assignment02

May 7, 2019

1 Functional Programming SS19

2 Assignment 02 Solutions

2.1 Exercise 1

In this exercise we consider priority queues storing arbitrary data.

2.1.1 1a)

Give a definition for a data type `PriorityQueue` for priority queues storing data of arbitrary type. The type `PriorityQueue` should have the two constructors: `Push`, for inserting an element with a priority of type `Int` into the priority queue, and `EmptyQueue`, for the empty queue. Furthermore, define a variable `p` of type `PriorityQueue Int` reflecting the queue as shown in Fig. 1.

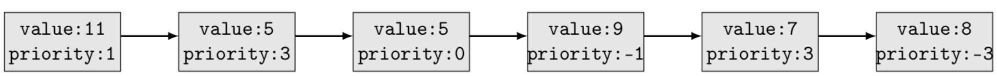

Figure 1: Priority queue storing values of type `Int`.

priority_queue

```
In [1]: data PriorityQueue a = EmptyQueue | Push a Int (PriorityQueue a) deriving Show
```

```
In [2]: p :: PriorityQueue Int
        p = Push 11 1 (Push 5 3 (Push 5 0 (Push 9 (-1) (Push 7 3 (Push 8 (-3) EmptyQueue)))))
```

```
In [3]: p
```

```
Push 11 1 (Push 5 3 (Push 5 0 (Push 9 (-1) (Push 7 3 (Push 8 (-3) EmptyQueue)))))
```

2.1.2 1b)

Write a function `isWaiting` that gets an element of type `a` and a queue of type `PriorityQueue a` and returns `True` if and only if the element is stored in the queue. For example, `isWaiting 5 p == True` and `isWaiting 6 p == False`. The function should be applicable for as many types `a` as possible.

Hints: Remember that the predefined type class `Eq` contains all types providing the equality operator ==.

```
In [4]: {- |
          Returns True if an element m is contained
          in a given priority queue; returns False otherwise.
        -}
        isWaiting :: Eq a => a -> PriorityQueue a -> Bool
        isWaiting m EmptyQueue = False
        isWaiting m (Push x _ q) | m == x = True
                                 | otherwise = isWaiting m q
```

Test cases

```
In [5]: isWaiting 5 p
```

```
True
```

```
In [6]: isWaiting 6 p
```

```
False
```

2.1.3 1c)

Write a function `fromList` that given a list of type `[(a,Int)]` computes a priority queue of type `PriorityQueue a` such that for every element `(x,n)` of the list the resulting queue contains the element `x` with priority `n`. For example, `fromList [(11,1), (5,3), (5,0), (9,-1), (7,3), (8,-3)]` should yield an expression of type `PriorityQueue Int` as in Fig. 1.

```
In [7]: {- |
          Creates a PriorityQueue instance from a list
          of (x, p) tuples, where x is an element and
          p is its priority.
        -}
        fromList :: [(a,Int)] -> PriorityQueue a
        fromList [] = EmptyQueue
        fromList ((x,p):xs) = Push x p (fromList xs)
```

Test cases

```
In [8]: fromList [(11,1), (5,3), (5,0), (9,-1), (7,3), (8,-3)]
```

```
Push 11 1 (Push 5 3 (Push 5 0 (Push 9 (-1) (Push 7 3 (Push 8 (-3) EmptyQueue)))))
```

```
In [9]: fromList []
```

```
EmptyQueue
```

```
In [10]: fromList [(1,1), (2,2), (3,3)]
```

```
Push 1 1 (Push 2 2 (Push 3 3 EmptyQueue))
```

2.1.4 1d)

Write a function `pop` that given a nonempty priority queue `queue` returns a pair `(x,q)` where the first entry `x` is the value with highest priority in `queue` and the second entry `q` is `queue` where the element with value `x` and the highest priority is deleted. For example `pop p` should yield a pair `(5,q)`, where `q::PriorityQueue Int` corresponds to the queue given in Fig. 2.

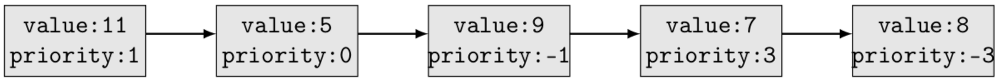

Figure 2: Reduced priority queue from Fig. 1.

reduced_priority_queue

Hints:

