Higher-Order Functions 03

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Collatz Sequence

- The Collatz Sequence (Chain or $3x + 1$) of a natural number `n` is defined as follows:
  - The first number is `n`
  - If the number is `1`, stop
  - If the number is even, divide it by 2
  - If the number is odd, multiply by 3 and add 1
  - Repeat with the resulting number
- Example: Suppose the first number is `5`, the sequence is

```
5, 16, 8, 4, 2, 1
```

- Example: Suppose the first number is `11`, the sequence is

```
11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
```

Collatz Sequence

- Define the function `collatzSeq`:

```
collatzSeq 1 = [1]
collatzSeq n | even n = n : collatzSeq (div n 2)
             | odd n  = n : collatzSeq (3 * n + 1)
```

- Some tests:

```
ghci> collatzSeq 5
[5,16,8,4,2,1]
ghci> collatzSeq 11
[11,34,17,52,26,13,40,20,10,5,16,8,4,2,1]
ghci> collatzSeq 27
[27,82,41,124,62,31,94,47,142,71,214,107,322,161,484,242,121,364,
182,91,274,137,412,206,103,310,155,466,233,700,350,175,526,263,
790,395,1186,593,1780,890,445,1336,668,334,167,502,251,754,377,
1132,566,283,850,425,1276,638,319,958,479,1438,719,2158,1079,3238,
1619,4858,2429,7288,3644,1822,911,2734,1367,4102,2051,6154,3077,
9232,4616,2308,1154,577,1732,866,433,1300,650,325,976,488,244,122,
61,184,92,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

Collatz Sequence

- Given a number `n` and an upperbound `b`, give a list all starting numbers between `1` and `b` of the Collatz sequences that has length greater than or equal to `n`
- All starting numbers between `1` and `b`:

```
[1..b]
```

- Create the list of all Collatz sequences (from `1` to `b`):

```
map collatzSeq [1..b]
```

- Filter just those that has length greater than or equal to `n`:

```
filter longerThan (map collatzSeq [1..b])
       where longerThan xs = length xs >= n
```

- Get the heads of all filtered sequences:

```
startNumbers n b = map head (filter longerThan
                                    (map collatzSeq [1..b]))
                   where longerThan xs = length xs >= n
```

List of Functions using `map`

- Recall that if we partially apply a function, we get another function in return
  - `max` is a function
  - `max 5` is another function
- What is this expression?

```
map max [3,5,1,2,4]
```

  - It is a list of functions

    ```
    [max 3, max 5, max 1, max 2, max 4]
    ```

  - Note that we cannot see the result on the screen since functions are not instances of the `Show` type class
- However, the following is fine:

```
ghic> ((map max [3,5,1,2,4]) !! 1) 3
5
ghci> ((map max [3,5,1,2,4]) !! 1) 7
7
```

Lambda

- **Lambdas** are anonymous functions (functions without names)
- We generally use a lambda when we need the function only once
- Mathematical Syntax: $\lambda x. f(x)$
- Examples:
  - $\lambda\ x.\ x + 1$ where $(\lambda\ x.\ x + 1)$ 5 is 6
  - $\lambda\ x\ y.\ x \times y$ where $(\lambda\ x\ y.\ x \times y)$ 5 12 is 60
- In Haskell, we use \ to represent lambda and -> instead of .
- Above examples in Haskell

```
ghci> (\x -> x + 1) 5
6
ghci> (\x y -> x * y) 5 12
60
```

Lambda

- Recall the function `startNumber` defined earlier:

```
startNumbers n b = map head (filter longerThan
                                    (map collatzSeq [1..b]))
                   where longerThan xs = length xs >= n
```

- The function `longerThan` is used only once and only available inside the function definition
- We can use the following instead:

```
startNumbers n b = map head (filter (\xs -> length xs >= n)
                                    (map collatzSeq [1..b]))
```

Lambda

- Lambdas are expressions:
- A lambda has a type

```
ghci> :t (\x y -> x + y)
(\x y -> x + y) :: Num a => a -> a -> a
```

- Partially apply is allowed

```
ghci> :t (\x y -> x + y) 5
(\x y -> x + y) 5 :: Num a => a -> a
```

- It can be passed as an argument

```
ghci> map (\x -> x + 1) [3,5,1,2,4]
[4,6,2,3,5]
```

Lambda

- A Lambda can take multiple arguments

```
ghci> zipWith (\x y -> x * y) [1,2,3,4] [3,7,1,2]
[3,14,3,8]
```

- A Lambda can take a pair as an argument

```
ghci> map (\(x,y) -> x * y) [(1,2), (3,4), (5,6), (7,8)]
[2,12,30,56]
```

- These three functions are equivalent:

