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

WS 2020/21
18.12.2020
Deadline: 11.01.2021, 23:59

Functional Programming and Verification

Sheet 7

Exercise T7.1 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
   ```

   (Optional) 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?

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 T7.2 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. (Optional) Use `foldTree` to create an instance of `Foldable Tree`.

Exercise T7.3 by simp

Give alternative definitions with as few parameters to the left of the equality symbol as possible for the following functions. Simplify 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
f8 f g x y = f (g x y)
```

Homework

As one can tell based on the fabulous snowflakes frame of this document, this week is rather unique. Indeed, it is our Christmas special, which is why we host a fun game exercise! This special game can be found on exercise sheet 08, and you can not only take part in a fun online tournament, but also collect an additional coin and sweet Wettbewerb points for it.

However, we also need to do serious business, and so there are some (not so special) exercises for you below to collect a coin too.

You need to collect 11 out of 15 Christmas stars ($\star$) to collect a coin.

Exercise H7.1 Christmas Trees! [a–d,f: 1$\star$ each, e: $\star\star$]

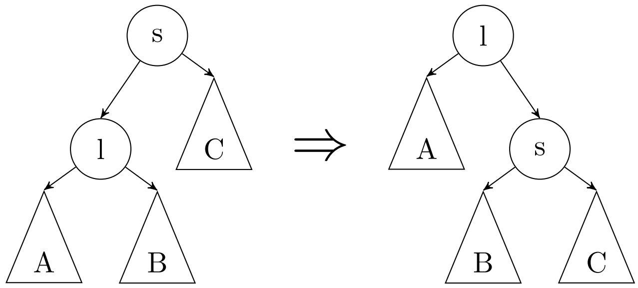
We define a datatype of trees and, in addition, directions to navigate the tree. Here, `L` corresponds to navigating to the left subtree while `R` corresponds to the right subtree.

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
data Direction = L | R
```

a) Write a function `subtreeAt :: [Direction] -> Tree a -> Maybe (Tree a)` that navigates to the subtree specified by the list of directions and returns it (if it exists). For example:

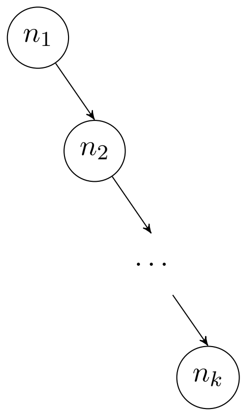
```
subtreeAt [L,R] (Node (Node Leaf 'a' (Node Leaf 'b' Leaf)) 'c' Leaf)
    = Node Leaf 'b' Leaf
```

b) Define a function `rotateR :: Tree a -> Maybe (Tree a)` that performs a right-rotation on the tree. If that is not possible, return `Nothing`. The picture below exemplifies a right-rotation.



c) Now combine the approaches of the two previous functions in order to write a function `rotateRAt :: [Direction] -> Tree a -> Maybe (Tree a)` that performs a right-rotation at the location specified by the list of directions.

The rest of the exercise concerns itself with right-linear chains which are a degenerate version of trees that are isomorphic to lists. Drawn as a tree they look as follows:



d) Define a function `isRchain :: Tree a -> Bool` that returns `True` if, and only if, the given tree is a right-linear chain.

e) Every tree can be transformed into a right-linear chain using only right-rotations. In fact, you need at most n rotations where n is the number of nodes in the tree. Implement a function `rotateRchainDirs :: Tree a -> [[Direction]]` that returns a list l with $|l| \leq n$ where each entry of the list is a list of directions specifying the subtree where a right-rotation should be performed.

f) Write a function `rotateRchain :: [[Direction]] -> Tree a -> Maybe (Tree a)` that performs all rotations in the order that the first argument specifies, i.e. it is expected that `rotateRchain (rotateRchainDirs t) t` returns a right linear chain.

Exercise H7.2 Finite Data, Infinite Fun [a–d: 1⋆ each, e+f: 2⋆ each]

We define a typeclass `Finite` to represent finite types. An instance `Finite a` must give an enumeration of all values of type `a` without duplicates:

```
class Finite a where
  -- finite list of all values of type a without duplicates
  enumerated :: [a]
```

We ignore that `undefined` is of any type in Haskell (i.e. `undefined :: a` for any type `a`) for this exercise unless explicitly said otherwise.

a) Assuming there is an instance `Finite a`, define an instance `Finite (Maybe a)` for any type `a`.

b) We define a data type `Time` that should store dates using the 24-hour clock convention from 00:00:00 (midnight) to 23:59:59 (one second to midnight).

