Introduction

The major theme of this assignment is to gain experience with monadic computations and lazy evaluation in Haskell.

Read and follow the course Haskell Style Guide (https://www.cs.rit.edu/~mtf/teaching/20191/psfp/style.html). Especially, look for opportunities to eliminate unnecessary complexity (use anonymous functions, use partial applications, use sectioning, use functionals). Also, look for opportunities to use standard library functions.
Array State Monad

Download Array.hs, which defines a type class for “arrays”, and ArrayStateMonad.hs, which defines a monad which threads a single array of elements through a computation.

The ArrayState arr e a type is defined as follows:

newtype ArrayState arr e a = ArrayState (arr e -> (a, arr e))

The arr is the “type” of the array (e.g., SBBTree for a size-balanced-binary-tree-based array), the e is the type of the elements of the array, and the a is the type of the computation result.

Add all expressions for this section to the file named ArrayStateMonad.hs.

1. (15pts) Define functions
   • fmap :: (a -> b) -> ArrayState arr e a -> ArrayState arr e b
   • (<*>) :: ArrayState arr e (a -> b) -> ArrayState arr e a -> ArrayState arr e b
   • pure :: a -> ArrayState arr e a
   • return :: a -> ArrayState arr e a
   • (>>=) :: ArrayState arr e a -> (a -> ArrayState arr e b) -> ArrayState arr e b
   
   that make ArrayState arr e an instance of Functor, Applicative, and Monad. In particular, return x should be the ArrayState action that, when run, leaves the array unchanged and yields x as the computation result and act1 >>= mkAct2 should be the ArrayState action that, when run, executes the action act1 with the current array arr0 to obtain a result res1 and new array arr1, evaluates mkAct2 res1 to obtain an action act2, and executes the action act2 with the array arr1.

2. (9pts) Define functions
   • sizeAS :: Array arr => ArrayState arr e Integer
   • idxAS :: Array arr => Integer -> ArrayState arr e (Maybe e)
   • updAS :: Array arr => Integer -> e -> ArrayState arr e ()
   
   that applies the functions of the Array type class to the array threaded through the computation. In particular, sizeAS should be the ArrayState action that, when run, leaves the array unchanged and yields the size of the array as the computation result; idxAS i should be the ArrayState action that, when run, leaves the array unchanged and yields either Just the element at index i (0-based indexing) of the array, if the index i is in bounds for the array, or Nothing, if the index i is out of bounds for the array, as the computation result; and, updAS i y should be the ArrayState action that, when run, either updates the element at index i (0-based indexing) of the array to y, if the index i is in bounds for the array, or leaves the array unchanged, if the index i is out of bounds for the array, and yields () as the computation result.

3. (3pts) Define a function
   • runArrayState :: Array arr => ArrayState arr e a -> arr e -> (a, arr e)
   
   such that runArrayState act arr executes the ArrayState action act with the initial array arr to obtain a result res and final array arr' and returns the pair (res, arr').

Discussion Note that a client of the ArrayStateMonad module may only construct an ArrayState action using return, (>>=), sizeAS, idxAS, and updAS. Within an ArrayState action, it is not possible to yield the array as the computation result, much as within an IO action, it is not possible to yield the RealWorld as the computation result. Although not exemplified by this implementation, with appropriate compiler support, it would be possible to implement ArrayStateMonad using a “true” mutable, O(1)-index and O(1)-update array threaded through the computation.
Bubble Sort

Download SizeBalancedBinaryTreeArray.hs, which defines an instance of the Array type class that represents arrays as size-balanced binary trees, InsertionSort.hs, which implements an in-place insertion sort as an ArrayState action, and BubbleSort.hs, which implements an in-place bubble sort as an ArrayState action.

Add all expressions for this section to the file named BubbleSort.hs.

1. (10pts) Define an ArrayState action

   • bsortAS :: (Ord e, Array arr) => ArrayState arr e ()

that, when run, sorts the array using an in-place bubble sort. Assuming an implementation of ArrayStateMonad using a “true” mutable, O(1)-index and O(1)-update array threaded through the computation, bsortAS should use O(1) space and O(n^2) time. (Recall that bubble sort should use only O(n) time if the input is already sorted.)

Hints:

• idxAS returns Nothing when the index is out of bounds.

• See InsertionSort.hs for an example of an in-place insertion sort implemented as an ArrayState action.
Infinite Streams

Download `Stream.hs`, which defines a type of infinite streams, and `StreamTests.hs`, which provides a (large) test suite.

