Homework Assignment 5

Functional and Logic Programming, 2024

Due date: Thursday, June 20th, 2024 (20/06/2024)

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 `HW5_<id1>_<id2>.zip` where `<id1>` and `<id2>` are the submitters IDs.
  - Or `HW5_<id>.zip` if submitting alone.
  - You do not need special permission to submit alone.
- The zip file should contain **two (2!), top-level files** (no folders!) named `HW5.hs` and `Deque.hs`!
  - The contents of these files will be explained later.
- 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 HW5.hs Deque.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**.
- If you don't know how to implement some function, **do not remove it**! Use `undefined` in the implementation, otherwise your entire submission will fail compilation, which will also result in a 0 grade.
  - This is especially true to for the bonus section!

General notes

- The instructions for this exercise are split between this file and `HW5.hs`. This file offers a more high-level overview of the exercise, as well as offering a few hints. The Haskell file details all the required functionality for this assignment.
- You may not modify the **`import`** statements at the top of the file.
  - N.B. HLS has the habit of adding **unnecessary imports** when it doesn't recognize an identifier, **so please double-check this before submitting**!
  - If you are unsure what some function does, you can either ask HLS or Hoogle.
  - Hoogle also supports module lookups, e.g., `Prelude.not`.

- Do be aware however, that some functions from the standard library are more general than what we have learned so far in class.
  - And in some cases their definition may not be entire clear just yet!

- The exercises and sections are defined in a linear fashion. 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! Many of them are simple one-liners, and were designed to aid you in solving the more complex functions.
  - 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. 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!

- If possible, please ask your questions first in Piazza, so all students can take part in the discussion.

Special instructions

- Unfortunately, we have again not yet covered all the material necessary to implement all the sections in this assignment. In particular, not all lectures have seen `IO` yet, which is used in the last exercise. We will cover `IO` in the next Lecture.

Section 1: Fold maps

In this section, we will exercise our `foldMap` abilities by re-implementing basic functionality using just `foldMap`. To help guide you, and ensure you're using just `foldMap`, instead of using `foldMap` directly, we have defined a new data type, which `foldMap` will use[1]:

```
data FoldMapFunc a m result = FoldMapFunc {agg :: a -> m, finalize :: m -> result}
```

Where `a` is the type of the `Foldable` elements, `m` is an intermediary type, and `result` is the final result. The first function `agg` converts to the intermediary type `m` which will be used directly in `foldMap`, and the `finalizer` converts the intermediary type to the final result type. This data type will be used thus:

```
foldMap' :: (Foldable t, Monoid m) => FoldMapFunc a m result -> t a -> result
foldMap' FoldMapFunc{agg, finalize} = finalize . foldMap agg
```

Below are a couple of examples

```
fmproduct :: Num a => FoldMapFunc a (Product a) a
fmproduct = FoldMapFunc Product getProduct

foldMap' fmproduct [1, 2, 3]
6


fmand :: FoldMapFunc Bool All Bool
fmand = FoldMapFunc All getAll

foldMap' fmand [True, False, True]
False
foldMap' fmand [True, True, True]
True
```

To avoid *over*-guidance, the intermediate type is left for you to decide, so if those functions would appear in the code file, they would appear with a **wildcard** '_', which you would need to fill in:

```
fmproduct :: Num a => FoldMapFunc a _ a -- Fill this in with Product!
fmand :: FoldMapFunc Bool _ Bool -- Fill this in with All!
```

Below are example usages of the functions you will need to implement:

```
aux = (`foldMap'` [1, 2, 3])
aux fmsum
6
-- If the Foldable elements are already a monoid, just fold it!
foldMap' fmfold $ map show [1, 2, 3]
"123"
aux $ fmelem 2
True
aux $ fmfind (> 2)
Just 3
aux $ fmfind (> 3)
Nothing
aux fmlength
3
aux fmnull
False
foldMap' fmnull []
True
```

[1] As you might expect, this is *not* the canonical way of interacting with `foldMap`! It's just a good template for the exercise.

```
aux fmmaximum
Just 3
aux fmminimum
Just 1
foldMap' (fmmaxBy length) ["foo", "bar", "bazz"]
Just "bazz"
-- Notably, (a, b) implement Ord if a and b do, using lexicographical order!
foldMap' (fmminBy $ \ x -> (length x, x)) ["foo", "bar", "bazz"]
-- NonEmpty is also a foldable!
-- One of the most useful abilities of any fold is that it can be turned into a
    list!
-- This can also helps you implement other instances, in the homework or tests!
foldMap' fmtoList $ 1 :| [2, 3]
[1,2,3]
```

It's very important to note here that all these functions (and more!) are available for all `Foldable`s. A more comprehensive list will be available in the cheat-sheet attached to the exam.

