Real-world Functional Programming

Coursework Part I Report

14274056 Junsong Yang (psyjy3)

November 5, 2019

1 Task I.1

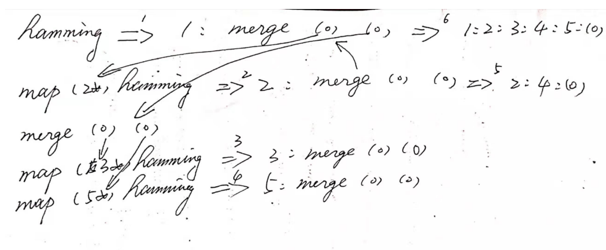
```haskell


hamming :: [Int]
hamming = 1 : merge (map (2*) hamming) (merge (map (3*) hamming) (map (5*) hamming))

merge :: [Int] -> [Int] -> [Int]
merge xss@(x:xs) yss@(y:ys)
  | x == y = x : merge xs ys
  | x < y  = x : merge xs yss
  | x > y  = y : merge xss ys

```

(a) Hamming Function Definition



(b) Cyclic Graph

Figure 1: TaskI.1

The hamming function can be defined as an infinite recursive list. As Figure 1 (a) shows, the type of hamming function is a list of Int. This implementation using the map function to calculate 2x, 3x and 5x and then merge them together as Figure 1 (b) demonstrated how it was evaluated.

2 Task I.2

```haskell
data Exp = Lit Double
         | Ref CellRef
         | Sum CellRef CellRef
         | Avg CellRef CellRef
         | App BinOp Exp Exp



evalCell :: Sheet Double -> Exp -> Double
evalCell _ (Lit v)        = v
evalCell s (Ref r)        = s ! r
evalCell s (Sum c1 c2)    = foldr (+) 0 l
  where
    l = [ s ! (c, r) | c <- [(fst c1)..(fst c2)],
                       r <- [(snd c1)..(snd c2)] ]
evalCell s (Avg c1 c2)    = (foldr (+) 0 l) / (fromIntegral llen)
  where
    l = [ s ! (c, r) | c <- [(fst c1)..(fst c2)],
                       r <- [(snd c1)..(snd c2)] ]
    llen = length l
evalCell s (App op e1 e2) = (evalOp op) (evalCell s e1) (evalCell s e2)

```

Figure 2: Extended Evaluator

Figure 2 demonstrates how evalCell function was extended to support Sum and Avg expression. The process of this implementation is given two CellRef c1 and c2, find every cell in between. Next, lookup corresponding values in the given sheet s. Then put all the values found in sheet s in list l. Finally using foldr to calculate the sum of all values.

The similar idea was used to implement the evaluation of Avg expression but a further step was taken to yield the average value.

The most obvious weakness of this evaluator is that it will be stuck in an infinite loop when there are circular references in the given sheet. As the type signature indicates the evalCell function will return a Double for every given sheet s and an Exp, which means when the Exp is Ref, it will keep evaluating until a Double can be returned. If there is a circular reference, this evaluator will keep evaluating without return.

One simple solution to this problem is to maintain a separate list to keep track of the evaluation of Ref Exp. This list works like a stack but disallowing duplicated elements. A cell will be pushed to the stack if it contains Ref Exp as well as the reference cell. Then continue evaluating that reference cell, if it returns, which means there is no circular reference. Hence those cells can be popped off the stack. If a cell that contains Ref Exp already exists in the stack that there must be a circular reference. Therefore, the evaluator can just return fail without continue evaluating.

3 Task I.3

In skew binary random access list, a number is represented by a sequence of digits starting from least significant bit to the most significant bit. Each digit is associated with weight predefined by the positional system.

```haskell

drop :: Int -> RList a -> RList a
drop _ [] = []
drop 0 t  = t
drop i (t:ts)
  | i == fst t = ts
  | i > fst t  = drop (i - fst t) ts
  | otherwise  = drop' i (fst t) (snd t) ++  ts

drop' :: Int -> Int -> Tree a -> RList a
drop' 0 _ (Leaf a) = [(1, (Leaf a))]
drop' _ _ (Leaf _) = []
drop' 0 w n@(Node _ _ _) = [(w, n)]
drop' i w (Node l _ r)
  | i <= half = drop' (i - 1) half l ++ [(half, r)]
  | otherwise = drop' (i - half -1) half r
  where
    half = w `div` 2

testDrop :: Int -> Bool
testDrop i = l1 == l2
  where
    l1 = last $ take i $ iterate (cons 1) empty
    l2 = drop i $ last $ take (i*2) $ iterate (cons 1) empty

~
- 2.2k SBRAL.hs   Haskell   Git-master
```

