Programming Skills: Functional Programming and Haskell
CSCI-541/CSCI-641
Term 20191

Homework 04
Due: October 18, 2019

Introduction

The major theme of this assignment is to implement purely functional data structures and associated operations. A minor theme of this assignment is to see the use of types to enforce invariants of data structures.

For full credit, complete EITHER Part 1. (Append Lists) and Part 2. (Size-Balanced Binary Tree Arrays) OR Part 2. (Size-Balanced Binary Tree Arrays) and Part 3. (Binary Random Access Lists with Non-Uniform Recursive Types and Polymorphic Recursion), based on how you can most effectively further your understanding of Haskell and functional programming. Briefly,

- Part 1. (Append Lists): Asks you to practice extending the standard first-order and higher-order list-processing functions to an implementation of the list abstract data type that makes construction cheap, but makes destruction expensive.

- Part 2. (Size-Balanced Binary Tree Arrays): Asks you to exploit the invariants of a data structure to efficiently query and update a data structure. In addition, you will need to efficiently construct the data structure to maximize sharing.

- Part 3. (Binary Random Access Lists with Non-Uniform Recursive Types and Polymorphic Recursion): Asks you to learn about and use a sophisticated type definition that ensures that certain data structure invariants are not violated. Although understanding the type can be challenging and the functions more difficult to initially implement, the payoff is that the overall implementation is significantly more concise and simpler.

Note: Although each part has a different number of questions and points, each part will be weighted equally in the Homework 04 grade.

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.
1 Append Lists

Download AppendList.hs.

In class, we noted that list append (++) must copy the first list and, therefore, is a linear time operation. Algorithms (such as quicksort) that make extensive use of list append may suffer (in running time).

In class, we also noted that a different implementation of the same abstract data type can sometimes “shift” a portion of the running time from one operation to another.

An append list (also known as a catenable list) is a (simple) implementation of the list abstract data type that makes construction cheap ($O(1)$), but makes destruction expensive ($O(n)$). In particular, appending two append lists is a constant time operation, but splitting a list into a head and a tail is a linear time operation. It is useful for algorithms that construct large lists in an irregular fashion (e.g., logging, pretty printing, web templating) and need only convert to a (standard) list at the end.

The \texttt{AListNonEmpty a} and \texttt{AList a} types are defined as follows:

```hs
data AListNonEmpty a = NEList (Data.List.NonEmpty.NonEmpty a)       
  | Append (AListNonEmpty a) (AListNonEmpty a)
  deriving (Show, Read, Eq)

data AList a = Empty | NonEmpty (AListNonEmpty a)
  deriving (Show, Read, Eq)
```

The \texttt{AListNonEmpty a} type represents the non-empty append lists, while the \texttt{AList a} type represents arbitrary (empty or non-empty) append lists. Note that the \texttt{NEList} constructor of the \texttt{AListNonEmpty a} type wraps a value of the \texttt{NonEmpty a} type from the \texttt{Data.List.NonEmpty} module (https://downloads.haskell.org/~ghc/8.6.5/docs/html/libraries/base-4.12.0.0/Data-List-NonEmpty.html); review this documentation for useful functions.

1. (5 pts) Define a function
   - \texttt{alistFromList :: [a] -> AList a}

   that converts a (regular) list to an append list. The function \texttt{alistFromList xs} should run in $O(1)$ time and should not use any partial functions.

2. (5 pts) Define a function
   - \texttt{alistAppend :: AList a -> AList a -> AList a}

   that appends two append lists. The function \texttt{alistAppend xs ys} should run in $O(1)$ time.

   Note: This function is used to make \texttt{AList} an instance of the \texttt{Monoid} type class.

3. (5 pts) Define functions
   - \texttt{alistCons :: a -> AList a -> AList a}
   - \texttt{alistSnoc :: AList a -> a -> AList a}

   that adds a new element to the front of an append list and adds a new element to the rear of an append list, respectively. (\texttt{snoc} is “backwards” \texttt{cons}.) The functions \texttt{alistCons x xs} and \texttt{alistSnoc xs x} should run in $O(1)$ time.

