

# Real-world Functional Programming

## Coursework Part I Report

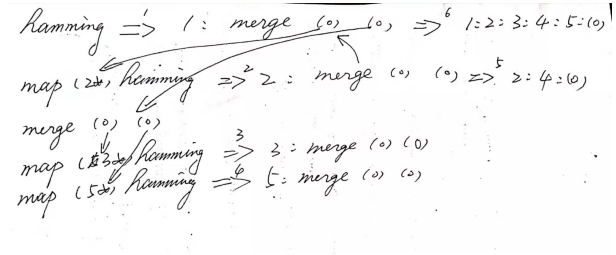
14274056 Junsong Yang (psyjy3)

November 5, 2019

### 1 Task I.1

```
1
2
3 hamming :: [Int]
4 hamming = 1 : merge (map (2*) hamming) (merge (map (3*) hamming) (map (5*) hamming))
5
6 merge :: [Int] -> [Int] -> [Int]
7 merge xs@(x:xs') yss@(y:ys)
8   | x == y = x : merge xs' ys
9   | x < y = x : merge xs yss
10  | x > y = y : merge xss ys
11
```

(a) Hamming Function Definition



(b) Cyclic Graph

Figure 1: TaskI.1

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

### 2 Task I.2

```
11 data Exp = Lit Double
12          | Ref CellRef
13          | Sum CellRef CellRef
14          | Avg CellRef CellRef
15          | App BinOp Exp Exp
16
17
18
19 evalCell :: Sheet Double -> Exp -> Double
20 evalCell _ (Lit v) = v
21 evalCell s (Ref r) = s ! r
22 evalCell s (Sum c1 c2) = foldr (+) 0 l
23   where
24     l = [ s ! (c, r) | c <- [(fst c1)..(fst c2)],
25                           r <- [(snd c1)..(snd c2)] ]
26 evalCell s (Avg c1 c2) = (foldr (+) 0 l) / (fromIntegral llen)
27   where
28     l = [ s ! (c, r) | c <- [(fst c1)..(fst c2)],
29                           r <- [(snd c1)..(snd c2)] ]
30     llen = length l
31 evalCell s (App op e1 e2) = (evalOp op) (evalCell s e1) (evalCell s e2)
32
```

Figure 2: Extended Evaluator

Figure 2 demonstrates how evalCell function was extended to support Sum and Avg expression. The key idea of this implementation is to given to 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 stuck in infinite loop when there are cyclic 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 cyclic 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 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 to evaluate that reference cell, if it returns, which means there is no cyclic reference. Hence those cell can be popped off the stack. If a cell that contains Ref Exp already exists in the stack that there must be a cyclic 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.

```

65
66 drop :: Int -> RList a -> RList a
67 drop _ [] = []
68 drop 0 t = t
69 drop i (t:ts)
70 | i == fst t = ts
71 | i > fst t = drop (i - fst t) ts
72 | otherwise = drop' i (fst t) (snd t) ++ ts
73
74 drop' :: Int -> Int -> Tree a -> RList a
75 drop' 0 _ (Leaf a) = [(1, (Leaf a))]
76 drop' _ _ (Leaf _) = []
77 drop' 0 w n@(Node _ _ _) = [(w, n)]
78 drop' i w (Node l _ r)
79 | i <= half = drop' (i - 1) half l ++ [(half, r)]
80 | otherwise = drop' (i - half - 1) half r
81 where
82   half = w `div` 2
83
84 testDrop :: Int -> Bool
85 testDrop i = l1 == l2
86 where
87   l1 = last $ take i $ iterate (cons 1) empty
88   l2 = drop i $ last $ take (i+2) $ iterate (cons 1) empty
89