The `Stream a` type is defined as follows:

```haskell
infixr 5 :$

data Stream a = a :$ (Stream a)
deriving (Eq)
```

Note that `Stream` is defined with an infix, symbolic constructor. Moreover, note that streams are like lists but with only a “cons” constructor; there is no “nil” constructor to signal the end of a stream. Thus, all streams are must be infinite (whereas lists may be either finite or infinite).

For convenience, we define a `Stream` instance of `Show` that displays the first 15 elements of the stream.

We also define a number of basic functions for streams similar to functions for lists. Note that `sHead` and `sTail` are not partial functions on streams. Also note that `sTake` returns a list, while `sDrop` returns a stream. `

`sUnfoldr` is a higher-order stream builder, which takes a function (from a “seed” to the next stream element and the next seed) and an initial seed and returns a stream, while `sRepeat` is a simple stream builder that repeats an element. `sIterate`, `sMapMaybe`, `sMap`, `sFilter`, `sZipWith`, `sZip`, `sUnzip`, `sIntersperse`, and `sInterleave` adapt the corresponding functions for lists to streams. `sAppend` appends a (finite) list and a stream and a stream and `sLConcat` concatenates a stream of (finite) lists. Finally, we make `Stream` an instance of `Functor` and make `Stream` of monoidal elements an instance of `Monoid`.

1. (5pts) Define an infinite stream

   - `facts :: Stream Integer`

   such that `facts` is the stream of factorial numbers.

   Examples:

   ```haskell
   facts
   ~> [1,1,2,6,24,120,5040,40320,362880,3628800,39916800,479001600,
      6227020800,87178291200,1307674368000,20922789888000,...]
   ```

   Note:

   - Follow the model of the provided infinite stream `fibs :: Stream Integer`; submissions of the form `sMap fact nats` will receive no credit.

2. (5pts) Define the stream

   - `nnss :: Stream Integer`

   such that `nnss` is the stream with zero 0s, one 1, two 2s, three 3s, ..., `n` ns, ....

   Examples:

   ```haskell
   nnss
   ~> [1,2,2,3,3,3,4,4,4,4,5,5,5,5,5,6,6,6,6,6,...]
   nnss !! 99 ~> 14
   nnss !! 999 ~> 45
   nnss !! 9999 ~> 141
   nnss !! 99999 ~> 447
   ```
3. (5pts) Define the function

- \(\text{sCrossWith} :: (a \to b \to c) \to \text{Stream } a \rightarrow \text{Stream } b \rightarrow \text{Stream } (\text{Stream } c)\)

such that \(\text{sCrossWith } f \, xs \, ys\) returns the stream of streams obtained by applying \(f\) to each pair of elements from \(xs\) and \(ys\). In particular, the \(j^{th}\) element of the \(i^{th}\) stream of the output is obtained by applying \(f\) to the \(i^{th}\) element of \(xs\) and the \(j^{th}\) element of \(ys\).

Examples:

- \(\text{sCrossWith } (,) \, \text{nats nats} \rightarrow [((0,0),(0,1),(0,2),(0,3),(0,4),...),((1,0),(1,1),(1,2),(1,3),(1,4),...),((2,0),(2,1),(2,2),(2,3),(2,4),...),((3,0),(3,1),(3,2),(3,3),(3,4),...),((4,0),(4,1),(4,2),(4,3),(4,4),...),...]\)

- \(\text{sCrossWith } (,) \, \text{nats nats} \, !! 20 \, !! 30 \sim (20,30)\)

- \(\text{sCrossWith } (*) \, \text{fibs facts} \rightarrow [0,1,2,6,24,...],[1,1,2,6,24,...],[2,2,4,12,48,...],[3,3,6,18,72,...],[4,4,8,24,96,...],...]\)

- \(\text{sCrossWith } (*) \, \text{fibs facts} \, !! 20 \, !! 30 \sim 1794435596629472511674626867200000000\)

There are a number of ways of “reducing” a stream of streams of elements (such as that returned by \(\text{sCrossWith}\)) to a stream of elements. The following problems consider some of the more interesting ones.

