Functional Programming Assignment Report

Key	Value
Author	Haihao Yan
Date	04/04/2019
Source Code	https://github.com/hyan36/fdr

Introduction

The balance puzzle requires us to find the coin with different weight. In our implementation, we are going to use a list to represent a pile of coins, the element of the list represents the weight of a coin. For example, given a list of coins [1,1,2,1,1], the coin weighs 2 is the fake coin.

Firstly, we will need to import the required libraries.

```
import Data.List
import Data.Ord
import Test.QuickCheck
```

The following function will weigh 3 coins and spot the fake coin. Assume there are 3 coins in the pile, a, b and c. We will have the following conditions:

1. a == b && b == c : all coins are equal in weight
2. a == b && a /= c : c is the fake coin
3. a /= b && a == c : b is the fake coin
4. a /= b && a /= c && b == c : a is the fake coin

```
weigh :: [Int] -> Int
weigh (a:b:xs)
   | a == b && a == c = -1
   | a == b && a /= c = c
   | a /= b && a == c = b
   | otherwise = a
   where
       c = head xs
weigh xs = -1
```

My strategy is to weigh 3 coins each time until we spot the fake coin. However, this strategy creates an obstacle, when the size of the pile cannot be `mod` by 3, we will have 1 or 2 coins left in the pile which cannot be determined whether it is fake or not.

My solution is to take **n** coins from the genuine pile. (Given we have only one fake coin in the pile, if the program hits the last few coins we can assume that all previously tested coins are genuine.)

- n = 3 - ((length xs) `mod` 3)

We will then define a function to prefill the original list. So we can split the pile into smaller chunks e.g. [[1,2,1],[1,1,1]...[1,1,1]].

The following method will take an `n` as an input which indicates the size of the chunks and list as original input. It will return a new list which will always be able to `mod` by n without remaining.

```
prefill::Int -> [Int] -> [Int]
prefill _ [] = []
prefill n xs = xs ++ (take (if size == n then 0 else size) xs)
    where
        size = n - ((length xs) `mod` n)
```

Now that we have a list of test cases for every coin. We just need to weigh every 3 coins until find out the fake coin.

```
findFake::[Int] -> Int
findFake [] = -1
findFake (x:y:z:xs) = if test /= -1 then test else findFake xs
    where
        pile = prefill 3 (x:y:z:xs)
        test = weigh [x,y,z]
```

This is one of the most obvious ways to identify the fake coin. The complexity of this solution is O(n/3). When there are 12 coins we will use 4 weighs to find the fake coin.

Solution

The prime motivation of the assignment is to simulate a physical solution to the problem. The program will calculate all possible strategies of weighing and then choose the most efficient way of weighing coins.

States and test

Firstly, we need to define the data type `State` and `Test` to represent the state of the simulation (State) and our strategy of next move (Test).

```
data State = Pair Int Int | Triple Int Int Int
    deriving (Eq, Show)
```

```
data Test = TPair (Int,Int) (Int,Int) | TTrip (Int,Int,Int) (Int,Int,Int)
    deriving (Eq,Show)
```

Q1 - Define `valid`

The `valid` function returns True when it meets the following conditions:

- Pair can only be tested by TPair and Triple can only be tested by TTrip
- **validTest**: A valid test case requires the number of the coins on the right side matches the coins the left. We learn nothing if we weigh 3 coins against 4.
- **validSample**: You can't take more coins than the pile can offer. For example, assume we have a pile of 5 coins. We've taken 3 coins to the left-hand side of the scales. There are 2 coins remaining in the pile. We can't take more than 2 coins for the right side of the scale.

All other situations are invalid.

