assignment03

May 15, 2019

1 Functional Programming SS19

2 Assignment 03 Solutions

2.1 Exercise 1 (Lazy Evaluation)

In this exercise we will consider the "Collatz conjecture"[1] *from mathematics. In this exercise you are allowed to use any predefined function from Haskell's module Prelude.*

Let the function $f : \mathbb{N} \mapsto \mathbb{N}$ *be defined as*

$$n \mapsto \begin{cases} \frac{n}{2}, & n \bmod 2 = 0 \\ 3 \cdot n + 1, & n \bmod 2 = 1 \end{cases}$$

For any $n, k \in \mathbb{N}$*, let* $f^k(n)$ *denote* $\underbrace{f(...f(n)...)}_{k \text{ times}}$*. The Collatz total stopping time of a positive natural number* n *is defined to be the smallest* $k \in \mathbb{N}$ *such that* $f^k(n) = 1$ *and* ∞ *otherwise. For example, the Collatz total stopping time of 1 is 0 (since* $f^0(1) = 1$*), the Collatz stopping time of 2 is 1 (since* $f^0(2) = 2, f^1(2) = 1$*) and the Collatz total stopping time of 3 is 7. The Collatz conjecture states that any positive natural number has a finite Collatz total stopping time. This conjecture is a famous open problem.*

$^{1} $https://en.wikipedia.org/wiki/Collatz_conjecture

2.1.1 1a)

Implement a function `collatz :: Int -> [Int]` *such that for any positive natural number* n *the expression* `collatz n` *is the infinite list with kth entry* $f^k(n)$*. This function may behave arbitrarily on non-positive inputs.*

Furthermore implement a function `total_stopping_time :: Int -> Int`*. For any positive natural number* n *the function computes its Collatz total stopping time. This function may behave arbitrarily on non-positive inputs.*

Hints

- *The predefined function* `takeWhile :: (a -> Bool) -> [a] -> [a]` *computes the shortest prefix of the list where the first argument of* `takeWhile` *is true for every list element.*
- *The predefined function* `div :: Int -> Int -> Int` *computes integer division, the predefined function* `mod :: Int -> Int -> Int` *implements the modulo operation from mathematics.*

```
In [7]: f :: Int -> Int
        f n | m == 0 = n `div` 2
            | otherwise = 3 * n + 1
            where m = n `mod` 2

        collatz :: Int -> [Int]
        collatz n | n <= 0 = []
                  | otherwise = n : collatz r
                  where r = f n

        total_stopping_time :: Int -> Int
        total_stopping_time n = length (takeWhile (/= 1) (collatz n))
```

Test cases

```
In [8]: -- the first test case in the exercise description
        total_stopping_time 1
```

```
0
```

```
In [9]: -- the second test case in the exercise description
        total_stopping_time 2
```

```
1
```

```
In [10]: -- the third test case in the exercise description
         total_stopping_time 3
```

```
7
```

```
In [17]: -- another test case
         total_stopping_time 7
```

```
16
```

```
In [18]: -- yet another test case
         total_stopping_time 9
```

```
19
```

2.1.2 1b)

Implement a function `check_collatz :: Int -> Bool`*. For any positive natural number* `n` *the function returns* `True` *if the Collatz conjecture holds for the first* `n` *positive natural numbers. If the input* `n` *is not positive or the conjecture does not hold for the first* `n` *positive numbers, then the function may behave arbitrarily.*

```
In [4]: check_collatz :: Int -> Bool
        check_collatz n | n <= 0 = True
                        | otherwise = (total_stopping_time n) < maxBound
                                      && check_collatz (n - 1)
```

Test cases

```
In [ ]: -- if the conjecture holds, the following
        -- line should never terminate
        takeWhile (==True) [check_collatz n | n <- [1..]]
```

2.2 Exercise 2 (List Comprehensions)

In this exercise you ***may not*** *write auxiliary functions and you* ***may not*** *use* `where` *or* `let`*.*

2.2.1 2a)

Consider the following definition for an infinite list containing all primes from the lecture:

```
drop_mult :: Int -> [ Int ] -> [ Int ]
drop_mult x xs = [ y | y <- xs , y `mod` x /= 0]

dropall :: [ Int ] -> [ Int ]
dropall ( x : xs ) = x : dropall ( drop_mult x xs )

primes :: [ Int ]
primes = dropall [2 ..]
```

Write a function `goldbach :: Int -> [(Int,Int)]`[2] *that returns a list of pairs indicating all possibilities to write a positive number as a sum of two odd primes. Each possibility up to permutation has to occur exactly once but the order of the pairs in the resulting list is irrelevant. For example,* `goldbach 50 == [(3,47),(7,43),(13,37),(19,31)]`*. Of course,* `goldbach n == []`*, whenever* `n` *is odd and positive! For non-positive numbers, the function may behave arbitrarily. A famous open problem is whether there exists an even positive number* `n` *with* `goldbach n == []`*.*

