Functional Programming for Logicians
Homework 7

Péter Mekis
Department of Logic, ELTE Budapest

Deadline: 2019 April 1 17:59 pm

- Solve *four* of the fifteen exercises below. Some of them are follow-ups to others; it may be a good idea to choose them together.
- Create *a miniproject* of your own, including
  - the definition of a datatype (preferably recursive) that hasn't been discussed in the course yet,
  - the instantiation of a few type classes,
  - a few function definitions.

1. In the session we defined the `List` type, deriving the `Show` class. This way lists were printed in a not so readable fashion. Instantiate `Show` with an explicite definition so that '`Colon 1 (Colon 2 (Colon 3 Empty))`' will be printed as '<1,2,3>' (to avoid confusion with the built-in list type).

2. We also derived `Ord` for the `List` type. The derived default ordering uses the following definition:

   ```
   x < Empty         = False
   Empty < y         = True
   (Colon x xs) < (Colon y ys)
     | x < y         = True
     | otherwise     = xs < ys
   ```

   This definition makes
   `Colon True (Colon False Empty) < Colon True (Colon True Empty)` true, and
   `Colon True (Colon True Empty) < Colon True (Colon False (Colon False Empty))` false. Define a function that generates an infinite descending chain of lists of Booleans, witnessing that the derived ordering of `List a` is not a well-ordering even if the ordering of `a` is a well-ordering. (In an infinite descending chain, every element is greater than its successor.)

3. Instantiate `Ord` for `List a` so that the defined ordering of lists will be a well-ordering whenever the ordering of `a` is a well-ordering.

4. Instantiate the `Foldable` class for the `Tree` type defined in the session.

5. In homework 6, exercise 18 the task was to define a version of the Tree type with two parameters, so that the type of the data at the nodes might be different from the type of the data at the leaves. Define that type if you haven't done yet. Look up the Bifunctor class in the Haskell documentation, and make this new Tree type an instance of it.

6. Instantiate the `Foldable` class for the version of `Tree` with two parameters fromm the previous exercise.

7. Instantiate the `Foldable` class for the `HunMaybe` type introduced in the session.

8. Another frequently used type of Haskell somewhat similar to '`Maybe a`' is '`Either a b`'. The documentation says:

   "The Either type represents values with two possibilities: a value of type `Either a b` is either `Left a` or `Right b`.

   The `Either` type is sometimes used to represent a value which is either correct or an error; by convention, the `Left` constructor is used to hold an error value and the `Right` constructor is used to hold a correct value (mnemonic: "right" also means "correct")."

   Define your own version of the type, just as we did with `Maybe`, `Bool`, and `List`, deriving from `Eq`, `Show`, and `Ord`. Try to find a meaningful way to use the type based on the explanation above, and define a few functions that exploit its potential.

9. Does it make sense to make your version of `Either` an instance of `Functor`? How about `Bifunctor` (see exercise 5)? (Resist the temptation to look up the answers in the documentation of `Either`.)

10. Does it make sense to make your version of `Either` an instance of `Foldable`? (Resist the temptation to look up the answer in the documentation of `Either`.)

11. Consider the following definition:

    ```
    data Set a = Set [a]
    ```

    The `Set` constructor creates a set from a list. Instantiate `Eq` so that two sets will be the same if and only if their elements are the same, with no regard to their order and the numbers of their occurrences in the lists from which they are constructed. Instantiate `Show` so that '`Set [1,2,7,7,2]`' will be shown as '{`1,2,7`}' (the order of the elements might be different).

12. Define the following functions for the `Set` type: `element`, `elementlist` (not just an accessor function that returns the list from which a set is constructed, it should remove duplicates), `boolUnion`, and `boolIntersection`.

13. Instantiate the `Functor` class for the `Set` type defined above.

14. Why is the instantiation of the `Foldable` class for the `Set` type problematic? Can you find a solution to this problem?

15. Finally a more advanced exercise: The Zermelo–Fraenkel style concept of pure sets used in set theory is different from the above one. In this approach every set belongs to the same type. This set concept is not recursive, but it has a recursive core: finite sets are the result of iterated application of the pair, unary union, and power set operations to the empty set. This core can be implemented by the following type definition:

    ```
    data PureSet  = EmptySet
                  | Pair PureSet PureSet
                  | Union PureSet
                  | Power PureSet
    ```

    Find a way to define equality and elementhood for this type in accordance with the set-theoretical meaning of *empty set*, *pair*, *union*, and *power set*.