Thus, we will have the following definition of `valid`.

```
valid:: State -> Test -> Bool
valid (Pair x y) (TPair (a,b) (c,d)) = validTest && validSample
    where
        validTest   = (a + b) == (c + d)
        validSample = (a + c) <= x && (b + d) <= y
valid (Triple x y z) (TTrip (a,b,c) (d,e,f)) = validTest && validSample
    where
        validTest   = (a + b + c) == (d + e + f)
        validSample = (a + d) <= x && (b + e) <= y && (c + f) <= z
valid _ _ = False
```

Choosing and conducting a test

Now that we have a way to determine whether a state and test is valid or not. We will focus on getting the program to simulate the tests.

Q2 - Define `outcomes`

There will only be 3 outcomes for each test. [lighter, balanced, heavier]. The basic rules of Pair have been explained very well in the assignment requirement. The triple situation is slightly more complicated.

Given `(TTrip (a,b,c) (d,e,f))`, we will have the following combinations:

- balanced - all tested coins go to pile `g`
- lighter - when the left pan is lighter than the right pan, the fake coin can only be in pile `a` or pile `e`. All other coins left on the table goes to pile `g`.
- heavier - when the left pan is heavier than the right pan, the fake coin must be in pile `b` or pile `b`. All other coins left on the table goes to pile `g`.

Thus, we will define `outcomes` as the following:

```
outcomes::State -> Test -> [State]
outcomes s t
    | valid s t = [(lighter s t),(balanced s t),(heavier s t)]
    | otherwise = []
    where
        balanced (Pair x y) (TPair (a,b) (c,d))         = Pair (x - a - c) (y + a
+ c)
        balanced (Triple x y z) (TTrip (a,b,c) (d,e,f)) = Triple (x - a - d) (y -
b - e) (z + a + d + b + e)
        lighter (Pair x y) (TPair (a,b) (c,d))          = Triple a c (x - a - c)
        lighter (Triple x y z) (TTrip (a,b,c) (d,e,f))  = Triple a e (x + y + z -
a - e)
        heavier (Pair x y) (TPair (a,b) (c,d))          = Triple c a (x - a - c)
        heavier (Triple x y z) (TTrip (a,b,c) (d,e,f))  = Triple d b (x + y + z -
b - d)
```

Q3 - Define `weighings Pair`

Given `TPair (a,b) (c,d)`, a sensible test must meet following requirements:

- a + b == c + d : in our implementation, a + b == (a+b) + 0, thus there is no need to validate
- a + b > 0 : we will use as it is
- b * d == 0 : in our implementation, d = 0 thus b * d will always equals 0, thus there is no need to validate
- a + c <= u : since c = a + b thus we will translate this condition to 2 * a + b <= u
- b + d <= g : since d = 0 thus, b + d = b. In our implementation b <- [0..g], it will always be smaller than g, thus there is no need to validate that
- (a,b) <= (c,d) : since c = a + b, d = 0. We translate the equation to (a,b) <= (a+b,0)

```
weighings::State -> [Test]
weighings (Pair u g) = [TPair (a,b) (a + b, 0) | a <- [0..u], b <- [0..g],
                                                 a + b > 0,
                                                 2 * a + b <= u,
                                                 (a,b) <= (a + b, 0)]
```

Q5 - Define `weighings Triple`

The strategy is to pick `k` number of coins for each pans. We will have following conditions:

- k <= (l + h + g) / 2 : you can't pick more coins than total amount of coins
- a + b + c == d + e + f : both pans need to have identical amount of coins
- a + b + c > 0 : you need to put something on the both pans
- c * f == 0 : don't pull genuine coins in both pans
- a + d <= l : enough light coins
- b + e <= h : enough heavy coins
- c + f <= g : enough genuine coins
- (a,b,c) <= (d,e,f) : symmetry breaker

```
weighings (Triple l h g) = [ TTrip (a,b,c) (d,e,f) | k <- [1..((l + h + g) `div`
2)],
                                                     (a,b,c) <- choices k (l,h,g),
                                                     (d,e,f) <- choices k (l,h,g),
                                                     a + b + c == d + e + f,
                                                     a + b + c > 0,
                                                     c * f == 0,
                                                     a + d <= l,
                                                     b + e <= h,
                                                     c + f <= g,
                                                     (a,b,c) <= (d,e,f)]
```

