Homework Assignment 4

Functional and Logic Programming, 2024

Due date: Thursday, June 6h, 2024 (06/06/2024)

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 HW4_<id1>_<id2>.zip where <id1> and <id2> are the submitters IDs.
  - Or HW4_<id>.zip if submitting alone.
  - You do not need special permission to submit alone.
- The zip file should contain **three (3!), top-level files** (no folders!) named `HW4.hs`, `EqSet.hs`, and `EqMap.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 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**.
- 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 `HW4.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`** statement at the top of the file, nor add new **`import`**s.
  - 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 not yet covered all the material necessary to implement all the sections in this assignment. In particular, we have just began `Semigroups` in the lectures, and have not yet taught the `Arg` data type. We will complete `Semigroups` and `Monoid` in the next lecture, and `Arg` in the next tutorial. Importantly, most of the assignment does not require these concepts, so you can start working on the assignment without those concepts.
  - To make things easier, we have marked sections requiring `Semigroup` and `Monoid` with † and sections requiring `Arg` with ‡.
  - If you are truly eager to get started, then you can just read up on `Arg`, as it's not a very complex concept.
- In questions where you can assume the input is valid, it's fine to use `error` or `undefined`. This isn't very good design, but it's fine for an assignment.
- For this exercise, you can assume all lists are finite.

Section 1: Abstract data types: `EqSet` and `EqMap`

In this section, you will implement two **abstract** data types: `EqSet` and `EqMap`. These implementations have very poor performance characteristics—most operations are linear at best, quadratic at worst—but they are useful for understanding the basics of abstract data types (as well having the dubious benefit of being constrained on `Eq` instead of `Ord`...). Since these are **abstract** data types, they reside in their own modules, aptly named `EqSet` and `EqMap`. While you can add helper functions to these modules, including exported ones, you should of course never export the constructors of the data types themselves!

`EqSet`

- `EqSet` is a very basic implementation of a set using a list.
- All operations should run in $O(n)$ time, with the exception of `(==)` and `(<>)` which can take $O(nm)$, where n and m are the sizes of the two sets.
- As expected from a `Set`, `insert`ing an element which already exists should return the same, or equivalent[1], set, and likewise for `remove` on an element which does not exist.
- `elems` returns all the values in the set, in no particular order.
- Lastly, you should implement the following basic **`instance`**s:
  1. `Eq`, such that two sets are equal if they contain the same elements.
     - Of course, ignoring the order of the underlying list
  2. The `Show` should `show` the elements of a set in any order, using {} as wrappers, e.g.:

```
insert 1 $ insert 2 empty
-- Not necessarily in this order:
{1,2}
```

  3. `Semigroup`† should be equivalent to the union (∪) of two sets.

```
set1 = insert 1 $ insert 2 empty
set2 = insert 3 $ insert 2 empty
set1 <> set2 -- Again, not necessarily in this order...
{1,2,3}
```

     - From this definition, you should be able to infer the correct implementation for `Monoid`†.

[1]By equivalent, we of course mean that the two sets are `(==)` to each other

`EqMap`‡

- An `EqMap` can be implemented on top of an `EqSet` by storing key-value pairs in the set.
  - Hint‡: Using `Arg` instead of a basic tuple here can be very useful!
  - Like sets, all operations are expected to run in $O(n)$ time, with the exception of `(==)` and `(<>)` which can take $O(nm)$, where n and m are the sizes of the two maps.
  - The keys themselves form a logical set, so inserting a value for an existing key **overrides** the old value, and removing a key which does not exist should return the same, or equivalent, map.
  - Like `EqSet`, you should also implement `Eq`, `Show`, `Semigroup`†, and `Monoid`† for `EqMap`.
    - Two `EqMap`s are equal if they contain the same key-value pairs, regardless of order.
    - `Show` should `show` the key-value pairs in any order, with {} wrappers and → between keys and values:

```
insert 1 'a' $ insert 2 'b' empty
-- Again, not necessarily in this order...
{1->'a',2->'b'}
```

    - `Semigroup`† should be equivalent to the union of two maps, with the **second** (or **right**) map **overriding** the first in case of a key conflict.

```
map1 = insert 1 'a' $ insert 2 'b' empty
map2 = insert 2 'c' $ insert 3 'd' empty
map1 <> map2
-- Again, not necessarily in this order...
{1->'a',2->'c',3->'d'}
```

    - And again, from this definition, you should be able to infer the correct implementation for `Monoid`†.
  - Lastly, you should implement a special kind of `Semigroup` for `EqMap`, where the values are combined using `(<>)` (combining the left value with the right value):

```
map1 = insert 1 "a" $ insert 2 "b" empty
map2 = insert 2 "c" $ insert 3 "d" empty
getCombiningMap $ CombiningMap map1 <> CombiningMap map2
-- Again, not necessarily in this order, but the value of 2 should be "
    bc"!
{1->"a",2->"bc",3->"d"}
```

Section 2: The `Serialize` type class

In this section, we will generalize the `serialize` and `deserialize` functions from the last assignment to a type class.

