Functional Programming (FPR)

Pawel Sawicz

25th March 2019; Total number of pages: 13

1. Introduction

1.1 Getting started

It is an essay for Functional Programming course at Software Engineering Programme. I have been given two tasks, firstly to solve "Twelve Coins" puzzle in Haskell, secondly to explain my decisions behind the code.

At the beginning of this essay, you might find a complete explanation of the fundamentals of functional programming and Haskell syntax. As we move along, you will find a more compacted description of my approach, as there is no point to repeat myself.

It should be noted here that puzzle related narrative is either copy or rephrased text from the assignment. I am aware of the `module` concept in Haskell, but in the case of this submission, I did not use it.

The puzzle described as follows:

> There are n $>$ 2 coins, identical in appearance; either all are genuine, or exactly one of them is fake. It is unknown whether the fake coin is lighter or heavier then genuine one. You have two-pan balance scale without weights. The problem is to find whether all the coins are genuine and, if not, to find the fake coin and to establish whether it is lighter or heavier than the genuine ones. Design an algorithm to solve the problem, in the minimum number of weighings.

1.2 Importing libraries

Haskell by default provides us access to `Prelude` library, which contains core functions of the language. To use functions from other libraries (modules), you need to use keyword `import` with a name.

```
import Data.List
import Data.Ord
import Data.Function
```

2. State and Test algebraic data types

Our algorithm simulates a physical solution to the problem. The solution must maintain up to four piles of coins: `U` as unknown, `G` as genuine, `L` as light, `H` as heavy and it goes through two distinct phases. The first phase we do not know whether any coin is fake; we can only consider pile U and G, that is our first state. If the two groups don't balance, then one of those coins must be fake; we move coins to L pile and H pile and rest of them to G pile. Now we are in the second state.

2.1 State

We are going to use algebraic data types (ADT) to represent some data in our program. In short, ADT is a composition type which is formed by a combination of other types. You can think of it similarly to a struct in C. We model state of a simulation by introducing `State` ADT, with two data constructors `Pair` and `Triple`. Those two data constructors are also a function.

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

2.2 Partial application

The `Pair` constructor is a type of `Int -> Int -> State`.

It means that we can look at this type in two ways. First, as a function that takes two arguments and returns a value of `State` type, or it takes one argument and returns a function that takes one argument and returns a value of `State`. In the latter case, we say that the function can be partially applied.

2.3 Test

We model conducted weightings by the data type `Test`, with two data constructors, `TPair` and `TTrip`. The `TPair` constructor takes two 2-tuples. The `TTriple` constructor takes two 3-tuples.

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

2.4 Product type or series of functions

As you can see, there are two different ways of representing an argument of a function as a product type (tuple) or as a series of unary functions. In Haskell, we can translate arguments represented as an n-tuple to a series of unary functions by using currying, and vice versa.

2.5 The keyword deriving

Another piece that needs an explanation is the keyword `deriving`. The keyword deriving allows us to make a datatype an instance of typeclass. To put it another way, we say that datatype has some behaviour. In the case of `State` and `Test`, it derives "Eq" and "Show" typeclass.

Generally, the compiler automatically finds out the default implementation for functions to make an instance of typeclass. Otherwise, we are forced to denote it manually as below:

```
instance Show Test where
    show :: Test -> String
```

Now we can start implementing our first function to solve this puzzle. We shall define `valid` function such that determines whether a given test is valid in a given state.

By using pattern matching, it makes sure that `TPair` test is conducted in a `Pair` state and a `TTrip` test in a `Triple` state. In addition to that, predicates check the validity of the test against the state:

1. The number of coins is the same in each pan of the scale
2. There are sufficiently many coins in the various piles for the test

```
valid :: State -> Test -> Bool
valid (Pair u g) (TPair (a, b) (c ,d)) =
    (a+b) == (c+d) &&
    (a+c) <= u &&
    (b+d) <= g
valid (Triple l h g) (TTrip (a, b, c) (d, e, f)) =
    (a+b+c) == (d+e+f) &&
    (a+d) <= l &&
    (b+e) <= h &&
    (c+f) <= g
```

3. Choosing and conducting a test

3.1 Constructing outcomes

The next function that we need to define is `outcomes` that for a given state `s` and test `t` such that `valid s t = True`, works out possible outcomes.

