Homework Assignment 4

Functional and Logic Programming, 2023

Due date: Thursday, June 15, 2023 (15/06/2023)

Bureaucracy

- Submission is in pairs, but solo submission is also allowed. We very much suggest you pair-up, as solving this exercises with another person greatly enhances your learning (as well as being more fun!).
- To submit, create a zip file named `HW4_<id1>_<id2>.zip` where `<id1>` and `<id2>` are the submitters IDs.
  - Or `HW4_<id>.zip` if submitting alone.
- The zip file should contain **a single file** named `HW4.hs`!
- Make sure your submission compiles successfully. Submissions which do not compile will receive a 0 grade!
  - We will be using the following command to compile the file: `ghc -Wall -Werror HW4.hs`.
- You may submit the assignment after the due date even without approval (e.g., excluding reserve duty, serious illness, or other cases covered by the student administration).
  - You will be penalized for **5 points for every late day**.
  - The **maximum** extension allowed by this is **3 days**.
- For any late submissions, with or without approval, please E-mail your submission directly to ofir.yaniv@post.runi.ac.il.

General notes

- You may not modify the **`import`** statement at the top of the file, nor add new **`import`**s.
  - If you are unsure what some function does, you can either ask HLS or Hoogle.
  - Hoogle also support module lookups, e.g., `Prelude.notElem`.
  - Do be aware however, that some functions from the standard library are more general than what we have learned so far in class.
    - And some cases their definition may not be entire clear just yet!

- You may not add any new `LANGUAGE` pragmas.
- The exercises and sections are defined linearly. That is, it is a good idea to use previously defined functions (either from the same section or ones before it). It is also a good idea to use functions you saw in class; some of them are already imported, and some of them you would have to define yourself.
  - Do not be alarmed by the large amount of functions! Unlike previous assignments there are no "big" functions. All functions can be implemented using one-lines (or one line per pattern).
  - In general, you may define as many helper functions as you wish.
- Try to write elegant code, as taught in class. Use point-free style, η-reductions, and function composition to make your code shorter and more declarative. Prefer `foldr` over manual recursion where possible, and use functions like `map` and `filter` when appropriate. Although it is not required in the homework assignments, non-elegant code *will* be penalized in the test, so this is a good exercise. HLS and hlint can be very helpful in this.
  - Do note that in some cases, hlint may suggest functions which are not imported, or which you are trying to implement right now!
- If possible, please ask your questions first in Piazza, as this will allow all students to take part in the discussion.

Section 1: `Foldable` functions

In this section you will implement a few functions we saw for lists for `Foldable`, plus a few extra ones. Note that functions which can early exit—`elem`, `null`, etc.—should support it.

Here are example usages of the new functions:

```
getSum $ fold $ map Sum [1, 2, 3]
6
toList $ Just 4
[4]
toList $ Nothing
[]
-- Not part of HW4.hs.
-- You can take the implementation and instances from the lectures.
single a = Tree Empty a Empty
toList $ Tree (single 1) 2 (single 3)
[1,2,3]
maxBy length ["foo", "bar", "bazz"]
Just "bazz"
minBy length ["bar", "bazz"]
Just "bar"
```

Hint: The `Arg` type from the previous homework can also be useful here (it is already pre-imported from `Data.Semigroup`).
Tip: Since you already saw how to implement a few of these using `foldr`, it is **a very good exercise** to implement them using `foldMap`!

Section 2: `Functor` functions

In this section you will implement a few functions on `Functor`s. Here are example usages of the new functions:

```
fmapToFst length ["foo", "bar"]
[(3,"foo"),(3,"bar")]
fmapToSnd length $ Just "foo"
Just ("foo",3)
strengthenL 42 $ Right "foo"
Right (42,"foo")
strengthenR "x" [1, 2, 3]
[(1,"x"),(2,"x"),(3,"x")]
unzip $ Just (1,2)
(Just1,Just2)
coUnzip (Right [1,2,3] :: Either String [Int])
[Right 1,Right 2,Right 3]
coUnzip (Left "foo" :: Either String [Int])
[Left 'f',Left 'o',Left 'o']
```

Section 3: `Unfoldables`

If `Foldable` is used for iterating over some structure as if it were a list, `Unfoldable` does the opposite: it *builds* a structure from a list, or from an **unfolding** function.

```
class Unfoldable t where
    fromList :: [a] -> t a -- opposite of toList
    unfoldr :: (b -> Maybe (a, b)) -> b -> t a -- opposite of foldr
    {-# MINIMAL fromList | unfoldr #-}
```

