Assignment 4

Haskell

Due Date: Monday, March 13th 2017, 11:59pm

Directions

Answers to English questions should be in your own words; don't just quote from articles or books. We will take some points off for: code with the wrong type or wrong name, duplicate code, code with extra unnecessary cases, or code that is excessively hard to follow. You should always assume that the inputs given to each function will be well-typed, thus your code should not have extra cases for inputs that are not of the proper type. (Assume that any human inputs are error checked before they reach your code. Avoid duplicating code by using helping functions, library functions (when not prohibited in the problems), or by using syntactic sugars and local definitions (using **let** and **where**). It is a good idea to check your code for these problems before submitting. Be sure to test your programs on Eustis to ensure they work on a Unix system.

This is a group assignment. Make a group of 3-4 in WebCourses and use that to turn in the assignment.

Deliverables

For this assignment, turn in the following files:

- Matrix.hs
- MatrixAdd.hs
- InfSet.hs
- ConcatMap.hs
- FoldWindowLayout.hs
- TotalHeight.hs
- SplitScreen.hs

For coding problems, put your solution in the appropriate Haskell source file (named with ".hs" or ".lhs" suffix).

Problem 1 (30 pts)

In this problem you will write operations for an abstract data type Matrix, by writing definitions for the module Matrix that completes the module definition below.

```
module Matrix (Matrix, fillWith, fromRule, numRows, numColumns,
               at, mtranspose, mmap) where

-- newtype is like "data", but has some efficiency advantages
newtype Matrix a = Mat ((Int,Int), (Int,Int) -> a)

fillWith :: (Int,Int) -> a -> (Matrix a)
fromRule :: (Int,Int) -> ((Int,Int) -> a) -> (Matrix a)
numRows :: (Matrix a) -> Int
numColumns :: (Matrix a) -> Int
at :: (Matrix a) -> (Int, Int) -> a
mtranspose :: (Matrix a) -> (Matrix a)
mmap :: (a -> b) -> (Matrix a) -> (Matrix b)
```

That is, you are to complete the module by writing a function definition for each function declared in the module. To explain these, note that an $m \times n$ matrix is one with m rows and n columns. Element indexes range from 1 to the number of rows or columns (unlike the convention in C or Haskell). With that convention you are to implement the following functions:

- `fillWith` takes a pair (m, n) and an element e and produces an $m \times n$ matrix whose elements are e.
- `fromRule` takes a pair (m, n) and a function g (the rule), and produces an $m \times n$ matrix whose $(i, j)^{th}$ element is $g(i, j)$.
- `numRows` takes an $m \times n$ matrix and returns the number of rows in the matrix, m.
- `numColumns` takes an $m \times n$ matrix and returns the number of columns in the matrix, n.
- at takes an $m \times n$ matrix and a pair of Ints, (i, j), and returns the $(i, j)^{th}$ element of the matrix, provided that $1 \leq i \leq m$ and $1 \leq j \leq n$. If either index is outside of those ranges, an error occurs (at runtime).
- `mtranspose` takes an $m \times n$ matrix and returns an $n \times m$ matrix where the $(i, j)^{th}$ element of the result is the $(j, i)^{th}$ element of the augment matrix.
- `mmap` takes an $m \times n$ matrix and a function f and returns an $m \times n$ matrix whose $(i, j)^{th}$ element is the result of applying f to the $(i, j)^{th}$ element of the argument matrix.

There are test cases contained in `MatrixTests.hs`.

To aid in testing, we have also provided code to make `Matrix` an instance of the Haskell type classes Show and Eq. These instances are found in the file `MatrixInstances.hs`.

To make the tests work, you have to put your code in a module named `Matrix`. As specified on the first page of this homework, turn in both your code file and the output of your testing.

Problem 2 (20 pts)

Complete the module definition below, by defining the functions `sameShape`, `pointwiseApply`, `add`, and `sub`.

```
module Matrix (Matrix, fillWith, fromRule, numRows, numColumns,
               at, mtranspose, mmap) where

-- newtype is like "data", but has some efficiency advantages
newtype Matrix a = Mat ((Int,Int), (Int,Int) -> a)

fillWith :: (Int,Int) -> a -> (Matrix a)
fromRule :: (Int,Int) -> ((Int,Int) -> a) -> (Matrix a)
numRows :: (Matrix a) -> Int
numColumns :: (Matrix a) -> Int
at :: (Matrix a) -> (Int, Int) -> a
mtranspose :: (Matrix a) -> (Matrix a)
mmap :: (a -> b) -> (Matrix a) -> (Matrix b)
```

The predicate `sameShape` takes arguments of type `Matrix a` and returns **`True`** when they have the same dimensions.

