Programming Paradigms Fall 2023 — Problem Sets

by Nikolai Kudasov and Khaled Ismaeel

October 5, 2023

1 Problem set №6

1. Implement the following functions over a list of binary digits in Haskell using **explicit recursion**. Each function should be implemented independently.

   (a) Count a given `Char` in a `String`:

   ```
   countChar 'l' "hello"          -- 2
   ```

   (b) Convert a binary string represented a list of `0`s and `1`s into a (decimal) number:

   ```
   binaryToDecimal [1,0,1,1,0]    -- 22
   ```

   (c) Encode a binary string by removing leading zeros and replacing each consecutive substring of digits with its length. For example, `[0,0,0,1,1,0,1,1,1,0,0]` has some leading zeros,then 2 ones,then 1 zero,then 3 ones,then 2 zeros,so it should be encoded as `[2,1,3,2]`):

   ```
   encodeWithLengths [0,0,0,1,1,0,1,1,1,0,0]    -- [2,1,3,2]
   ```

   (d) Decrement a binary number. Decrementing zero should produce zero:

   ```
   decrement [1,0,1,1,0]    -- [1,0,1,0,1]
   decrement [1,0,0,0,0]    -- [1,1,1,1]
   decrement [0]            -- [0]
   ```

   (e) Implement function `propagate :: (Bool, [Int]) -> [(Bool, Int)]` that pairs a given boolean value with every integer in the list:

   ```
   propagate (False, [1, 2, 3])    -- [(False,1),(False,2),(False,3)]
   propagate (True, [1, 1])        -- [(True,1),(True,1)]
   ```

2. Implement in Haskell a function `sumAndProduct` that computes a sum and a product of a list of numbers.
   For example, `sumAndProduct [6, 2, 4, 1]` should compute `[13, 48]`.

   (a) Implement `sumAndProduct` using explicit recursion (i.e. **without** higher-order functions).

   (b) Use the equational reasoning to verify that `(fst (sumAndProduct [x, y, z]))` is equal to `(x + y + z)`.

3. Consider the following definitions:

   ```
   data Days   = Days Double      -- 7 days   = Days 7
   data Weeks  = Weeks Double     -- 4 weeks  = Weeks 4
   data Months = Months Double    -- 1 month  = Months 1
   ```

   Implement the following functions that convert between different time intervals, assuming that one month is exactly 30 days:

   ```
   daysToWeeks :: Days -> Weeks
   weeksToMonths :: Weeks -> Months
   mothsToDays :: Months -> Days
   ```