Monad-Based Programming SS 2019

Assignment 6

Deadline for solutions: 24.07.2019, **12.15 a.m.**

Exercise 1 Make Trees Foldable (Again) (10 Points)

The following code implements a breadth-first traversal of a tree, using the formalization of trees from `Data.Tree`:

```
import Data.List
import Data.Monoid
import Data.Tree (Tree(..))

newtype BFSTree a = BFS (Tree a)

instance Foldable (BFSTree) where
  foldMap f (BFS tr) = go [tr]
    where
      go q = case q of
        [] -> mempty
        (Node x xs):qs -> f x `mappend` go (qs ++ xs)
```

Hence one can run programs like

```
foldr (:) [] (BFS some_tree)
```

to obtain a breadth-first unfolding of a tree `some_tree` into a list, instead of the default depth-first unfolding by

```
foldr (:) [] some_tree
```

This implementation is based on using the list type `[BFSTree]` as a *queue* in which new trees are added at the back with the `qs ++ xs` command, which is highly inefficient, because it requires full traversal of the pending queue `qs` at every iteration.

(a) Introduce a monad class `(QMonad s)` supporting the following operations

```
empty :: m b
pop_front :: m s
push_back :: s -> m b
```

for initializing the background queue, for popping elements (of type `a`) in front of the queue, and for pushing elements at the back of the queue (this requires the *{-## LANGUAGE FunctionalDependencies ##-}* extension). Implement a function

```
foldMapM :: (Monoid m, QMonad s q) => (s -> m) -> (s -> [s]) -> q m
```

working analogously to the above `foldMap`, but using the above operations `empty`, `pop_front`, `push_back` for working with the queue. The first argument of `foldMapM` is a map extracting a root label of a tree (or some more general data structure) and interpreting it in a monoid, and second argument is a function that yields a list of immediate subtrees of a given tree.

(b) Implement a concrete instance of `(QMonad s)` based on the above breadth-first traversal example using `[s]` as the underlying storage (like in the state monad).

(c) Implement another instance of `(QMonad s)` using a pair of lists to simulate a queue. Compare the performance of both implementations by running tests on exponentially growing trees, e.g.

```
expTree a b = Node (a, b) [expTree (a + 1) b, expTree a (b + 1)]
```

(d) Implement a CPS-transform of the monad from (c) and make it into a yet another instance of `(QMonad s)`.