4. (5pts) Define the function

- \(\text{sDiag} :: \text{Stream } (\text{Stream } a) \rightarrow \text{Stream } a\)

such that \(\text{sDiag } xss\) returns the stream obtained by taking the main (down-right) diagonal elements of \(xss\).

\[
\begin{array}{ccccccc}
(0,0) & (1,0) & (2,0) & (3,0) & (4,0) & \cdots \\
(0,1) & (1,1) & (2,1) & (3,1) & (4,1) & \cdots \\
(0,2) & (1,2) & (2,2) & (3,2) & (4,2) & \cdots \\
(0,3) & (1,3) & (2,3) & (3,3) & (4,3) & \cdots \\
(0,4) & (1,4) & (2,4) & (3,4) & (4,4) & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots \\
\end{array}
\]

Note that the “outer” stream runs left-to-right and the “inner” streams run top-to-bottom.

Examples:

- \(\text{sDiag } (\text{sCrossWith } (,) \, \text{nats nats}) \sim [(0,0),(1,1),(2,2),(3,3),(4,4),...],[(1,0),(1,1),(1,2),(1,3),(1,4),...],[(2,0),(2,1),(2,2),(2,3),(2,4),...],[(3,0),(3,1),(3,2),(3,3),(3,4),...],[(4,0),(4,1),(4,2),(4,3),(4,4),...],...]\)

- \(\text{sDiag } (\text{sCrossWith } (,) \, \text{nats nats}) \, !! 99 \sim (99,99)\)

- \(\text{sDiag } (\text{sCrossWith } (,) \, \text{nats nats}) \, !! 999 \sim (999,999)\)

- \(\text{sDiag } (\text{sCrossWith } (*) \, \text{fibs facts}) \sim [0,1,2,12,72,...],\)

- \(\text{sDiag } (\text{sCrossWith } (*) \, \text{fibs facts}) \, !! 9 \sim 12337920\)

- \(\text{sDiag } (\text{sCrossWith } (*) \, \text{fibs facts}) \, !! 19 \sim 508598164809326592000\)

Using \(\text{sDiag}\), \(\text{Stream}\) can be made a (proper) instance of \(\text{Monad}\); see “Streams and Monad Laws” by Nicolas Wu (http://zenzike.com/posts/2010-10-21-streams-and-monad-laws) and “The stream monad” by Jeremy Gibbons (https://patternsinfp.wordpress.com/2010/12/31/stream-monad/). Also note that lazy evaluation is critical; the implementations of \(\gg=\) and \(\leftrightarrow\) create many applications of functions to arguments, the vast majority of which are discarded by \(\text{sDiag}\).

Note: For full credit, \(\text{sDiag}\) must be implemented without using \(\text{!!}\) (stream indexing).
5. (5pts) Define the function

```
• sSquish :: Stream (Stream a) -> Stream a
```

such that \( sSquish \) \( sss \) returns the stream obtained by interleaving the first stream of \( sss \) with the squishing of the remaining streams of \( sss \). (The code is prettier than the picture.)

Examples:

- \( sSquish (sCrossWith (,) nats nats) \) \( \rightarrow \) \([0,0),(1,0),(0,1),(2,0),(0,2),...\]$
- \( sSquish (sCrossWith (,) nats nats) \) \( !! 99 \rightarrow (2,12) \)
- \( sSquish (sCrossWith (,) nats nats) \) \( !! 999 \rightarrow (3,62) \)
- \( sSquish (sCrossWith (*) fibs facts) \) \( \rightarrow \) \([0,1,0,1,0,...]\$
- \( sSquish (sCrossWith (*) fibs facts) \) \( !! 9 \rightarrow 2 \)
- \( sSquish (sCrossWith (*) fibs facts) \) \( !! 19 \rightarrow 2 \)
- \( sSquish (sCrossWith (*) fibs facts) \) \( !! 99 \rightarrow 479001600 \)

Using \( sSquish \), define the stream

```
• ruler :: Stream Integer
```

such that \( ruler \) is the stream where the \( n^{th} \) element of the stream is the largest power of 2 that (evenly) divides \( n + 1 \).

Examples:

- \( ruler \rightarrow \) \([0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,4,0,1,0,2,0,1,0,3,0,...]\$

Strive for a concise solution; the reference solution has a definition of \( ruler \) that is only 7 tokens
6. (5pts) Define the function

- \( sWalk :: \text{Stream (Stream } a \text{)} \rightarrow \text{Stream } a \)

such that \( sWalk f xss \) returns the stream obtained by concatenating all of the up-right diagonal elements of \( xss \).

\[
\begin{array}{cccccc}
(0,0) & (1,0) & (2,0) & (3,0) & (4,0) & \cdots \\
(0,1) & (1,1) & (2,1) & (3,1) & (4,1) & \cdots \\
(0,2) & (1,2) & (2,2) & (3,2) & (4,2) & \cdots \\
(0,3) & (1,3) & (2,3) & (3,3) & (4,3) & \cdots \\
(0,4) & (1,4) & (2,4) & (3,4) & (4,4) & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

Note that the “outer” stream runs left-to-right and the “inner” streams run top-to-bottom.

