Weekly Assignments 8

Optional

Note that some tasks may deliberately ask you to look at concepts or libraries that we have not yet discussed in detail. But if you are in doubt about the scope of a task, by all means ask.

Please try to write high-quality code at all times! This means in particular that you should add comments to all parts that are not immediately obvious. Please also pay attention to stylistic issues. The goal is always to submit code that does not just correctly do what was asked for, but also could be committed without further changes to an imaginary company codebase.

W8.1 Deciding whether a number is even or odd

Consider the type of Peano natural numbers `Nat`, the corresponding singleton type `SNat`, type-level addition and the types `EvenN` and `OddN` defined as follows:

```
data Nat = Z | S Nat deriving Show

data SNat (n :: Nat) :: * where
    SZ :: SNat 'Z
    SS :: SNat n -> SNat ('S n)

type family (+) (m :: Nat) (n :: Nat) :: Nat where
    'Z   + n = n
    'S m + n = 'S (m + n)

data EvenN (n :: Nat) :: * where
    EvenN :: SNat m -> (n :~: m + m) -> EvenN n

data OddN (n :: Nat) :: * where
    OddN :: SNat m -> (n :~: 'S 'Z + m + m) -> OddN n
```

A value of type `EvenN n` provides a “witness” `m` for the “even-ness” of `n` (a proof for `m + m = n`). Similarly, a value of type `OddN n` provides a “witness” `m` for the “oddness” of `n` (a proof for `1 + m + m = n`).

Show that whether a natural number is even or odd is *decidable* by implementing a function

```
evenOddDec :: SNat n -> Either (EvenN n) (OddN n)
```

W8.2 Vectors

Consider the type of fixed-length lists ("vectors"):

```
infixr 5 :*

data Vec (n :: Nat) (a :: *) :: * where
    VNil :: Vec 'Z a
    (:*) :: a -> Vec n a -> Vec ('S n) a
```

Subtask W8.2.1

Try to derive instances of `Show`, `Eq`, `Ord`, `Functor`, `Foldable` and `Traversable` for vectors. You won't have to write any code, but you might need to add appropriate preconditions.

Subtask W8.2.2

Write an `Applicative` instance for `Vec Z` and one of the form

```
instance Applicative (Vec n) => Applicative (Vec ('S n))
```

Why can't you simply write an instance for `Vec n`? Or can you?

Subtask W8.2.3

Write an instance

```
instance (Applicative (Vec n), Monoid m) => Monoid (Vec n m)
```

Subtask W8.2.4

Write a function.

```
splitV' :: SNat m -> SNat n -> Vec (m + n) a -> (Vec m a, Vec n a)
```

that splits a vector into two parts (the opposite of `(++)`). Using class `SNatI` as defined in the lecture, define

```
splitV :: (SNatI m, SNatI n) => Vec (m + n) a -> (Vec m a, Vec n a)
```

W8.3 Matrices

Consider the following type of *matrices* (two-dimensional arrays):

```
newtype Matrix m n a = Matrix (Vec m (Vec n a))
    deriving (Show, Eq, Ord, Functor, Foldable, Traversable)
```

Subtask W8.3.1

Write an `Applicative` instance

```
instance (Applicative (Vec m), Applicative (Vec n)) => Applicative (Matrix m n)
```

Subtask W8.3.2

Write a function to *transpose* a matrix (i.e. swap rows and columns):

```
transpose :: Applicative (Vec n) => Matrix m n a -> Matrix n m a
```

Using the class instances we have, that's a half-liner! Can you do it without the `Applicative` constraint?

Subtask W8.3.3

Write a function *diag* that takes a vector and generates a (square) matrix with the given vector on the main diagonal (first row and first column, second row and second column and so on) and zeros everywhere else:

```
diag :: Num a => Vec n a -> Matrix n n a
```

W8.4 Binary Lookup

We want to implement a version of fixed-length lists with efficient (logarithmic) indexing. To this end, let's first define a binary variant of (positive) natural numbers:

```
data Pos = One | Even Pos | Odd Pos
```

So for example `1 = One`, `2 = Even One`, `3 = Odd One`, `4 = Even (Even One)`, `5 = Odd (Even One)`, `6 = Even (Odd One)`, `7 = Odd (Odd One)`, `8 = Even (Even (Even One))` and so on.

Let's define corresponding "vectors"

```
data PVec (p :: Pos) (a :: *) :: * where
    POne  :: a -> PVec 'One a
    PEven :: PVec n a -> PVec n a -> PVec ('Even n) a
    POdd  :: a -> PVec n a -> PVec n a -> PVec ('Odd n) a
```

and "pointers" into such vectors:

```
data PPtr (p :: Pos) :: * where
    PtrOne   :: PPtr 'One
    PtrEvenL :: PPtr p -> PPtr ('Even p)
    PtrEvenR :: PPtr p -> PPtr ('Even p)
    PtrOddF  :: PPtr ('Odd p)
    PtrOddL  :: PPtr p -> PPtr ('Odd p)
    PtrOddR  :: PPtr p -> PPtr ('Odd p)
```

Define functions

```
find :: (a -> Bool) -> PVec p a -> Maybe (PPtr p)
```

and

```
(!!) :: PVec p a -> PPtr p -> a
```

to *find* a vector-element satisfying a given predicate and to *lookup* the element a given pointer points to.

Note that lookup happens in logarithmic time!

W8.5 Vector to Binary (*)

This exercise involves doing proofs in Haskell and is not for the faint-at-heart!

We extend the type `Pos` and the associated vectors from the last exercise to cover *all* natural numbers:

```
data Bin = Zero | Pos Pos

data BVec (b :: Bin) (a :: *) :: * where
    BNil :: BVec 'Zero a
    BPos :: PVec p a -> BVec ('Pos p) a
```

Subtask W8.5.1

Define a type family to convert from Peano naturals to binary naturals:

```
type family NatToBin (n :: Nat) :: Bin
```

Subtask W8.5.2

Define a function to convert from “normal” vectors to “binary” vectors of the same length:

```
vecToBVec :: SNatI n => Vec n a -> BVec (NatToBin n) a
```

Tip: Subtask W8.2.4 might be useful. You will probably have to prove some facts about the type family `NatToBin`.