G54RFP Coursework Part I

Jack Ellis
psyje5@nottingham.ac.uk
4262333

Contents

1 Task I.1

1.1 Part 1

```
hamming = let twos   = fmap (2*) hamming
              threes = fmap (3*) hamming
              fives  = fmap (5*) hamming
          in 1 : twos `merge` threes `merge` fives
```

Here we begin with a list containing only `1`, to which we cons the merged list containing 2 times itself, 3 times itself, and 5 times itself. These additional lists are formed of recursive calls to `hamming`.

1.2 Part 2

The following is the execution trace for the first 4 digits of hamming:

```
hamming = 1:map (*2) hamming `merge` map (*3) hamming `merge` (*5) hamming
        = 1:merge (map *2) hamming) (map (*3) hamming `merge` (*5) hamming)
        = 1:merge (map *2) hamming) (merge (map (*3) hamming)) (map (*5) hamming)
        = 1:merge (2:map (*2) hamming) (merge (3:map (*3) hamming) (5:map (*5) hamming))
        = 1:merge (2:map (*2) hamming) (3:merge (map (*3) hamming) (5:map (*5) hamming))
        = 1:2:merge (map (*2) hamming) (3:merge (map (*3) hamming) (5:map (*5) hamming))
        = 1:2:merge (4:map (*2) hamming) (3:merge (map (*3) hamming) (5:map (*5) hamming))
        = 1:2:3:merge (4:map (*2) hamming) (merge (6:map (*3) hamming) (5:map (*5) hamming))
        = 1:2:3:merge (4:map (*2) hamming) (3:merge (map (*3) hamming) (5:map (*5) hamming))
        = 1:2:3:merge (4:map (*2) hamming) (5:merge (map (*3) hamming) (5:map (*5) hamming))
        = 1:2:3:4:merge (map (*2) hamming) (6:merge (map (*3) hamming) (5:map (*5) hamming))
```

2 Task I.2

2.1 Part 1

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

Here I extended the `Exp` type to include what is shown in the coursework issue sheet.

```
mean    :: [Double] -> Double
mean xs = (sum xs)/((fromIntegral . length) xs)
```

A function to calculate the `mean` of a list was required and not present, so here I defined one.

```
evalCell s (Sum (r1,c1) (r2,c2))
  = sum [evalCell s (Ref (r,c))
         | r <- range (r1, r2), c <- range (c1,c2)]
evalCell s (Avg (r1,c1) (r2,c2))
  = mean [evalCell s (Ref (r,c))
          | r <- range (r1, r2), c <- range (c1,c2)]
```

Here I used list comprehensions to obtain the full 2D range of cells described by the function call, map to them their value via an evalCell call, and sum or mean that list of values based upon what was called originally.

2.2 Part 2

The problem with the evaluator is that if any cells are empty (i.e. there is no item in the list with their reference), the evaluator fails completely. This could be fixed by padding the list with empty (i.e. a payload of `Lit 0`) cells where there is no cell matching a required reference. Additionally to this, if the `array` value declared at the start of a `Sheet` declaration does not match the number of cells in the `Sheet`, it again fails. This could be solved by removing the bounds call and determining the bounds of a `Sheet` as a first step in evaluating it.

3 Task I.3

```
drop :: Int -> RList a -> RList a
drop _ [] = []
drop 0 ts = ts
drop n ts = drop (n-1) (tail ts)
```

My implementation of the `drop n` function works in much the same way as the `Prelude` version of drop works on lists; recursive application of `tail` until the counter (specified by `n` and decrementing on each recursive iteration) reaches 0. It has the desired time complexity (*O(log n)*), because `tail` runs in constant time, and the function will run a number of times based on the size of the SBRAL, whose size grows in *log n* time.

For testing purposes I extended the derivations of the `Tree` type to include `Eq`, and created a `treeGen` function via iterated applications of `cons 1` to an empty list. Hence if `treeGen 255` is called, an Rlist of size 255 is generated.

To test this, the function `treeTest :: Int -> Bool`

4 Task I.4

```
data Ivl = Ivl Double Double deriving Show
```

Here the data type `Ivl` is defined per the coursework issue sheet.

```
isWellDef :: Ivl -> Bool
isWellDef (Ivl l u) = l <= u

wellDef1 :: Ivl -> Ivl -> Ivl
wellDef1 i i' = if isWellDef i
                then i'
                else error "Poorly defined argument"

wellDef2 :: Ivl -> Ivl -> Ivl -> Ivl
wellDef2 i1 i2 i' | isWellDef i1 && isWellDef i2 = i'
```

```
                  | not (isWellDef i1) && isWellDef i2
                    = error "First argument is not well-defined."
                  | isWellDef i1 && not (isWellDef i2)
                    = error "Second argument is not well-defined."
                  | otherwise
                    = error "Neither argument is well-defined."

makeWellDef :: Ivl -> Ivl
makeWellDef i@(Ivl l u) | isWellDef i = i
                        | otherwise   = Ivl u l
```