Section 2: `instances`

In this section we will define **`instance`**s for the `Deque` data structure shown in class. In particular, we will implement **`instance`**s both internally—or within the module—and externally—or outside the module.

1. A file `Deque.hs` is provided, which contains a basic definition of the data structure. You should implement the required **`instance`**s in this file directly. You *should not* modify export list of `Deque.hs`, and definitely shouldn't export the `Deque` constructor!

2. For external **`instance`**s, we need to wrap the `Deque` in a **`newtype`**, and define the **`instance`** for this **`newtype`**.[2] When implementing **`instance`**s externally for an abstract data type, we can only its external functions, i.e., `empty`, `pushr`, `pushl`, `popr`, and `popl`. As well as functions provided by the type class instances implemented for the type (e.g., `fmap`). Note that while it is possible to implement using a simple reduction to the existing implementations in `Deque.hs`, this wouldn't be a very good exercise.[3]

Regardless of whether the **`instance`** is internal or external, its behavior should be the same. In particular, note that the `liftA2`/`(<*>)` and `(>>=)` implementations should use Cartesian product. Usage examples:

```
q1 = pushl 1 $ pushl 2 $ pushr 3 empty
q2 = pushr 30 $ pushr 20 $ pushl 10 empty

q1 <> q2
[1,2,3,10,20,30]
empty == mempty
True
foldMap show q1
"123"
sum q2 -- Works since sum works on all Foldables
60
fmap show q1
["1","2","3"]
liftA2 (+) q1 q2 -- The order is important!
[11,21,31,12,22,32,13,23,33]
q1 >>= \x -> pushl (x * 10) $ pushr (x + 1) empty
[10,2,20,3,30,4]
```

[2]As discussed in class, this is done to avoid orphan **`instance`**s, which are generally considered bad practice.
[3]And if this kind of questions were to appear in the exam, you wouldn't be able to it either!

Section 3: Error handling and `traverse`

In this section we will learn how we can use `Appliactive` in combination with `traverse` to handle different error strategies. Given a list of expressions `[Expr]`, we wish to evaluate them, but we may encounter two types of errors as discussed in class:

1. Division by zero.
2. Missing variables.

There are multiple ways we can deal with these errors, such as:

1. Just apply `map` to the list of expressions with whatever error you want, e.g., getting a `[Maybe Int]` or `[Either Err Int]`, etc.
   - Been there, done that
   - While it should be noted this is actually possible to achieve this using `traverse`—since it's possible to implement `map` using `traverse`—it requires techniques we havent' seen yet in class.
2. Fail on the first error, be it `Maybe` or `Either Err`.
3. Use an error handler for default values.

Luckily, this is all possible to achieve using just `traverse`!

First, define a new **`class`** for handling specific errors:

```
class Monad f => CalculatorError f where
  divideByZero :: f Int
  missingVariable :: String -> f Int
```

Then, implement the following function:

```
data Expr
  = Val Int
  | Var String
  | Add Expr Expr
  | Sub Expr Expr
  | Mul Expr Expr
  | Div Expr Expr
  deriving (Show, Eq)
runCalculator :: CalculatorError f => Map String Int -> Expr -> f Int
```

Lastly, implement the following **`instance`**s:

```
instance CalculatorError Maybe where
data Err = DivByZero | MissingVar String deriving (Show, Eq)
instance CalculatorError (Either Err) where
data Defaults
  = Defaults
  { defaultForDivisionByZero :: Int
  , defaultForVariable :: String -> Int
  }
instance CalculatorError (Reader Defaults)
-- This one aggregates all the failures
```

