I/O, Evaluation

Jonas Hübotter

August 19, 2020

1 Exercises

1.1 I/O

1. Use the Prelude function `getLine` and the function `readMaybe` from `Text.Read` to define a function `getInt :: IO (Maybe Int)` that safely reads an integer from standard input.

2. [sheet 11] In this exercise you will implement the following game: The program selects a random number in the range `[0..100]` and asks the user to guess the number. If the user's guess is incorrect, the program tells them whether the actual number is larger or smaller and waits for another guess. If the user guesses correctly, the program prints the number of guesses made.

   Hints:

   - Use `getInt` from the previous exercise to read numbers from the user.
   - Use `randomRIO` from `System.Random` to get a random number in a certain range.

3. [sheet 15] We consider a game of matches for two players. In the beginning, there are 10 matches on the table. The players take turns in taking matches off the table (at least 1 and at most 5). The winner is the player who takes the last match.

   Define an IO action `match :: IO ()` that implements the game of matches. Before any player's turn the program should print the number of the player and of the remaining matches. When a player wins the program should print the winner and exit afterwards. The program should ensure that the player only takes a valid number of matches. If not enough matches remain, the player takes all of them.

   You can use the function `putStrLn :: String -> IO ()` to print a string to standard output and `readLn :: Read a => IO a` to read from standard input.

```
Matches: 10. Player 1?
4
Matches: 6. Player 2?
6
The input must be between 1 and 5.
5
Matches: 1. Player 1?
1
Player 1 wins!
```

1.2 Evaluation

1. [sheet 14] Identify all redexes in the following `Integer`-expressions. Determine for each redex whether it is innermost, outermost, both, or neither.

   (a) `1 + (2 * 3)`

(b) `(1 + 2) * (2 + 3)`

(c) `fst (1 + 2, 2 + 3)`

(d) `fst (snd (1, 2 + 3), 4)`

(e) `(\x -> 1 + x) (2 * 3)`

2. [sheet 14] Evaluate the following expressions according to Haskell's evaluation strategy:

```
map (*2) (1 : threes) !! 1
(\f -> \x -> x + f 2) (\y -> y * 2) (3 + 1)
head (filter (/=3) threes)
```

Which evaluations do not terminate?

The functions used in the expressions above are defined as follows:

```
map _ [] = []
map f (x:xs) = f x : map f xs

filter _ [] = []
filter f (x:xs) | f x = x : filter f xs
                | otherwise = filter f xs

(x:xs) !! n | n == 0 = x
            | otherwise = xs !! (n - 1)

threes = 3 : threes
```

1.3 Infinite data structures

1. *(ranger)* Define a function

```
enumFromThen' :: Integer -> Integer -> [Integer]
```

using list comprehensions such that `enumFromThen' a b = [a,b..]`.

Use `enumFromThen'` to define a function

```
enumFromThenTo' :: Integer -> Integer -> Integer -> [Integer]
```

such that `enumFromThenTo' a b c = [a,b..c]`.

2. Using the Prelude function `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]` that given a list $[x_1, \ldots, x_n]$, a list $[y_1, \ldots y_n]$ and a function f returns the list $[f(x_1, y_1), \ldots, f(x_n, y_n)]$ define the function `fib :: [Integer]` that computes the (infinite) list of Fibonacci numbers.

   Explain which components of the implementation require lazy evaluation such that the function can be (partially) evaluated. Which functions can we use to evaluate the function partially?

3. [sheet 14] Consider an alternative implementation for `fib`.

```
fib' :: [Integer]
fib' = map f [0..]
  where
    f 0 = 0
    f 1 = 1
    f n = fib' !! (n - 1) + fib' !! (n - 2)
```

Compare the alternative implementation with the original one. Which function performs better and how could the slower function be improved?

Since normal Fibonacci numbers are boring, we want to generalise them to n-onacci numbers. We can construct the n-onacci numbers by letting $f_0 = 0, f_1 = 0, \ldots, f_{n-2} = 0, f_{n-1} = 1$ and $f_a = f_{a-n} + f_{a-n+1} + \cdots + f_{a-1}$. Implement the function `nonacci :: Int -> [Integer]` in two ways:

(a) Come up with a function `zipWithN :: ([a] -> b) -> [[a]] -> [b]` and define `nonacci` analogously to `fib`.

(b) Use `fib'` as a template to define `nonacci`.

2 Homework

2.1 I/O

1. Derive the functions `putStr' :: String -> IO ()` and `putStrLn' :: String -> IO ()` from the Prelude function `putChar :: Char -> IO ()`.

2. Derive the function `getLine' :: IO String` from the Prelude function `getChar :: IO Char`.

3. [1, p. 461] Give a definition of the function

   ```
   fmap :: (a -> b) -> IO a -> IO b
   ```

   the effect of which is to transform an interaction by applying the function to its result. Define `fmap` using the `do` construct.

