Functional Programming for Logicians
Homework 8

Péter Mekis
Department of Logic, ELTE Budapest

Deadline: 2019 April 7 17:59 pm

Solve two of exercises 1-5, two of exercises 6-10, and exercise 11. Solving more is appreciated, but not necessary.

Reasoning about algorithms Prove the following claims by induction on the complexity of lists. Use the sample proof given in the slides for the April 1 session.

1. The values of '`foldr f y xs`' and '`foldl (flip f) y (reverse xs)`' are the same for any list '`xs`', function '`f`' and value '`y`' of compatible types.
2. The values of '`(map f) . (map g) xs`' and '`map (f . g) xs`' are the same for any list '`xs`', and function '`f`' and '`g`' of compatible types.
3. The values of '`fs <*> pure x` and `pure (\f x) <*> fs`' are the same for any list of functions '`fs`', and value '`x`' of compatible types.
4. The values of '`pure (.) <*> fs <*> gs <*> xs`' and '`fs <*> (gs <*> xs)`' are the same for any list of functions '`fs`' and '`gs`', and lists of values '`xs`' of compatible types.
5. The quicksort function defined in the March 11 session sorts any finite lists of integers in finitely many steps.

Applicatives

6. We saw in the class that there is another way of instantiating the `Applicative` class for lists. An example of applying the *bind* operation in this "zipping style" version:
   `[(+2), (*2)] <*> [3, 4] == [5, 16]`
   If one list is longer than the other, the extra elements are ignored:
   `[(+2), (*2)] <*> [3, 4, 5] == [5, 16]`
   Define this version.
7. Instantiate the `Applicative` class for the Tree type as defined in the March 25 session, following the logic of instantiating it for lists.
8. Instantiate once again the `Applicative` class for the Tree type, now following the logic of the "zipping style" version above.
9. Look up the `Biapplicative` class in the Haskell documentation. Make the `Tree2` type (tree with two parameters, one for the nodes, one for the leaves) as it was defined in Homework 6, exercise 8, an instance of the `Biapplicative` class, just as Use Homework 7, exercise 5, instantiating `Bifunctor`, as a basis.
10. Instantiate the `Applicative` class for the `Set` type defined in Homework 7, exercise 11. Use exercise 12, instantiating `Functor`, as a basis.

Input and output

11. The following Haskell code contains a minimal code to handle input and output. Read it and try to make sense of it without looking up definitions of `do`, `getLine`, and `putStrLn`. These will be explained in the next session.

    ```
    foo :: [String] -> Int -> Bool
    foo xs n = maximum (map length xs) < n

    main :: IO()
    main = do
    par1 <- getLine
    par2 <- getLine
    putStrLn $ show $ foo (read par1 :: [String]) (read par2 :: Int)
    ```

    Now use it as a sample to convert one of your previous homework exercises into a buildable code. You can find the code in the file "`haskell_hw8_basic_io.hs`".