Figure 3: Drop Function

Figure 3 shows the implementation of drop function which drops the first n elements from the list. The drop function takes an integer and an RList as input and returns an RList without the first n elements as output. If the tree is empty which means nothing can be dropped, hence an empty list is returned. If the input integer is zero which means nothing needs to be dropped, then the same list taken as input will be returned. If the size of the first element in the list equals the number of elements needs to be dropped, then the tail of the input list will be returned. If the number of elements need to be dropped is greater than the size of the first element in the list, then the first element will be dropped and the drop function will be called recursively to continue processing the remaining list. The operations mentioned above would run in constant time.

The next case of the drop function determines the overall time complexity. If the number of elements needs to be dropped is less than the size of the first element in the list, then a helper function drop' will be called. The drop' function takes two integers and a tree as input and returns an RList. Similar to the first two cases of drop function, the first three cases of drop' function are practically doing nothing. Since the input tree is complete binary leaf tree, the size of the tree will be halved for every element dropped from the tree. Therefore, the overall time complexity is $O(logn)$. If the number of elements needs to be dropped is less than or equals to the half size of the input tree, then only the left children of the input tree will be processed. Otherwise, the left children will be dropped and the right children will be processed accordingly. The drop' function will be called recursively until the drop is finished or there is nothing to drop and return the final result.

A test function called testDrop is created to test the drop function. The key idea is the RList which represents n should equal to the RList that represents 2n without the first n elements.

4 Task I.4

```haskell
module Intervals (newIvl, (+/-))  where

data Ivl = Ivl Double Double deriving Show

instance Num Ivl where
  Ivl a b + Ivl c d = Ivl (a + c) (b + d)
  Ivl a b - Ivl c d = Ivl (a - d) (b - c)
  Ivl a b * Ivl c d
      | (not . null) $ filter isNaN l = error "operation undefined -- NaN"
      | otherwise = Ivl (minimum l) (maximum l)
    where
      l = [ x * y | x <- [a, b],
                    y <- [c ,d]]

  abs i@(Ivl a b) | a >= 0    = i
                  | b <= 0    = Ivl (-b) (-a)
                  | otherwise = Ivl 0 (max (-a) b)

  signum (Ivl a b) = Ivl (signum a) (signum b)

  fromInteger i = Ivl (fromInteger i) (fromInteger i)

instance Fractional Ivl where
  fromRational r = Ivl (fromRational r) (fromRational r)

  Ivl a b / Ivl c d
    | (not . null) $ filter isNaN l = error "operation undefined -- NaN"
    | c == 0 || d == 0              = error "operation undefined -- divided by zero"
    | otherwise                     = Ivl (minimum l) (maximum l)
    where
      l = [ x / y | x <- [a, b],
                    y <- [c, d]]

(+/-) :: Double -> Double -> Ivl
(+/-) a b = Ivl (min x y) (max x y)
  where
    x = a + b
    y = a - b

newIvl :: Double -> Double -> Ivl
newIvl a b | a <= b = Ivl a b
           | otherwise = error "lower bound cannot be greater than upper bound"

-- for testing
instance Eq Ivl where
  Ivl a b == Ivl c d = a == c && b == d

testIvl :: Bool
testIvl = (-3) +/- (-4) == Ivl (-7) 1   &&
          1 +/- 0.5     == Ivl 0.5 1.5  &&
          1 +/- 2       == Ivl (-1) 3
```

Figure 4: Intervals

Figure 4 shows how Ivl was made an instance of Num and Fractional class. As an instance of Num class, (+), (-) and abs functions were implemented according to the rule of interval arithmetic.

As for (*) function, there are a few cases that NaN may be yield which means the arithmetic operation is illegal. Hence, the (*) function will return error every time NaN occurs. The signum function takes an Ivl as input and returns an Ivl with the signum value of lower bound and signum value of the upper bound. The fromIntegral function takes an int as input and returns an Ivl with the same lowwer and upper bound value converted from the input.