- *If `x` occurs several times with the highest priority in queue, then only one of the corresponding elements should be deleted in `q`.*
- *If `queue` contains several values with the highest priority, then your implementation should choose one of them. So `pop p` could also yield `(7,q')`, where `q'` results from `p` by deleting the element with value `7` and priority `3`.*
- *You may need `minBound :: Int`, the smallest `Int`, and `max :: Int -> Int -> Int`.*

```
In [11]: -- Note: A potential limitation of the current solution is that
         -- the queue is assumed to only store bounded numeric values

         getmaxP :: Num a => Bounded a => PriorityQueue a -> Int -> (a, Int, Int)
         getmaxP EmptyQueue _ = (-1, minBound, -1)
         getmaxP (Push x p q) currIdx = (newX, newP, newIdx)
                                 where (maxX, maxP, maxIdx) = getmaxP q (currIdx+1)
                                       newP = max p maxP
                                       newIdx | newMaxPFound = currIdx
                                              | otherwise = maxIdx
                                       newX | newMaxPFound = x
                                            | otherwise = maxX
                                       newMaxPFound = p >= maxP


         {- |
           Returns a priority queue with the element at a given index
           removed from the given queue.
         -}
         rmAt :: PriorityQueue a -> Int -> Int -> PriorityQueue a
         rmAt EmptyQueue _ _ = EmptyQueue
         rmAt (Push x p q) i currIdx | i == currIdx = rmAt q i (currIdx+1)
                                     | otherwise = Push x p (rmAt q i (currIdx+1))

         {- |
           Removes a tuple of the form (x, q), where x is the element
           with highest priority in the given queue and q is the given
```

```
  queue without the element with highest priority.
-}
pop :: Num a => Bounded a => PriorityQueue a -> (a, PriorityQueue a)
pop EmptyQueue = (minBound, EmptyQueue)
pop q = (maxX, updatedQ)
        where (maxX, _, maxIdx) = getmaxP q 0
              updatedQ = rmAt q maxIdx 0
```

Test cases

```
In [12]: (x, newP) = pop p
```

```
In [13]: x
```

```
5
```

```
In [14]: newP
```

```
Push 11 1 (Push 5 0 (Push 9 (-1) (Push 7 3 (Push 8 (-3) EmptyQueue))))
```

2.1.5 1e)

Write a function `toList` that given a `PriorityQueue` returns a list containing the values from the queue sorted in decreasing priority. For example, `toList p == [5,7,11,5,9,8]`.

```
In [15]: {- |
           Returns the elements in a PriorityQueue
           in a list format, such that the list entries
           are obtained by sorting the elements'
           priorities in descending order.
         -}
         toList :: Num a => Bounded a => PriorityQueue a -> [a]
         toList EmptyQueue = []
         toList q = x : toList reducedQ
                   where (x, reducedQ) = pop q
```

Test cases

```
In [16]: -- the example from the exercise description
         toList p
```

```
[5,7,11,5,9,8]
```

2.2 Exercise 2

2.2.1 2a)

Consider the type `List a` from the lecture that is defined as follows:

```
data List a = Nil | Cons a (List a) deriving Show
```

Declare `List a` as an instance of the type class `Eq` whenever `a` is an instance of `Eq`. Implement the method (==) such that it computes equality between lists entrywise.

```
In [17]: data List a = Nil | Cons a (List a) deriving Show
```

```
In [18]: instance Eq a => Eq (List a) where
             (==) Nil Nil = True
             (==) Nil _ = False
             (==) _ Nil = False
             (==) (Cons x l1) (Cons y l2) | x /= y = False
                                          | otherwise = l1 == l2
```

Test cases

```
In [19]: -- we verify the equality function with two equivalent lists
         x = Cons 1 (Cons 2 (Cons 3 (Nil)))
         y = Cons 1 (Cons 2 (Cons 3 (Nil)))
         x == y
```

```
True
```

```
In [20]: -- we verify the equality function with two different lists
         -- (same elements as above, but in different order)
         x = Cons 1 (Cons 2 (Cons 3 (Nil)))
         y = Cons 3 (Cons 2 (Cons 1 (Nil)))
         x == y
```

```
False
```

```
In [21]: -- another test with a different order
         -- of the list elements
         x = Cons 1 (Cons 2 (Cons 3 (Nil)))
         y = Cons 1 (Cons 3 (Cons 2 (Nil)))
         x == y
```

```
False
```

```
In [22]: -- we verify that two lists with
         -- different lenghts are not equal
         x = Cons 1 (Cons 2 (Cons 3 (Nil)))
         y = Cons 1 (Cons 2 (Nil))
         x == y
```

```
False
```

```
In [23]: -- we verify that two empty lists are equal
         x = Nil
         y = Nil
         x == y
```

```
True
```

2.2.2 2b)

*Give a declaration for a type class `Mono` for monoids as a subclass of `Eq` with the following methods: * `binOp :: a -> a -> a` (for "binary operation"). * `one :: a` (the neutral element of the monoid). * `pow :: Word -> a -> a` (for "power"). `Word` is the predefined type for nonnegative integers. Here, for an object `x` of type `a` and a number `n :: Word` the expression `pow n x` should stand for `binOp x (binOp x ...(binOp x one))` with `n` occurences of `binOp`. So `pow x 0 == one`.*

