Prof. Dr. Peter Thiemann
Janek Spaderna
janek.spaderna@pluto.uni-freiburg.de

Winter Term 2022/23
Issued 2022-11-11
Discussion on 2022-11-18

Functional Programming

https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2022/

Exercise Sheet 4

Exercise 1 (List functions IV – infinite lists)

Implement the following functions from `Data.List`:

- `iterate`
- `cycle`

Make sure to define `cycle` in a way that leads to a cyclic data structure like `ones` or the second version of `repeat` from the lecture.

Note The type signature of `cycle` might mention a `HasCallStack` constraint, depending on which version of the documentation you're looking at. This constraint is not required in your implementation.

Exercise 2 (Folding & laziness)

Remember the two functions `foldl` and `foldr`:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ nil []     = nil
foldl f nil (a:as) = foldl f (nil `f` a) as

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ nil []     = nil
foldr f nil (a:as) = a `f` foldr f nil as
```

Answer the following questions:

1. Which, if any, of these two functions, when used on infinite lists, (may) terminate?

   If you have identified either of the functions as potentially terminating verify your claim by providing a suitable function `f` and starting value `nil` such that folding over the infinite list `[0..]` terminates.

2. Given a finite list `nums` = [x_1, x_2, ..., x_n]. Identify a problem which occurs both with `foldl (+) 0 nums` and `foldr (+) 0 nums` and is only exacerbated with increasing `n`.

Exercise 3 (Vectors)

Define a datatype for 2D vectors with `Double` components. Write `Eq`, `Show` and `Num` instances for your type (without deriving, obviously).

Exercise 4 (Monoids)

Two typeclasses that are used quite frequently in Haskell programs are `Monoid` and its superclass `Semigroup`. They model the algebraic structures of the same name: semigroups and monoids.

A semigroup is a set with a closed, associative, binary operation. In Haskell this operation is represented by the operator `(<>)`. Every monoid is a semigroup but additionally there exists one element in the set which is a left and right identity for the binary operation. This element is called `mempty` in Haskell.

The most prominent instances for `Monoid` and `Semigroup` are lists with the binary operation being append: `(<>) = (++)` and `mempty = []`.

1. For the set of integers there exist two obvious interpretations as monoids: the binary operation can be either summation or multiplication.

   Create two new data types, `Sum` and `Product`, each wrapping an `Integer`. Write `Semigroup` and `Monoid` instances corresponding to the two interpretations. Use QuickCheck to verify that your instances follow the monoid laws.

2. A very versatile way to fold a list (or any kind of container-like data structure) is to map the elements into a monoid and combine these using the binary operation. Write the function

   ```
   foldMap :: Monoid m => (a -> m) -> [a] -> m
   ```

   Implement `sum` and `product` using `foldMap` and your datatypes from above.

For the interested Take a look through the modules `Data.Monoid` and `Data.Semigroup`. They provide a lot of wrapping types with different `Monoid`/`Semigroup` behaviors, including a more general version of your `Sum` and `Product` types.