The fromRational function in Fractional class works similarly as the fromIntegral function. The (/) function also employed the same manner to deal with NaN. Furthermore, in this implementation, divided by zero is not allowed. Hence, if divided by zero ever occurs, an error will be returned.

As for the (+/-) function, the key idea is to calculate the symmetric interval around a specific number. The input numbers may be negative, to deal with this situation, the lower bound of the resulting Ivl is the smaller value of the result of plus and minus of the two input value while the upper bound is the larger value of the result of plus and minus of the two input values. A few test cases are provided in the testIvl function to test the result of the (+/-) function.

5 Task I.5

```haskell
statistics :: Drawing Object -> Statistics
statistics dobject =
  Statistics {
    avgArea = asSumArea accum / (fromIntegral $ asCount accum),
    avgCircumference = asSumCircumference accum / (fromIntegral $ asCount accum),
    maxArea = asMaxArea accum,
    maxCircumference = asMaxCircumference accum }
  where
    accum = accumStats (AccumStats 0 0 0 0 0) dobject


accumStats :: AccumStats -> Drawing Object -> AccumStats
accumStats accum (Element object) =
  AccumStats {
    asCount = asCount accum + 1,
    asSumArea = asSumArea accum + area object,
    asSumCircumference = asSumCircumference accum + circum object,
    asMaxArea = max (asMaxArea accum) (area object),
    asMaxCircumference = max (asMaxCircumference accum) (circum object) }
accumStats accum (Group [])     = accum
accumStats accum (Group (x:xs)) = accumStats (accumStats accum x) (Group xs)


area :: Object -> Area
area (Rectangle{ .. }) = width * height
area (Circle{ .. })    = pi * radius * radius

circum :: Object -> Length
circum (Rectangle { .. }) = (width + height) * 2
circum (Circle { .. })    = pi * radius * 2

```

(a) Recursive Statistics

```haskell
instance Foldable Drawing where
  foldMap f (Element object) = f object
  foldMap f (Group [])       = mempty
  foldMap f (Group (x:xs))   = foldMap f x <> foldMap f (Group xs)

instance Semigroup AccumStats where
  a <> b =
    AccumStats {
      asCount = asCount a + asCount b,
      asSumArea = asSumArea a + asSumArea b,
      asSumCircumference = asSumCircumference a + asSumCircumference b,
      asMaxArea = max (asMaxArea a) (asMaxArea b),
      asMaxCircumference = max (asMaxCircumference a) (asMaxCircumference b) }

instance Monoid AccumStats where
  mempty = AccumStats 0 0 0 0 0

foldMapStatistics :: Drawing Object -> Statistics
foldMapStatistics dobject =  Statistics {
    avgArea = asSumArea accum / (fromIntegral $ asCount accum),
    avgCircumference = asSumCircumference accum / (fromIntegral $ asCount accum),
    maxArea = asMaxArea accum,
    maxCircumference = asMaxCircumference accum }
  where
    accum = foldMap accumStats' dobject

accumStats' :: Object -> AccumStats
accumStats' object =
  AccumStats {
    asCount = 1,
    asSumArea = area object,
    asSumCircumference = circum object,
    asMaxArea = area object,
    asMaxCircumference = circum object }

```

(b) Statistics using foldMap

Figure 5: TaskI.5 1 2

Figure 5 shows how the statistics functions were implemented using two different approaches. Figure 5 (a) illustrates how the statistics function was defined using recursion. The statistics function takes a Drawing Object as input and returns Statistics as a result by calling the accumStats function to do the computation. The accumStats was implemented using recursion by taking the result of last computation and the remaining Drawing Object as input and eventually return the final result. The area and circum functions were defined to aid the computation. They calculated the area and circumference of a single object repsectively.

Figure 5 (b) shows a different approach of defining the statistics function by making Drawing an instance of Foldable and making AccumStats an instance of Semigroup. By making Drawing an instance of Foldable, a function can be applied to every object in drawing and the results can be combined. By making AccumStats an instance of Semigroup, it provided a way of merging two AccumStats to form a new AccumStats.

