Functional Programming (FPR)

Pawel Sawicz

25th March 2019

Introduction

Getting started

This document 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 and 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 find more compacted descriptions of my approach, as there is no point to repeat myself.

It should be noted here that puzzle related narrative uses the content of assignment text. Those paragraphs are either copy or rephrased.

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

In Haskell by default, you are given access to `Prelude` library, which contains core functions of the language. To use functions from other libraries, you need to use keyword `import`, which imports `module`.

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

State and Test Algebraic data types

Our algorithm simulates 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. The solution 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 k coin to L pile, k coins to H pile and rest of them to G pile. Now we are in the second state.

State

We model state of the simulation by the data type `State`.

State data type has two data constructors `Pair` and `Triple`.

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

```
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, a function that takes two arguments and returns `State`, or it takes one argument and returns a function that takes one argument and returns `State`. In latter, we say that function can be partially applied.

Test

We model conducted weightings by the data type `Test.`

`Test` datatype has two data constructors, `TPair` and `TTrip`. `TPair` constructor takes two 2-tuple.

```
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. In the case of `State` and `Test`, it derives (Eq and Show typeclass). Generally, the compiler automatically finds out default implementation to make an instance of typeclass. Otherwise, we are forced to make an instance of the desired typeclass by denoting manually:

```
instance Eq Test where
```

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

By using pattern matching this guard that `TPair` test is conducted in a `Pair` state and a `TTrip` test in a `Triple` state.

In addition to that, some 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 &&
    (a+b+c+d) <= (u+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

We shall define a function `outcomes` such that for state `s` and test `t` that `valid s t = True`, works out the possible outcomes.

There is always three outcomes 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 pile and the heavier pan.

`outcomes` is a partial function, if `valid s t` = False then return an error, otherwise proceed with 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)) == True =
        [Pair un gc] ++
        [Triple l h gcc] ++
        [Triple l h gcc]
    | otherwise = error ("Invalid state or test" ++ (show (Pair u g)))
        where
           un =  (u - (a + c))
           gcc = (u - (a + c)) + g
           gc  = 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)) == True =
        [Triple (a+d) (b+e) (g+(l-(a+d))+(h-(b+e)))] ++
        [Triple (a+d) (b+e) (g+(l-(a+d))+(h-(b+e)))] ++
        [Triple (l-(a+d)) (h-(b+e)) (g+a+b+d+e)]
    | otherwise = error ("Invalid state or test:" ++ " "
        ++ (show (Triple l h g)) ++ " "
        ++ (show (TTrip (a, b, c) (d, e, f))))
```

Weighings

Next function that we would like to implement is to generate sensible tests. All criteria for a sensible test are described below (predicates for `Pair`, `Triple`). Later we will make sure that tests make progress on a state by introducing special ordering.

I used set comprehension with below predicates to generate valid weighings. Set comprehension is the only way to generate data set in Haskell. `Pair` case implementation is straightforward, `Triple` case uses subsidiary function `choices`.

Predicates for `Pair`:

1. `a + b > 0`

2. `2*a+b <= u`
3. `b <=g`

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+
       where
           k = (l+h+g) `div` 2
```

`choices` function uses set comprehension with predicates to generate valid selections of `k` coins.

```
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) <= g,
                      (k-i-j) >= 0]
```

Special purpose ordering as instance of `Ord` typeclass

Third criterion. 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.

In order to implement this special purpose ordering, we going to make an instance of `Ord` type class on `State` type.

Previously in the section about `deriving` keyword, I explained that the compiler usually finds out default implementation for type class functions, in a case where it can't we need to provide a custom implementation of typeclass functions.

Typeclass `Ord` has two functions that are mandatory to be provided with implementation, if you want an instance of this type class.

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 implement `productive` a predicate, that checks if all outcomes are making progress. I used `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 composed 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 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 weighting, and it's a node of the tree.

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

Constructing a tree

Let's now implement some functions that 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.

```haskell
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.

```haskell
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 calculates the height of each of the element, then returns a 2-tuple of Tree and its height. Then sort a list by height, lastly select the first element.

```haskell
minHeight :: [Tree] -> Tree
minHeight [] = error "Tree cannot be empty"
minHeight xs = snd
    $ head
    $ sortBy (compare `on` (\(y,_) -> y))
    $ map (\x -> (height x, x)) xs
```

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.

```haskell
mktree :: State -> Tree
mktree s
    | (final s) == True = Stop s
    | otherwise = minHeight
        $ map subTree
        $ productiveTests
        where
            productiveTests = tests s
            subTree = (\t -> (Node t (map mktree (getOutcomes s t))))
            getOutcomes = (\s t -> (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 ((+) 1 $ maximum $ map heightH ths) t 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. Let's do quick type check. `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) == True = (StopH s)
    | otherwise = head $ sortBy (compare `on` heightH) $ subTree $ productiveTests
```

```
        where
            productiveTests = tests s
            subTree = map (\t -> nodeH t (map mktreeH (getOutcomes s t)))
            getOutcomes = (\s t -> (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.

```
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) == True = (StopH s)
    | otherwise = subTree $ bestTest
        where
            bestTest = head $ bestTests s
            subTree = (\t -> nodeH t (map mktreeG (getOutcomes s t)))
            getOutcomes = (\s t -> (outcomes s t))
```

`mktreesG` builds trees based on all optimal tests.

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

Conclusions

Firstly, I did not manage to solve this puzzle correctly. Constructing a tree does not return correctly minimal height tree. For n=8, I am getting, tree of height four rather than three. I presume that I have made a mistake in the implementation of `outcomes` function.

Haskell is a perfect tool for mathematical puzzles, domain modelling concurrent and parallel computation. The code is much more compacted in compare with popular languages like (C, C#, Java, Python). That is because Haskell by default support features that other languages do not, like :

1. tail-recursion
2. lazy evaluation
3. high-order functions
4. strong types

This puzzle merely shows us the power of the functional paradigm and Haskell. There is a whole separate field of study that concerns about patterns of behaviour in abstractions (Category Theory).

I am aware of the `module` concept in Haskell, in the case of this submission I did not use it.

There are a few areas where I would like to improve 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. I could try to make Tree foldable; it could simplify some functions.