4. (5 pts) Define functions
   - \texttt{alistUncons :: AList a -> Maybe (a, AList a)}
   - \texttt{alistUnsnoc :: AList a -> Maybe (AList a, a)}

   that splits an append list into its first element and the remaining elements (as an append list) and that splits an append list into its last element and the remaining elements (as an append list), respectively. The functions \texttt{alistUncons xs} and \texttt{alistUnsnoc xs} should run in $O(n)$ time.
5. (5pts) Define a function

- \textbf{alistMap} :: (a \to b) \to AList a \to AList b

that performs a \textit{map} on an append list. The function \textbf{alistMap} \( f \; xs \) should run in \( O(n \times t) \) time (where \( t \) is the running time of the function \( f \)).

Note: repeatedly calling \textbf{alistUncons} or \textbf{alistUnsnoc} will not achieve the required running time.

Note: This function is used to make \textbf{AList} an instance of the \textbf{Functor} type class.

6. (5pts) Define a function

- \textbf{alistFilter} :: (a \to \textbf{Bool}) \to AList a \to AList a

that performs a \textit{filter} on an append list. The function \textbf{alistFilter} \( f \; xs \) should run in \( O(n \times t) \) time (where \( t \) is the running time of the function \( f \)).

Note: repeatedly calling \textbf{alistUncons} or \textbf{alistUnsnoc} will not achieve the required running time.

7. (5pts) Define functions

- \textbf{alistFoldr} :: (a \to b \to b) \to b \to AList a \to b
- \textbf{alistFoldl} :: (b \to a \to b) \to b \to AList a \to b

that performs a \textit{right(-to-left)-fold} of an append list and that performs a \textit{left(-to-right)-fold} of an append list, respectively. The functions \textbf{alistFoldr} \( f \; b \; xs \) and \textbf{alistFoldl} \( f \; b \; xs \) should run in \( O(n \times t) \) time (where \( t \) is the running time of the function \( f \)).

Note: repeatedly calling \textbf{alistUncons} or \textbf{alistUnsnoc} will not achieve the required running time.

Note: These functions is used to make \textbf{AList} an instance of the \textbf{Foldable} type class.

8. (5pts) In a comment, explain why the definition of \textbf{AList} \( a \) given above is better than this seemingly simpler definition:

\begin{verbatim}
data AList a = Empty | List [a] | Append (AList a) (AList a)
\end{verbatim}
2 Size-balanced Binary Tree Arrays

Download SizeBalancedBinaryTreeArray.hs.

The *SBBTree* a type is defined as follows:

```haskell
data SBBTree a = Leaf
                 | Node Int (SBBTree a) a (SBBTree a)
  deriving (Read, Show, Eq)
```

Note the invariant that asserts that these trees are *size balanced*: the sizes of the subtrees of a *Node* differ by at most one. Also note the invariant that asserts that the *Int* component of a *Node* records the size of the tree.

1. (5pts) Define a function

   • *new :: Int -> a -> SBBTree a*

   such that *new n x* returns a size-balanced binary tree of *n* *xs*. If *n* is less than or equal to 0, then *new n x* returns *Leaf*. The function *new n x* should run in $O(\log n)$ time and return a size-balanced binary tree that shares identical subtrees and uses $O(\log n)$ space.

Examples (figures demonstrate sharing):

   • *new 0 'Z' :: SBBTree Char ~ Leaf*

   ![Diagram 1](https://example.com/diagram1.png)

   • *new (-5) 'Z' :: SBBTree Char ~ Leaf*

   ![Diagram 2](https://example.com/diagram2.png)

   • *new 2 'Z' :: SBBTree Char ~ Node 2 (Node 1 Leaf 'Z' Leaf) 'Z' Leaf*

   ![Diagram 3](https://example.com/diagram3.png)