```
ghci> addTwo x y = x + y
ghci> :t addTwo
addTwo :: Num a => a -> a -> a

ghci> addTwo' = \x y -> x + y
ghci> :t addTwo'
addTwo' :: Num a => a -> a -> a

ghci> addTwo'' = \x -> \y -> x + y
ghci> :t addTwo''
addTwo'' :: Num a => a -> a -> a
```

  but the first one is easier to read

- Do not need to try to use lambdas

Folds

- Recall when we try to do something on a list
  - We need to decide what to do with the empty list
  - Then handle non-empty list like `x:xs`
- We do this with almost every function on lists
- Haskell provides **fold** functions to handle these common pattern
- Folds allow you to reduce a list into a single value
- Folds are suitable for operations that require to travel a list once, element-by-element
- A fold takes the following arguments:
  - A binary function (a function that takes two arguments)
  - A starting value
  - A list

Left Fold

- To easily understand the left fold, let's look at the definition of the `foldl` function:

```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- Sometime we call `z` as an accumulator
- Let's trace what is going on with `foldl (+) 0 [1,2,3]`

```
foldl (+) 0 [1,2,3] = foldl (+) 0 1:[2,3]
                    = foldl (+) (0 + 1) [2,3]
                    = foldl (+) (0 + 1) 2:[3]
                    = foldl (+) ((0 + 1) + 2) [3]
                    = foldl (+) ((0 + 1) + 2) 3:[]
                    = foldl (+) ((0 + 1) + 2) + 3) []
                    = (((0 + 1) + 2) + 3)
                    = ((1 + 2) + 3)
                    = (3 + 3)
                    = 6
```

- The list is folded from the left side
- The binary function is applied between the starting accumulator and the head of the list
- The result becomes the new starting value of the rest of the list

Right Fold

- To easily understand the right fold, let's look at the definition of the `foldr` function:

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- Let's trace what is going on with `foldl (+) 0 [1,2,3]`

```
foldr (+) 0 [1,2,3] = foldr (+) 0 1:[2,3]
                    = 1 + (foldr (+) 0 [2,3])
                    = 1 + (foldr (+) 0 2:[3])
                    = 1 + (2 + (foldr (+) 0 [3]))
                    = 1 + (2 + (foldr (+) 0 3:[]))
                    = 1 + (2 + (3 + (foldr (+) 0 [])))
                    = 1 + (2 + (3 + 0))
                    = 1 + (2 + 3)
                    = 1 + 5
                    = 6
```

- The list is folded from the right side
- The binary function is applied between the head of the list and the result of the rest of the fold

Let's Double Check

- Let's try to define our own `map` function called `myMapr` using `foldr`
- Since the result of a `map` is a list, the accumulator should be the empty list

```
myMapr f xs = foldr g [] xs
              where g y ys = f y : ys
```

- Let's try to define our own `map` function called `myMapl` using `foldl`

```
myMapl f xs = foldl g [] xs
              where g ys y = ys ++ [f y]
```

- Note that `:` will be a lot faster than `++`

```
1:[2,3] = [1,2,3]
```

```
[1,2] ++ [3] = 1:([2] ++ [3])
             = 1:(2:[] ++ [3])
             = 1:(2:[3])
             = [1,2,3]
```

- What is `foldr (:) [] [1..]` evaluated to?

```
ghci> foldr (:) [] [1..]
[1,2,3,4,5,6,7,8,9...Interrupted.
```

- Let's trace `take 3 (foldr (:) [] [1..])`

```
take 3 (foldr (:) [] [1..]) = take 3 (foldr (:) [] 1:[2..])
                            = take 3 ((:) 1 (foldr (:) [] [2..]))
                            = (:) 1 (take 2 (foldr (:) [] [2..]))
                            = (:) 1 (take 2 (foldr (:) [] 2:[3..]))
                            = (:) 1 (take 2 ((:) 2 (foldr (:) [] [3..])))
                            = (:) 1 ((:) 2 (take 1 (foldr (:) [] [3..])))
                            = (:) 1 ((:) 2 (take 1 (foldr (:) [] 3:[4..]))))
                            = (:) 1 ((:) 2 (take 1 ((:) 3 (foldr (:) [] [4..])))))
                            = (:) 1 ((:) 2 ((:) 3 (take 0 (foldr (:) [] [4..])))))
                            = (:) 1 ((:) 2 ((:) 3 [])))
```

- We only need the first three element.