There are always three outcomes being generated. In a case when coins in both pans balance out, we know that all those coins are genuine, then we move them to G pile. Otherwise, we move coins from pans respectively to the lighter, heavier and genuine pile.

Note that the `outcomes` is a partial function if `valid s t` = False then it throws an error, otherwise proceed with the generation of outcomes.

```
outcomes :: State -> Test -> [State]
outcomes (Pair u g) (TPair (a, b) (c, d))
    | valid (Pair u g) (TPair (a, b) (c, d)) =
        [Pair unew gnew'] ++
        [Triple l h gnew] ++
        [Triple l h gnew]
    | otherwise = error "Invalid state or test"
        where
           unew = u - (a + c)
           gnew = unew + g
           gnew' = g + a + c
           l    = a
           h    = c
outcomes (Triple l h g) (TTrip (a, b, c) (d, e, f))
    | valid (Triple l h g) (TTrip (a, b, c) (d, e, f)) =
        [Triple lnew hnew gnew] ++
        [Triple lnew hnew gnew] ++
        [Triple lnew' hnew' gnew']
    | otherwise = error "Invalid state or test"
            where
                lnew = a+d
                lnew' = l-lnew
                hnew = b+e
                hnew' = h-hnew
                gnew = g+lnew'+hnew'
                gnew' = g+lnew+hnew
```

3.2 Weighings

The next function that we would like to implement is a function that generates sensible tests. See predicates for `Pair`, `Triple` below. Later we will make sure that tests make progress on a state by introducing special ordering (gives more information about genuine coins).

I used a set comprehension with predicates to generate accurate weighings. The `Pair` case implementation is straightforward. The `Triple` is a bit trickier because we need to use the subsidiary function `choices`.

Predicates for `Pair`:

1. `a + b > 0` - no point in weighting only air
2. `2*a+b <= u` - enough unknown coins
3. `b <=g` - enough genuine coins

Predicates for `Triple`:

1. `a+b+c = d+e+f` - same number of coins per pan
2. `a+b+c > 0` - no point in weighting only air
3. `c x f = 0` - don't put genuine coins in boh pans
4. `a + b <= l` - enough light coins
5. `b + e <= h` - enough heavy coins
6. `c + f <= g` - enough genuine coins
7. `(a,b,c) <= (d,e,f)` - symmetry breaker

```
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,
     b <= g]
weighings (Triple l h g) = [TTrip (a, b, c) (d, e, f) | k1<-[1..k],
     (a, b, c) <- choices k1 (l, h, g),
     (d, e, f) <- choices k1 (l, h, g),
     c == 0 || f == 0,
     (a,b,c) <= (d,e,f),
     (c+f) <= g, (b+e) <= h, (a+d) <= l,
     (a+b+c) == (d+e+f),
     (a+b+c) > 0]
        where
            k = (l+h+g) `div` 2
```

The `choices` function uses a set comprehension with predicates to generate valid selections of `k` coins. My initial implementation did not include `let e = (k-i-j)`, so a value of `(k-i-j)` would have been recalculated every time, by introducing local variable I saved a significant amount of CPU cycles due to the space of the solution.

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

3.3 Special purpose ordering as instance of `Ord` typeclass

To check if the `State` is making progress, we model this in term of special purpose ordering, that we use to determine whether the state gives strictly more information about the coins. Because the number of coins is preserved, it generally suffices to make a comparison by the number of genuine coins. Moreover `Triple` state always represent progress from `Pair` state.