   • *new 7 'Z' :: SBBTree Char ~ Node 7 (Node 3 (Node 1 Leaf 'Z' Leaf) 'Z' (Node 1 Leaf 'Z' Leaf)) 'Z' (Node 3 (Node 1 Leaf 'Z' Leaf) 'Z' (Node 1 Leaf 'Z' Leaf))*

   ![Diagram 4](https://example.com/diagram4.png)
• new 10 'Z' :: SBBTree Char
  ~ node 10 (node 5 (node 2 (node 1 leaf 'Z' leaf) 'Z' leaf) 'Z')
    (node 2 (node 1 leaf 'Z' leaf) 'Z' leaf))
  (node 4 (node 2 (node 1 leaf 'Z' leaf) 'Z' leaf) 'Z')
    (node 1 leaf 'Z' leaf))

Note:
• The explicit type signatures select the SBBTree instance of the Array type class.

Hints:
• Introduce a helper function new2 :: Int -> a -> (SBBTree a, SBBTree a) such that new2 n x returns a pair of size-balanced binary trees, the first of size n and the second of size n + 1, with x in every Node.

2. (5pts) Define a function
• idx :: SBBTree a -> Int -> Maybe a
such that idx t i returns Just the element at index i (0-based indexing, according to an in-order, left-to-right traversal). If the index i is out of bounds for the tree t, then idx t i returns Nothing. The function idx t i should run in O(log n) time.

Examples:
• map (idx (new 10 'Z' :: SBBTree Char)) [-2 .. 11]
• map (idx (upd (new 10 'Z' :: SBBTree Char) 5 'A')) [-2 .. 11]
3. (5pts) Define a function

\[ \text{upd} :: \text{SBBTree} \, a \rightarrow \text{Int} \rightarrow a \rightarrow \text{SBBTree} \, a \]

such that \( \text{upd} \, t \, i \, y \) returns a size-balanced binary tree that is identical to \( t \) except that \( y \) is the element at index \( i \) (0-based indexing, according to an in-order traversal). If the index \( i \) is out of bounds for the tree \( t \), then \( \text{upd} \, t \, i \, y \) returns \( t \). The function \( \text{upd} \, t \, i \) should run in \( O(\log n) \) time.

Examples (figures demonstrate sharing):

- \( \text{upd} \, t \, 0 \, 'A' \)
  \[ \sim \text{Node} \, 10 \, (\text{Node} \, 5 \, (\text{Node} \, 2 \, (\text{Node} \, 1 \, \text{Leaf} \, 'A' \, \text{Leaf}) \, 'Z' \, \text{Leaf}) \, 'Z') \]
  \[ \sim (\text{Node} \, 4 \, (\text{Node} \, 2 \, (\text{Node} \, 1 \, \text{Leaf} \, 'Z' \, \text{Leaf}) \, 'Z' \, \text{Leaf}) \, 'Z') \]

- \( \text{upd} \, t \, 2 \, 'A' \)
  \[ \sim \text{Node} \, 10 \, (\text{Node} \, 5 \, (\text{Node} \, 2 \, (\text{Node} \, 1 \, \text{Leaf} \, 'Z' \, \text{Leaf}) \, 'Z' \, \text{Leaf}) \, 'A') \]
  \[ \sim (\text{Node} \, 4 \, (\text{Node} \, 2 \, (\text{Node} \, 1 \, \text{Leaf} \, 'Z' \, \text{Leaf}) \, 'Z' \, \text{Leaf}) \, 'Z') \]
• upd t 5 'A'
  ~ Node 10 (Node 5 (Node 2 (Node 1 Leaf 'Z' Leaf) 'Z' Leaf) 'Z'
       (Node 2 (Node 1 Leaf 'Z' Leaf) 'Z' Leaf))
        'A'
       (Node 4 (Node 2 (Node 1 Leaf 'Z' Leaf) 'Z' Leaf) 'Z'
        (Node 1 Leaf 'Z' Leaf))