Right Fold vs Left Fold

- Let's trace the `foldr` again with a binary function `f`

```
foldr f z [1,2,3] = foldr f z 1:[2,3]
                  = f 1 (foldr f z [2,3])
                  = f 1 (foldr f z 2:[3])
                  = f 1 (f 2 (foldr f z [3]))
                  = f 1 (f 2 (foldr f z 3:[]))
                  = f 1 (f 2 (f 3 (foldr f z [])))
                  = f 1 (f 2 (f 3 z))
```

- Let's trace the `foldl` again with a binary function `f`

```
foldl f z [1,2,3] = foldl f z 1:[2,3]
                  = foldl f (f z 1) [2,3]
                  = foldl f (f z 1) 2:[3]
                  = foldl f (f (f z 1) 2) [3]
                  = foldl f (f (f z 1) 2) 3:[]
                  = foldl f (f (f (f z 1) 2) 3) []
                  = f (f (f z 1) 2) 3
```

- In `foldr`, `f 1 ...` may already produce a partial result
- Right folds works on infinite lists but left folds do not

`foldl1` and `foldr1`

- Let's check their types:

```
ghci> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
ghci> :t foldl1
foldl1 :: Foldable t => (a -> a -> a) -> t a -> a

ghci> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
ghci> :t foldr1
foldr1 :: Foldable t => (a -> a -> a) -> t a -> a
```

- `foldl1` and `foldr1` do not need a starting value
- They use the first element (`foldl1`) and the last element (`foldr1`) as their starting values
- For `foldl` and `foldr`, their starting values do not have to have the same type as elements in a given list

Exercise with Folds

- Let's try to define the `reverse` function using `foldl`
- Initial value should be the empty list and we will slowly accumulate it
- So, it should be

  ```
  foldl f [] [1,2,3]
  ```

  for a function `f`
- The above expression is evaluated to

  ```
  foldl f (f [] 1) [2,3]
  ```

  and eventually to

  ```
  f (f (f [] 1) 2) 3
  ```
- What should `f` be?

Exercise with Folds

- Obviously, `f [] 1` should be `[1]`
  - `1:[] = [1]` or
  - `[] ++ [1] = [1]`?
- To decide which one is correct, we need to consider the next application as well
- Consider `f (f [] 1) 2`
  - If `f [] 1 = 1:[]`

```
f (f [] 1) 2 = f (1:[]) 2
             = f [1] 2
             = 2:[1]
             = [2,1]
```

  - If `f [] 1 = [] ++ [1]`

```
f (f [] 1) 2 = f ([] ++ [1]) 2
             = f [1] 2
             = [1] ++ [2]
             = [1,2]
```

Exercise with Folds

- Obviously, `f xs x` should be `x : xs`
- In other words, `f` should have type `[a] -> a -> [a]`
- Recall that `:` has type `a -> [a] -> [a]`
  - The first argument an element and
  - The second argument is a list
- Thus, we cannot use the `:` function directly
- We need to flip the arguments

Exercise with Folds

- Solution 1: Use a local binding

```
reverse_l1 lst = foldl f [] lst
                 where f xs x = x : xs
```

- Solution 2: Use a Lambda:

```
reverse_l2 lst = foldl (\xs x -> x : xs) [] lst
```

- Solution 3: Use the `flip` function:

```
reverse_l3 lst = foldl (flip (:)) [] lst
```

Exercise with Folds

- How about `foldr`?
- At least it should be

```
foldr f [] [1,2,3]
```

  for a function `f`
- The above expression is evaluated to

```
f 1 (foldr [] [2,3])
```

  and eventually to

```
f 1 (f 2 (f 3 []))
```

- What should `f` be?
- In this case, `f 3 []` should be `[] ++ [3] = [3]`
- Then `f 2 [3]` will be `[3] ++ [2] = [3,2]`

Exercise with Folds

- Solution 1: Use a local binding

```
reverse_r1 lst = foldr f [] lst
                 where f x xs = xs ++ [x]
```

- Solution 2: Use a Lambda:

```
reverse_l2 lst = foldr (\x xs -> xs ++ [x]) [] lst
```

- Flip cannot be used since arguments have type `a` and `[a]` but `++` has type `[a] -> [a] -> [a]`

Scans

- `scanl` and `scanr` functions are the same as `foldl` and `foldr`
- Instead of showing the final value, scans show immediate accumulators as a list

```
ghci> scanl (+) 0 [1,2,3]
[0,1,3,6]
ghci> scanr (+) 0 [1,2,3]
[6,5,3,0]
```

- `scanl1` and `scanr1` are also available
- They are good tools for debugging functions that utilize folds

Exercise

- How many elements does it take for the sum of the square roots of all natural numbers to exceed 1000?
- First, the list of square roots of all natural numbers:

```
map sqrt [1..]
```

- Now, turn it into the list of immediate accumulators

```
scanl1 (+) (map sqrt [1..])
```

- We will take all elements that are less than 1000

```
takeWhile (<1000) (scanl1 (+) (map sqrt [1..]))
```

