Prof. Dr. Peter Thiemann
Janek Spaderna
janek.spaderna@pluto.uni-freiburg.de

Winter Term 2022/23
Issued 2022-11-18
Discussion on 2022-11-25

Functional Programming

https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2022/

Exercise Sheet 5

Exercise 1 (Maybe, maybe not)

A very versatile data type available from the `Prelude` is `Maybe`. A value of type `Maybe a` holds either a value of type `a` or no value. The goal of this exercise is to make you comfortable working with `Maybe` values which you will come across in the following exercises.

The module `Data.Maybe` defines functions for working with `Maybe` values. Take a look through the documentation and choose three functions to implement yourself. You can check the solution yourself by clicking on the `Source` links.

Exercise 2 (Unfolding)

As a dual to `foldr` there exists the function `unfoldr :: (b -> Maybe (a, b)) -> b -> [a]`

https://hackage.haskell.org/package/base-4.16.3.0/docs/Data-List.html#v:unfoldr

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 `f` to the (updated) `seed`. Once `f seed` returns `Nothing` the list is terminated.

1. Define `unfoldr`.
2. Using `unfoldr`, define `map`.
3. Using `unfoldr`, write a function `range` such that `range m n` produces the ordered list of integers from `m` to `n` (inclusively).

Exercise 3 (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.

Note We will make use of the `Map` type from the containers library. A value of type `Map k v` is a dictionary/finite map from keys of type `k` to values of type `v`. The type and its functions are documented in `Data.Map.Lazy`. You will have to add the library to either your `package.yaml` file (stack) or the `.cabal` file (cabal), just as you did for QuickCheck.

We consider the following definition of tries, where each node contains a boolean (indicating if the string from the root node to this node 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 stubbed functions in `Trie.hs` linked on the lecture home page.

   Additionally, derive or implement appropriate instances. You can also look through the API of `Data.Set` and `Data.Map.Lazy` for inspiration for additional functions.

2. Test your implementation using QuickCheck. Use the function `fromList` 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."

   Make sure, that your `delete` returns a minimal `Trie`: "if I insert a word `w` into some trie `t` and I delete `w` again the result should be structurally equal to `t`."

3. The generalization from our `Trie` type, which stores words over `Char`, to storing words over any type `a` is relatively easy. As this translation is quite mechanical we will only consider what changes would be necessary.

   a) Give the new data definition.
   b) Are there required typeclass constraints? How do the type signatures change?
   c) How do the function implementations change?

4. We now consider the case of a dictionary-trie, where each word is associated with a value. Create a new module containing the new data type `TrieMap`. Adapt the various functions and tests.