Homework 2

ITI0212

Due: 29/03/2024

Place your solutions in a module named `Homework2` in a file with path `homework/Homework2.idr` within a repository called `iti0212-2024` on the TalTech GitLab server (`https://gitlab.cs.ttu.ee/`). Submit only your Idris source file. Do not include any build artifacts, such as `.tmp`, `.ttc`, or `.ttm` files. At the beginning of the file include a comment containing your name. Precede each problem's solution with a comment specifying the problem number.

Whether or not it is complete, the solution file that you submit should load without errors. If you encounter a syntax or type error that you are unable to resolve, use comments or holes to isolate it from the part of the file that is interpreted by Idris.

Your solutions will be pulled automatically for marking shortly after the due date.

Problem 1. A *queue* is a data structure with two "ends". An *empty* queue contains no data. Given any queue we can *push* an element onto its back end, and if it is not empty, we can also *pop* an element off of its front end in a first-in–first-out manner.

```
interface  Queue (queue : Type -> Type)  where
  emp  :  queue a
  push :  a -> queue a -> queue a
  pop  :  queue a -> Maybe (Pair a (queue a))
```

A list can be used as a simple (if inefficient) form of queue. Write a `Queue` implementation for the `List` type in Idris, so that:

```
Homework2> the (List Nat) ((push 3 . push 2 . push 1) emp)
[1, 2, 3]
Homework2> pop (the (List Nat) [1, 2, 3])
Just (1 , [2, 3])
Homework2> pop (the (List Nat) [])
Nothing
```

Problem 2. Recall that `List`s are necessarily finite sequences of data, `Stream`s are necessarily infinite sequences of data, and `Colist`s are sequences of data that may be either finite or infinite. Because of this, it should always be safe to convert a `List` or a `Stream` to a `Colist` containing the same elements in the same order.

Write `Cast` instances to convert both `List`s and `Stream`s to the corresponding `Colist`s.

```
implementation  Cast (List a) (Colist a)  where
implementation  Cast (Stream a) (Colist a)  where
```

Both `cast` methods should be total.

Problem 3. Write the (proper) predecessor function for natural numbers:

```
pred  :  Nat -> Maybe Nat
```

then write the unroll function (see lab 7, task 5) for colists:

```
unroll  :  (a -> Maybe a) -> a -> Colist a
```

so that

```
Homework2> take 5 (unroll pred 5)
[5, 4, 3, 2, 1]
Homework2> take 5 (unroll pred 3)
[3, 2, 1, 0]
```

Why does the result sequence type of this function need to be a `Colist` and not a `List` or a `Stream`?

Problem 4. Write `Eq` and `Ord` instances for the coinductive type of conatural numbers such that each number is equal to only itself, and is greater than another number just in case it contains more `Succ` constructors.

For example:

```
Homework2> coN 42 == coN 42
True
Homework2> coN 42 == coN 43
False
Homework2> coN 42 < coN 43
True
Homework2> coN 42 < infinity
True
```

You will need to mark your implementations as `partial` because Idris will (rightly) suspect that they are not total.

```
partial
implementation  Eq Conat  where

partial
implementation  Ord Conat  where
```

Demonstrate the partiality of these implementations by giving an example of an equality or ordering comparison that does not produce a result in finite time.

Problem 5. Write functions with each of the following types:

```
joinIO :  IO (IO a) -> IO a
mapIO  :  (a -> b) -> IO a -> IO b
```

Do this without using standard library functions that we haven't yet discussed in this course. Specifically, your definitions should be written in terms of the `IO` combinators that we learned about, `pure : a -> IO a` and `(>>=) : IO a -> (a -> IO b) -> IO b`, or the `do`-notation syntactic sugar, if you prefer.

Problem 6. Write a function that takes either a computation that when run produces a result of type `a` or a computation that when run produces a result of type `b`, and returns a computation that when run, runs whichever computation was given and produces the corresponding result:

```
eitherIO  :  Either (IO a) (IO b) -> IO (Either a b)
```

Now write a function that takes both a computation that when run produces a result of type `a` and a computation that when run produces a result of type `b`, and returns a computation that when run, runs the two computations in order and produces the pair of their results:

```
bothIO  :  Pair (IO a) (IO b) -> IO (Pair a b)
```

Problem 7. Write an IO computation,

```
get_number  :  IO (Maybe Integer)
```

that has the following behaviour: it prints a message asking the user to provide a number, then returns `Just n` if the user's input can be parsed as the number `n`, and `Nothing` otherwise.

Hint: You may want to import `Data.String` and `:doc parsePositive`.

Now write a function,

```
insist  :  IO (Maybe a) -> IO a
```

which returns a computation that when run, runs the argument computation repeatedly until it results in a value `Just x` for some `x : a`, at which point it produces the result `x`.

For example,

```
Homework2> :exec insist get_number >>= printLn
Please enter a number: no
Please enter a number: i don't want to
Please enter a number: okay, fine
Please enter a number: 42
42
```