Q4 - Define `choice`

In order to achieve the previous function, we need to define a `choice` function which picks `k` number of coins from `Triple l h g`. Given we are picking `i` coins from pile `l` and `j` coins from pile `h`, a valid pick must meet the following conditions:

- i + j <= k: you can't pick more than `k` number of coins.
- k - i - j <= g: you can't pick more coin than the pile can offer.

```
choices::Int -> (Int,Int,Int)->[(Int,Int,Int)]
choices k (l,h,g) = [(i,j,k-i-j) | i <- [0..l], j <-[0..h], k-i-j >= 0, k-i-j <=
g]
```

Q6 - Define `Ord state`

For any two `State`, we will have the following rules:

- `Triple` always less than `Pair` - `Triple` always provides more information than pair
- For either `Pair` to `Pair` / `Triple` to `Triple` - we always compare pile `g`, whoever has more coins in pile `g` will provide more information than the other.

```
instance Ord State where
    compare (Triple _ _ _) (Pair _ _)          = compare 0 1
    compare (Pair _ _) (Triple _ _ _)          = compare 1 0
    compare (Pair u g) (Pair u' g')            = compare g' g
    compare (Triple l h g) (Triple l' h' g') = compare g' g
```

Q7 - Define `productive`

A productive test must give more information than the existing `State`. Thus we will have `s' <= s`. When there is no legit outcome of a given test, the function always returns False.

```
productive::State -> Test -> Bool
productive s t
   | length results > 0 = foldr (\x y -> x && y) True [ s' < s | s'<-results]
   | otherwise = False
   where
       results = outcomes s t
```

Q8 - Define `tests`

We will filter the results of `weighings` with the following conditions:

- check if a test is **valid**
- check if a test is **productive**

```
tests::State -> [Test]
tests s = filter (\t -> (valid s t) && (productive s t)) (weighings s)
```

Decision trees

In this section, we are going to define a decision tree to record the steps of simulation.

Firstly, we will define a recursive `Tree` data type.

```
data Tree = Stop State | Node Test [Tree]
   deriving (Eq,Show)
```

Q9 - Define `final`

In the meanwhile, we need a function to determine if a `State` is final. The final `State` must meet the following rules:

1. For `(Pair u g)` we will have `u` == 0 && g > 0 : no fake coins on the table
2. For `(Triple l h g)` we will have exactly 1 coin in `l` or exactly 1 coin in `h`
3. Otherwise, it is not final

```
final::State -> Bool
final (Pair u g) = u == 0 && g > 0
final (Triple l h g)
   | l == 1    = h == 0 && g > 0
   | h == 1    = l == 0 && g > 0
   | otherwise = False
```

Q10 - Define `height`

Function `height` will traverse the tree and count the height of the decision tree.

```
height::Tree -> Int
height (Stop x) = 0
height (Node t xs)
  | length xs > 0 = 1 + (maximum $ map height xs)
  | otherwise     = 0
```

Q11 - Define `minHeight`

Instead of defining our own `minimum` function, we will just try to utilize the existing minimum function provided by `Data.List`.

As a result, we will need to define a new instance for `Ord Tree`. The order of two trees is determined by their height. The benefit of this approach is that we can utilize the "out of box" functions instead of defining our own. However, it changes the default of Ord of `t1 < t2`. Given we didn't have other requirements of using this expression, I decide to use this approach.

```
instance Ord Tree where
    compare x y = compare (height x) (height y)
minHeight::[Tree] -> Tree
minHeight = minimum
```

Q12 - Define `mktree`

We will define a `mktree` function to generate the most efficient strategy. The most efficient strategy is the tree which has the lowest height.

```
mktree::State -> Tree
mktree s
    | final s || l == 0  = Stop s
    | otherwise          = minHeight (map (\t -> Node t [ mktree s' | s'<-
(outcomes s t)]) ts)
    where
        l  = (length ts)
        ts = tests s
```

Caching heights

In the previous section, we have got `mktree` to produce the best options of weighings. However, the performance of which is relatively low. It takes 16 sec on my laptop. One of the potential reasons is that we are calculating `height` recursively. In this section, we are going to try to optimize performance by caching the height of trees.

