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

WS 2020/21
27.11.2020
Deadline: 07.12.2020, 23:59

Functional Programming and Verification

Sheet 4

Pen and paper proofs are out of fashion. We use *Check your Proof* (CYP) to formally verify our proofs. CYP proofs are split into theory files (*.cthy) and proof files (*.cprf). The former contains the background theory and statement of the goal, the latter the proof of the goal.

In order to check your proofs, you can clone CYP and run it on your machine as follows:

```
git clone https://github.com/lukasstevens/cyp
cd cyp
stack run cyp <pathToTheoryFile> <pathToProofFile>
```

Note that each proof step must use *one* of the defining equations or axioms of the theory file or the inductive hypothesis (IH). You have to state the justification for each step. The syntax is described in the CYP repository. You can also checkout the test examples.

You can install the CYP syntax highlighting plugin for VSCodium or syntax highlighting plugin for VIM written by our tutors.

Tutorial Exercises

You can find the template on our website.

Exercise T4.1 Hello Type Constraints

Without using any list library functions (except the cons operator `(:)`), implement the following functions:

a) `lSub :: Num a => [a] -> [a]` takes a list $[x_1, \dots, x_n]$ and returns the list $[x_1 - x_2, x_2 - x_3, \dots, x_{n-1} - x_n, x_n]$.

b) `noDupSnoc :: Eq a => [a] -> a -> [a]` takes a list $[x_1, \dots, x_n]$ and an element y and returns the list $[x_1, \dots, x_n, y]$ if y is not already contained in the list; otherwise, it returns the input list.

c) `addAbsLt :: Num a => Ord a => [a] -> a -> a` that takes a list $[x_1, \dots, x_n]$ and an element y and returns $\sum_{i \in \{i | x_i < y\}} |x_i|$.

   Hint: use `abs` from the typeclass `Num` to compute the absolute value of a number.

Exercise T4.2 Structural Induction □

We define two functions `snoc :: [a] -> a -> [a]` and `reverse :: [a] -> [a]` as follows:

```
snoc [] y = [y]
snoc (x : xs) y = x : snoc xs y

reverse [] = []
reverse (x : xs) = snoc (reverse xs) x
```

a) Use structural induction to prove the following equation:

```
reverse (snoc xs x) = x : reverse xs
```

b) Use structural induction to prove the following equation:

```
reverse (reverse xs) = xs
```

Exercise T4.3 Accumulators

Again, *without* using any list library functions (except the cons operator `(:)`), implement the following functions using an accumulator based approach:

a) Rewrite the function `addAbsLt` from the previous exercise.

b) `maxAbs` takes a list xs and returns $\max\{|x| \mid x \in xs\}$ if xs is not empty and 0 otherwise.

c) `countSigns` returns the number of negative, zero, and positive elements in a given list. For example,

```
countSigns [-1,0,0,-5,2,0] = (2,3,1)
```

Hint: use `signum` from the typeclass `Num` to compute the sign (-1, 0, or 1) of a number.

d) (Optional) `ltAndGt` returns `True` for a given list `xs` and element `y` if and only if `xs` contains both an element less than and greater than `y` while iterating the list only once.

Exercise T4.4 The Anti-Pattern

In the lecture (on slide 111), an example of an accumulating parameter was given: Function `ups :: Ord a => [a] -> [[a]]` was defined that chops up a list into maximal ascending sub-lists. The given solution is an anti-pattern. A solution without an accumulating parameter is simpler (and faster!). Define function ‘ups‘ directly, without using any auxiliary functions or aditional parameters.

Homework

You need to collect 7 out of 10 points (P) to collect a coin.

Exercise H4.1 (**Wettbewerb**) Hole In One [5P]

It is *Wettbewerb*-time again. This time, we ask you to check the syntax of a string for the rules specified below. The function is called `xmlLight` and has the type `String -> Bool`. In addition to returning `True` if and only if the string is well-formed according to below rules, your solution should be code-golfed to claim the top spot of prestigious Wettbewerb. This means that the goal is to implement the function using as few tokens as possible.

Here are our simplified XML rules:

1. For every opened `<tag>`, you need a closed `</tag>` with the same tag-ID.
2. You can assume that tag-IDs are non-empty.
3. Trailing whitespaces following the tag-ID but preceeding the closing bracket are allowed and not part of the tag-ID. Other whitespaces in the tag are not allowed. As an example, `<a ></a>` is valid while `<a b></a b>` is not valid.
4. The brackets `<` and `>` may not be used in any other way.
5. A tag-ID will not contain the character "/".
6. Each tag opens a new level and all tags started within this level must be closed within it using the same order. Therefore `"<A><B></B><C></C></A>"` is valid and `"<A><B></A></B>"` is invalid.
7. Plaintext may be contained between an opened and closed tag but also outside of any xml contained in the string.
8. There are no other rules; XML attributes are not allowed (i.e. only the tag-ID is inside the bracket).

This exercise was designed and implemented in coorporation with our tutors. Special thanks to all of them!

Remark: For a deeper dive into parsing including techniques beyond the scope of the lecture, we recommend this excellent tutorial that teaches you how to implement a JSON parser in 111 lines of Haskell.

Exercise H4.2 Catena catenae [2P]

Use structural induction to prove the following identity:

```
concat (xss ++ yss) = concat xss ++ concat yss
```

Check the template cthy for the definitions of `concat` and `++`. You may also use the fact that `++` is associative in your proof, the corresponding axiom is defined in the template as well.

Exercise H4.3 Nullsummen [3P]

We define a summation sum for two lists as follows:

```
vectorSum [] ys = []
vectorSum xs [] = []
vectorSum (x:xs) (y:ys) = x + y : vectorSum xs ys
```

Use structural induction to prove the following identity:

```
vectorSum xs (replicate (length xs) 0) = xs
```

No more ”proofs” that look more like LSD trips than coherent chains of logical arguments.

— Scott Aaronson