Examples:

- \( sWalk (sCrossWith (,) nats nats) \sim [\{(0,0),(0,1),(1,0),(0,2),(1,1),\ldots\}] \)
- \( sWalk (sCrossWith (,) nats nats) \text{!! 99} \sim (8,5) \)
- \( sWalk (sCrossWith (,) nats nats) \text{!! 999} \sim (9,35) \)
- \( sWalk (sCrossWith (*) fibs facts) \sim [0,0,1,0,1,\ldots] \)
- \( sWalk (sCrossWith (*) fibs facts) \text{!! 9} \sim 2 \)
- \( sWalk (sCrossWith (*) fibs facts) \text{!! 19} \sim 3 \)
- \( sWalk (sCrossWith (*) fibs facts) \text{!! 99} \sim 2520 \)

Using \( sWalk \), we can construct the classic enumeration of the rationals.

Note: For full credit, \( sWalk \) must be implemented without using \( \text{!!} \) (stream indexing).

Hints:

- The first diagonal has length 1, the second diagonal has length 2, the third diagonal has length 3, \ldots.
- After removing the previous diagonals, where is the next diagonal?
Make Change

Download MakeChange.hs.

1. (15pts) Define a function

   • makeChange :: [Int] -> Int -> Maybe [Int]

   such that solves the “make change” problem (https://en.wikipedia.org/wiki/Change-making_problem). Assume that 
   \( ds = [d_1, \ldots, d_k] \) is a (finite) list of positive integers, representing coin denominations, and that \( n \geq 0 \). Then 
   makeChange \( ds \) \( n \) returns a minimal length list of coins with total value equal to \( n \) (if one exists). Specifically, if 
   makeChange \( ds \) \( n \) returns \( \text{Just} \ [c_1, \ldots, c_l] \), then it must be the case that \( c_1 + \cdots + c_l = n \), each \( c_j \) is equal to 
   some \( d_i \), and there does not exist a list \( [c'_1, \ldots, c'_l] \) such that \( l' < l \), \( c'_1 + \cdots + c'_l = n \), and each \( c'_j \) is equal to 
   some \( d'_i \). And, if makeChange \( ds \) \( n \) returns \( \text{Nothing} \) then there must not exist a list \( [c_1, \ldots, c_l] \) such that \( l < l \), 
   \( c_1 + \cdots + c_l = n \), and each \( c_j \) is equal to some \( d_i \).

   The “make change” problem is a classic example of dynamic programming and, as described in lecture and recitation, 
   many dynamic programming problems can be elegantly solved in Haskell using the lazy evaluation.

   The Wikipedia page describes a dynamic-programming algorithm for the problem using a two-dimensional array. 
   However, it may be simpler to use a dynamic-programming algorithm for the problem using a one-dimensional 
   array. Consider the simpler problem of determining \( l \), the minimum number of coins that sum to the value \( n \). 
   Define \( L[m] \) to be the minimum number of coins that sum to the value \( m \) (and, therefore, \( l = L[n] \)); entries of \( L \) 
   may be \( \infty \), representing that there is no list of coins of any length that sum to the value \( m \). \( L \) can be recursively 
   defined:

   \[
   L[m] = \begin{cases} 
   \infty & m < 0 \\
   0 & m = 0 \\
   1 + \min \{L[m - d_i] \mid 1 \leq i \leq k\} & m > 0 
   \end{cases}
   \]

   To solve the full problem, simply keep track of the denomination \( d_i \) chosen for each minimum. The other minor 
   challenge with coding this in Haskell is that the \( m < 0 \) case must be handled explicitly.

   Be sure to review the documentation of the Data.Array.IArray module 
   (https://downloads.haskell.org/ghc/8.6.5/docs/html/libraries/array-0.5.3.0/Data-Array-IArray.html).

Requirements and Submission

Your submission must :load into ghci without errors; submissions that have parse errors or type errors will receive no 
credit. Submissions that violate code style guidelines will lose up to 25%.

Submit ArrayStateMonad.hs, BubbleSort.hs, Stream.hs, and MakeChange.hs to the Homework06 Assignment on My-
Courses by the due date.

Document History

October 25, 2019
Original version