Now, `traverse` automagically works out of the box! Example definitions:

```
variables = M.fromList [("x", 10), ("y", 20)]
exprs1 =
  [Val 1, Add (Var "x") (Val 2), Div (Val 30) (Var "y"), Mul (Var "x") (Var "y")]
exprs2 = [Val 1, Add (Var "z") (Val 2), Div (Var "y") (Val 0)]
go :: CalculatorError f => [Expr] -> f [Int]
go = traverse $ runCalculator variables
```

Example uses:

```
go exprs1 :: Maybe [Int]
Just [1,12,1,200]
go exprs2 :: Maybe [Int]
Nothing

go exprs1 :: Either Err [Int]
Right [1,12,1,200]
go exprs2 :: Either Err [Int]
Left (MissingVar "z")

defaults =
  Defaults
    { defaultForDivisionByZero = 0
    , defaultForVariable = sum . map ord
    }
runReader (go exprs1) defaults
[1,12,1,200]
runReader (go exprs2) defaults
[1,124,0]
```

Section 4: Hangman game

In this section you will implement a Hangman game, interacting with the user using the `IO` monad. The entry point is the function `hangman :: String -> IO Score`, which accepts a word, and returns the score of the game. The score of the game is the number of guesses it took to fill in the word—a lower score is better—minus the **optimal** number of guesses. For example, in the word `haskell`, the minimum number of guess is 6, since there are 6 unique letters in the word. Below is an example usage of the function from `GHCi` (user input is in green):

```
*HW5> hangman "Haskell Curry"
_______ _____
Guess a letter: a
_a_____ _____
Guess a letter: b
Wrong guess!
_a_____ _____
Try again: h
Ha_____ _____
Guess a letter: s
Has____ _____
Guess a letter: l
Has__ll _____
Guess a letter: y
Has__ll ____y
Guess a letter: i
Wrong guess!
Has__ll ____y
Try again: c
Has__ll C___y
Guess a letter: u
Has__ll Cu__y
Guess a letter: r
Has__ll Curry
guess a letter: q
Wrong guess!
Has__ll Curry
Try again: b
Has__ll Curry
Wrong guess!
Try again: &
Invalid letter guess &!
Has__ll Curry
Try again: k
Hask_ll Curry
Guess a letter: k
Hask_ll Curry
Guess a letter: e
Very good, the word is:
Haskell Curry
4
```

The last line is the final output of the function, so it shouldn't actually be printed to the user.

- In the above example, `GHCi` prints the final output of any function, including `IO` ones!

Guidelines:

- You can assume the input is valid, i.e., it contains only letters and spaces, and does end or start with a space.
- Since the testing is **automatic**, please make sure your output text is **exactly** as shown above!
- Use `getChar :: IO Char` to read a single character from the user.
- Use `putStr "something"` to print something to the user.
- Use `putStrLn "something"` to print something to the user, followed by a newline.

- In particular, you can use `putStrLn ""` to print a newline.

- If that character is anything other than a letter, ignore it.
  - It does not count as a guess.
  - While the word can contain spaces, these should not be guessable, but are shown to the user.
  - For the bonus section, you should also support `'?'`
- While the word itself is case-sensitive, the guesses are case-**in**sensitive.
  - In other words, the word should be displayed to the user in its original case, but the guessed characters should be treated as lower case only.
- Guessing the same wrong letter multiple times *should* count as a guess.
- Guessing a *correct* letter multiple times should *not* count as a guess.
- Once the user has guessed all the letters in the word, the game should end

Section 4: Bonus (10 points)

If the user enters `'?'` as a guess, the game should display to the user the list of which have not been guessed yet, in alphabetical order.

- Both letters which were guessed correctly and incorrectly *not* should be displayed.
- This should *not* count as a guess.

For example:

```
*HW5> hangman "Cat"
___
Guess a letter: ?
Remaining letters: [abcdefghijklmnopqrstuvwxyz]
___
Guess a letter: c
C__
Guess a letter: b
Wrong guess!
C__
Guess a letter: a
Ca_
Guess a letter: a
Ca_
Guess a letter: ?
Remaining letters: [defghijklmnopqrstuvwxyz]
Ca_
Guess a letter: t
Very good, the word is:
Cat
1
```