P. Thiemann, G. Radanne Wintersemester 2019/20

Functional Programming

http://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2019/

Exercise Sheet 4 – High order functions, Functional data structures

2017-06-05

1 High order functions

Exercise 1 (Folding)
Fold is a very common functional programming idiom:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

1. Define `foldr`.
2. Using `foldr`, implement:
   - `or`, returns `True` if at least one item in the list of booleans is `true`
   - `filter`
   - `map`
   - `foldl`, the left-associative variant of `foldr`:

     ```
     foldl :: (b -> a -> b) -> b -> [a] -> b
     foldl _ acc [] = acc
     foldl f acc (x: xs) = foldl f (f acc xs) xs
     ```

   - `remdups`, removes consecutive duplicates from a list

Exercise 2 (Unfolding)
There is also a dual function to `foldr`, `unfoldr`:

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

Instead of reducing a list to a final result, `unfoldr f seed` builds a new list: The elements of the list are created by repeatedly applying the `f` function to the accumulator `b`. If `f b` returns the value `Nothing`, the list is over. If `f b` returns the value `Just (a, b ')`, then `a` is added as the foremost element. The value `b '` is then passed to `f` to calculate the next element.

1. Define `unfoldr`.
2. Using `unfoldr`, define `map`.
3. Another standard function of functional programming is `iterate :: (a -> a) -> a -> [a]`
   What could this function do? Implement `iterate` using `unfoldr`.

2 Functional data structures

Exercise 3 (Lazy Lists – Hamming numbers)

1. Write a function `mergeBy :: Ord a => [a] -> [a] -> [a]` which merges two sorted lists in one sorted list.
2. The Hamming numbers are a sequence of number of the form $2^i * 3^j * 5^k$ for all i, j, k positive integers. Define `hamming :: [Integer]`, the infinite lists of sorted Hamming numbers.

Exercise 4 (Tries)

The goal of this exercise is to implement Tries. Tries, or "prefix trees", are trees where each branch is indexed by a character. Each path in the trie then represent a list of characters, aka a string.

We consider the following definition of tries, where each node contains a boolean (indicating if the string considered so far is in the trie) and the branches of the tries represented as a map from characters to sub-tries.

```
import qualified Data.Map as Map

data Trie = Trie Bool (Map.Map Char Trie)
```

1. Implement the following functions:

```
empty :: Trie

insert :: [Char] -> Trie -> Trie
member :: [Char] -> Trie -> Bool

prefix :: [Char] -> Trie -> Trie
union :: Trie -> Trie -> Trie

ofList :: [[Char]] -> Trie
```

   Which other functions could you implement? Look at the API of `Data.Set` and `Data.Map` for inspiration. You can also derive a few appropriate instances.

2. Is the `remove :: [Char] -> Trie -> Trie` function easy to write? Write a first naive version, and consider how you would write one that minimizes the size of the trie after deletion.

3. Test your implementation using quickCheck. Use the function `ofList` to generate arbitrary tries. You can consider tests such as "For any trie t, if I insert something in t, it is now a member".

4. Generalize the previous definition of `Trie` to lists of any elements (not only characters). Adapt the various function definitions. Do you need a typeclass constraints on the elements? How much do you need to change your code?

5. We now consider the case of a dictionary-trie, where each "string" (or list) is associated to a value. How would you change the original definition? Adapt the various function definitions and your tests.