The class declaration should contain a default implementation for `pow`. In contrast, the functions `binOp` and `one` have to be implemented in the instances of the type class `Mono`.

```
In [24]: class Eq a => Mono a where
             binOp :: a -> a -> a
             one :: a

             pow :: Word -> a -> a
             pow 0 x = one
             pow n x = binOp x (pow (n-1) x)
```

2.2.3 2c)

Declare the built-in type `Integer` and `List a` from a) as instances of the type class `Mono` for as many types `a` as possible. For `Integer` the binary operation should be (*) and the neutral element is 1. For `List a` the binary operation is concatenation of the lists where the empty list is the neutral element. For example `binOp (-3) 2 = -6` and `binOp (Cons 'a' Nil) (Cons 'b' (Cons 'c' Nil)) = Cons 'a' (Cons 'b' (Cons 'c' Nil))`.

```
In [25]: instance Mono Integer where
             one = 1
             binOp x y = x * y
```

```
In [26]: instance Eq a => Mono (List a) where
             one = Nil

             binOp Nil xs = xs
             binOp xs Nil = xs
             binOp (Cons x xs) ys = Cons x (binOp xs ys)
```

Test cases

```
In [27]: binOp (-3) 2
```

```
-6
```

```
In [28]: binOp (Cons 'a' Nil) (Cons 'b' (Cons 'c' Nil))
```

```
Cons 'a' (Cons 'b' (Cons 'c' Nil))
```

```
In [29]: binOp (Cons 'b' (Cons 'c' Nil)) (Cons 'a' Nil)
```

```
Cons 'b' (Cons 'c' (Cons 'a' Nil))
```

2.2.4 2d)

For any monoid `a`, implement a function `multiply` that given a list of type `[(Word,a)]` multiplies the powers of the pairs from the list, i.e., `multiply [(x 1 ,y 1 ),(x 2 ,y 2 ),...,(x n ,y n )] == binOp (pow x 1 y 1 ) (binOp (pow x 2 y 2 ) (binOp ...(binOp (pow x n y n ) one)))`. The empty product `multiply []` is defined to be the neutral element of the monoid `a`.

For example `multiply [(3,Cons 'a' Nil),(1,Cons 'b' Nil),(2,Cons 'c' (Cons 'd' Nil))] == Cons 'a' (Cons 'a' (Cons 'a' (Cons 'b' (Cons 'c' (Cons 'd' (Cons 'c' (Cons 'd' Nil)))))))`.

```
In [30]: multiply :: Mono a => [(Word, a)] -> a
         multiply [] = one
         multiply ((x1, y1):xs) = binOp (pow x1 y1) (multiply xs)
```

Test cases

```
In [31]: -- the example from the exercise description
         multiply [(3,Cons 'a' Nil), (1,Cons 'b' Nil), (2,Cons 'c' (Cons 'd' Nil))]
```

```
Cons 'a' (Cons 'a' (Cons 'a' (Cons 'b' (Cons 'c' (Cons 'd' (Cons 'c' (Cons 'd' Nil)))))))
```

2.3 Exercise 3

*In this exercise you may **not** use any predefined functions except `map`, `foldr`, `filter`, `+`, constructors and comparisons. Also, you may **not** use explicit recursion.*

2.3.1 3a)

Implement a function `removeDuplicates :: Eq a => [a] -> [a]` that given a list of an instance of `Eq` computes a list that contains exactly the same elements as the input list but only with a single occurrence. For example `removeDuplicates [1,1,2,1,-1,1,1,2,3,1] == [1,2,-1,3]`.

```
In [32]: {- |
           Inserts an element x into a list xs
           only if x is not already in xs.
         -}
         insertUnique :: Eq a => a -> [a] -> [a]
         insertUnique x xs | isUnique  = x : xs
                           | otherwise = xs
                           where isUnique = (filter (==x) xs) == []

         {- |
           Given a list xs, returns a list xs' in which
           all duplicate elements have been removed.
         -}
         removeDuplicates :: Eq a => [a] -> [a]
         removeDuplicates [] = []
         removeDuplicates xs = foldr insertUnique [] xs
```

Test cases

```
In [33]: -- the example from the exercise description
         removeDuplicates [1,1,2,1,-1,1,1,2,3,1]
```

```
[-1,2,3,1]
```

```
In [34]: -- a simple test with a singleton list
         removeDuplicates [1]
```

```
[1]
```

```
In [35]: -- we check that a list with unique
         -- elements is left untouched
         removeDuplicates [1,2,3,4,5]
```

```
[1,2,3,4,5]
```

```
In [36]: -- another list with repeated elements
         removeDuplicates [1,1,1,2,2,2,3,3,3]
```

```
[1,2,3]
```

2.3.2 3b)

Implement a function `differentDigits :: Int -> Int` that counts the number of different digits occurring in the decimal representation of an integer. For example, `differentDigits 08052019 == 6` and `differentDigits 111231112111 == 3`.

