Higher Order Functions

Reading: Thompson Ch.9

COMP1100 — Introduction to Programming and Algorithms

Clem Baker-Finch

Australian National University
Semester 1, 2006

Mapping

We often want to transform each element of a list in some way, eg:

- double every element of a list of numbers
- increment each element of a list of numbers
- convert each character of a string to upper case

For the PPM example, to select the red channel for an image, we apply a function that selects the red channel of a **pixel** to every pixel in a **row**, then apply that process to every row in the **image**.

We call this pattern of computation, **mapping**.

Patterns of Recursion

Polymorphism allows us to *reuse* code because the same function can be applied to *different types*.

Another mechanism for reuse is to recognise **patterns of computation** and design functions that *embody those patterns*.

For example, we often need to:

- transform every list element in some uniform way
- select out the list elements that satisfy some property
- combine the elements of a list using some operator

Three examples:

```
double :: [Int] -> [Int]
double []      = []
double (x:xs)  = 2*x : double xs
```

```
incr :: [Int] -> [Int]
incr []      = []
incr (x:xs)  = x+1 : incr xs
```

```
upperCase :: [Char] -> [Char]
upperCase []      = []
upperCase (x:xs)  = toUpper x : upperCase xs
```

How do they differ?

```
double []      = []
double (x:xs)  = 2*x : double xs
```

```
incr []        = []
incr (x:xs)    = x+1 : incr xs
```

```
upperCase []   = []
upperCase (x:xs) = toUpper x : upperCase xs
```

- They all have the same form (**pattern**)
- Only differ in function applied to the elements
- Obviously, other functions could be used in the same way

Idea!

Make those transformation functions *arguments* to a function representing the **form of the definition**. Such a mapping function will take two arguments:

- a function to transform the elements
- a list

This is exactly the function `map` defined in the prelude:

```
map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs
```

Functions that take other functions as arguments are called **higher-order**.

Notice that `map` is also **polymorphic**.

Change all the names to something meaningless to further emphasise to pattern even more:

```
f []      = []
f (x:xs)  = 2*x : f xs
```

```
g []      = []
g (x:xs)  = x+1 : g xs
```

```
h []      = []
h (x:xs)  = toUpper x : h xs
```

Now we can use `map` to define `double`, `incr` and `upperCase`:

```
double xs = map (2*) xs
```

```
incr xs = map (+1) xs
```

```
upperCase xs = map toUpper xs
```

Sample trace of computation:

```
map (2*) [3, 9] =
map (2*) (3 : (9 : []))
⇒ 2*3 : (map (2*) (9 : []))
⇒ 2*3 : 2*9 : (map (2*) [])
⇒ 2*3 : 2*9 : []
⇒ [6, 18]
```

Advantages

What are the advantages of this approach?

- The `map` function is a good example of **abstraction** and **generalisation**.
- If you understand *WHAT* `map` does, definitions that use it are easier to understand. You don't need to think about *HOW* it works.
- *Code re-use* is a cornerstone of modern software engineering practices. `map` can be re-used for all functions of this form.
(Similar advantage to polymorphism.)

How do they differ?

```
getDigits [] = []
getDigits (x:xs)
  | isDigit x = x : getDigits xs
  | otherwise =   getDigits xs
```

```
getNegs [] = []
getNegs (x:xs)
  | x < 0    = x : getNegs xs
  | otherwise =   getNegs xs
```

Filtering

We may want to *select* the elements of a list with some property in common.

For example, we may wish to select the digits in a string:

```
getDigits [] = []
getDigits (x:xs)
  | isDigit x = x : getDigits xs
  | otherwise =   getDigits xs
```

or the negative numbers in a list:

```
getNegs [] = []
getNegs (x:xs)
  | x < 0    = x : getNegs xs
  | otherwise =   getNegs xs
```

Extract a function to represent the common form of the selection algorithm:

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

`filter` takes a *predicate* (a `Bool`-valued function) and a list as arguments, and returns the sub-list as result.

Now we can write:

```
getDigits xs = filter isDigit xs

getNegs xs   = filter (< 0) xs
```

Folding

We often want to combine all the elements of a list into a single value in a uniform way.

For example, the standard function `sum` adds together all the elements of a list. `concat` combines a list of lists by the operator `(++)`.

Recall the `sum` computation:

```
x1 + (x2 + (x3 + ... (xn + 0 )))
x1 : (x2 : (x3 : ... (xn : [])))
```

which has the same pattern as `concat`:

```
x1 ++ (x2 ++ (x3 ++ ... (xn ++ [])))
x1 : (x2 : (x3 : ... (xn : [])))
```

foldl

There is also a **fold left**:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []      = z
foldl f z (x:xs) = foldl f (f z x) xs
```

In fact, the standard Prelude definition of `sum` is in terms of `foldl`:

```
sum xs = foldl (+) 0 xs
```

For example, `sum [x1, x2 .. xn]` is:

```
foldl (+) 0 [x1, x2 .. xn] = (((0 + x1) + x2) + ... xn)
```

foldr

The pattern on the previous slide is **fold right**:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []      = z
foldr f z (x:xs) = f x (foldr f z xs)
```

which is a bit harder to grasp than `map` or `filter`.

Think of what it does, rather than the definition.

```
foldr (+) 0 [x1, .. xn] = x1 + (x2 + .. (xn + 0)))
foldr (++) [] [x1, .. xn] = x1 ++ (x2 ++ .. (xn ++ [])))
```

The fold functions are extremely general patterns of recursion (even `map` can be defined using `fold`) ...

... and they correspond nicely to some loop structures in procedural languages ...

... but I think they are harder to understand ...

... so they might be a bit difficult for an introductory programming course.

Look for some other useful standard higher-order functions, e.g.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
```