Fakultät für Informatik und Mathematik Summer semester 2023
Prof. Dr. Martin Griebl

Exercises for Lecture: Functional Programming
Exercise Sheet 12 (Strictness Analysis, Lazy Evaluation, Interpretation)

Problem 1 (Strictness Analysis)

Determine (using the method shown in the lecture) in which arguments the following functions are strict. Assume that the functions $<$, $+$, $-$ are strict in all their arguments.

(a) `f (x,y,z) =      if x<0 then f (z,z,-x) else y+z`

(b) `f (x,y,z) = y + (if x<0 then f (z,z,-x) else y+z)`

Problem 2 (Lazy Evaluation)

It is possible to define infinite and cyclic data structures in Haskell using laziness (a terminating program can only use a finite part of an infinite structure, of course).

(a) Define the list `evenNumbers :: [Integer]` of even numbers `[0,2,4,...]`. Define `evenNumbers` by a computation of the following form:

```
evenNumbers = 0 :  ... evenNumbers ...
```

Do not write a recursive function `f`, which generates the list (to define, for example, `evenNumbers = f 0`) but define `evenNumbers` itself in a recursive way.

(b) Define the list `fibs :: [Integer]` of Fibonacci numbers. Define it (similar to (a)) as

```
fibs = 0 : 1 :  ... fibs ...
```

(c) Define the list `primes :: [Integer]` of prime numbers. This requires a different approach from (a) and (b). A method to compute the prime numbers is the Sieve of Eratosthenes. Start with the list of numbers $2, \dots, n$. The first number (2) is prime. Now all multiples of the prime just found are removed (i.e., $4, 6, \dots$). The next number which remains in the list (3) is the next prime. Again, the multiples of this prime are removed, and so on.

Due to the laziness, one does not need to fix an upper bound n but can define the list of all prime numbers using the sieve. Define the list of primes as an application of the function `sieve` to the list of numbers from 2:

```
primes = sieve [2..]
    where
    sieve (x:xs) =  ...
```

Function `sieve` returns the next prime, removes its multiples from the remaining list and continues recursively on the resulting list.

(d) Write a function `roundrobin :: [a] -> [a]` which repeats a given list infinitely. For example, `roundrobin [1,2,3] = [1,2,3,1,2,3,1,2,3,...]`. Turn on the statistics about time and memory consumption in GHCi (with `:set +s`) and test `roundrobin [1,2,3] !!` n for $n = 10^5, 10^6, 10^7, 10^8, 10^9$. The infinite list returned by `roundrobin` can be represented as a cyclic structure in memory, i.e., the access to the element with index n needs $\Theta(n)$ time but only $\Theta(1)$ memory (in contrast to (a)–(c) where an access to index n needs $\Omega(n)$ memory). Try to implement `roundrobin` such that only a constant amount of memory is needed.

Problem 3 (Enriched λ-Calculus)

Let us consider again the following abstract syntax known from previous exercise sheet:

```
data LExp = V String                  -- variable
         | CInt Int                   -- Int constant
         | CBool Bool                 -- Bool constant
         | LExp :@: LExp              -- application
         | L String LExp              -- λ-abstraction
         | If LExp LExp LExp          -- If c t e  means  if c then t else e
         | Y LExp                     -- fixed-point operator
         | Prim Op [LExp]             -- predefined Operator
         deriving (Show)
```

In this exercise we want to define the semantics not by β- and δ-reductions (as before) but assign to each construct an interpretation in Haskell. You can find the necessary files in Stud.IP.

We assign to every expression of type `LExp` a function from `Env Domains` to `Domains` by

```
eval :: LExp -> Env (Domains) -> Domains
```

`Domains` is a data type which can represent the values of our language, namely `Int`, `Bool` and functions. `Env Domains` keeps a mapping from variable names to values from `Domains`, i.e., it represents the environment.

Complete the implementation of function `eval` in module `Eval`. In the course of the recursion, the environment (parameter `env`) is updated with values for the variables bound in the expression.

Test your implementation with the example in module `Main` by calling `main`.

(The modules `Reduce` and `Syntax` can be found in the soultion for the previous sheet.)

Due date: Tuesday, July 11, 2023 at 16:00

Upload your solutions to the `Submissions` folder for this exercise in Stud.IP.