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 weighting puzzle in Haskell and explain my reasoning and decisions behind the code.

At the begining of this essay you might find a lot of explanation about fundamentals of functional programing (FP) theory and Haskell syntax.

As we move along you will find less text and more compacted descriptions of my approach, as there is not point to repeat myself.

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

It should be noted here that puzzle related explanation uses content of assignemnt text.

Importing liblaries

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

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

State and Test Algebraic datatypes

What is Algebraic datatype ?

First we shall define two algebraic datatypes, `State` and `Test`.

State

State datatype has two data contructors `Pair` and `Triple`. It should be noted here that everything in Haskell is a function therefore data constructor is also a function.

Partial application

`Pair` constructor is a type of `Int -> Int -> State`. It means that we can looks at this in two ways. First, function 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.

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

Test

Test datatype has two data constructors, `TPair` and `TTrip`. `TPair` constructor takes two tuples. Tuple represents cartisian product of an arguments.

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

Cartisian product vs series of functions

As you can see there are two diffrent ways of representing an argument of a function. As a cartesian product or as a series of functions.

In Haskell we have ability to transpose cartisian product to series of functions by using currying, and vice versa.

deriving keyword

Another piece that needs an explenation is keyword `deriving`. Keyword deriving allows us to make a datatype an instance of typeclass, in the instance of `State`, `Test` it derrives (Eq and Show). Compiler automatically finds out default implementation for instance of typeclass.

Alternatively we can manually make an instance of desired typeclass by denoting

```
instance Eq Test where
```

We shall define valid function such that determine whether a given test is valid in a given state.

By using pattern matching this guards that `TPair` test will be only conducted in a `Pair` state, and a `TTrip` test in a `Triple` state.

In addition to that there are predicates that checks validity of test against state.

We denoted following predicates :

1. The number of coins is the same in each pan of the scale
2. There is sufficiently many coins in the variouse 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 define function outcomes such that for state `s` and test `t` that `valid s t = True`, works out the possible outomes.

This 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

We uses set comprehension in order to generate valid weighings. As you can notice there are few predicates to generate sensible tests.

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]
```

Instance of typeclass

Previously I mentioned about `deriving` keyword that informs compiler that this datatype has some behaviour. `deriving` keyword works in most of the cases, but if you need custom implementation of functions that are part of typeclass, then you need to manually set up an instance of typeclass.

Typeclass `Ord` has two functions.

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

In order to check if `State` is making a progress, we modell this in therm of special purpose ordering, which we use to determine whether state gives strictly more information about the coins.

```
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

Productive function checks if all outcomes making a progress. I used `all` function that checks if all elements fulfied predicate.

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

Test function is composed by three other functions: `weighings`, `productive` and `filter`. First two are defined by us. Filter is part of library that we imported. Filter function as name indicates it filters out an collection by provided predicate, and preserving just those elements which follow predicate

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

Decision tree

Now we can introduce `Tree` datatype that represents a weighting process. It's a ternary tree that contains itself. We can also say that it's recursive datatype. `Tree` has two data constructors:

1. `Stop` represents final state, it's a leafe of the tree.
2. 'Node represtens weighting, 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 help us to construct a valid weighting process and represent it as Tree.

Final is a predicate that determine whether `State` is final.

```
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 height of a Tree. For `(Stop s)` simply returns 0. For `(Node _ xs)`, it recursivly calculates height of a tree and then selects 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 empty list. For non empty list it calculates height of each of the elements, then returns a tuple of Tree and its height. At the end collection is sorted by height we select first element.

```
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 will construct weighting process as a Tree. For all productive tests generates recursivly tree for each of the outcome.

```
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))
```

Running example for `mktree (Pair 6 0)`, some conclusions ??

Caching heights

Introducing height

Program in the previouse sections works but it is rather slow. One of the problems is that it is recomputing the heights of trees. Now we will introduce 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 simillar to previouse `Tree` with one change. We introduced notion of height in the `NodeH` constructor.

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

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

heightH function extracts height out of `TreeH` type.

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

treeH2tree function that maps `TreeH` type to `Tree`. After mapping `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 tries, it will construct new `TreeH`.

```
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 `heightH . tree2treeH = 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` it can be simplified to `Tree -> Int` (composition law), which is a type sygnature of our `height` function.

We could also prove it by induction.

Finally we can implement function that constructs tree for 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 an assignment. This approach does not massivly improve performance.

Greedy solution

This function was copied from 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)))
```

Function that filters unoptiman weighings, leaving just optimals.

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

Function that builds tree out of the first optimal test.

```
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))
```

Function that builds tries based on all optimal tests

```
mktreesG :: State -> [TreeH]
mktreesG s
    | (final s) == True = [StopH s]
```

```haskell
    | otherwise = concat $ makeTree
        where
            optimalTests = bestTests s
            makeTree = map (\t -> map mktreeG (outcomes s t)) $ optimalTests
```

Conclusion

Haskell is a perfect tool for mathematical puzzles. Code is much more compacted in compare with more popular languages like (C, C#, Java). That is because Haskell by default support features that other languages do not, like :

1. tail-recursion
2. lazy evaluation
3. high-orderism

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

What to improve, ideas ?

Error handling patter, by using Either bifunctor. pattern