• upd t (~2) 'A'
  ~ Node 10 (Node 5 (Node 2 (Node 1 Leaf 'Z' Leaf) 'Z' Leaf) 'Z'
       (Node 2 (Node 1 Leaf 'Z' Leaf) 'Z' Leaf))
        'Z'
       (Node 4 (Node 2 (Node 1 Leaf 'Z' Leaf) 'Z' Leaf) 'Z'
        (Node 1 Leaf 'Z' Leaf))
• upd (upd t 3 'A') 8 'B'
  ~ Node 10 (Node 5 (Node 2 (Node 1 Leaf 'Z' Leaf) 'Z' Leaf)
    'Z'
    (Node 2 (Node 1 Leaf 'A' Leaf) 'Z' Leaf))
  'Z'
  (Node 4 (Node 2 (Node 1 Leaf 'Z' Leaf) 'Z' Leaf)
    'B'
    (Node 1 Leaf 'Z' Leaf))

where

\[
t :: \text{SBBTree Char}
\]
\[
t = \text{new 10 'Z'}
\]

4. (5pts) Define a function
   
   • \text{fmap} :: (a \to b) \to \text{SBBTree a} \to \text{SBBTree b}

   that makes \text{SBBTree} an instance of the \text{Functor} type class.

5. (5pts) Define a function
   
   • \text{foldr} :: (a \to b \to b) \to b \to \text{SBBTree a} \to b

   that performs a \textit{right(-to-left)-fold} of a size-balanced binary tree array and makes \text{SBBTree} an instance of the \text{Foldable} type class.

6. (5pts) In addition to \text{foldr}, there is (at least) one method of the \text{Foldable} type class that should be defined for the \text{SBBTree} type because a direct implementation will be more efficient than the default implementation. Add the method(s) to the \textit{instance Foldable SBBTree} and, in a comment, explain why the direct implementation(s) will be more efficient than the default implementation(s). This question asks you to imagine how the non-required methods of the \text{Foldable} type class are likely to be implemented in terms of the required \text{foldr} method and, therefore, which methods would be more efficient to give a direct implementation. Review the methods of the \text{Foldable} type class (https://downloads.haskell.org/~ghc/8.6.5/docs/html/libraries/base-4.12.0.0/Data-Foldable.html#t:Foldable).
3 Binary Random-Access Lists with Non-Uniform Recursive Types and Polymorphic Recursion

Download BinaryRandomAccessList.hs.

Recall the implementation from lecture of a random-access list (a.k.a., a one-sided flexible array) inspired by a binary numerical representation:

```haskell
data BRAList a = Nil | ConsOne a (BRAList (a,a)) | ConsZero (BRAList (a,a))

-- Invariant:: forall (BRAList ds). snd (foldl aux (1,True) ds)
-- where aux (p,b) Zero = (p+1,b)
-- aux (p,b) (One t) = (p+1, b && 2^p == sizeTree t)
```

Note that there are a number of invariants that are expected to hold; in particular, each `Tree a` is a complete binary leaf tree of the appropriate size for its position in the list. While we can carefully write and test our code to check that this invariant holds, it would be more satisfying if the invariants could be reflected in the type of the data structure and any violations of the invariants would manifest as compile-time type errors.

In this section, we examine an alternative implementation that ensures that the correct number of elements are stored at the correct positions. The (new) `BRAList a` type is defined as follows:

```haskell
data BRAList a = Nil | ConsOne a (BRAList (a,a)) | ConsZero (BRAList (a,a))

-- Invariant:: A "Tree a" is a complete binary leaf tree.
-- Invariant:: for all (Node s l r). sizeTree l == sizeTree r && s = sizeTree l + sizeTree r
```