- Just like the last assignment, the only requirement is for `deserialize . serialize` ≡ `id`.
  - And you can assume the input to `deserialize` is a valid output of `serialize` for the same type.
- Just like the last assignment, you may not assume anything of the values you are serializing, other than the constraints of the `instance` itself.
- As shown in class, it is a good idea to use simpler `instance`s to build more complex ones.
- For the two abstract data types, the important thing is for the initial value and the value computed using `serialize . deserialize` are both `(==)` to each other, not that they necessarily have the same underlying data!
- Hint: you can use `ord` and `chr` to convert between `Char` and `Int` and vice-versa.

Section 3: `Metric spaces`

In this section, we will implement a type class for measuring the distance between two values. The `Metric` type class is defined as follows:

```
class Metric a where
  distance :: a -> a -> Double
```

`Metric` is an implementation of a metric space, meaning the following laws are expected to hold:

- a distance is always non-negative: `distance x y >= 0` for all `x` and `y`, i.e., **positivity**.
  - It can be ∞ for some cases, which are explicitly detailed.
  - We have predefined `infinity = 1 / 0` for you in the Haskell file.
- `distance x y == 0` if and only if `x == y`.
- `distance x y == distance y x` for all `x` and `y`, i.e., **symmetry**.
- `distance x z <= distance x y + distance y z` for all `x`, `y`, and `z`, i.e., **triangle inequality**.

Guidance:

- For numbers (`Int` and `Double` in our case), the distance is the absolute difference.
  - You can use `fromIntegral` to convert an `Int` to a `Double`
- For `char`, use `ord` to convert to `Int` and then compute the distance.
- For tuples and lists, we can define the distance using either the Euclidean distance or the Manhattan distance.

```
distance (0, 1), (5, 13)
13.0
distance (ManhattanTuple 0 1) (ManhattanTuple 5 13)
17.0
distance [0, 1, 2] [2, 4, 8]
7.0
distance (ManhattanList [0, 1, 2]) (ManhattanList [2, 4, 8])
11.0
```

There are also four functions—actually two variant pairs of `elem` and `sort`—you should implement.

1. `closest` is a generalization of `elem` in a metric space, returning the closest element to the input, or `Nothing` if there is no element whose distance is less than ∞.

```
closest 0.75 [0.0,1.0,2.0,3.0]
Just 1.0
closest Nothing [Just 42, Just 0]
Nothing
closestOn (uncurry ManhattanTuple) (5, 5) [(3, 3), (4.6, 4.5), (6, 6), (4, 5)
    ]
Just (4.6, 4.5)
```

   - Of course, you can implement `closest` and `closestOn` in terms of one another. You're free to choose which one you'd prefer to implement "for real".
   - Implementing `closest` in terms of `cloestOn` is trivial, but implementing `closestOn` on its own is more complicated than implementing `closest`!
   - ‡ Using `Arg` can be useful here as well, as well as defining an **`instance`** for `Arg` (like `Eq` and `Ord`, you would of course compute the distance on the first element).

2. `metricBubbleSort` is a special kind of bubble sort where we only swap between `x` and `y` if `x < y` *and* their distance is at least `d`.

```
metricBubbleSort 1.5 [4, 2, 3, 1] :: [Int]
[2,1,4,3]
```

- All the notes from `closest` vs. `closestOn` apply to `metricBubbleSort` and `metricBubbleSortOn` as well!

Bonus (10 points): `clusters`

Implement a function `clusters` which takes a list of values, and clusters them together such that all values in a cluster have a finite distance between them.

```
clusters [Just [1,2], Nothing Just [1,2,3], Just [4,5], Just [4,5,6], Nothing]
-- The order of the clusters or the order of the elements inside is not important.
[[Just [1,2],Just [4,5]],[Just [1,2,3],Just [4,5,6]], [Nothing, Nothing]]
```