Functional Programming (FPR)

Pawel Sawicz

25th March 2019

Pages : Total number of pages : 10

Introduction

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.

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.

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.

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

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.

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 a little bit like a struct in C. We model state of a simulation by introducing `State` ADT, it has two data constructors `Pair` and `Triple`.

Everything in Haskell is a function; therefore data constructor is also a function.

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

Partial application

`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 latter, we say that function can be partially applied.

Test

We model conducted weightings by the data type `Test`, it has two data constructors, `TPair` and `TTrip`. `TPair` constructor takes two 2-tuple. `TTriple` constructor takes two 3-tuple partial application also applies to both constructors.

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

Cartesian product and series of functions

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

deriving keyword

Another piece that needs an explanation is the keyword `deriving`. 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 default implementation for functions to make an instance of typeclass. Otherwise, we are forced to denote it manually as below:

```haskell
instance Eq Test where
    show :: a -> 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

```haskell
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
```

Choosing and conducting a test

Constructing outcomes

Next function that we need to define is `outcomes` such that for state `s` and test `t` 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.

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

```haskell
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
```

Weighings

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

I used set comprehension with predicates to generate accurate weighings. `Pair` case implementation is straightforward, `Triple` is a bit trickier, because we need to uses 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
```

`choices` function uses 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 some CPU cycles.

```
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]
```

Special purpose ordering as instance of `Ord` typeclass

How to keep just productive tests? In order to check if `State` is making progress, we model this in term of special purpose ordering, which we use to determine whether state gives strictly more information about the coins. Because the number of coins is preserved, it generally suffices to do 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 manually implement an instance of typeclass, it is when the compiler cannot generate an implementation for type class function for an unusual data type, or when there is a need for custom implementation in a specific context. In our case, it is this special ordering that we would like to have on the number of genuine coins.

Typeclass `Ord` has two functions that are mandatory to be provided with implementation:

```
1. (<) :: a -> a -> Bool
2. (<=) :: a -> a -> Bool
```

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

    (Pair _ _) <= (Triple _ _ _) = False
    (Pair _ g1) <= (Pair _ g2) = g2 <= g1
    (Triple _ _ g1) <= (Triple _ _ g2) = g2 <= g1
```

Productive tests

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

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

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

`test` function is made up of three other functions: `weighings`, `productive` and `filter`. Filter function as the name indicates it filters out a collection by a provided predicate and preserve 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)
```

Decision tree

Now we can introduce `Tree` data type that 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 the 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)
```

Constructing a tree

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

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

```
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
```

`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)
```

`minHeight` is a partial function that throws an error for an empty list. For non-empty list, it returns 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. It uses `sortBy`. At first sight, those two functions look the same, but in fact performance wise they are not.

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

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))
```

Caching heights

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 weighting process as TreeH.

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

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

treeH2tree maps TreeH type to Tree, after map TreeH loses direct access to its height.

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

nodeH is a function that for, given Test and list of trees, construct a new tree with height.

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

tree2treeH is just an inverse of 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.

If you don't belive me, let's do quick type reasoning. `heightH` has a type `TreeH -> Int`, `tree2treeH` has a type `Tree -> TreeH`. Composition of those two functions has type `Tree -> TreeH -> TreeH -> Int`, so there exist a function such that has a type of `Tree -> Int` (composition law), which is a type sygnature of our `height` function.

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.

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)))
```

`bestTests` filters out not optimal weighings. I used a similar approach as in `tests` function.

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

`mktreeG` builds tree similarly to `mktreeH`, out of the first optimal test, rather than exploring all possible tests as it's a case in `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)
```

`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
```

Conclusions

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. In function `tests` which returns three tests, however, it supposes to return four productive tests (as it is noted 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, a code is much more compacted in compare with popular languages like (C, C#, Java, Python). Another characteristic of Haskell is that it is strongly typed, which allows us to spot type errors at the compilation time instead of runtime.

Haskell 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.

Tooling around the platform is also quite good, there is a build system, styling checkers (HLint) which I was actively using during my coding and it hints me that I should not use lambdas (there were few lambdas here and there).

In a some functions' implementation, I decited to be more explicit in terms of puzzle logic (predicates on set comprehension: function `outcomes`, `weightings`) mainly to increase the visibility of puzzle requirements.

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 a monoid; it could simplify some functions. Although in current definition it does not follow monoid laws.