Most polymorphic types that we have seen, such as the `Tree a` type above, are uniform recursive types (a.k.a., regular types): in the places where there is a recursive use of the type being defined, it is identical to the type being defined (that is, `Tree a` is defined in terms of `Tree a`). In contrast, the `BRAList a` type is a non-uniform recursive type (a.k.a., nested type): in the places where there is a recursive use of the type being defined, it is different from the type being defined (that is, `BRAList a` is defined in terms of `BRAList (a,a)`). Thus, we can understand `BRAList a` as a “list” of increasingly nested pairs; for example, a `BRAList Char` starts with a character, followed by a pair of characters, followed by a pair of pair of characters, .... Note that the recursion on the type `BRAList (a,a)` exactly forces the number of elements in the nested pairs to double each time. Thus, each of these nested pairs corresponds to a complete binary leaf tree and also corresponds to the increasing weights of positions of a binary numerical representation. For example, the sequence 'A' to 'K' is represented as:

```haskell
ConsOne 'A' (
  ConsOne ('B', 'C') (ConsZero (ConsOne (((('D', 'E'), ('F', 'G'))), (((('H', 'I'), ('J', 'K')))))) (Nil)))
```

Note that this sequence has $11 = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 10112$ elements. Moreover, no other organization of 11 elements is well typed; for example, the expression:

```haskell
ConsOne 'A' (
  ConsOne ('B', 'C') (ConsOne (((('D', 'E'), ('F', 'G')))
    ConsOne (((('H', 'I'), ('J', 'K')))) (Nil)))
```

has a type error:

* Couldn't match type `Char` with `Char Char`
  Expected type: `BRAList (((Char, Char), (Char, Char)), ((Char, Char), (Char, Char)))`
  Actual type: `BRAList (((Char, Char), (Char, Char)), (Char, Char))`

* In the second argument of `ConsOne`, namely
  (ConsOne (((('H', 'I'), ('J', 'K'))) Nil)
In the second argument of `ConsOne`, namely
(ConsOne (((('D', 'E'), ('F', 'G')) (ConsOne (((('H', 'I'), ('J', 'K')))) Nil))
Recursive functions over non-uniform recursive types use polymorphic recursion: in the places where there is a recursive use of the function being defined, it may have a different type from the type of the function being defined. For example, here is a `bralistToString` function that converts a binary random-access list to a string, making use of a function to convert elements to strings:

```haskell
bralistToString :: (a -> String) -> BRAList a -> String
bralistToString aToString xs =
  case xs of
    Nil -> ""
    ConsOne y ys -> aToString y ++ ',' : aToString pairToString ys
    ConsZero ys -> bralistToString pairToString ys
where pairToString (z0,z1) = aToString z0 ++ ',' : aToString z1
      _ ?: [] = []
      z ?: zs = z:zs
```

Note that `bralistToString` is being defined with the type `(a -> String) -> BRAList a -> String` but the recursive calls `bralistToString pairToString ys` use `bralistToString` with the type `((a,a) -> String) -> BRAList (a,a) -> String`. In particular, the `aToString :: a -> String` function is “upgraded” to `pairToString :: (a,a) -> String` and used as the element-to-string function for the recursive calls.

This example demonstrates the main “trick” to writing higher-order polymorphic-recursive functions over non-uniform recursive types: at the point of the recursive call, the argument function must be “upgraded” from working on the “old” element type to working on the “new” element type. There is one other “trick” to writing polymorphic-recursive functions: such functions always require a type signature. Type inference of polymorphic-recursive functions is undecidable, so no compiler attempts to infer types of polymorphic-recursive functions. This is rarely a problem for top-level functions, which should have a type signature, or instance methods of type classes, which “know” the type signature from the type class definition, but does require polymorphic-recursive helper functions defined in `where` clauses to be given an explicit type signature; a type error of the form `Occurs check: cannot construct the infinite type: t ~ (t, u)` may indicate that a polymorphic-recursive function is missing a type signature.

1. (5pts) Define a function
   - `push :: a -> BRAList a -> BRAList a`
   that adds a new element to the front of a binary random-access list. The function `push x x` should run in `O(log n)` time.

   Examples:
   - `push 'Z' Nil ~> ConsOne 'Z' Nil`
   - `push 'Y' (push 'Z' Nil) ~> ConsZero (ConsOne ('Y','Z') Nil)`
   - `push 'X' (push 'Y' (push 'Z' Nil))`  
     ~> ConsOne 'X' (ConsOne ('Y','Z') Nil)

   Hints:
   - This function works like an “add-with-carry”; see the `consDigits` function from the old implementation of binary-random access lists.
2. (5pts) Define a function
   
   \[ \text{pop} :: \text{BRAList } a \rightarrow \text{Maybe} (a, \text{BRAList } a) \]

   that splits a binary random-access list into its first element and the remaining elements (as a binary random-access list). The function \[\text{alistUncons} \quad xs\] should run in \(O(\log n)\) time.