Firstly, we need to define `TreeH` to record the height of a tree.

```
data TreeH = StopH State | NodeH Int Test [TreeH]
   deriving (Eq,Show)
```

Secondly, we will define `heightH` to calculate the height of `TreeH`.

```
heightH::TreeH -> Int
heightH (StopH s) = 0
heightH (NodeH h t ts) = h
```

Last but not least, we will define `minHeightH` function to get the tree with the lowest height.

```
instance Ord TreeH where
    compare x y = compare (heightH x) (heightH y)
minHeightH:: [TreeH] -> TreeH
minHeightH = minimum
```

Q13 - Define `treeH2tree`

Following function transforms a given `Tree` to `TreeH` by mapping nodes recursively.

```
treeH2tree::TreeH -> Tree
treeH2tree (StopH s) = Stop s
treeH2tree (NodeH n t ts) = Node t [ treeH2tree x | x <- ts]
```

Q14 - Define smart constructor `nodeH`

Following function constructs a `NodeH` with a given `Test` and 3 `TreeH`.

```
nodeH :: Test -> [TreeH] -> TreeH
nodeH t ts = NodeH (1 + maximum (map heightH ts)) t ts
```

Q15 - Define `tree2treeH`

Following function transforms a given `Tree` to `TreeH` (similar to Q13).

```
tree2treeH::Tree -> TreeH
tree2treeH (Stop s) = StopH s
tree2treeH (Node t ts) = nodeH t [ tree2treeH x | x <- ts]
```

Justify `heightH . tree2treeH = height`:

- when input `t` is final, we have `heightH t` and `height t` both equals to 0
- when input `t` is not final, `tree2treeH` t will calculate the height of each layer recursively `(1 + maximum (map heightH ts))` until hit final states. Becaue function `heightH` equals to `height` (0) when given `t` is final, the statement is equivant to `height` by `(1 + (maximum $ map height xs))`.

As a result, I believe `heightH . tree2treeH = height`.

To increase my confidence level, I've also implemented a quick check test for this assumption. Please see **Appendix** for full implementation.

```
prop_testHeight t = heightH (tree2treeH t) == height t
```

After running the test, we had following result:

```
Main> quickCheck prop_testHeight
+++ OK, passed 100 tests.
```

Q16 - Define `mktreeH`

I've found three possible approaches to achieve this function.

Approach 1 - using 1st level tree only

Instead of traversing all trees nodes, we only transform the first layer of the trees.

```
```

```
mktreeH :: State -> TreeH
mktreeH s = minHeightH ( map ( \t -> nodeH t [ tree2treeH (mktree s') | s' <-
(outcomes s t)]) (tests s))
```

Approach 2 - traverse all node during generation

This approach will calculate the height of all tree nodes during generation.

```
mktreeH' :: State -> TreeH
mktreeH' s
   | final s || length (tests s) == 0  = StopH s
   | otherwise  = minHeightH ( map ( \t -> nodeH t [ mktreeH' s' | s' <- (outcomes
s t)]) (tests s))
```

Approach 3 - convert the final result of mktree

This approach is very straight forward, it converts whatever output of `mktree` function.

```
mktreeH'' :: State -> TreeH
mktreeH'' = tree2treeH . mktree
```

Peformance Indication

The following table summarized our experiment for each approach.

