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, solve weighting puzzle, and explain haskell. To separate those two tasks and to improve clarity of this essay I have introduced two sections `Puzzle` and `Haskell`.

Puzzle section in an narrative around a problem and setps that leads to solution. Haskell section contains explanation to syntax and functional programming knowledge.

Haskell

In order to use some functions, I imported two modules. Import keyword imports modules, and makes its content aviable for us.

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

State and Test Algebraic datatypes

State

Puzzle

Intro

Haskell

First we shall define two algebraic datatypes. `State` and `Test`. State datatype has two contructors `Pair` and `Triple`. It should be noted here that everything in Haskell is a function therefore constructor is also a function.

`Pair` constructor is a type of `Int -> Int -> State`. It means that we can looks at this two ways. It takes two arguments and returns `State`, or it takes one argument and returns a function that takes one argument and returns `State`. Which is called partial application.

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

Test

Test data type has two constructors, `TPair` and `TTrip`. `TPair` constructor takes two tuples. Tuple represents cartisian product. As you can see there are two diffrent ways of representing a arguments. As cartesian product or as a series of functions in the case of `State` constructors.

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

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

Another piece that needs an explenation is keyword `deriving`. Keyword deriving allows us to make a instance of type classes (Eq and Show).

Otherwise we would need to make manually an instance of desired type class by denoting

```
instance Eq Test where
```

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. if the number of coins is the same in each pan of the scale if there is sufficiently many coins in the variouse piles.

```
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

OUTCOMES TBD

```
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)))] ++ -- 1 1 10
        [Triple (a+d) (b+e) (g+(l-(a+d))+(h-(b+e)))] ++ -- 1 1 10
        [Triple (l-(a+d)) (h-(b+e)) (g+a+b+d+e)] -- 2 2 8
    | otherwise = error ("Invalid state or test:" ++ " "
        ++ (show (Triple l h g)) ++ " "
        ++ (show (TTrip (a, b, c) (d, e, f))))
```

Weighings

Weighings function has diffrent implementation for each of State.

```
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

```
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]
```

This is a case of manually set up an instance of type class.

```
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
```

Test function uses two other functions. `weighings`, `productive` and `filter`. First two are defined by us. Filter is part of lib. Which filters out an collection by provided predicate

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

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

Decision tree

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

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

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

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

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

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

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

nodeH :: Test -> [TreeH] -> TreeH
nodeH t ths = NodeH 0 t ths

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

Greedy solution

```
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` (1-a-d+h-b-e
        where
            p = 3 ^ (t - 1)
            t = ceiling (logBase 3 (fromIntegral (l+h)))

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