Till Mossakowski
Benjamin Junge

5. Excercise sheet

Issued: 2023-11-14
Due: 2023-11-20 & 2023-11-21

5.1 *Comprehensible Streams*

The lecture introduced the syntactic sugar of list comprehension as a way to compactly write map and filter applications to lists. To get a firm grasp on how it can be used, implement the following streams using list comprehension only (and predicates/non recursive functions).

- The stream of all even numbers `ev` that got introduced in 4.3
- The stream of all members of the harmonic sequence `harmonic` that got introduced in 4.3 ($a_n = \frac{1}{n}$)
- The stream of all members of the palindrome sequence `palin` for Integers that got introduced in 4.3
- The stream `partsum xs :: Num a ⇒ [a] → [a]` which is the stream of all partial sums of a given sequence xs. (you may use foldr but **not** the sum function)
  e.g:

```
take 11 partsum [0..] ⇝ [0,1,3,6,10,15,21,28,36,45,55]
take 6 (partsum harmonic) ⇝
[0.0,1.0,1.5,1.8333333333333333,2.083333333333333,2.283333333333333]
```

5.2 *The Functor class and its limits*

Out of the following types which ones can be instances of the Functor class? Which ones can't and why?:

```
data Bool_ = True_ | False_
data List a = List a (List a) | Empty
data Either_ a b = Right_ a | Left_ b
data Maybe_ a = Nothing_ | Just_ a
data Pair a b = Pair a b
data LList a = LList [a] (a,a) a
```

Is there a way to make them fit the Functor class requirements?

5.3 *The Functor class and its limits continued*

Implement Functor instances for all types given in 5.2 that can be instances. If a type can't be an instance but there is a way to make it fit the requirement implement that kind of solution.

5.4 *Folding*

Consider the following type of sequences of single elements, pairs and lists of a type a:

```
data Seq a = End | Seq a (Seq a) | Lseq [a] (Seq a) | Pseq (a,a) (Seq a) deriving(Show)
```

Implement the Foldable and the Functor instance for the type Seq a.
e.g:

```
fmap (+3) (Seq 1 (Lseq [1,2,3] (Pseq (1,2) End))) ⇝ Seq 4 (Lseq [4,5,6] (Pseq (4,5) End))
foldr (+) 0 (Seq 1 (Lseq [1,2,3] (Pseq (1,2) End))) ⇝ 10
```

5.5 *Functor Functor and Foldable Foldable*

As we have seen in the previous task 5.2 there are limitations to the regular typeclasses Functor and Foldable. However, there are the two analogous classes called Bifunctor and Bifoldable that extend the idea of Functor and Foldable and which are defined as follows:

```
class Bifunctor p where
 bimap :: (a → b) → (c → d) → p a c → p b d
class Bifoldable p where
 bifoldr :: (a → c → c) → (b → c → c) → c → p a b → c
```

To use both classes in your module use ***import*** *Data.Bifoldable and* ***import*** *Data.Bifunctor.*
Familiarize yourself with the concepts of bimap and bifoldr and explain their signature.

Implement the instances Bifoldable and Bifunctor for the types Either_ and Pair defined in 5.2. [1]
e.g:

```
bimap (+2) (show) (Pair 1 10) ⇝ Pair 3 "10"
bifoldr (+) (*) 3 (Pair 2 10) ⇝ 32
bimap (+1) (+2) (Left_ 1) ⇝ Left_ 2
bimap (+1) (+2) (Right_ 1) ⇝ Right_ 3
bifoldr (+) (*) 3 (Left_ 1) ⇝ 4
bifoldr (+) (*) 3 (Right_ 1) ⇝ 3
```

5.6 * *Not enough Folding and Mapping: Functor^3 and Foldable^3*

As we have seen in 5.5 the concept of the Functor/Foldable class can be extended to the class Bifunctor/Bifoldable. However both of those classes have limitations to them like their base cousin. Consider the type

```
data Triple a b c = Triple a b c
```

If you try and define a Bifunctor or Bifoldable instance for Triple you will run into the same (or at least equivalent) problem that we had with defining a Functor instance for Pair. Sadly there is no already defined Trifunctor or Trifoldable class, but that can't stop us from defining those classes on our own.
Based on how bimap extends fmap and bifoldr extends foldr, figure out an analogues type signature for trimap and trifoldr and write them into the class template:

```
class Trifunctor p where
 trimap :: ???
class Trifoldable p where
 trifoldr :: ???
```

After defining those two classes write an instance of both for our type Triple.
e.g:

```
trifold (+) (*) (^) 2 (Tri 1 2 3) ⇝ 19
trimap show (+1) (replicate 3∘show) (Tri 2 3 4) ⇝ Tri "2" 4 ["4","4","4"]
```

[1] bimap and bifoldr are sufficient to write an instance of the corresponding class. However there are many more possible functions that can be implemented instead. For more informations see Data.Bifoldable and Data.Bifunctor