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
06.12.2019
Deadline: 16.12.2019, 23:59

Functional Programming and Verification

Sheet 8

Tutorial Exercises

Exercise T8.1 Is This a Type Class?

Define a type `Fraction` with one constructor `Over :: Integer -> Integer -> Fraction` to represent fractions over integers.

1. Define an instance `Num Fraction`.

   ```
   class Num a where
     (+), (-), (*)        :: a -> a -> a
     negate               :: a -> a
     fromInteger          :: Integer -> a
     -- The functions 'abs' and 'signum' should
     -- satisfy the law: abs x * signum x = x
     abs                  :: a -> a
     signum               :: a -> a
   ```

   After each operation that may increase the numerator/denominator, the fraction should be reduced as far as possible.

2. Haskell can automatically derive instances for `Eq` (and also `Show`) using `deriving Eq`. Is this automatically derived instance useful in this case? If not, what should it look like?

Note:

- `Num` contains no division operator. `(/)` is defined by the typeclass `Fractional`. In a more extensive library, one would hence declare `Fraction` as an instance of `Fractional`.
- The function `fromInteger` is used by Haskell to embed integers into the required type. The expression `3 :: Fraction` is hence equivalent to `(fromInteger 3) :: Fraction`. The function `fromRational` in `Fractional` does the same for decimal numbers (e.g. `3.14`).

Exercise T8.2 I Can't See The Forest For The Trees

1. In a binary tree, we can only descend to the left or to the right. Define a data type `Tree` with constructors `Leaf` and `Node` for binary trees.
2. Implement a function `sumTree :: Num a => Tree a -> a` that returns the sum of all values in a tree.
3. Implement a function `cut :: Tree a -> Integer -> Tree a` that cuts off a tree after a given height.

4. Implement a function `foldTree :: (a -> b -> b) -> b -> Tree a -> b` that folds a function over a tree. The fold should process the right children of a node first, then the node itself, and lastly the left children.
5. Use `foldTree` to implement a function `inorder :: Tree a -> [a]` that returns all elements of a tree in left to right order.
6. Use `foldTree` to implement a function `findAll :: (a -> Bool) -> Tree a -> [a]` that returns an all elements of a tree that satisfy a given predicate.

Exercise T8.3 by simp

Give alternative definitions with as few parameters to the left of the equality symbol as possible for the following functions. Rewrite all λ-expressions and do not introduce new ones. Make use of the combinators in Data.Function.

```
f1 xs = map (\x -> x + 1) xs
f2 xs = map (\x -> 2 * x) (map (\x -> x + 1) xs)
f3 xs = filter (\x -> x > 1) (map (\x -> x + 1) xs)
f4 fs = foldr (\f acc -> f acc) 0 (map (\f -> f 5) fs)
f5 f g x = f (g x)
f6 f (x,y) = f x y
f7 f x y z = f z y
```

Homework

You need to collect 7 out of 9 points (P) to pass this sheet.

Exercise H8.1 Homegrown Trees [1: 1P, 2a: 1P, 2b+c: 1P, 2d: 1P]

We define the recurisve datatype tree with

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

for which we provide you with a function `showGraphviz :: Show a => Tree a -> String` that prints a tree in dot-format.

1. For this definition of trees write a function `symmetric :: Eq a => Tree a -> Bool` that determines whether a tree is symmetric along its vertical axis. See Figure 1 for examples.

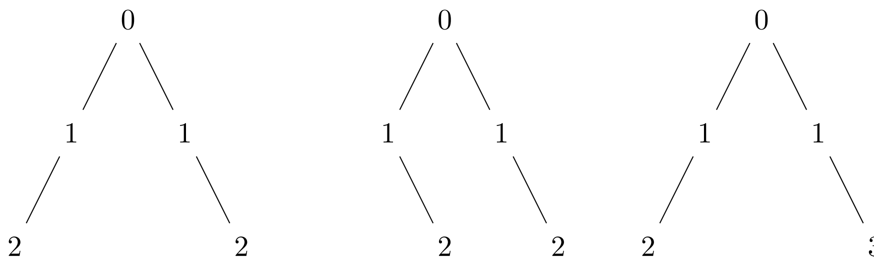

Figure 1: The first tree is symmetric while the second and third a not. For the second tree the shapes of the subtrees do not match. In the third tree the shape of the subtrees match but they contain different nodes.

2. The remainder of this task is about binary search trees (BST). These are trees where all the nodes in the left subtree of a node have a strictly smaller value than the node itself whereas the nodes in the right subtree have strictly larger value. This holds for all nodes in the tree.

   a) Write a function `isBST :: Ord a => Tree a -> Bool` that checks whether a tree is a binary search tree.

   b) Write a function `isAlmostComplete :: Tree a -> Bool` that determines whether a tree `t` is an almost complete binary tree, i.e. every level of the tree `t`, except possibly the last, is completely filled with nodes.

   c) Implement a function `buildBST :: Ord a => [a] -> Tree a` that builds an almost complete BST out of a sorted list.

   d) Finally, come up with a procedure

   ```
   rangeSubtree :: Ord a => Tree a -> (a, a) -> Tree a
   ```

that only retains those nodes in the tree whose values are within the specified (inclusive) range. Subtrees whose parents are not retained may need to be re-attached such that the result is still a BST.

Exercise H8.2 (**Competition**) SCHUFA Madness [1P]

Just recently, the MC Jr had to request a *SCHUFA-Selbstauskunft*[1] for his unpleasant adventure "Find an affordable place in Munich". One day, after one of his very long strolls on the campus, he received the letter expecting nothing else than a perfect score. Surprisingly, the SCHUFA just awarded him a 96%, meaning "Geringes bis überschaubares Risiko". Bewildered, he dug through his past few bank statements, discovering that his balance had more swings than a vibrating guitar string.

