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
22.11.2019
Deadline: 02.12.2019, 23:59

Functional Programming and Verification

Sheet 6

IMPORTANT: We use the cyp ("Check your proof") format for our proofs. Use the templates from the submission website to check your proofs by structural induction. The submission system can check these proofs against the provided templates. Proofs by computation induction cannot automatically be checked by the system but still have to be written in the cyp format.

Each proof step must use *one* of these defining equations, the inductive hypothesis (IH), or an axiom. You have to state the justification for each step.

Tutorial Exercises

Exercise T6.1 Be More General

We define functions `sum :: Num a => [a] -> a` and `(++) :: [a] -> [a] -> [a]` as follows:

```
sum xs = sum_aux xs 0

sum_aux [] acc = acc
sum_aux (x:xs) acc = sum_aux xs (acc+x)

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

Use structural induction to show that

```
sum (xs ++ ys) = sum xs + sum ys
```

Exercise T6.2 Computation Induction

We define the functions `sum :: Num a => [a] -> a` and `sum2 :: Num a => [a] -> [a] -> a`:

```
sum [] = 0
sum (x:xs) = x + sum xs

sum2 [] [] = 0
sum2 [] (y:ys) = y + sum2 ys []
sum2 (x:xs) ys = x + sum2 xs ys
```

Use computation induction to show that `sum2 xs ys = sum xs + sum ys`.

Note that cyp does not support computation induction. We nevertheless encourage you to write your proof in a style similar to cyp's format. In particular, you should always state what you are proving and any inductive hypotheses. As an example, given the following definition

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

and a proposition `P xs ys`, we suggest you to write:

```
Lemma: P xs ys
Proof by myFunc-induction on xs ys
Case 1
  To show: P xs []
  Proof
  ....
  QED
Case 2
  To show: P [] (y:ys)
  IH: P (y:ys) []
  Proof
  ...
  QED
Case 3
  To show: P (x:xs) (y:ys)
  IH1: P (y:xs) ys
  IH2: P xs ys
  Proof
  ...
  QED
QED
```

Exercise T6.3 Iteration

1. Write a function `iter :: Int -> (a -> a) -> a -> a` that takes a number n, a function f, and a value x and applies f n-times with initial value x, that is `iter` n f x computes $f^n(x)$. A negative input for n should have the same effect as passing $n = 0$. For example, `iter 3 sq 2 = 256`, where `sq x = x * x`
2. Use `iter` to implement the following functions *without* recursion:
   a) Exponentiation: `pow :: Int -> Int -> Int` such that `pow` n $k = n^k$ (for all $k \geq 0$).
   b) The function `drop :: Int -> [a] -> [a]` from Haskell's standard library that takes

a number k and a list $[x_1, \ldots, x_n]$ and returns $[x_{k+1}, \ldots, x_n]$. You can assume that $k \leq n$.

c) The function `replicate :: Int -> a -> [a]` from Haskell's standard library that takes a number $n \geq 0$ and a value x and returns the list $[\underbrace{x, \ldots, x}_{n\text{-times}}]$.

Homework

IMPORTANT: The submission system can process multiple file at once. You have to upload all files (h61.cprf, h62.cprf, and Exercise_6.hs) at once. Moreover, you have to use the provided filenames when uploading, that is do not change the filenames of the template files.

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

The needed background theories/definitions (*.cthy files) for the following exercises can be found on moodle.

Exercise H6.1 It Is All The Same [2P]

Remember the function `addAbsLt :: Num a => Ord a => [a] -> a -> a` from sheet 4 that takes a list $[x_1, \ldots, x_n]$ and an element y and returns $\sum_{i \in \{i | x_i < y\}} |x_i|$. We implemented two versions of it: one that does use an accumulator and one that does not. Since you now possess the power to prove that they are equivalent, we challenge you to show that indeed

```
addAbsLt xs y = itAddAbsLt xs y
```

using structural induction so that you can check your proof by cyp.

Exercise H6.2 Fight Against Inequality [2P]

We define:

```
length [] = 0
length (x:xs) = 1 + length xs

countGt [] ys = 0
countGt (x:xs) [] = length (x:xs)
countGt (x:xs) (y:ys) = if x > y then 1 + countGt (x:xs) ys
  else countGt (y:ys) xs
```

Show that `countGt xs ys <= length xs + length ys` using computation induction. Follow the cyp-like template as described in the tutorials. Partial credits can be rewarded for almost correct, comprehensible proofs.

Note: Given a rule `P x ==> y <= z` with name `myRule` and a proof `p` of `P x`, you can use `(by myRule OF p)` to apply the inequality between `y` and `z`. For example:

```
axiom leAddMono: y <= z ==> x + y <= x + z
axiom zeroLeOne: 0 <= 1

Lemma: 0 + 0 <= 0 + 1
Proof
```

```
                                        0 + 0
  (by leAddMono OF zeroLeOne) <=        0 + 1
QED
```

Exercise H6.3 FRACTRAN (**Competition**) [1P]

For next year's FPV lecture, the *Übungsleitung* wants to move to a more elegant programming language than Haskell. After much deliberation, the choice has fallen on FRACTRAN, invented by John Horton Conway.

A FRACTRAN program is simply a list of positive fractions and a positive integer n. To run the program, one finds the first fraction f in the list such that nf is an integer and then replaces n by nf. This process is iterated until no fraction in the list produces an integer when multiplied by n. The final n is the output of the program.

Puzzlingly, there is an embarrasing dearth of production grade FRACTRAN interpreters. Since the *Übungsleitung* cannot be bothered to write one themselves, we pawn this task off on you.

Write a function `traceFractran :: [Rational] -> Integer -> [Integer]` such that `traceFractran rs n` executes the program given by the list of fractions `rs` with the starting value `n`, returning a list containing the values of `n` after each iteration. You may assume that the input is well formed, i.e. that `rs` contains positive rational numbers and that `n` is a positive integer.

An example FRACTRAN program is `[3%2]`, which adds two integers a and b encoded as 2^a3^b, producing the result 3^{a+b}. The invocation

```
traceFractran [3%2] 144
```

produces the result `[144,216,324,486,729]`, where $144 = 2^43^2$ and $729 = 3^6$.

A more complex example is the program

```
primeprog = [17%91,78%85,19%51,23%38,29%33,77%29,95%23,77%19,1%17,
             11%13,13%11,15%14,15%2,55%1]
```

which produces the prime numbers when given the initial value 2. Concretely, all n produced during the execution of the program that are powers of two will have successive prime numbers as exponents. The invocation

```
take 10 [x | x <- traceFractran primeprog 2, isPowerOfTwo x],
```

where `isPowerOfTwo` returns `True` iff its argument is a power of 2, yields

```
[2,4,8,32,128,2048,8192,131072,524288,8388608]
```

which is

```
[2^1,2^2,2^3,2^5,2^7,2^11,2^13,2^17,2^19,2^23]
```

Donald Knuth

John Horton Conway

Good FRACTRAN interpreters share many properties with FRACTRAN programmes: they are concise, esoteric, and mystically beautiful. The submissions will hence be judged by their number of tokens. For reference, a naive implementation using only material covered in the lecture has 49 tokens, an optimized version by the MC Sr. makes due with 30.

Important: If you submit a competition exercise, you agree that we are allowed to publish your name as part of the competition on our website. If you just want to submit a competition exercise as part of your homework without taking part in the competition, you can just remove the {-WETT-}...{-TTEW-} comments of your submission.

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> — Donald Knuth