Test case: `Pair 8 0`

Function	Performance	Analysis
mktreeH	Almost identical to `mktree`, with less than 0.1-second difference	According to our experiment in **Appendix 2**, the performance of `height` and `heightH` is unnoticeable when given tree or data set is small.
mktreeH'	Significantly slower than `mktree`	We have to calculate the height for all possible nodes during tree generation.
mktreeH''	Slightly slower than `mktree`	It won't improve the performance because `mktree` is always executed first. It's a bit slower because it takes time to get the height of a Tree.

As a result, we pick `mktreeH` as our final submission.

A greedy solution

According to our experiment, our previous attempt didn't optimize the performance as we would hope it would be. In this section, we are going to try a greedy solution which filters the test cases at local bases.

Firstly, we will copy the `optimal` method from the assignment sheet.

```
optimal::State -> Test -> Bool
optimal (Pair u g) (TPair (a,b) (ab, 0))
    = (2 * a + b <= p) && (u - 2 * a - b <= q)
        where
            p = 3 ^ (t - 1)
            q = (p - 1) `div` 2
            t = ceiling (logBase 3 (fromIntegral (2 * u + k)))
            k = if g == 0 then 2 else 1
optimal (Triple l h g) (TTrip (a, b, c) (d, e,f ))
   = (a + e) `max` (b + d) `max` (l - a - d + h - b - e) <= p
       where
           p = 3 ^ (t - 1)
           t = ceiling (logBase 3 (fromIntegral( l + h )))
```

Q17 - Define `bestTests`

Following function filters the tests by checking if the test case is optimal.

```
bestTests::State -> [Test]
bestTests s = filter (\t -> optimal s t) (tests s)
```

Q18 - Define `mktreeG`

The following function returns a tree with the best strategy based on all optimal tests. The algorithm is very similar to `mktree`.

```
mktreeG::State -> TreeH
mktreeG s
   | final s || length (bestTests s) == 0 = StopH s
   | otherwise  = minHeightH ( map ( \t -> nodeH t [ mktreeG s' | s' <- (outcomes
s t)]) (bestTests s))
```

Q19 - Define `mktreesG`

This function returns a list of decision trees based on all optimal tests.

```
mktreesG::State -> [TreeH]
mktreesG s = map (\t -> nodeH t [mktreeG  s' | s' <- (outcomes s t)]) (bestTests
s)
```

There is only 1 tree in the list for `Pair 12 0`. The reason behind this is because there is only 1 optimal test for `bestTests (Pair 12 0)`.

Appendix 1 - Automated Tests

In order to justify the assumption of question 15, I implemented quick check tests to validate our theory. Although this can not prove the equation is always correct, this still increases the user's confidence significantly.

"as an inverse to treeH2tree. Convince yourself that => heightH . tree2treeH = height"

Firstly, we need to create a new instance for our customized data type State.

```
instance Arbitrary State where
    arbitrary = sized state'
        where
           state' 0 = do
              u <- arbitrary
              g <- arbitrary
              return (Pair u g)
           state' n = do
              l <- arbitrary
              h <- arbitrary
              g <- arbitrary
              return (Triple l h g)
```

Secondly, we need to create a new instance for our customized data type Test.

```
instance Arbitrary Test where
    arbitrary = sized test'
        where
          test' 0 = do
              u <- arbitrary
              g <- arbitrary
              u' <- arbitrary
              g' <- arbitrary
              return (TPair (u,g) (u',g'))
          test' n = do
              l <- arbitrary
              h <- arbitrary
              g <- arbitrary
              l' <- arbitrary
              h' <- arbitrary
              g' <- arbitrary
              return (TTrip (l,h,g) (l',h',g'))
```

Thirdly, we have to create a new instance for `Tree`. `Tree` is a recursive data type. It is important that we control the size of recursion. That is why I choose to use (n `div` 2) to avoid long test case generation.

```
instance Arbitrary Tree where
   arbitrary = sized tree'
       where
          tree' 0 = do
               a <- arbitrary
               return  (Stop a)
          tree' n
              | n > 0 = do
                 t <- arbitrary
                 ts <- vectorOf 3 (tree' (n `div` 2))
                 return (Node t ts)
```