Hints: The function `show :: Int -> String` converts an integer into its decimal string representation. You are allowed to use it in this subexercise.

```
In [37]: {- |
           Returns the number of unique digits
           in the decimal representation of a
           given integer. Uses the removeDuplicates
           function from 3a).
         -}
         differentDigits :: Int -> Int
         differentDigits x = foldr (+) 0 oneList
                             where oneList = map (\x -> 1) uniqueDigits
                                   uniqueDigits = removeDuplicates (show x)
```

Test cases

```
In [38]: -- the first example from the exercise description
         differentDigits 08052019
```

```
6
```

```
In [39]: -- the second example from the exercise description
         differentDigits 111231112111
```

```
3
```

```
In [40]: -- sanity check with a single-digit number
         differentDigits 0
```

```
1
```

```
In [41]: -- a number with unique digits
         differentDigits 12345
```

```
5
```

2.4 Exercise 4

Consider the following data type which represents univariate polynomials with arbitrary coefficients:

`data Polynomial a = Coeff a Int (Polynomial a)| Null deriving Show`

With `Null` we denote the zero polynomial und `Coeff c n p` represents the polynomial $c \cdot x^n + p$. For example, the polynomial $4 \cdot x^3 + 2 \cdot x + 5$ with integer coefficients is represented by the term `q` with `q = Coeff 4 3 (Coeff 2 1 (Coeff 5 0 Null))` and type `q :: Polynomial Int`.

```
In [42]: data Polynomial a = Coeff a Int (Polynomial a)| Null deriving Show
```

```
In [43]: q :: Polynomial Int
         q = Coeff 4 3 (Coeff 2 1 (Coeff 5 0 Null))
```

```
In [44]: q
```

```
Coeff 4 3 (Coeff 2 1 (Coeff 5 0 Null))
```

2.4.1 4a)

Write a function `foldPoly :: (a -> Int -> b -> b) -> b -> Polynomial a -> b` that behaves similar to the `foldr` function on lists, i.e., `foldPoly f e p` replaces every occurrence of the constructor `Coeff` by `f` and every occurrence of `Null` by `e` in `p`. For example, `foldPoly (\c n m -> c *3^n + m) 0 q` evaluates to `4 * 3^3 + 2 * 3 + 5 == 119`.

```
In [45]: foldPoly :: (a -> Int -> b -> b) -> b -> Polynomial a -> b
         foldPoly f e Null = e
         foldPoly f e (Coeff c n p) = f c n (foldPoly f e p)
```

Test cases

```
In [46]: -- the example from the exercise description
         foldPoly (\c n m -> c *3^n + m) 0 q
```

```
119
```

2.4.2 4b)

*Write a function `degree :: Polynomial Int -> Int` that computes the degree of a polynomial with integer coefficients. We define `degree Null == minBound`, where `minBound :: Int` is the smallest integer on your system, as a placeholder for $-\infty$. The degree of a nonzero polynomial is the maximal power of x occurring with a nonzero coefficient. For example the degree of $4 \cdot x^3 + 2 \cdot x + 5$ is 3. For simplicity, we assume that if a polynomial contains the expression `...Coeff c n ...` then `n` does not occur as the second argument of `Coeff` again. You may **not** use any predefined functions except comparisons. You may **not** use explicit recursion.*

```
In [47]: maxCoeff :: Int -> Int -> Int -> Int
         maxCoeff c n m | (c < 0) = m
                        | (c > 0) && (n > m) = n
                        | otherwise = m

         degree :: Polynomial Int -> Int
         degree poly = foldPoly (maxCoeff) minBound poly
```

Test cases

```
In [48]: -- the example from the exercise description
         degree q
```

```
3
```

```
In [49]: -- a Null polynomial
         degree Null
```

```
-9223372036854775808
```

```
In [50]: -- x^2 + 2x + 1
         degree (Coeff 1 2 (Coeff 2 1 (Coeff 1 0 Null)))
```

```
2
```

```
In [51]: -- x^2 + 5x^4 + 3x^3 + x^7 + x
         degree (Coeff 1 2(Coeff 5 4(Coeff 3 3(Coeff 1 7 (Coeff 1 0 Null)))))
```

```
7
```

```
In [52]: -- x^3 + 0x^2 + x + 1
         degree (Coeff 1 3(Coeff 0 2(Coeff 1 1(Coeff 1 0 Null))))
```

```
3
```