```

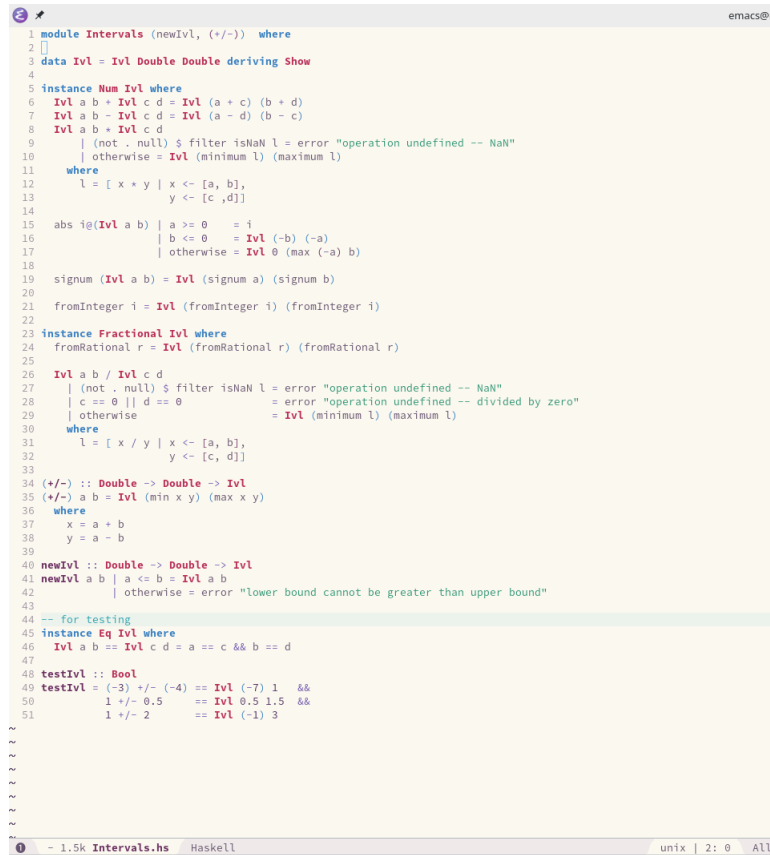
Figure 3: Drop Function

Figure 3 shows the implementation of drop function which drops the first n elements from the list. The drop function takes an integer and a RList as input and return a RList without the first n elements as output. If the tree is empty which means nothing can be dropped, hence 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 is equals to the number of elements need to be dropped, then the tail of the input list will be returned. If the number of elements need to be dropped is greater that the size of 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 need to be dropped is less than the size of 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 a 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 tree will be halved for every element dropped from the tree. Therefore, the overall time complexity is  $O(\log n)$ . If the number of elements need 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 nothing to drop and return the final result.

A test function called testDrop id 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



```

1 module Intervals (newIvl, (+/-)) where
2
3 data Ivl = Ivl Double Double deriving Show
4
5 instance Num Ivl where
6   Ivl a b + Ivl c d = Ivl (a + c) (b + d)
7   Ivl a b - Ivl c d = Ivl (a - c) (b - c)
8   Ivl a b * Ivl c d
9     | (not . null) $ filter isNaN l = error "operation undefined -- NaN"
10    | otherwise = Ivl (minimum l) (maximum l)
11   where
12     l = [ x * y | x <- [a, b],
13               y <- [c, d]]
14
15   abs (Ivl a b) | a >= 0 = Ivl a b
16                | b <= 0 = Ivl (-b) (-a)
17                | otherwise = Ivl 0 (max (-a) b)
18
19   signum (Ivl a b) = Ivl (signum a) (signum b)
20
21   fromInteger i = Ivl (fromInteger i) (fromInteger i)
22
23 instance Fractional Ivl where
24   fromRational r = Ivl (fromRational r) (fromRational r)
25
26   Ivl a b / Ivl c d
27     | (not . null) $ filter isNaN l = error "operation undefined -- NaN"
28     | c == 0 || d == 0 = error "operation undefined -- divided by zero"
29     | otherwise = Ivl (minimum l) (maximum l)
30   where
31     l = [ x / y | x <- [a, b],
32                   y <- [c, d]]
33
34   (+/-) :: Double -> Double -> Ivl
35   (+/-) a b = Ivl (min x y) (max x y)
36   where
37     x = a + b
38     y = a - b
39
40   newIvl :: Double -> Double -> Ivl
41   newIvl a b | a <= b = Ivl a b
42              | otherwise = error "lower bound cannot be greater than upper bound"
43
44 -- for testing
45 instance Eq Ivl where
46   Ivl a b == Ivl c d = a == c && b == d
47
48 testIvl :: Bool
49 testIvl = (-3) +/- (-4) == Ivl (-7) 1 &&
50           1 +/- 0.5 == Ivl 0.5 1.5 &&
51           1 +/- 2 == Ivl (-1) 3
52
53 ~
54 ~
55 ~
56 ~
57 ~
58 ~
59 ~
60 ~
61 ~
62 ~
63 ~
64 ~
65 ~
66 ~
67 ~
68 ~
69 ~
70 ~
71 ~
72 ~
73 ~
74 ~
75 ~
76 ~
77 ~
78 ~
79 ~
80 ~
81 ~
82 ~
83 ~
84 ~
85 ~
86 ~
87 ~
88 ~
89 ~
90 ~
91 ~
92 ~
93 ~
94 ~
95 ~
96 ~
97 ~
98 ~
99 ~
100 ~