Previously I mentioned that sometimes we are forced to implement an instance of typeclass manually. It is when the compiler cannot generate an implementation for a type class function for an unusual data type, or when there is a need for custom implementation in a specific domain context. In our case, it is this special ordering that we would like to have on the number of genuine coins.

According to the documentation (hackage), an instance of the `Ord` requires a either of function to be implemented: `(<=) :: a -> a -> Bool` or `compare :: a -> a -> Ordering`

```
instance Ord State where
    (Pair _ _) <= (Triple _ _ _) = False
    (Pair _ g1) <= (Pair _ g2) = g2 <= g1
    (Triple _ _ g1) <= (Triple _ _ g2) = g2 <= g1
```

3.4 Productive tests

Now with all previous work done, we can implement a `productive` predicate, that checks if all outcomes are making progress. I used the higher order function `all` with a partially applied predicate on the outcomes function.

```
productive :: State -> Test -> Bool
productive s t = all (s > ) (outcomes s t)
```

Finally, we are ready to fulfil the third criterion by keeping only the `productive` tests among the possible `weighings`.

The `test` function is made up of three other functions: the `weighings`, the `productive` and the `filter`. The filter function filters out a collection by a provided predicate and preserves just those elements which follow predicate. Similarly to the previous function, I can take advantage of a partially applied predicate..

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

4. Decision tree

The `Tree` data type represents a weighting process. It's a ternary tree that contains itself. In other words, it's a recursive datatype.

`Tree` has two data constructors:

1. `Stop`, represents a final state; it's a leaf of the tree.
2. 'Node represents a weighting, and it's a node of the tree.

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

4.1 Constructing a tree

Let's now implement functions that will help us to construct a valid weighting process and represent it as our `Tree` data type.

Firstly, the `final` is a predicate that determines whether a `State` is final. The state is final when all coins are genuine or that one coin is fake. I use a pattern matching and guards to check that.

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

The `height` function calculates the height of a tree. If it's a leaf (Stop) then returns zero. Otherwise (Node), it recursively calculates the height of a tree and then selects a maximum value.

```
height :: Tree -> Int
height (Stop s) = 0
height (Node _ xs) = 1 + maximum (map height xs)
```

The `minHeight` function is a partial function that throws an error for an empty list. For a non-empty list, it returns a tree with the smallest height from the collection.

```
minHeight :: [Tree] -> Tree
minHeight [] = error "Tree cannot be empty"
minHeight xs = minimumBy (compare `on` height) xs
```

Compare above with my initial implementation that is below. Ones use `sortBy`. At first sight, those two functions look similar, but in fact, their performance differs.

```
minHeight' xs = head $ sortBy (compare `on` height) xs
```

4.2 Tree

Finally, we can define our function that constructs a solution process as a tree. For all productive tests generates tree recursively for each of the outcomes of each such test, then pick up the one that yields the best tree overall.

```
mktree :: State -> Tree
mktree s
    | final s = Stop s
    | otherwise = minHeight (map subTree (tests s))
        where
            subTree t = Node t (map mktree (outcomes s t))
```

5. Caching heights

5.1 Introducing height

Program in the previous version works, but it is rather slow. One of the problems is that it is recomputing the heights of trees. Now we introduce the notion of height on each Node so that we can compute it in constant time.

We define new datatype `TreeH`. As you can see it's very similar to the previous `Tree` with one change, which is height in (NodeH) constructor.

```
data TreeH = StopH State | NodeH Int Test [TreeH]
 deriving Show
```

Now we need to implement a set of new functions that work on `TreeH` rather than on `Tree` so that later on we can use it to construct the weighting process as `TreeH`.

The `heightH` extracts height out of `TreeH` type. This function is predefined in the assignment text.