While `fromList` is obvious, `unfoldr` is not so: it accepts a function—called the **unfolding** function—from `b` to `Maybe (a, b)` and an initial value `b`. It applies the function recursively so long as it returns `Just`, and stops on the first `Nothing`. Note that it can also never stop, resulting in an infinite structure. For example, Suppose we had an instance of `Unfoldable` for `[]` (which you will implement in the next section):

```
fromList [1, 2, 3] :: [Int]
[1,2,3]
-- An infinite list, but take stops it!
take 5 $ fromList [1..] :: [Int]
[1,2,3,4,5]
unfoldr (\ x -> if x > 5 then Just (x, x + 1) else Nothing) 1 :: [Int]
[]
unfoldr (\ x -> if x <= 5 then Just (x, x + 1) else Nothing) 1 :: [Int]
[1,2,3,4,5]
-- An infinite list, but take stops it!
take 5 $ unfoldr (\ x -> Just (x, x + 1)) 1 :: [Int]
[1,2,3,4,5]
```

3.1 Interchangeable implementations

Just like other `class`es we saw, `Unfoldable` can be implemented using either `fromList` or `unfoldr`. Implement `fromList` using `unfoldr` and `unfoldr` using `fromList`.

3.2 Unfoldable `instances`

Next, implement `Unfoldable` `instance`s for `[]`, `Deque`, and `PersistenArray`. You need to download the two files `PersistenArray.hs` and `Deque.hs` and place them in the same directory as `HW4.hs`. You may not modify these files; in fact, you shouldn't even submit them! Although their implementations are a bit different than the ones we saw in class, their API is identical. As these are abstract data types, their implementation shouldn't matter.
The implementaton of `Unfolable` should satisfy the following rules:

1. For `Deque`, a series of `popl`s should return the elements in the order of the original list.
2. For `PersistentArray`, the element at index `i` in the array should be the element at index `i` in the original list.
3. For `unfoldr`, the same holds: for `Deque` a series of `popl`s should in the order that the unfolding function returned, and for `PersistenArray` the element at index `i` should be the i^{th} element returned by the unfolding function.

Example output:

```
dequeFromList :: Deque Int
dequeFromList = fromList [1, 2, 3]

dq1 :: (Int, Deque Int)
dq1 = fromJust $ DQ.popl dequeFromList
dq2 = fromJust $ DQ.popl $ snd dq1
dq3 = fromJust $ DQ.popl $ snd dq2
map fst [dq1, dq2, dq3]
[1,2,3]

arrayFromList :: PersistentArray Int
arrayFromList = fromList [1, 2, 3]
[PA.lookup 0 arrayFromList, PA.lookup 1 arrayFromList, PA.lookup 2 arrayFromList]
[Just 1,Just 2,Just 3]
```

Tip: Remember, you only need to implement `fromList` *or* `unfoldr`!

Section 4: More data structure instances

In this section you will implement `Foldable`, `Semigroup`, `Monoid`, and `Functor` **`instance`**s for both `Deque` and `PersistentArray`.

Implementation rules:

- For `Deque`:
  1. Folding should be done from **left to right**.
  2. The instance of a `Semigroup` should maintain the order of the elements in the deque.
- For `PersistenArray`:
  1. Folding should be done from the lowest to the highest index.
  2. The instance of a `Semigroup` should maintain the order of the elements in the array.
  3. `Monoid` and `Functor` should satisfy the regular laws with regard to `mepmty <> a` and `fmap id === id`

Examples:

```
dq1 = DQ.pushl 1 $ DQ.pushr 2 DQ.empty
dq2 = DQ.pushl 3 $ DQ.pushr 4 DQ.empty
toList $ dq1 <> dq2
[1,2,3,4]

array1 = PA.pushr 2 $ PA.pushr 1 $ PA.empty
array2 = PA.pushr 4 $ PA.pushr 3 $ PA.empty
toList $ array1 <> array2
[1,2,3,4]
```

Hints:

- You can use one **`instance`** to help in the implementation of other **`instance`**s.
- Use the previous sections **`class`**es and functions.

Section 5: Bonus: `Ziplists` (10 points)

Implement a `Semigroup` and `Monoid` **`instance`** for `ZipList`, such that `(<>)` is applied as a dot product.

```
map getSum $ getZipList $ ZipList (map Sum [1, 2, 3]) <> ZipList (map Sum [4, 5])
[5,7]
take 5 $ map getProduct $ getZipList $
    ZipList (map Product [1..]) <> ZipList (map Product [0..])
[0,2,6,12,20]
```

Note: Remember the `Monoid` laws: `mempty <> a == a` and `a <> mempty == a` for all `a` (including infinite lists)! What would be the correct implementation of `mempty` for `ZipList`s?