The first step here is defining a series of functions to detect and enforce well-definedness. `isWellDef` takes an `Ivl` and returns true if its lower bound is less than or equal to its upper bound, else it returns false. With this function, we can create two more: `wellDef1` and `wellDef2`. They perform in exactly the same way bar the fact that they deal with functions containing 1 and 2 `Ivl` values respectively. If all of the `Ivl`s are well-defined, the function passed into them may be computed and its result returned. If any of the `Ivl`s passed as arguments are poorly formed it will return an `error` detailing which `Ivl` it took was poorly formed and halting computation. Finally, the function `makeWellDef` takes an `Ivl` and makes it well-defined. This is useful for the later functions whose results may be poorly formed by default, in many cases however the result is mathematically guaranteed to be well-formed.

```
instance Num Ivl where
  (+) i1@(Ivl l1 u1) i2@(Ivl l2 u2)
    = wellDef2 i1 i2 $ makeWellDef (Ivl (l1+l2) (u1+u2))
  (-) i1@(Ivl l1 u1) i2@(Ivl l2 u2)
    = wellDef2 i1 i2 $ makeWellDef (Ivl (l1-u2) (u1-l2))
  (*) i1@(Ivl l1 u1) i2@(Ivl l2 u2)
    = wellDef2 i1 i2 $ makeWellDef (Ivl (l1*l2) (u1*u2))
  abs i@(Ivl l u)
    = let abl = abs l
          abu = abs u
      in wellDef1 i (if l < 0 && 0 < u
                     then if abl > abu
                          then Ivl 0.0 abl
                          else Ivl 0.0 abu
                     else makeWellDef (Ivl abl abu))
  signum i@(Ivl l u)
```

```
    = wellDef1 i (Ivl (signum l) (signum u))
  fromInteger n
    = let fin = fromInteger n
      in Ivl fin fin
```

Here the Num instance of Ivl is defined. Addition, subtraction, and multiplication are all defined as per the issue sheet, with checks on well-definedness before the values are computed. The results will be well-defined Ivls mathematically; the sum of two lower bounds will be lower than that of two upper bounds, subtracting a large number from a small one will be smaller than subtracting a small one from a large one, and multiplying will follow the same rules of addition. abs is defined such that if the Ivl straddles 0 it will return 0 as the lower bound, and the larger absolute value of the two as the upper bound. Otherwise it will return a well-defined Ivl with the bounds being the absolute values of the bounds passed to it. signum returns a well-defined Ivl with the signum values of both bounds as its bounds. The rationale here is that we want to see the signum values of the highest and lowest possible values of the Ivl. Finally, fromInteger converts an Int value into an Ivl with a range of 0, given that the Int has no uncertainty or bounds.

```
instance Fractional Ivl where
  (/) i1@(Ivl l1 u1) i2@(Ivl l2 u2)
    = if u2 /= 0 && l2 /= 0
      then wellDef2 i1 i2 (Ivl (l1/u2) (u1/l2))
      else error "Arguments result in attempt to divide by 0"
  recip i@(Ivl l u)
    = wellDef1 i $ makeWellDef (Ivl (recip l) (recip u))
  fromRational n
    = let frn = fromRational n
      in makeWellDef (Ivl frn frn)
```

Here the Fractional instance of Ivl is defined. Division is less trivial to define than the other arithmetic symbols; we must ensure that at no point does division by 0 occur, and ensure well-definedness of the result. Hence in my implementation the new lower bound is defined by l1/u2 to give the smallest possible value, and the new upper is u1/l2 to give the largest possible value. recip is the reciprocal function, and will divide 1 by any value given to it. Here the recip instance for Ivl will return a new Ivl with the reciprocals of the bounds. If (and this is the case for much of the mathematical functions) the input Ivl is well-defined, the switching of the reciprocals will mathematically ensure that the new Ivl is well-defined, however for the sake of ensuring correctness I have left the `makeWellDef` function in place. Finally, `fromRational` works much the same as `fromInteger`, with the same justification.

```
(+/-) :: Double -> Double -> Ivl
(+/-) n i = makeWellDef (Ivl (n-i) (n+i))
```

The `(+/-)` operator will construct a well-defined `Ivl` with a symmetric interval around the number given as its first argument of a size given by its second argument. `makeWellDef` is used here because it is possible to pass a negative number as the second argument and thus flip the lower and upper bounds of the resulting `Ivl`, leading to a poorly-defined `Ivl`.

For every function defined, the function `makeWellDef` ensures well-definedness for the results, a feature I tested using the Haskell library QuickCheck. Here I defined an `arbitrary` instance of `Ivl`.

```
instance Arbitrary Ivl where
  arbitrary = do i1 <- arbitrary
                 i2 <- arbitrary
                 return $ if i1 <= i2 then Ivl i1 i2
                                      else Ivl i2 i1
```

Using this I defined two functions: both check the well-definedness of a function's results, with one instance for those functions with a single argument, and one for those with multiple.

```
prop_wellFormedResult1 ::(Ivl -> Ivl) -> Ivl -> Bool
prop_wellFormedResult1 f i = let Ivl l u = f i
                             in l <= u

prop_wellFormedResult2 ::(Ivl -> Ivl -> Ivl) -> Ivl -> Ivl -> Bool
prop_wellFormedResult2 f i1 i2 = let Ivl l u = f i1 i2
                                 in l <= u
```

Running every function through these (in the form `quickCheck $ prop_wellFormed<N> <f>`, where N is the number of arguments of the function being checked, and f is the function) yielded no failures.