# Functional Programming Assignment Report

| Key | Value |
| --- | --- |
| Author | Haihao Yan |
| Date | 31/03/2019 |
| Source Code | https://github.com/hyan36/fdr |

# Introduction

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

Firstly, we will need to import 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 is 3 coins in the pile, a, b and c. We will have following coinditions:

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 find the fake coins. However, this strategy creates a problem, when the size of pile can not be `mod` by 3, we will have 1 or 2 coins left in the pile whic can not be determined wether it is fake or not.

The solution is to take **n** coins from the genue pile. (Given we have only one fake coin in the pile. If the program hits the last few coins, all previous tested coins are genue.)

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

We then define a function to prefill the original list. So we can split the pile into smaller sizes 3. E.g. [ [1,2,1], [1,1,1]...[1,1,1] ].

```
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 coins. 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 the most obvious way of finding the fake coins. The complexity of this solution is O(n/3). When there is 12 coins we will use 4 weighs to find the fake coin.

# Solution

The core of our approach is to simulate a physical solution to the problem problems. The program will simulate all possible strategies and then choose the most optimized ways of weighing coins.

## States and test

Firstly, we need to define the data type State and Test to represent the state of the coins and our strategy of testing them.

```
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 only returns true when matching following condition

- 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 situation is invalid.

Thus, we will have the following definition.

```
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 we have a way to deterimine weather a state and test is valid or not. We will focus on getting the program to simulate the tests.

## Q2 - Define `outcomes`

We will now define a function to determine all possible outcomes given a pair of valid state and tests.

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

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

- balanced - all test coins goes to pile `g`
- ligther - when left pan is ligther than right pan, the fake coin can only comes from pile `a` or pile `e`. All other coins left on the table goes to pile `g`.
- heavier - when left pan is heavier than right pan, the fake coin can only comes from pile `b` or pile `b`. All other coins left on the table goes to pile `g`.

Thus, we will define `outcomes::State -> Test -> [State]` as follow:

```
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 requirement

- 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`

We will pick `k` number of coins for both 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`

We need to define a function to pick `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 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 instance of State, we will have following rules:

- Triple always less than Pair - it always provides more information than pair
- For either Pair to Pair / Triple to Triple - we always compare pile `g`, whomever has more coins in pile `g` provides 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 given more information than the existing state. Thus we will have `s' <= s`. When there is no outcomes it always return 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 all possible weighings by following conditions:

- if it is a **valid** test cases
- if it is a **productive** test cases

```
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 tree structure to represent the possible simulation.

Firstly, we will define a recursive `Tree` structure.

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

### Q9 - Define `final`

We will need a function to determine if a state is final. A final states will have 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 read the tree object recursively to cout the height of 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 are determined by their height. The benefit of this approach is that we can utilize the out of functions instead of defining our own. However, it changes the default of definition of `t1 < t2`. Given that in this assignment, we won't have other definition of the expression, I choose decide to follow this approach.

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

### Q12 - Define `mktree`

Now we will define a function which returns the tree with minimum height. The method will get all possible trees and return the tree with the smallest 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 a function which produces the tree with minium height (best strategy of weighing). However, the performance of which is relatively low. One of the potential reasons that been hinted is that we are calling `height` recursively. In this section, we are going to try to optimized performance by caching the height of trees.

Firstly, we need to define a `TreeH` data type.

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

Secondly, we will define `heightH` to return 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 lowest height.

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

### Q13 - Define `treeH2tree`

Following function will transform a given `Tree` to `TreeH`.

```
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 will make a `NodeH` with given `Test` and 3 nodes.

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

### Q15 - Define `tree2treeH`

Following function will covert given `Tree` to `TreeH`.

```
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. This 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 execution, we had following result:

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

### Q16 - Define `mktreeH`

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

#### Approach 1 - 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 just convert whatever output from `mktree` function.

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

#### Peformance Indication

Following table summarized our experiment of 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 experiement in **Appendix 2**, the performance of `height` and `heightH` is unnoticeable when given tree or data set is small. |
| mktreeH' | a lot slower than `mktree` | We have to calculate the height for more node than first approach and `mktree` |
| mktreeH'' | always slightly slower than `mktree` | obviously, it won't improve the performance because `mktree` is still going to be executed. Also according to **Appendix 2** the time lapse of transform a small tree (3 in height) is almost unnoticeable |

As a result, we will present `mktreeH` as our final implementation.

## 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 filter the test cases at local bases.

Firstly, we will copy the `optimal` method from 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 method will filter 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 will return a tree with the optimal tests. The algorithm is very similar to `mktree`. We didn't try other approach like what we've done for Q16. The reason for that is in this scenario, we want all node to be optimized.

```
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 will return a list of trees.

```
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 test for `bestTests (Pair 12 0)`.

# Appendix 1 - Unit Test

In order to justify the assumption in question 15, I implemented quick check tests to validate our theory. Although this can not proof the equation in theory, this still increases user's confidence.

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

Firstly, we need to create 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 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 new instance for Tree. Tree is recursive data type. It is important that we control the size of recursion. That is why I choose to use (n `div` 2) so it can 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 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)]]
```

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 |
| minimum (map (\x -> b) [0..100]) | 0.04 sec |

| Test Case | Time Usage |
|---|---|
| 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 unnoticable. However, the cached method works a lot better when there is more maps in the list as it doesn't need to traverse all the child nodes. Unfortunately, this is not the main bottle neck of `mktree (Pair 8 0)`.