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

The `treeH2tree` maps `TreeH` type to `Tree`.

```
treeH2tree :: TreeH -> Tree
treeH2tree (StopH s) = Stop s
treeH2tree (NodeH h t ths) = Node t (map treeH2tree ths)
```

The `nodeH` is a smart constructor that for, a given `Test` and a list of trees, construct a new tree with height. We could use a function composition: `((+ 1) . maximum))` but in my opinion, it would deteriorate the readability.

```
nodeH :: Test -> [TreeH] -> TreeH
nodeH t ths = NodeH h t ths
            where
                h = (+ 1) $ maximum (map heightH ths)
```

The `tree2treeH` is an inverse function of the `treeH2tree`.

```
tree2treeH :: Tree -> TreeH
tree2treeH (Stop s) = StopH s
tree2treeH (Node t ts) = nodeH t (map tree2treeH ts)
```

As you can notice a composition of `heightH` and `tree2treeH` (such that `heightH` after `tree2treeH`) is equal to = `height`. This equality holds due to function composition law.

Finally, we can implement a function that constructs a tree for a given state.

```
mktreeH :: State -> TreeH
mktreeH s
    | final s = StopH s
    | otherwise = minimumBy (compare `on` heightH) $ map subTree (tests s)
        where
            subTree t = nodeH t (map mktreeH (outcomes s t))
```

As it was stated in the assignment text. This approach does not massively improve performance.

6. Greedy solution

This function was copied from the assignment description.

```
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)))
```

The `bestTests` filters out non-optimal weighings. I used a similar approach as in the `tests` function.

```
bestTests :: State -> [Test]
bestTests s = filter (optimal s) (weighings s)
```

The `mktreeG` builds a tree similar to the `mktreeH`, but it only uses a first optimal test, rather than exploring all possible tests as it is a case in the `mktreeH` and `mktree`.

```
mktreeG :: State -> TreeH
mktreeG s
    | final s = StopH s
    | otherwise = (\t -> nodeH t (map mktreeG (outcomes s t))) bestTest
        where
            bestTest = head (bestTests s)
```

The `mktreesG` builds trees based on all optimal tests.

```
mktreesG :: State -> [TreeH]
mktreesG s
    | final s = [StopH s]
    | otherwise = makeTree
        where
            optimalTests = bestTests s
            makeTree = map (\t -> nodeH t (map mktreeG (outcomes s t))) optimalTests
```

7. Conclusion

Firstly, I did not manage to solve this puzzle correctly. Constructing a tree does not return the correct minimal height tree. For n=8, Program outputs tree of height four, rather than three. I presume that I have made a mistake in the implementation of `outcomes` function. There is also another discrepancy. The function `tests` for (Triple 3 0 6), returns three tests. However, it supposes to return four productive tests (as in the assignment).

My program runs slow in comparison with an example's answer from the assignment. For `Pair (8 0)` elapsed time is about (150sec.) and allocated space (94GB).

Haskell is a perfect tool for mathematical puzzles, domain modelling, concurrent and parallel computation. For instance, code is much more compacted in compare with popular languages like (C, C#, Java, Python). Another characteristic of Haskell is that it is statically typed, which allows us to spot type errors at the compilation time instead of runtime. That encourages programmers to sketch a scaffold of a program by decomposing problem into smaller pieces and denoting just a functions' types. Then we can implement the most naive version of our program and work on improvements.

There exists tooling around the platform, there is a build tool, styling checker (HLint) which I was actively using during my coding, and it warned me that I should not use lambdas (there were few lambdas here and there).

In some functions' implementation, I decided to be more explicit in terms of puzzle logic (mainly in predicates of the `outcomes`, `weightings`), primarily to increase the readability of the puzzle related logic.

Last but not least, there are a few areas where I would like to apply further improvements on my code:

1. Error handling, by using `Either` monad pattern (although it could be an overkill in this case)
2. Add test coverage for some functions, like `outcomes` using QuickCheck.
3. Try to make Tree and TreeH a monoid; it could simplify some functions. Although in current definition it does not follow monoid laws.