```
-- Time stores dates in this order: hours, minutes, seconds
-- For example, 17:32:59 is modelled as Time 17 32 59
data Time = Time Int Int Int
```

Write an instance `Finite Time` that enumerates all valid `Time` values using the 24-hour clock convention.

c) Assuming that there are instances `Finite a` and `Finite b`, define an instance `Finite (Either a b)` for any types `a` and `b`. Recall that `Either` has exactly two constructors `Left :: a -> Either a b` and `Right :: b -> Either a b`.

d) Assuming that there are instances `Finite a` and `Finite b`, define an instance `Finite (a, b)` for any types `a` and `b`.

e) Assuming that there is an instance `Finite a`, define an instance `Finite (Tree a)` (as defined in H7.1) of trees that contain each value of type `a` exactly once for any type `a`.

f) Assuming that there are instances `Eq a`, `Finite a`, `Eq b`, and `Finite b`, define an instance `Finite (a -> b)` that enumerates the list of all total, surjective functions from `a` to `b`.

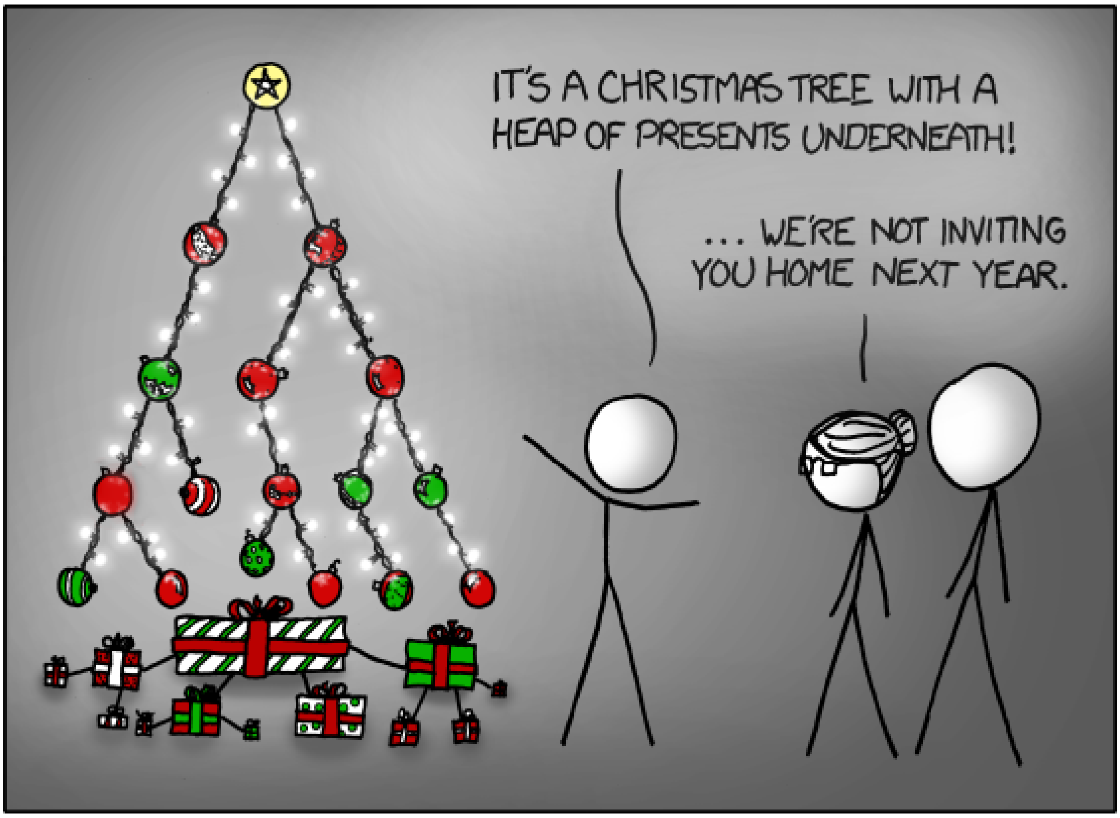
Reminder: A function `f :: a -> b` is total if it maps each value `v :: a` to some non-undefined `f v :: b`. A function `f :: a -> b` is surjective if for each `u :: b`, there is some `v :: a` such that `f v = u`.

Note: The empty function is the only function from an empty domain. The empty function should be represented by either `\x -> undefined` or `undefined`.

****

We all wish you a merry Christmas and a happy new year!

****



Source: https://www.xkcd.com/835/