The function pointwiseApply takes a curried function op of two argument and two matrices, m1, and m2, which have the same shape and whose elements are the same type as the argument types of op, and returns a matrix of the same shape as m1 and m2, in which the $(i,j)^{th}$ element is `(m1 `at` (i,j)) `op` (m2 `at` (i,j))`. If the two matrices do not have the same shape, then an error should be raised (using Haskell's `error` function).

The `add` and `sub` operations are the usual pointwise addition and subtraction of matrices.

You must define `add` and `sub` by using `pointwiseApply`.

There are test cases contained in `MatrixAddTests.hs`.

Problem 3 (30 pts)

A set can be described by a "characteristic function" (whose range is **`Bool`**) that determines if an element occurs in the set. For examples, the function ϕ such that $\phi("coke") = \phi("pepsi") = \text{True}$ for all other arguments x, $\phi(x) = \text{False}$ is the characteristic function for a set containing the strings "coke" and "pepsi", but nothing else. Allowing the user to construct a set from a characteristic function gives one the power to construct sets that may "contain" an infinite number of elements (such as the set of all prime numbers).

In a module named InfSet, you will declare a polymorphic type constructor Set, which can be declared something like as follows:

```
type Set a = ...
-- or perhaps something like –
data Set a = ...
```

Hint: think about using a function type as part of your representation of sets.

Then fill in the operations of the module InfSet, which are described informally as follows.

1. The function

   ```
   fromFunc :: (a -> Bool) -> (Set a)
   ```

   takes a characteristic function, f and returns a set such that each value x (of type a) is in the set just when $f\ x$ is `True`.

2. The function

   ```
   unionSet :: Set a -> Set a -> Set a
   ```

   takes two sets, with characteristic functions f and g, and returns a set such that each value x (of type a) is in the set just when either (fx) or (gx) is true.

3. The function

   ```
   intersectSet :: Set a -> Set a -> Set a
   ```

   takes two sets, with characteristic functions f and g, and returns a set such that each value x (of type a) is in the set just when both (fx) and (gx) are true.

4. The function

   ```
   inSet :: a -> Set -> Bool
   ```

   tells whether the first argument is a member of the second argument.

5. The function

   ```
   complementSet :: Set a -> Set a
   ```

   which returns a set that contains everything (of the appropriate type) not in the original set.

Tests are given in `InfSetTests.hs`.

Note (hint, hint) that the following equations must hold, for all f, g, and x of appropriate types.

```
inSet x (unionSet (fromFunc f) (fromFunc g)) == (f x) || (g x)
inSet x (intersectSet (fromFunc f) (fromFunc g)) == (f x) && (g x)
inSet x (fromFunc f) == f x
inSet x (complementSet (fromFunc f)) == not (f x)
```

Problem 4 (10 pts)

Using Haskell's built-in foldr function, write the polymorphic function

```
concatMap :: (a -> [b]) -> [a] -> [b]
```

This function can be considered to be an abstraction of problems like deleteAll. An application such as `(concatMap f ls)` applies `f` to each element of `ls`, and concatenates the results of those applications together (preserving the order). Note that application of `f` to an element of types a returns a list (of type `[b]`), and so the overall process collects the elements of these lists together into a large list of type `[b]`.

Your solution must have the following form:

```
module ConcatMap where
import Prelude hiding (concatMap)
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f ls = foldr ...
```

where the "…" is where you will put the arguments to `foldr` in your solution.

Note: your code in … must not call `concatMap` (let `foldr` do the recursion).

There are test cases contained in `ConcatMapTests.hs`.

Problem 5 (30 pts)

In this problem you will write a function

```
foldWindowLayout :: ((String, Int, Int) -> r) -> ([r] -> r) -> ([r] -> r)
                 -> WindowLayout -> r
```

that abstract from all the WindowLayout examples we have seen (such as those earlier in this homework and on the course examples page). For each type r, the function foldWindowLayout takes 3 functions: wf, hf, and vf, which correspond to the three variants in the grammar for WindowLayout. In more detail:

- `wf`, operates on a tuple of the information from a `Window` variant and returns a value of type `r`,
- `hf`, takes a list of the results of mapping `(foldWindowLayout wf hf vf)` over the list in a `Horizontal` variant, and
- `vf`, takes a list of the results of mapping `(foldWindowLayout wf hf vf)` over the list in a `Vertical` variant.

There are test cases contained in `FoldWindowLayoutTests.hs`.

Problem 6 (30 pts)

Use your `foldWindowLayout` function to implement

(a) (15 pts) `totalHeight` from Problem 1 in Assignment 3
(b) (15 pts) `splitScreen` from Problem 2 in Assignment 3

Write these solutions into modules `TotalHeight` and `SplitScreen`, respectively, that both import the module FoldWindowLayout from the previous problem, and then run the tests appropriate to each of the window layout problems. Hand in your code for these new implementations of `totalHeight` and `splitScreen`.