Finally, we define our test cases, given a random tree, we will always have the following equation:

`heightH (tree2treeH tree) == height tree`

```
prop_testHeight t = heightH (tree2treeH t) == height t
```

After execution, we had following result:

```
Main> quickCheck prop_testHeight
+++ OK, passed 100 tests.
```

Appendix 2 - Comparing `height` and `heightH`

In order to test the performance difference between `heighH` and `height`. We designed the following test cases:

Define `a` as a `Tree` with height 3.

```
a = Node (TPair (2,0) (2,0)) [Node (TTrip (0,0,1) (0,1,0)) [Stop (Triple 0 1 7),Node (TTrip (1,0,0) (1,0,0)) [Stop (Triple 1 0 7),Stop (Triple 0 1 7),Stop (Triple 1 0 7)],Stop (Triple 0 0 8)],Node (TPair (0,2) (2,0)) [Node (TTrip (0,0,1) (0,1,0)) [Stop (Triple 0 1 3),Stop (Triple 0 1 3),Stop (Triple 0 0 4)],Node (TPair (1,0) (1,0)) [Stop (Triple 1 1 0),Stop (Pair 0 8),Stop (Triple 1 1 0)],Node (TTrip (0,0,1) (1,0,0)) [Stop (Triple 0 0 4),Stop (Triple 1 0 3),Stop (Triple 1 0 3)]],Node (TTrip (0,0,1) (0,1,0)) [Stop (Triple 0 1 7),Node (TTrip (1,0,0) (1,0,0)) [Stop (Triple 1 0 7),Stop (Triple 0 1 7),Stop (Triple 1 0 7)],Stop (Triple 0 0 8)]]
```

Define `b` as a `TreeH` with height 3.

```
b = NodeH 3 (TPair (2,0) (2,0)) [NodeH 2 (TTrip (0,0,1) (0,1,0)) [StopH (Triple 0 1 7),NodeH 1 (TTrip (1,0,0) (1,0,0)) [StopH (Triple 1 0 7),StopH (Triple 0 1 7),StopH (Triple 1 0 7)],StopH (Triple 0 0 8)],NodeH 2 (TPair (0,2) (2,0)) [NodeH 1 (TTrip (0,0,1) (0,1,0)) [StopH (Triple 0 1 3),StopH (Triple 0 1 3),StopH (Triple 0 0 4)],NodeH 1 (TPair (1,0) (1,0)) [StopH (Triple 1 1 0),StopH (Pair 0 8),StopH (Triple 1 1 0)],NodeH 1 (TTrip (0,0,1) (1,0,0)) [StopH (Triple 0 0 4),StopH (Triple 1 0 3),StopH (Triple 1 0 3)]],NodeH 2 (TTrip (0,0,1) (0,1,0)) [StopH (Triple 0 1 7),NodeH 1 (TTrip (1,0,0) (1,0,0)) [StopH (Triple 1 0 7),StopH (Triple 0 1 7),StopH (Triple 1 0 7)],StopH (Triple 0 0 8)]]
```

On top of that, we also have `a == (treeH2tree b)`:

For example:

```
*Main> a == (treeH2tree b)
True
```

We've run the following tests:

Test Case	Time Usage
height a	0.00 sec
heightH b	0.00 sec
minimum (map (\x -> a) [0..100])	0.04 sec

Test Case	Time Usage
minimum (map (\x -> b) [0..100])	0.04 sec
minimum (map (\x -> a) [0..10000])	0.30 sec
minimum (map (\x -> b) [0..10000])	0.05 sec

In conclusion, when given data set is small (either a smaller `Tree`, or smaller dataset), the performance difference between `height` and `heightH` is unnoticeable. However, the cached method works a lot better when there are more maps in the list as it doesn't need to traverse all the child nodes. Unfortunately, in our case, this is not the main bottleneck of `mktree (Pair 8 0)`.