4. [1, p. 461] Define the function

   ```
   repeat' :: IO Bool -> IO () -> IO ()
   ```

   so that `repeat' test oper` has the effect of repeating `oper` until the condition `test` is `True`.

5. [1, p. 461] Define the higher-order function `whileG` in which the condition and the operation work over values of type `a`. Its type should be

   ```
   whileG :: (a -> IO Bool) -> (a -> IO a) -> (a -> IO a)
   ```

   Using the function `whileG`, define an interaction which reads a number n and then reads a further n numbers and finally returns their average.

6. [1, p. 461] Define a function

   ```
   accumulate :: [IO a] -> IO [a]
   ```

which performs a sequence of interactions and accumulates their result in a list.

Also give a definition of the function

```
sequence :: [IO a] -> IO ()
```

which performs the interactions in turn, but discards their results. Finally, show how you would sequence a series, passing values from one to the next:

```
seqList :: [a -> IO a] -> a -> IO a
```

What will be the result on an empty list?

7. [endterm 2013] Define a IO action `vowelCounter :: IO ()` that reads strings line-wise from standard input and counts the number of lowercase vowels (`'a','e','i','o','u'`). The program should not terminate and print the total number of vowels in all inputs after every entered line.

   Example:

```
>>> Hello world!
vowels: 3
>>> How are you?
vowels: 8
>>> brb
vowels: 8
>>> lol
vowels: 9
>>> aAaAaA
vowels: 12
>>> aeiou
vowels: 17
```

8. [endterm 2020] Define an IO action `main :: IO ()` that waits for user input in form of a binary number. The binary number is given as a string $0bx$ where x is a (potentially empty) string consisting of 0s and 1s. The string $0b$ represents 0. The program should output `"Invalid input"` if the given number does not adhere to this format. Otherwise, the program should print the number to the standard output after converting it to decimal. For example, the program should output 5 for the input $0b0101$. The program should continue to listen for the next input in either of the above cases. As an example, consider the following excerpt of the execution of the program.

```
>>> 0b12
Invalid input
>>> 0b010
2
>>> 0b111
7
...
```

   You can read from standard input with the function `getLine :: IO String` and print a string to the standard output with `putStrLn :: String -> IO ()`.

2.2 Evaluation

1. [endterm 2015] Given

```
f :: [a] -> [Bool] -> Bool
f _ (y:ys) = y
f [] _ = True
```

   Find defined expressions a and b such that the evaluation of the expression `f` a b fails.

2. [retake 2015] Using Haskell's evaluation strategy as introduced in the lecture, evaluate the following expressions step-by-step as far as possible. Indicate infinite reductions by "..." as soon as nontermination becomes apparent.

   (a) `head (map (\x -> x * x) [1,2,3])`

   (b) give a list `xs` such that `elem 0 (xs ++ [0])` does not evaluate to `True`. Then evaluate `elem 0 (xs ++ [0])`.

   The following definitions are given:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

```
head :: [a] -> a
head (x:_) = x
elem :: Eq a = > a -> [a] -> Bool
elem x [] = False
elem x (y:ys)
  | x == y    = True
  | otherwise = elem x ys
```

3. [endterm 2020] Given the following definitions:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

odds :: [Integer]
odds = 1 : map (+2) odds

(||) :: Bool -> Bool -> Bool
True || b = True
False || b = b
```

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)

inf :: [a]
inf = inf

instance Eq a = > Eq [a] where
    (==) :: Eq a = > [a] -> [a] -> Bool
    [] == [] = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _ == _ = False
```

   Using Haskell's evaluation strategy as introduced in the lecture, evaluate the following expressions step-by-step as far as possible. Indicate infinite reductions by "..." as soon as nontermination becomes apparent.

   (a) `(\f g -> g .  map f) (+1) head odds`

   (b) `False || inf == inf`

2.3 Infinite data structures

1. [1, p. 443] The *Sieve of Eratosthenes* is an ancient algorithm for generating prime numbers. It works by cancelling out all the multiples of numbers, once they are established as prime. The primes are the only elements which remain in the list.

   (a) Define a function `sieve :: [Integer] -> [Integer]` that establishes the first element x of the provided list as prime and cancels out all multiples of x. Given any list of integers xs as first argument, `sieve` should return the list of primes in xs.

   (b) Define `primes :: [Integer]` using your definition of `sieve`.

2. [1, p. 447] Give a definition of the function

   ```
   factors :: Integer -> [Integer]
   ```

   which returns a list containing the factors of a positive integer. For instance,

   ```
   factors 12 = [1,2,3,4,6,12]
   ```

   Using `factors`, define the list of numbers whose only prime factors are 2, 3, and 5, the so-called *Hamming numbers*:

   ```
   hamming = [1,2,3,4,5,6,8,9,10,12,15,...
   ```

References

[1] Simon Thompson. *Haskell - the craft of functional programming.* 3rd ed. Pearson, 2011.