```

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

return an Ivl with the signum value of lower bound and signum value of upper bound. the fromIntegral function takes an int as input and returns an Ivl with the same value converted from input.

the fromRational function in Fractional class works in a similar way 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 is provided in the testIvl function to test the result of the (+/-) function.

## 5 Task I.5

```

38 statistics :: Drawing Object -> Statistics
39 statistics dobject =
40   Statistics {
41     avgArea = asSumArea accum / (fromIntegral $ asCount accum),
42     avgCircumference = asSumCircumference accum / (fromIntegral $ asCount accum),
43     maxArea = asMaxArea accum,
44     maxCircumference = asMaxCircumference accum }
45   where
46     accum = accumStats (AccumStats 0 0 0 0 0) dobject
47
48
49 accumStats :: AccumStats -> Drawing Object -> AccumStats
50 accumStats accum (Element object) =
51   AccumStats {
52     asCount = asCount accum + 1,
53     asSumArea = asSumArea accum + area object,
54     asSumCircumference = asSumCircumference accum + circum object,
55     asMaxArea = max (asMaxArea accum) (area object),
56     asMaxCircumference = max (asMaxCircumference accum) (circum object) }
57 accumStats accum (Group []) = accum
58 accumStats accum (Group (x:xs)) = accumStats (accumStats accum x) (Group xs)
59
60
61 area :: Object -> Area
62 area (Rectangle{ .. }) = width * height
63 area (Circle{ .. }) = pi * radius * radius
64
65 circum :: Object -> Length
66 circum (Rectangle{ .. }) = (width + height) * 2
67 circum (Circle{ .. }) = pi * radius * 2
68

```

(a) Recursive Statistics

```

71 instance Foldable Drawing where
72   foldMap f (Element object) = f object
73   foldMap f (Group []) = mempty
74   foldMap f (Group (x:xs)) = foldMap f x <> foldMap f (Group xs)
75
76 instance Semigroup AccumStats where
77   a <<> b =
78     AccumStats {
79       asCount = asCount a + asCount b,
80       asSumArea = asSumArea a + asSumArea b,
81       asSumCircumference = asSumCircumference a + asSumCircumference b,
82       asMaxArea = max (asMaxArea a) (asMaxArea b),
83       asMaxCircumference = max (asMaxCircumference a) (asMaxCircumference b) }
84
85 instance Monoid AccumStats where
86   mempty = AccumStats 0 0 0 0 0
87
88 foldMapStatistics :: Drawing Object -> Statistics
89 foldMapStatistics dobject = Statistics {
90   avgArea = asSumArea accum / (fromIntegral $ asCount accum),
91   avgCircumference = asSumCircumference accum / (fromIntegral $ asCount accum),
92   maxArea = asMaxArea accum,
93   maxCircumference = asMaxCircumference accum }
94   where
95     accum = foldMap accumStats' dobject
96
97 accumStats' :: Object -> AccumStats
98 accumStats' object =
99   AccumStats {
100     asCount = 1,
101     asSumArea = area object,
102     asSumCircumference = circum object,
103     asMaxArea = area object,
104     asMaxCircumference = circum object }
105

```

(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 the last computation and the remaining Drawing Object as input and eventually returning the final result. The area and circum functions were defined to aid the computation. They calculated the area and circumference of a single object separately.

