Homework 2

Functional Programming (ITI0212)

due: 2022-03-23

Place your solutions in a module named `Homework2` in a file with path `homework/Homework2.idr` within your `iti0212-2022` repository on the TalTech GitLab server. Your solutions will be pulled automatically for marking shortly after the due date.

At the start of the file include a comment containing your name as it appears in your university records. Precede each problem's solution with a comment specifying the problem number.

The solution file that you submit should load without errors. If you encounter a syntax or type error that you are unable to resolve, please use comments or holes to isolate them from the part of the file interpreted by Idris.

Problem 1
Recall that the `Fin` types represent bounded prefixes of the natural numbers.

- Write a function called `forget_bound` that sends a bounded natural number to the corresponding (unbounded) natural number.
- Write a function called `impose_bound` that sends an (unbounded) natural number to the corresponding bounded natural number with the tightest possible bound.
- Write a function called `relax_bound` that sends a bounded natural number to the corresponding bounded natural number with the bound loosened by one.

For example:

```
Homework2> :set showtypes
Homework2> forget_bound $ the (Fin 42) 7
7 : Nat
Homework2> impose_bound $ forget_bound $ the (Fin 42) 7
FS (FS (FS (FS (FS (FS (FS FZ)))))) : Fin 8
Homework2> relax_bound $ impose_bound $ forget_bound $ the (Fin 42) 7
FS (FS (FS (FS (FS (FS (FS FZ)))))) : Fin 9
```

Problem 2
Recall that `Vect` types represent homogeneous finite sequences of elements indexed by a `Nat` specifying the sequence length. We can generalize this idea by using a `Vect` of types as an index for finite sequences types. This gives us the types of *heterogeneous vectors*:

```
data  HVect : Vect n Type -> Type  where
  Nil  :  HVect []
  (::) :  (x : t) -> (xs : HVect ts) -> HVect (t :: ts)
```

The type of each element of an `HVect ts` is given by the corresponding element of the indexing `Vect n Type`, here called `ts`. For example:

```
Homework2> :t [() , True , 42]
[(), True, 42] : HVect [Unit, Bool, Integer]
```

Copy the definition of the `HVect` type constructor into your script file and write the following familiar functions for finite sequence types for `HVect`s:

- `head` and `tail` for non-empty sequences,
- concatenation `(++)`,
- `index`.

Like their counterparts for `Vect`s these functions should all be total, with totality achieved by restricting the domain, if necessary. For example:

```
Homework2> head [() , True , 42]
()
Homework2> tail [() , True , 42]
[True, 42]
Homework2> [() , True] ++ [42]
[(), True, 42]
Homework2> index 2 [() , True , 42]
42
```

Hint: when specifying the types of `(++)` and `index` for `HVect`s it will be useful to use the corresponding functions for the indexing `Vect`s. These can be found in the `Data.Vect` module of the standard library under the same names. If you get stuck try reading the corresponding definitions for `Vect` types and think about what needs to change when the index is generalized from a `Nat` to a `Vect n Type`. Once you manage to express the types of the above functions Idris can write their definitions for you using case splitting and term search.

Problem 3
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 4
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 5
Suppose that we have a number of computations, each of type `IO (Either error Unit)`, which when run may yield either the result `Right ()` if they complete normally or else `Left e`, where `e` is an element of some error type, if something goes wrong. Write a function that takes a list of such computations and returns a computation that tries to run them in order, but stops if it encounters an error, returning the error and discarding any pending computations from the list:

```
tryIOs  :  List (IO (Either error Unit)) -> IO (Maybe error)
```

Now suppose that we again want to run our list of computations in order, but this time we want to run them all unconditionally and return a list of any errors that occurred:

```
batchIOs  :  List (IO (Either error Unit)) -> IO (List error)
```