Technical University of Munich
Chair for Logic and Verification
Prof. Tobias Nipkow, Ph.D.
J. Rädle, L. Stevens, K. Kappelmann, MC M. Eberl

WS 2019/20
24.01.2020
Deadline: 04.02.2020, 23:59

Functional Programming and Verification

Sheet 14

Tutorial Exercises

Exercise T14.1 Redexes

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

1. `1 + (2 * 3)`
2. `(1 + 2) * (2 + 3)`
3. `fst (1 + 2, 2 + 3)`
4. `fst (snd (1, 2 + 3), 4)`
5. `(\x -> 1 + x) (2 * 3)`

Exercise T14.2 Reductions

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
```

Exercise T14.3 Nooooooonacci

The lecture presented the following implementation of `fib` which produces an infinite list containing all Fibonacci numbers, i.e. `fib = [0,1,1,2,3,5,8,13,...]`.

```
fib :: [Integer]
fib = 0 : 1 : fib'
  where
    fib' = zipWith (+) fib (tail fib)
```

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?

Now, consider an alternative implementation for `fib`.

```
fib2 :: [Integer]
fib2 = map f [0..]
  where
    f 0 = 0
    f 1 = 1
    f n = fib2 !! (n - 1) + fib2 !! (n - 2)
```

Compare the latter implementation with the former 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:

1. Come up with a function `zipWithN ([a] -> b) -> [[a]] -> [b]` and define `nonacci` analogously to `fib`.
2. Use `fib2` as a template to define `nonacci`.

Homework

You need to collect 4 out of 5 points (P) to pass this sheet.

Exercise H14.1 Collatz Strikes Back [3P]

In the tutorial, we saw a very Haskellian take on the definition of the Fibonacci numbers. To develop this further, we consider the more complicated (at least from the perspective of a number theorist) Collatz function which is defined as

$$f(n) = \begin{cases} 1 & \text{if } n = 1, \\ f(\frac{n}{2}) & \text{if } n \text{ is even}, \\ f(3n+1) & \text{otherwise.} \end{cases}$$

The Collatz conjecture postulates that for every $n \in \mathbb{N}$ it holds that $f(n)$ terminates after finitely many recursive calls. As the arguments to the recursive calls may increase in the odd case, it may take a number of steps to reach 1. For example, if we start with $n = 12$ the sequence of arguments is $12, 6, 3, 10, 5, 16, 8, 4, 2, 1$, i.e. it takes 9 steps to reach 1. The goal is now to lazily generate a list that contains the number of steps it takes to reach 1 for all $n \in \mathbb{N}$. By convention, we set the number of steps for $n = 0$ to 0.

Computing this list can be made efficient by memoising the step count for each number we encounter. To avoid the expensive index lookup of lists, we use a tree as a memoisation data structure. As the list of all step counts is infinite, the tree also has to be infinite. In Haskell, we define the tree as follows:

```
data Tree a = Node a (Tree a) (Tree a)
```

Note that it is not possible to actually construct a `Tree` but the data type is still useful if we partially evaluate a `Tree` that is assembled lazily. Your first task is to define a function

```
nats :: Tree Integer
```

that returns a tree containing each natural number exactly once and where the children of a node should always have larger value than the node itself. Furthermore, define an operator

```
(!!!) :: Tree a -> Integer -> a
```

that performs a lookup in the tree, i.e. `nats !!! i` should return `i`. Finally, define a function `collatz_steps_list :: [Integer]` that uses `nats` and `(!!!)` to efficiently construct the list of all step counts of the collatz function in a lazy fashion.

Exercise H14.2 Probably approximately correct [1: 1P,2+3: 1P]

On exercise sheet 8, you wrote a library for polynomials, including a function that determines the number of roots within a range by applying Sturm's theorem. In this exercise, your task is to actually find those roots.

1. We begin by implementing a function to find a single root. We use the first derivative f' of our polynomial f and a guess x_n to find a new (better) guess for the root using this formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1}$$

   This is Newton's method. Use it to implement a function

```
newton :: (Eq a,Fractional a) => (Poly a) -> a -> [a]
```

   where `newton f guess` returns an infinite list of approximations of the root, starting with the initial `guess`.

   Note: Be careful about division by 0.

2. Use Newton's method to write a function

```
findRoots :: (Poly Rational) -> (Rational,Rational) -> [[Rational]]
```

   such that `findRoots f (low,high)` returns a list of infinite approximations of all roots of the polynomial `f` in the (exclusive) range `[low..high]`.

   Hint: Use the `countRootsBetween` function from exercise 8 and repeatedly split the search space.

3. Write a function

```
approxRoots :: (Poly Rational) -> (Rational, Rational) ->
                Rational -> [Rational]
```

   where `approxRoots f (low,high) eps` returns a list of approximations of roots of `f` in the range `[low..high]` such that for each approximate root `x`, `f(x)` is at most $\pm$`eps`.

> Mathematics may not be ready for such problems.
>
> — Paul Erdős (about the Collatz conjecture)
>
> This is an extraordinarily difficult problem, completely out of reach of present day mathematics.
>
> — Jeffrey Lagarias (about the Collatz conjecture)