   Examples:

   \[ \text{pop} (\text{ConsOne 'X'} (\text{ConsOne ('Y','Z')} \text{Nil})) \]
   \[ \sim \text{Just ('X',ConsZero (\text{ConsOne ('Y','Z')} \text{Nil}))} \]

   \[ \text{pop} (\text{ConsZero (ConsOne ('Y','Z')} \text{Nil})) \]
   \[ \sim \text{Just ('Y',ConsOne 'Z' Nil)} \]

   \[ \text{pop} (\text{ConsOne 'Z'} \text{Nil}) \sim \text{Just 'Z', Nil} \]

   \[ \text{pop Nil} \sim \text{Nothing} \]

   Hints:

   - This function works like an “sub-with-borrow”; see the \textit{unconsDigits} function from the old implementation of binary-random access lists.

3. (5pts) Define a function
   
   \[ \text{new} :: \text{Int} \rightarrow a \rightarrow \text{BRAList } a \]

   such that \[\text{new } n \ x \] returns a binary random-access list of \( n x \). If \( n \) is less than or equal to \( 0 \), then \[\text{new } n \ x \] returns \[\text{Nil}\]. The function \[\text{new } n \ x \] should run in \( O(\log n) \) time and return a binary random-access list that maximizes sharing and uses \( O(\log n) \) space.

   Note: repeatedly calling \textit{push} will not achieve the required running time.

   Examples:

   \[ \text{new 0 'Z' :: BRAList Char} \sim \text{Nil} \]

   \[ \text{new (-5) 'Z' :: BRAList Char} \sim \text{Nil} \]

   \[ \text{new 2 'Z' :: BRAList Char} \sim \text{ConsZero (ConsOne ('Z','Z')} \text{Nil}) \]

   \[ \text{new 7 'Z' :: BRAList Char} \]
   \[ \sim \text{ConsOne 'Z'} ( \]
   \[ \quad \text{ConsOne ('Z','Z')} ( \]
   \[ \quad \quad \text{ConsOne (('Z','Z')},('Z','Z')) \]
   \[ \quad \quad \text{Nil))} \]

   \[ \text{new 10 'Z' :: BRAList Char} \]
   \[ \sim \text{ConsZero (} \]
   \[ \quad \text{ConsOne ('Z','Z')} ( \]
   \[ \quad \quad \text{ConsZero (} \]
   \[ \quad \quad \quad \text{ConsOne (((('Z','Z')},('Z','Z'))),('Z','Z')) \]
   \[ \quad \quad \quad \text{Nil))})) \]

   Hints:

   - This function works much like converting a number to its binary representation.

4. (5pts) Define a function
   
   \[ \text{size} :: \text{BRAList } a \rightarrow \text{Int} \]

   such that \[\text{size } xs\] returns the size (number of elements). The function \[\text{size } xs\] should run in \( O(\log n) \) time.

   Hints:

   - This function works much like converting a binary representation to a number.
5. (5pts) Define a function
   \[
   \text{idx} :: \text{BRAList } a \to \text{Int } \to \text{Maybe } a
   \]
such that \(\text{idx } xs i\) returns \text{Just} the element at index \(i\) (0-based indexing, according to an in-order traversal). If the index \(i\) is out of bounds for the binary random-access list \(xs\), then \(\text{idx } xs i\) returns \text{Nothing}. The function \(\text{idx } xs i\) should run in \(O(\log n)\) time.

Warning: \(\text{idx}\) is tricky.