- Now, the answer is the length of the above list

```
length (takeWhile (<1000) (scanl1 (+) (map sqrt [1..])))
```

- To go over 1000, you need to add one more square root:

```
length (takeWhile (<1000) (scanl1 (+) (map sqrt [1..]))) + 1
```

Function Application using $

- Consider the expression `f a b c d`
- The expression `f a b c d` is `(((f a) b) c) d`
  - `f a` results in a new function
  - `(f a) b` results in a new function
- In other words, function application is left associative
- The following expression

```
sum filter even [1..20]
```

will cause an errorbecause it tries to perform

```
((sum filter) even) [1..20]
```

- We need parentheses:

```
sum (filter even [1..20])
```

Function Application using $

- $ is an infix function:

```
ghci> :t ($)
($) :: (a -> b) -> a -> b
```

- It takes a function of type `a -> b`, a value of type `a`, and returns a value of type `b`
- Consider the type of the function `head`

```
ghci> :t head
head :: [a] -> a
```

- Now, what about `($) head`?

```
ghci> :t ($) head
($) head :: [b] -> b
```

- They have the same type
- What is the purpose of the `($)` function then?

Function Application using `$`

- Function application with `$` is **right-associative**
- In other words, `f $ a b c d` is `f (a (b (c d)))`
- Note that `(a (b (c d)))` becomes one expression as the argument to the function `f`
- For simplicity you can think of `f $ exp1 exp2 ... expn` as `f (exp1 exp2 ... expn)`
- Use `$` will result in less number of parentheses
- For example, instead of

  ```
  sum (filter even [1..20])
  ```

  simply use

  ```
  sum $ filter even [1..20]
  ```

- Another example, instead of

  ```
  sum (filter even (map (+3) [1..20]))
  ```

  simply use

  ```
  sum $ filter even $ map (+3) [1..20]
  ```

Function Composition

- In mathematics, suppose f and g are functions, a function composition is defined as

$$(f \circ g)(x) = f(g(x))$$

- To perform function composition in Haskell, we use the `.` function

```
ghci> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

- The definition of the `.` function is defined as

```
f . g = \x -> f (g x)
```

- Note that the type of the argument of the `f` function must be the same as the type of the value that returned by the `g` function

Function Composition

- We generally use function composition to make functions on the fly to pass to other functions
- This is the same as one of the purpose of using lambda
- Function composition is generally clearer and more concise
- Example: Suppose we want to turn very number on a list to negative number
  - Use lambda:

    ```
    map (\x -> negate (abs x)) [3,-5,1,2,-4]
    ```

  - Use function composition

    ```
    map (negate . abs) [3,-5,1,2,-4]
    ```

Function Composition

- Function composition is right-associative
- `f (g (h x))` is simply `(f . g . h) x`
- Example:

```
ghci> map (negate . sum . tail) [[1,2,3],[4,1,5,6],[1,1,2,5]]
[-5,-12,-8]
```

Function Composition

- For every function in composition, it cannot take multiple argument
- To use a function that takes more than one arguments in a composition, we need to partially apply it

```
ghci> :t sum
sum :: (Foldable t, Num a) => t a -> a
ghci> :t replicate
replicate :: Int -> a -> [a]
ghci> :t max
max :: Ord a => a -> a -> a
```

- Instead of

```
ghci> sum (replicate 5 (max 5 12))
60
```

we can use

```
ghci> (sum . replicate 5 . max 5) 12
60
```

  - `replicate 5` only takes one argument
  - `max 5` only takes one argument

Point-Free Style

- Consider the following function `mySum`

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

- Because of **currying**, we can omit `xs` on both side

```
mySum = foldl (+) 0
```

- `foldl (+) 0` is a function that take a list as the argument
- **Point-free** style does not include information regarding its argument
  - Instead of $f(x) = g(x)$, we can simply say $f = g$
  - Instead of $f(g(x)) = h(x)$, we can simply say $f \circ g = h$
- Note that $f \circ g$ in mathematics is the same as `f . g` in Haskell
- **Point-free style does not mean no point `(.)` in an expression**
- Haskell use point `(.)` to represent $\circ$ in function composition

Point-Free Style

- Let's consider another example:

```
fn x = ceiling (negate (tan (cos (max 50 x))))
```

- We cannot simply remove `x` on both sides of the above expression
- `cos (max 50)` is an invalid expression
  - `cos` is a function that takes a number as an argument
  - `max 50` is a function
  - `cos (max 50) 20` is also invalid
- However, we can use function composition:

```
fn = ceiling . negate . tan . cos . max 50
```

- Often time, a point-free style is more readable and concise
  - Readers simply focus on just functions
- It may not suitable for complex functions