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.

```

9 newtype Length = Length { getLength :: Double } deriving (Eq, Ord, Num, Show)
10 newtype Area   = Area   { getArea   :: Double } deriving (Eq, Ord, Num, Show)
11
12 instance Bounded Length where
13   minBound = 0
14   maxBound = Length $ read "Infinity"
15
16 instance Bounded Area where
17   minBound = 0
18   maxBound = Area $ read "Infinity"
19

```

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.

```

89 statistics :: Drawing Object -> Statistics
90 statistics dobject =
91   Statistics {
92     avgArea = Area ( (getArea $ getSum $ asSumArea accum)
93                     / (fromIntegral $ getSum $ asCount accum) ),
94     avgCircumference = Length ( (getLength $ getSum $ asSumCircumference accum)
95                                / (fromIntegral $ getSum $ asCount accum) ),
96     maxArea = getMax $ asMaxArea accum,
97     maxCircumference = getMax $ asMaxCircumference accum }
98   where
99     accum = accumStats (AccumStats 0 0 0 0 0) dobject
100
101
102 accumStats :: AccumStats -> Drawing Object -> AccumStats
103 accumStats accum (Element object) =
104   AccumStats {
105     asCount = asCount accum + 1,
106     asSumArea = Sum ( Area (
107       (getArea $ getSum $ asSumArea accum) +
108       (getArea $ area object) ) ),
109     asSumCircumference = Sum ( Length (
110       (getLength $ getSum $ asSumCircumference accum) +
111       (getLength $ circum object) ) ),
112     asMaxArea = Max ( Area (
113       max (getArea $ getMax $ asMaxArea accum)
114       (getArea $ area object) ) ),
115     asMaxCircumference = Max ( Length (
116       max (getLength $ getMax $ asMaxCircumference accum)
117       (getLength $ circum object) ) ) }
118 accumStats accum (Group []) = accum
119 accumStats accum (Group (x:xs)) = accumStats (accumStats accum x) (Group xs)
120
121 area :: Object -> Area
122 area (Rectangle { .. }) = Area (getLength width * getLength height)
123 area (Circle { .. }) = Area (pi * getLength radius * getLength radius)
124
125 circum :: Object -> Length
126 circum (Rectangle { .. }) = Length ( (getLength width + getLength height) * 2 )
127 circum (Circle { .. }) = Length ( pi * getLength radius * 2 )

```

(a) Recursive Statistics for newtype

```

51 instance Foldable Drawing where
52   foldMap f (Element object) = f object
53   foldMap f (Group []) = mempty
54   foldMap f (Group (x:xs)) = foldMap f x <> foldMap f (Group xs)
55
56 instance Semigroup AccumStats where
57   a <> b =
58     AccumStats {
59       asCount = asCount a + asCount b,
60       asSumArea = asSumArea a + asSumArea b,
61       asSumCircumference = asSumCircumference a + asSumCircumference b,
62       asMaxArea = max (asMaxArea a) (asMaxArea b),
63       asMaxCircumference = max (asMaxCircumference a) (asMaxCircumference b) }
64
65 instance Monoid AccumStats where
66   mempty = AccumStats 0 0 0 0 0
67
68 foldMapStatistics :: Drawing Object -> Statistics
69 foldMapStatistics dobject = Statistics {
70   avgArea = Area ( (getArea $ getSum $ asSumArea accum)
71                   / (fromIntegral $ getSum $ asCount accum) ),
72   avgCircumference = Length ( (getLength $ getSum $ asSumCircumference accum)
73                               / (fromIntegral $ getSum $ asCount accum) ),
74   maxArea = getMax $ asMaxArea accum,
75   maxCircumference = getMax $ asMaxCircumference accum }
76   where
77     accum = foldMap accumStats' dobject
78
79 accumStats' :: Object -> AccumStats
80 accumStats' object =
81   AccumStats {
82     asCount = 1,
83     asSumArea = Sum (area object),
84     asSumCircumference = Sum (circum object),
85     asMaxArea = Max (area object),
86     asMaxCircumference = Max (circum object) }

```

(b) Statistics using foldMap for newtype

Figure 7: TaskI.5 3

Figure 7 demonstrates the two version of statistics function implemetated according newtype Length and Area. They work exactly the same as the version showed in figure 5 but there are some work arounds according to the type definitions. For example, some values may wrapped in Area then they need to be extracted out and do some computation and put back to Area. The same processes works for values in Length, Sum and Max.