COMP105 Lecture 15

Fold

Outline

Today

- `fold`

Relevant book chapters

- Programming In Haskell Chapter 7
- Learn You a Haskell Chapter 6

Recap: list recursion

Some functions take lists and turn them into a **single value**

```
sum' [] = 0
sum' (x:xs) = x + sum' xs

ghci> sum [1..10]
55
```

```
product' [] = 1
product' (x:xs) = x * product' xs

ghci> product [1..10]
3628800
```

Recap: list recursion

```
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

The only things that **change** are

- The initial value: `0`, `1`, ...
- The operation use in each recursive step: `+`, `*`, ...

These are examples of **folds**

Foldr

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

foldr' _ init []     = init
foldr' f init (x:xs) = f x (foldr' f init xs)
```

```
ghci> foldr' (+) 0 [1..10]
55

ghci> foldr' (*) 1 [1..10]
3628800
```

The folded function

```
sum'' list = foldr (\ x acc -> acc + x) 0 list
```

The folded function `f` takes two arguments

- `x` is an element from the list
- `acc` is the **accumulator**

The function outputs a **new** value for the accumulator

- The initial value for the accumulator is `init`
- The final value for the accumulator is the output of `foldr`

Foldr

Consider:

```
foldr (\ x acc -> acc + x) 0 [1,2,3,4]
```

Values for the accumulator:

```
init  = 0
0 + 4 = 4
4 + 3 = 7
7 + 2 = 9
9 + 1 = 10
```

Final output: 10

An imperative equivalent

```
foldr f init input_list
```

In **python** this would be implemented as

```
acc = init
input_list.reverse()

for i in range(len(input_list)):
    acc = f(input_list[i], acc)

return acc
```

Foldr examples

```
concat' list = foldr (++) [] list

ghci> concat' ["a", "big", "bad"]
"abigbad"


all' list = foldr (&&) True list

ghci> all' [True, True, True]
True


length' list = foldr (\_ acc -> acc + 1) 0 list

ghci> length' [1,2,3,4]
4
```

Foldr examples

```
count_ones list =
    foldr (\ x acc -> if x == 1 then acc + 1 else acc)
          0 list
```

```
ghci> count_ones [1,1,2,3,1,4]
3
```

Exercise

What do these functions do?

```
mystery list = foldr (\x acc -> acc+(x/2)) 0 list
```

```
mystery2 list = foldr (\x acc -> x : x : acc) [] list
```

```
mystery3 list =
    let f = (\x acc -> if x `mod` 2 == 0
                       then x+acc
                       else acc)
    in foldr f 0 list
```

Revisiting the fold type

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

Note that **two** type variables are used here

- The input list has type `[a]`
- The accumulator has type `b`

So a fold can output a **different** type to the input list

Folds that output different types

```
sum_of_lengths list =
    foldr (\x acc -> acc + length x) 0 list
```

```
ghci> sum_of_lengths ["one", "two", "three"]
11
```

Folds that output different types

```
to_csv list =
    foldr (\x acc -> show x ++ "," ++  acc) "" list

ghci> to_csv [1,2,3,4]
"1,2,3,4,"
```

foldr1

The function `foldr1` uses the **final value** of the list to initialize the accumulator

```
foldr1' _ []     = error "empty list"
foldr1' _ [x]    = x
foldr1' f (x:xs) = f x (foldr1' f xs)
```

```
ghci> foldr1' (+) [1,2,3,4,5]
15
```

`foldr1`

Note that the type of `foldr1` is

```
foldr1' :: (a -> a -> a) -> [a] -> a
```

The accumulator has the same type as the list elements

- So `foldr1` **cannot** be used to change the type of a list

foldr1 examples

```
sum' list = foldr1 (+) list

product' list = foldr1 (*) list

concat' list = foldr1 (++) list
```

```
ghci> concat [[1,2,3], [4], [3,2,1]]
[1,2,3,4,3,2,1]
```

foldr1 examples

```
maximum' list =
    foldr1 (\ x acc -> if x > acc then x else acc) list
```

```
ghci> maximum [1,2,3,4,3,2,1]
4
```

Folding right

`foldr` processes lists from the **right**

```
foldr (+) 0 [1..4]

= 1 + (2 + (3 + (4 + 0)))
```

```
foldr (/) 1 [1..4]

= 1 / (2 / (3 / (4 / 1)))
= 0.375
```

Folding left

`foldl` processes lists from the **left**

```
foldl (+) 0 [1..4]

= ((((0 + 1) + 2) + 3) + 4)
```

```
foldl (/) 1 [1..4]

= ((((1 / 1) / 2) / 3) / 4)
= 0.0416
```

The type of `foldl`

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

Observe that the function `f` has its type **flipped**

- `foldr (\ x acc -> ...`
- `foldl (\ acc x -> ...`

Reversing a list with `foldl`

```
reverse_list list = foldl (\ acc x -> x : acc) [] list
```

```
ghci> reverse_list "hello"
"olleh"
```

Exercises

1. Use `foldr` to write a function `odd_sum list` that sums all odd numbers in the input list.

2. Use `foldr` to write a function `count_odds list` that returns the number of odd numbers in the input list.

3. Use `foldr` to write a function `triple list` that returns a new list with each element of the input list repeated three times.

4. Use `foldr` and `reverse` to write a function `concat_reverse strings` that takes a list of strings, and returns a string where the strings appear in the same order with spaces between them, but each string is reversed. So `concat_reverse ["one", "two", "three"] = "eno owt eerht"`.

Exercises

5. Use `foldr1` to write a function `minimum' list` that finds the smallest element of the input list.

6. Implement `reverse_list` using `foldr` instead of `foldl`

7. Use `foldr` to write a function `make_palindrome string` that takes a string, and returns `string ++ reverse string`. So `make_palindrome "abc" = "abccba"`

Summary

- `fold`

Next time: More higher order programming functions