Hints:
   - A lookup on \text{Nil} necessarily fails.
   - A lookup on \text{ConsOne } x xs either succeeds with \(x\) (if the index is 0) or “skips” over \(x\) and performs a lookup on \text{ConsZero } xs with the index decremented.
   - A lookup on \text{ConsZero } xs (notionally, a sequence of elements) executes by a lookup on \(xs\) (notionally, a sequence of pairs of elements). To lookup an element at index \(i\) in a sequence of pairs, lookup the pair at index \(i \div 2\) and then extract the appropriate element from that pair (based on the value of \(i \mod 2\)).

6. (5pts) Define a function
   \[
   \text{upd} :: \text{BRAList } a \to \text{Int } \to a \to \text{BRAList } a
   \]
such that \(\text{upd } xs i y\) returns a binary random-access list that is identical to \(xs\) except that \(y\) is the element at index \(i\) (0-based indexing, according to an in-order, left-to-right traversal). If the index \(i\) is out of bounds for the binary random-access list \(xs\), then \(\text{upd } xs i y\) returns \(xs\). The function \(\text{idx } xs i\) should run in \(O(\log^2 n)\) or \(O(\log n)\) time.

Warning: \(\text{upd}\) is very tricky.

Hints:
   - An update on \text{Nil} necessarily fails.
   - An update on \text{ConsOne } x xs either succeeds with \text{ConsOne } y xs (if the index is 0) or \text{push}-es \(x\) onto the result of an update on \text{ConsZero } xs with the index decremented.
   - An update on \text{ConsZero } xs (notionally, a sequence of elements) executes by an update on \(xs\) (notionally, a sequence of pairs of elements). To update an element at index \(i\) in a sequence of pairs, lookup the pair at index \(i \div 2\), construct a replacement pair (based on the value of \(i \mod 2\)), and update the pair at index \(i \div 2\).

The above hints lead to an \(O(\log^2 n)\) running time. To achieve an \(O(\log n)\) running time, avoid the lookup-the-old-pair then construct-the-replacement-pair then update-to-the-new-pair sequence by introducing an \(\text{fupd} :: \text{BRAList } a \to \text{Int } \to (a \to a) \to \text{BRAList } a\) function that sends a function to construct the new pair from the old pair wherever the old pair is found (and defines \(\text{upd } xs i y = \text{fupd } xs i (\lambda \_ \to y)\)).

Think about how to “upgrade” the \(a \to a\) replacement function to an \((a,a) \to (a,a)\) replacement function that applies the original replacement to either the first or second element of the pair (based on the value of \(i \mod 2\)).

7. (5pts) Define a function
   \[
   \text{fmap} :: \text{BRAList } a \to (a \to b) \to \text{BRAList } b
   \]
that makes \text{BRAList} an instance of the \text{Functor} type class.

Hints:
   - Think about how to “upgrade” the \(a \to b\) mapping function to an \((a,a) \to (b,b)\) mapping function.

8. (5pts) Define a function
   \[
   \text{foldr} :: (a \to b \to b) \to b \to \text{BRAList } a \to b
   \]
that performs a \text{right(-to-left)-fold} of a binary random-access list and makes \text{BRAList} an instance of the \text{Foldable} type class.

Hints:
   - Think about how to “upgrade” the \(a \to b\) folding function to an \((a,a) \to b \to b\) (right-to-left) folding function.
9. (5pts) In addition to foldr, there is (at least) one method of the Foldable type class that should be defined for the BRAList type because a direct implementation will be more efficient than the default implementation. Add the method(s) to the instance Foldable BRAList and, in a comment, explain why the direct implementation(s) will be more efficient than the default implementation(s). This question asks you to imagine how the non-required methods of the Foldable type class are likely to be implemented in terms of the required foldr method and, therefore, which methods would be more efficient to give a direct implementation. Review the methods of the Foldable type class (https://downloads.haskell.org/~ghc/8.6.5/docs/html/libraries/base-4.12.0.0/Data-Foldable.html#t:Foldable).

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 either (AppendList.hs and SizeBalancedBinaryTreeArray.hs) or (SizeBalancedBinaryTreeArray.hs and BinaryRandomAccessList.hs) to the Homework04 Assignment on MyCourses by the due date.