You may only use ***one*** *defining equation and the right-hand side must be a single list comprehension.*

Hints

- *You may use the given functions* `primes`*,* `drop_mult`*, and* `dropall` *in addition to predefined functions in Haskell's module Prelude.*
- *Only test your implementation on relatively small numbers! Our implementation of primes is not the most efficient one.*

---

[2] https://en.wikipedia.org/wiki/Goldbach's_conjecture

```
In [5]: ----------------------------------------------------
        -- defined in the lecture notes
        ----------------------------------------------------
        drop_mult :: Int -> [ Int ] -> [ Int ]
        drop_mult x xs = [ y | y <- xs , y `mod` x /= 0]

        dropall :: [ Int ] -> [ Int ]
        dropall ( x : xs ) = x : dropall ( drop_mult x xs )

        primes :: [ Int ]
        primes = dropall [2 ..]
        ----------------------------------------------------

        goldbach :: Int -> [(Int,Int)]
        goldbach n | n < 0 || odd n = []
                   | otherwise = [(x,y) | x <- takeWhile (<=(n `div` 2)) primes,
                                          y <- takeWhile (<=n) primes,
                                          odd x, odd y, x + y == n]
```

Test cases

```
In [6]: goldbach 50
```

```
[(3,47),(7,43),(13,37),(19,31)]
```

```
In [7]: goldbach 100
```

```
[(3,97),(11,89),(17,83),(29,71),(41,59),(47,53)]
```

```
In [8]: goldbach 1000
```

```
[(3,997),(17,983),(23,977),(29,971),(47,953),(53,947),(59,941),(71,929),(89,911),(113,887),(137,
```

2.2.2 2b)

Implement a function `range :: [a] -> Int -> Int -> [a]`. *The expression* `range xs m n` *should yield the sublist of* `xs` *starting from the first element with an index >= m and ending with the last entry at an index <= n. The first entry has index 0, the last one has (length xs) - 1. For example,* `range [3,4,5] 1 2 == [4,5]`, `range [3,4,5] (-7) 2 == [3,4,5]` *and* `range [3,4,5] 10 7 == []`.

You may only use one defining equation and the right-hand side must be a single list comprehension.

You may use any predefined functions in Haskell's module Prelude ***except*** `take`, `drop`, `span`, `break`, `splitAt`, *or variants*[3] *of these functions.*

[3] like `takeWhile`, etc.

```
In [9]: range :: [a] -> Int -> Int -> [a]
        range xs m n = [x | (i,x) <- (zip [0,1..] xs),
                            i >= m, i <= n]
```

Test cases

```
In [10]: -- the first example from the exercise description
         range [3,4,5] 1 2
```

```
[4,5]
```

```
In [11]: -- the second example from the exercise description
         range [3,4,5] (-7) 2
```

```
[3,4,5]
```

```
In [12]: -- the last example from the exercise description
         range [3,4,5] 10 7
```

```
[]
```

2.3 Exercise 3 (IO)

In this exercise an interactive program simulating your personal library should be implemented.

The library has a list books in which the books of the library are stored. A book is represented by a tuple `(title,author): (String,String)`*. After displaying the welcome message, the library is still empty. Then an interactive loop starts where the user is prompted for input and the library reacts to it. If the user * enters* `Book:title;author`*, the user can choose to either put the book into the library (by entering the character* `'p'`*) or take it from the library (by entering the character* `'t'`*). If the book is to be taken but not contained in the library, an error message should be stated before starting from the beginning. * enters* `Author:author`*, all the books from this author are printed on the screen. Then the loop starts again. * enters* `Title:title`*, all the books with this title are printed on the screen. Then the loop starts again. * enters* `Exit`*, the loop terminates. * enters something else, an error message is shown and the loop starts again.*

After the interactive loop terminates, a goodbye message is displayed and the whole program terminates. A framework for the implementation is given in the file `library.hs`*. Only edit the parts between the comments replace with implementation: and end replace. Do not change any other code.*

Hints

To output a line of text use `putStrLn :: String -> IO ()`*, to read a line use* `getLine :: IO String`*. Furthermore you may use* `elem :: Eq a => a -> [a] -> Bool` *for checking membership in a list.*

3a) *Implement the function* `main` *that first displays a welcome message, then evaluates* `library []` *and then displays a goodbye message.*

3b) *Implement the function* `getInput` *that displays a prompt >, then reads a line from standard input, and finally returns a LibraryInput computed from the just-read string using the given function* `parseLibraryInput`*.*