As for the statistics using foldMap, the overall structure is similar to the recursive version, but the accumStats was defined differently. Instead processing all objects in Drawing, the accumStats now only need to convert one object to AccumStats and the results for all object in Drawing can be combined using foldMap.

```haskell
newtype Length = Length { getLength :: Double } deriving (Eq, Ord, Num, Show)
newtype Area   = Area   { getArea   :: Double } deriving (Eq, Ord, Num, Show)

instance Bounded Length where
  minBound = 0
  maxBound = Length $ read "Infinity"

instance Bounded Area where
  minBound = 0
  maxBound = Area $ read "Infinity"

```

Figure 6: newtype Bounded

Figure 6 shows the implementation details of Bounded class for Length and Area. They both bounded below by 0 and above by +Infinity.

```haskell
statistics :: Drawing Object -> Statistics
statistics dobject =
  Statistics {
    avgArea = Area ( (getArea $ getSum $ asSumArea accum)
                     / (fromIntegral $ getSum $ asCount accum) ),
    avgCircumference = Length ( (getLength $ getSum $ asSumCircumference accum )
                               / (fromIntegral $ getSum $ asCount accum) ),
    maxArea = getMax $ asMaxArea accum,
    maxCircumference = getMax $ asMaxCircumference accum }
  where
    accum = accumStats (AccumStats 0 0 0 0 0) dobject


accumStats :: AccumStats -> Drawing Object -> AccumStats
accumStats accum (Element object) =
  AccumStats {
    asCount = asCount accum + 1,
    asSumArea = Sum ( Area (
                       (getArea $ getSum $ asSumArea accum) +
                       ( getArea $ area object) ) ),
    asSumCircumference = Sum ( Length (
                               (getLength $ getSum $ asSumCircumference accum) +
                                (getLength $ circum object) ) ),
    asMaxArea = Max ( Area (
                       max (getArea $ getMax $ asMaxArea accum)
                           (getArea $ area object) )),
    asMaxCircumference = Max ( Length (
                                max (getLength $ getMax $ asMaxCircumference accum)
                                  (getLength $ circum object))) }
accumStats accum (Group [])     = accum
accumStats accum (Group (x:xs)) = accumStats (accumStats accum x) (Group xs)

area :: Object -> Area
area (Rectangle{ .. }) = Area (getLength width * getLength height)
area (Circle{ .. })    = Area (pi * getLength radius * getLength radius)

circum :: Object -> Length
circum (Rectangle { .. }) = Length ( (getLength width + getLength height) * 2 )
circum (Circle { .. })    = Length ( pi * getLength radius * 2 )
```

(a) Recursive Statistics for newtype

```haskell
instance Foldable Drawing where
  foldMap f (Element object) = f object
  foldMap f (Group [])       = mempty
  foldMap f (Group (x:xs))   = foldMap f x <> foldMap f (Group xs)

instance Semigroup AccumStats where
  a <> b =
    AccumStats {
      asCount = asCount a + asCount b,
      asSumArea = asSumArea a + asSumArea b,
      asSumCircumference = asSumCircumference a + asSumCircumference b,
      asMaxArea = max (asMaxArea a) (asMaxArea b),
      asMaxCircumference = max (asMaxCircumference a) (asMaxCircumference b) }

instance Monoid AccumStats where
  mempty = AccumStats 0 0 0 0 0

foldMapStatistics :: Drawing Object -> Statistics
foldMapStatistics dobject =  Statistics {
    avgArea = Area ( (getArea $ getSum $ asSumArea accum)
                     / (fromIntegral $ getSum $ asCount accum) ),
    avgCircumference = Length ( (getLength $ getSum $ asSumCircumference accum )
                               / (fromIntegral $ getSum $ asCount accum) ),
    maxArea = getMax $ asMaxArea accum,
    maxCircumference = getMax $ asMaxCircumference accum }
  where
    accum = foldMap accumStats' dobject

accumStats' :: Object -> AccumStats
accumStats' object =
  AccumStats {
    asCount = 1,
    asSumArea = Sum (area object),
    asSumCircumference = Sum (circum object),
    asMaxArea = Max (area object),
    asMaxCircumference = Max (circum object) }
```

(b) Statistics using foldMap for newtype

Figure 7: TaskI.5 3

Figure 7 demonstrates the two versions of the statistics function implemented according to Length and Area types. They work the same as the version shown in figure 5 but there are some workarounds according to the type definitions. For example, some values may be wrapped in Area then they need to be extracted and put back to Area after some computations. The same processes work for values in Length, Sum and Max.