ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A. Lochbihler and P. Müller

Formal Methods and Functional Programming

Exercise Sheet 3: Induction, Lists, and Higher-order Functions

Submission deadline: March 10th, 2014 (before 11:00 am)

Assignment 1:

(a) Write down the induction scheme for natural numbers as a proof rule.

Hint: The conclusion of your rule should be $\Gamma \vdash \forall n \in Nat.\ P(n)$.

(b) On the first exercise sheet, you saw two implementations for computing the Fibonacci numbers. Namely, Louis Reasoner wrote the Haskell program

```
fibLouis :: Int -> Int
fibLouis 0 = 0                                  -- fibLouis.1
fibLouis 1 = 1                                  -- fibLouis.2
fibLouis n = fibLouis (n-1) + fibLouis (n-2)    -- fibLouis.3
```

whereas Eva La Tour wrote the Haskell program

```
fibEva :: Int -> Int
fibEva n = fst (aux n)                          -- fibEva.1
    where aux 0 = (0,1)                         -- aux.1
          aux n = next (aux (n-1))              -- aux.2
          next (a,b) = (b, a+b)                 -- next.1
```

Prove that `fibLouis` and `fibEva` compute the same function on the naturals, i.e., show that

$$\forall n \in Nat.\ \texttt{fibLouis}\ n = \texttt{fibEva}\ n$$

You first have to prove that

$$\forall n \in Nat.\ \texttt{aux}\ n = \big(\texttt{fibLouis}\ n, \texttt{fibLouis}\ (n+1)\big)$$

using induction over n.

Note: Identify the predicate $P(n)$ used in the induction. Use the linear equational reasoning style and justify each reasoning step by referring to the equation names given in the comments above.

Assignment 2:

The *One-Time Pad* is a provably secure encryption scheme provided the key is only used once. A message of length n is encrypted with a key of the same length by performing a bit-wise XOR. Every bit of the key is set independently of all others to `True` with probability 0.5. The encrypted message is decrypted by performing bit-wise XOR with the same key again. Here we represent messages and keys as lists of Booleans, i.e., they have type `[Bool]`.

(a) Implement a function `otp` that takes a key and a message and performs the operation described above. You may assume that the key and the message always have the same length. Do you see a way to do this using `zip` and `map`?

(b) Is there a single function that can be used instead of `zip` and `map` here? (Tip: Use `www.haskell.org/hoogle` to search for a function with the right type.)

Example: With `key = [False, False, True]` and `msg = [False, True, True]` we have `otp msg key = [False, True, False]`

Assignment 3:

A natural number $n \geq 2$ is prime iff its only divisors are 1 and n. In other words n is prime iff

$$\{x \mid 1 \leq x \leq n \text{ and } n \bmod x = 0\} = \{1, n\}\,.$$

(a) Use list comprehension to turn this definition into an executable primality test.

(b) Write a function that returns the list of all primes up to a given number `m`.

(c) Write a function that returns the list of the first `m` primes.

Assignment 4:

In the lecture you have seen Haskell implementations for Insertion Sort and Quicksort. In this assignment you will have to implement *Merge Sort* in Haskell.

Recall: Merge Sort is based on the divide-and-conquer principle. First, it splits a list in two halves and sorts these lists separately. In the conquer step, it merges the two sorted lists. Note that this can be done recursively by comparing the two heads of the lists.

Implement a Haskell function `mergeSort :: [Int] -> [Int]` that sorts an integer list in ascending order by using Merge Sort. To split the list you can define a function that alternates putting elements into two lists, sort of like a zipper.

Assignment 5:

(a) Let `g` be the function defined as `g x y = 1 + y`. Evaluate `foldr g 0 [1,2,3]` by hand. What function is computed if `[1,2,3]` is replaced by an arbitrary list?

(b) What function is computed by `foldr (:) []`? Prove your claim!

Assignment 6:

(a) Define a function `split :: Char -> String -> [String]` that splits a string, which consists of substrings separated by a separator, into a list of strings. Examples:

```
split '#' "foo##goo" = ["foo","","goo"]
split '#' "#" = ["",""]
```

(b) Define a function `isASpace :: Char -> Bool` that returns `True` when the character under consideration is a whitespace (just considering `' '` is enough for now).

Define a function `toWords :: String -> [String]` that takes a string and creates a list of the words in that string. A word is a string consisting of consecutive non-whitespace characters (note: a single character is a word already, but at least one character is required, so no empty words). Whitespace is the delimiter between words and does not show up in the resulting list. Examples:

```
toWords "This is a sentence." = ["This","is","a","sentence."]
toWords "  lots of white    " = ["lots","of","white"]
```

(c) Define a function `countWords :: String -> Int` that counts the number of words in a String.