3c) *Implement the function library. It should first ask the user for input using* `getInput` *and then take an appropriate action, depending on the input as described above.*

In [13]:
```haskell
import Data.Char

data LibraryInput = Exit | Error String | Book (String, String) | Author String | Title

instance Show LibraryInput where
  show Exit = "Exit"
  show (Error xs) = "Invalid Input: " ++ xs
  show (Book (title, author)) = "Book: " ++ title ++ ";" ++ author
  show (Author author) = "Author: " ++ author
  show (Title title) = "Title: " ++ title

trim :: String -> String
trim = f . f
   where f = reverse . dropWhile isSpace

parseLibraryInput :: String -> LibraryInput
parseLibraryInput input   | lhs == "Book" = Book ((trim (drop 1 title)), (trim (drop 1
                          | lhs == "Author" = Author (trim (drop 1 rhs))
                          | lhs == "Title" = Title (trim (drop 1 rhs))
                          | elem (map toLower input) ["q", "e", "exit", "quit"] = Exit
                          | otherwise = (Error input)
                            where
                              (lhs, rhs) = span (/= ':') (trim input)
                              (title, author) = span (/= ';') (trim rhs)

-- exercise:
main :: IO ()
main = do
    -- task a)
    -- replace with implementation:

    putStrLn "Welcome to your personal library!"
    library []
    putStrLn "Thanks for visiting. Have a nice day!"
    return ()

    -- end replace

library :: [(String,String)] -> IO ()
library books =  do
  -- task c)
  -- replace with implementation:

  input <- getInput
  putStrLn (show input)
  case input of
    Book (t, a) -> do
      putStrLn "Do you want to (p)ut the book back or do you want to (t)ake the book?"
```

```
      book_input <- getLine
      case book_input of
        "p" ->
          if (elem (t, a) books) then do
            putStrLn "You already have this book!\n"
            library books
          else do
            putStrLn "Done!\n"
            library ((t, a) : books)
        "t" ->
          if not (elem (t, a) books) then do
            putStrLn "You do not have this book!\n"
            library books
          else do
            putStrLn "Done!\n"
            library (filter (/= (t, a)) books)
        otherwise -> do
          putStrLn "Wrong input!\n"
          library books
    Author a -> do
      putStrLn ("You have the following books from: " ++ a)
      putStrLn (concatMap (\x -> show x ++ "\n") [Book (t, a_name) |
                                                  (t, a_name) <- books,
                                                  a_name==a])
      library books
    Title t -> do
      putStrLn ("You have the following books with the title: " ++ t)
      putStrLn (concatMap (\x -> show x ++ "\n") [Book (t_name, a) |
                                                   (t_name, a) <- books,
                                                   t_name==t])
      library books
    Error xs -> library books
    Exit -> return ()

  -- end replace

getInput :: IO LibraryInput
getInput = do
  -- task b)
  putStrLn "Would you like to put back or take a book?\n Enter Book: Title's name; Auth
  -- task b)
  -- replace with implementation:

  putStr "> "
  line <- getLine
  input <- return (parseLibraryInput line)
  return input
```

```
    -- end replace
```

3d) *Is it possible in Haskell to write a function* `main' :: Int` *that behaves similar to the function main but returns the number of books stored in the library when the loop was exited? Explain your answer either on the paper you hand in or as a comment in your solution of the programming exercises. You must not refer to predefined Haskell functions which are not included in the module Prelude.*

Since the `library` function does not return its result directly, `main'` could only return the number of books if `library` is writing the library books to a file (e.g. using the `writeFile` function in *Prelude*); then, `main'` could read the list of books upon exiting the loop (e.g. using the `readFile` function in *Prelude*) and return the number.

2.4 Exercise 4 (Definedness)

a) *Consider the following values of the domain* $(\mathbb{Z}_\perp \times \mathbb{B}_\perp) \times \mathbb{Z}_\perp$*:*

- $y_1 = ((-1, False), 0)$
- $y_2 = ((-1, \perp), 2)$

Find all elements in the domain that are less defined than one of the values above, i.e., all x *such that* $x \sqsubseteq y_i \wedge x \neq y_i$ *for some* $i \in \{1, 2\}$*.*

