Homework 2: CS 558

Taylor Berger

October 1, 2015

1

1.1 Binary Trees

This is a simple recursive descent on the structure of the tree taking the previous paths returned and mapping on additional path information on how to get to the subtrees of the current node.

```
data T = Leaf | Node T T
data P = GoLeft P | GoRight P | This

allPaths :: T → [P]
allPaths (Node t1 t2) = This : map GoLeft (allPaths t1)
  ⧺ map GoRight (allPaths t2)
allPaths Leaf = [This]
```

1.2 General Trees

Foldtree takes a function of two arguments, the first being the value of the node and the second being the list of values returned by mapping foldTree onto the children of the node and returning a single value.

```
data Tree a = Node a [Tree a]
foldTree :: (a → [b] → b) → Tree a → b
foldTree f = (Node a subtrees) = f a $ map (foldtree f) subtrees
```

MapTree is written by composing the function you want to map across the nodes with the Node constructor itself to maintain the structure of the tree.

```
mapTree :: (a → b) → Tree a → Tree b
mapTree f = foldTree (Node ∘ f)
```

1.3 File Handling

Since the structure of these programs are essentially the same with the only difference being the function applied to the input, we can first define a skeleton function that handles the majority of the work itself.

```
import System.Environment (getArgs)
import System.IO (hGetContents, openFile, IOMode (ReadMode))
```

```
skeleton :: Show b ⇒ (a → b) → IO ()
skeleton fun = do
  [f, _] ← getArgs
  openFile f ReadMode ≫= hGetContents ≫= mapM_ print ∘ fun
```
This lazily reads the file into memory, applies the function to the input and prints the output line by line.
```
cat :: IO ()
cat = skeleton lines
```
Since cat just prints out the file, we don't change the file's contents, so we just split the output into lines
```
tac :: IO ()
tac = skeleton (reverse ∘ lines)
```
Tac splits the input into lines then reverses the list of lines thereby printing out the content in the file, reversed.
```
rev :: IO ()
rev = skeleton (map reverse ∘ lines)
```
rev takes each line and reverses the line itself by mapping reverse across each line, maintaining the structure of the file.
```
sort :: IO ()
sort = do
  files ← getArgs
  content ← liftM concat $ mapM (λf → hGetContents =≪ openFile f ReadMode) files
  mapM_ print (L.sort $ lines content)
```
Sort is slightly different, which reads in all files passed in as parameters and concats them together. Then sorts the lines based on ascii character value and prints them out in ascending order.

1.4 Drawing & Postscript

Some type information to make things easier to think about. The rest of the code is pretty straightforeward. We identify the bounding box by the appropriate min/max of all (x,y) coordinate pairs, then print the header and map a function which converts a list of points to a string command that prints the shape in PostScript format.
```
type Point = (Float, Float)
type Path = [Point]

makeCommand :: [Path] → String
makeCommand paths = header paths ⧺ "\n" ⧺ (concatMap toPost paths) ⧺ "showpage\n%%EOF"
  where header :: [Path] → String
    header paths = "%! PS-Adobe-3.0 EPSF-3.0\n%%BoundingBox: "
      ⧺ show left ⧺ " "
      ⧺ show bottom ⧺ " "
      ⧺ show right ⧺ " "
      ⧺ show top ⧺ "\n"

    bottom, top, left, right :: Float
```

```
bottom = foldl1 min $ map (foldl1 min) $ map (map snd) paths
top = foldl1 max $ map (foldl1 max) $ map (map snd) paths
left = fst $ foldl1 min $ map (foldl1 min) paths
right = fst $ foldl1 max $ map (foldl1 max) paths

toPost :: Path → String
toPost ((x, y) : ps) =
  (show x ++ " " ++ show y ++ " moveto\n") ++
  concatMap (λ(x, y) → show x ++ " " ++ show y ++ " lineto\n") ps
  ++ "closepath\nstroke\n\n"
```

1.5 Proofs

We can show this is true by inductively by inducting on the structure of the list itself. First, keep the type signatures in mind for both `foldr` and `foldl`:

$\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b$
$\mathit{foldl} :: (b \to a \to b) \to b \to [a] \to b$

We need to prove that for any list, `ls`:

$\mathit{foldl}\ f\ x\ \mathit{ls} = \mathit{foldr}\ (\mathit{flip}\ f)\ x\ (\mathit{reverse}\ \mathit{ls})$

is true

For the base case, we consider the base structure of the list – the empty list. By definition, we know:

$\mathit{foldr}\ f\ x\ [\,] = x$
$\mathit{foldl}\ (\mathit{flip}\ f)\ x\ (\mathit{reverse}\ [\,]) = x$
$\mathit{foldl}\ f\ x\ [\,] = \mathit{foldr}\ (\mathit{flip}\ f)\ x\ (\mathit{reverse}\ [\,])$
$x = x$

So, the base case holds. For the inductive hypothesis, we assume this is true for any structure of a list:

$\mathit{foldl}\ f\ x\ \mathit{ls} = \mathit{foldr}\ (\mathit{flip}\ f)\ x\ \mathit{ls}$

Now, for any list of the form:

$(l : \mathit{ls})$

We want to show that:

$\mathit{foldl}\ f\ x\ (l : \mathit{ls}) = \mathit{foldr}\ (\mathit{flip}\ f)\ x\ (\mathit{reverse}\ (l : \mathit{ls}))$

By the definition of `foldl` we can rewrite the LHS as:

$\mathit{foldl}\ f\ (f\ x\ l)\ \mathit{ls} = \mathit{foldr}\ (\mathit{flip}\ f)\ x\ (\mathit{reverse}\ (l : \mathit{ls}))$

Applying `reverse` to `(l:ls)`:

$\mathit{foldl}\ f\ (f\ x\ l)\ \mathit{ls} = \mathit{foldr}\ (\mathit{flip}\ f)\ x\ ((\mathit{reverse}\ \mathit{ls}) +\!\!+ [l])$

By the definition of foldr, we know that

$\mathit{foldr}\ f\ s\ (x : \mathit{xs}) = f\ x\ (\mathit{foldr}\ f\ s\ \mathit{xs})$

And, by repeatedly applying the definition we obtain:

$\mathit{foldr}\ f\ s\ [x\_1, x\_2, ...x\_n] = f\ x\_1\ (f\ x\_2\ (f\ x\_3\ (...(f\ x\_n\ s))))$

We can see that on the inside, the function is applied to the seed value, s and the last element of the list, x_n. Therefore, we can rewrite original line as:

$\mathit{foldl}\ f\ (f\ x\ l)\ \mathit{ls} = \mathit{foldr}\ (\mathit{flip}\ f)\ ((\mathit{flip}\ f)\ l\ x)\ (\mathit{reverse}\ \mathit{ls})$
$\mathit{foldl}\ f\ (f\ x\ l)\ \mathit{ls} = \mathit{foldr}\ (\mathit{flip}\ f)\ (f\ x\ l)\ (\mathit{reverse}\ \mathit{ls})$

And using the inductive hypothesis, we know that the above reduction is true, the application of `f` to `x` and `l` will always result in the same value, and therefore the entire statement is true.