He very quickly realised that the SCHUFA does not provide any transparent measure as to how such swings influence the final SCHUFA-score. He hence decided to invent his own measure of financial accountability: the "SHOEFA-score". The SHOEFA-score counts the number of left-to-right sign changes in a sequence of balances. Any zeros are ignored in the scoring process, e.g. [1,0,1] has a SHOEFA-score of 0 and [1,0,-1] has a SHOEFA-score of 1.

The MC Jr wants to publish an app (that he then sells for $$$) that computes this score. He hence asks you for your help to implement a function

```
shoefa :: (Num a, Ord a) => [a] -> Int
```

that computes the SHOEFA-score with the prospect of some shares ($ↄ) for the solutions taking the fewest number of tokens.

> **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.

Exercise H8.3 Sturm und Drang [1+2: 1P, 3: 1P, 4+5+6: 1P, 7+8+9: 1P]

The goal of this exercises is to extend our (Laurent) polynomial library from sheet 3. We use the same encoding as on sheet 3, except that this time, we work with polynomials over arbitrary integral domains. An integral domain is a non-zero (i.e. the carrier is not a singleton set) commutative ring without zero divisors, that is $ab \neq 0$ whenever $a \neq 0, b \neq 0$.

For this, we define the following polynomial type: `data Poly a = Poly [(a, Integer)]`. We will assume further constraints on the type variable `a` as needed when implementing our functions. Most notably, we will use `Num a` constraints for our carrier `a`. Strictly speaking, `Num a` does not require `a` to be an integral domain, but we might as well just assume it does for this exercise.

[1] of course he made use of § 34 BDSG to request a free one

We already provide you with functions for addition (`polyAdd`) and derivation of a polynomial (`polyDeriv`) in the template. For tasks 5–9, you can assume that all exponents of the polynomial are nonnegative.

1. Create an instance `Show (Poly a)` assuming that there is an instance `Show a`. Examples:

   ```
   show (Poly [(1,-2),(2,3)] :: Poly Int) = 1x^{-2} + 2x^{3}
   show (Poly [] :: Poly Int) = 0
   ```

2. Write a function `polyShift :: (Eq a, Num a) => (a, Integer) -> Poly a -> Poly a` that multiplies a monomial cx^e with a polynomial.
3. Create an instance `Num (Poly a)` given that `Eq a` and `Num a`. You can define the functions `abs` and `signum` as `undefined`.
4. Write functions

   ```
   leadCoeff :: Num a => Poly a -> a
   degree :: Poly a -> Integer
   ```

   that return the leading coefficient and degree of a polynomial, respectively. The leading coefficient of a polynomial is the coefficient of the monomial with the highest exponent contained in the polynomial. Similarly, the degree of a polynomial is the exponent of the monomial with the highest exponent contained in the polynomial. Conventions:

   ```
   leadCoeff [] =  0
   degree    [] = -1
   ```

5. Write a function

   ```
   polyDivMod :: (Fractional a, Eq a) => Poly a -> Poly a ->
                 (Poly a, Poly a)
   ```

   such that `polyDivMod p q` computes the quotient and remainder of the polynomial division p/q. If we call the quotient s and the remainder r then s and r are the unique polynomials with $\deg(r) < \deg(q)$ and $p = qs + r$. Conventions: $p/0 = 0$ and $p \bmod 0 = p$. The pseudocode for polynomial long division in Figure 2 might be of help.
6. Write functions

   ```
   polyDiv :: (Fractional a, Eq a) => Poly a -> Poly a -> Poly a
   polyMod :: (Fractional a, Eq a) => Poly a -> Poly a -> Poly a
   ```

   that return the quotient and remainder of the polynomial division p/q, respectively.
7. Write a function `polyEval :: Num a => Poly a -> a -> a` that evaluates a polynomial at a given position.
8. The *Sturm chain* or *Sturm sequence* – named after Jacques Charles François Sturm – of a

```
function p / q:
  require q /= 0
  s = 0
  r = p         -- at each step p = q * s + r
  while r /= 0 && degree r >= degree q:
     t = lead r / (lead q) -- divide the leading terms
     s = s + t
     r = r - t * q
  return (s, r)
```

Figure 2: Pseudo code for polynomial long division. Source: https://en.wikipedia.org/wiki/Polynomial_long_division

polynomial p is the sequence of polynomials recursively defined by

$$\begin{aligned} p_0 &= p, \\ p_1 &= \text{polyDeriv}(p), \\ p_{i+1} &= -\text{polyMod}(p_{i-1}, p_i). \end{aligned}$$

Write a function `sturm :: (Fractional a, Eq a) => Poly a -> [Poly a]` that computes the Sturm sequence up to but excluding point i where $p_i = 0$.

9. Let $p(x)$ be a polynomial over $\mathbb{R}$ and $p_0, \ldots, p_i$ be its Sturm sequence. Sturm's Theorem states that the number of distinct real roots of p in the half-open interval $(a, b]$ is given by the number of sign changes in $p_0(a), \ldots, p_i(a)$ subtracted by the number of sign changes in $p_0(b), \ldots, p_i(b)$ provided that neither a nor b is a multiple root of p.

   Use Sturm's Theorem to write a function

   ```
   countRootsBetween :: Poly Rational -> Rational -> Rational ->
                        Int
   ```

   such that `countRootsBetween p a b` computes the number of distinct real roots of $p \in \mathbb{Q}[x]$ in the interval $(a, b]$. You can assume that neither a nor b is a multiple root of p.

   Hint: Use the SHOEFA-score

> Trees sprout up just about everywhere in computer science.
>
> — Donald Knuth in "The Art of Computer Programming" Vol. IV