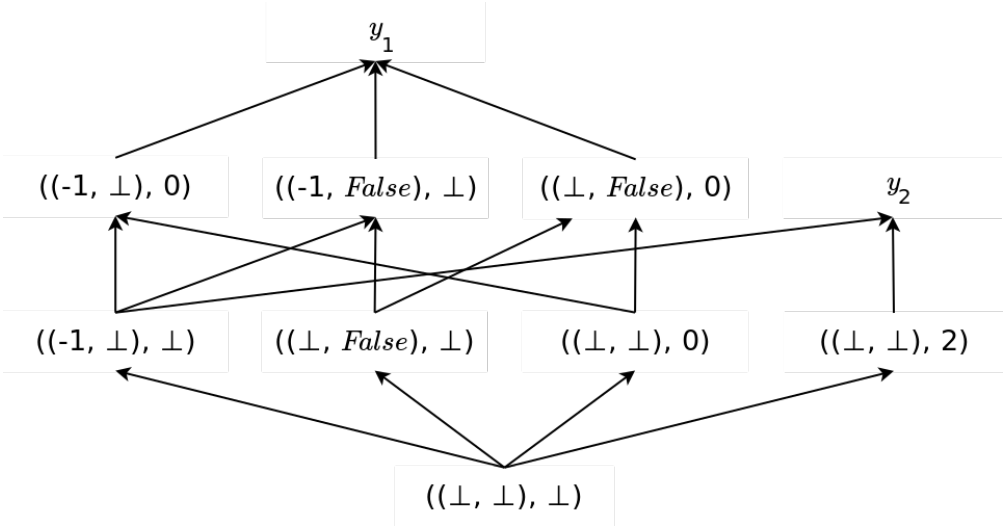
Draw a directed graph whose nodes are labeled with y_1*,* y_2 *and all these values* x*. Add arrows in such a way that there is a path from* x' *to* y' *if and only if* x' *is less defined than* y'*.*

According to definition 2.1.1 from the lecture notes, given a base domain D, $d \sqsubseteq d'$ for any $d, d' \in D$ iff $d = \perp$ or $d = d'$. Furthermore, according to Definition 2.1.2, $(d_1, ..., d_n) \sqsubseteq (d'_1, ..., d'_n)$, where $D = D_1 \times ... \times D_n$, iff $d_i \sqsubseteq d'_i$ for all $1 \leq i \leq n$. In this case, the domain is given as

$$D = D_{1,2} \times D_3 = (D_1 \times D_2) \times D_3$$

The domain can however also be written as $D' = D_1 \times D_2 \times D_3$, which is isomorphic to D[1].

A Hasse diagram illustrating the order of $(\mathbb{Z}_\perp \times \mathbb{B}_\perp) \times \mathbb{Z}_\perp$ for y_1 and y_2 is given below.



4a_hasse_diagram

[1] B. Milewski, "Category Theory for Programmers", ch. 7, pp. 74-75.

b) *Consider the domain* $D = \underbrace{\mathbb{Z}_\perp \times ... \times \mathbb{Z}_\perp}_{n \text{ times}}$ *for* $0 < n \in \mathbb{N}$. *A chain* $S \subseteq D$ *is a totally ordered subset of* D, *i.e. if* $x \neq y \in S$ *then either* $x \sqsubseteq y$ *or* $y \sqsubseteq x$. *Determine*

$$\sup\{|S| \mid S \subseteq D, S \text{ is a chain}\}$$

where $|S|$ *is the number of elements in* S. *In other words, what is the maximal number of elements a chain in* D *can have? Please prove the correctness of your answer.*

Let us first consider some examples to obtain an intuition for the problem. Given $n = 1$, a maximal chain S would be $S = \{\perp, 0\}$; for $n = 2$, a maximal chain is $S = \{(\perp, \perp), (\perp, 0), (0, 0)\}$; for $n = 3$, a maximal chain would be $S = \{(\perp, \perp, \perp), (\perp, \perp, 0), (\perp, 0, 0), (0, 0, 0)\}$, and so forth.

Theorem: Given a domain $D = \underbrace{Z_\perp \times ... \times Z_\perp}_{n \text{ times}}$ such that $n > 0 \in \mathbb{N}$, the maximum number of elements a chain S in D can have is $n + 1$.

Proof: Let $D = D_1 \times ... \times D_n$ and $S \subseteq D$ be a chain, where $S = \{d_1, ..., d_k\}$. Since we are interested in a maximal chain, $\perp$ is always included in S, such that we will denote $d_1 = \perp$. The highest element in a chain does not have $\perp$ for any D_i, $1 \leq i \leq n$ and is also always included in S; we will let d_k represent this element of the chain. For any other element d_i in S, $\perp$ can be included up to $n - 1$ times, such that, due to the fact that the chain is totally ordered, permutations of the elements of d_i are not allowed in S. But since we are looking for a maximal chain, $d_i \neq d_1 \neq d_k$ represent all possible combinations of ordered pairs in which $\perp$ appears at least once and at most $n - 1$ times. There are $n - 1$ such combinations; together with d_1 and d_k, there are a total of $n + 1$ elements in S, which is the maximum cardinality S can have. $\square$.