Homework 4

Functional Programming

due: 2024-04-10

Place your solutions in a module named `Homework4` in a file with path `homework/Homework4.idr` within your course git repository. Please submit *only* your Idris source file. At the beginning of the file include a comment containing your name. Precede each problem's solution with a comment specifying the problem number and `public export` each definition that you are asked to write so that it can be `import`ed for testing.

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 goals 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
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 2
Write the (proper) predecessor function for natural numbers:

```
prd  :  Nat -> Maybe Nat
```

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

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

so that

```
Homework4> (prd 3 , prd 0)
(Just 2, Nothing)
Homework4> take 5 (unroll prd 5)
[5, 4, 3, 2, 1]
Homework4> take 5 (unroll prd 3)
[3, 2, 1, 0]
```

Both functions should be total. Why should the result sequence type of this `unroll` function be a `Colist` rather than a `List` or a `Stream`?

Problem 3
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
Homework4> compare (coN 42) infinity
LT
```

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

```
partial
implementation  Eq Conat  where

partial
implementation  Ord Conat  where
```

Demonstrate the partiality of these implementations by giving an example of an equality and an ordering comparison that do not produce results in finite time.

Problem 4
The *Fibonacci function* is recursively defined as:

$$\text{fib } n = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ \text{fib } (n-1) + \text{fib } (n-2) & \text{otherwise} \end{cases}$$

You wrote this function by structural recursion in homework 1. That approach, while correct, is inefficient, because each call to `fib` with argument greater than `1` generates two new recursive calls.

A more efficient approach is to define the stream of successive pairs of Fibonacci numbers:

```
fibStream  :  Stream (Pair Nat Nat)
```

so that the nth element of this stream is the ordered pair (fib n , fib (S n)):

n :	0	1	2	3	4	...
fib n :	0	1	1	2	3	...
fib (S n) :	1	1	2	3	5	...

This is more efficient because in order to determine the nth ordered pair in the stream we need only the information contained in the $(n-1)$th ordered pair.

To compute the nth Fibonacci number we just extract the first component of the nth ordered pair in the stream:

```
fast_fib  :  Nat -> Nat
fast_fib n  =  (fst . index n) fibStream
```

where the stream versions of the `index` function is found in the standard library module `Data.Stream`.

Complete the definition of `fibStream` and verify that `fast_fib` agrees with – and runs faster than – your `fib` implementation from homework 1.

Problem 5
Write a(ny possible) total expression with each of the following types:

```
expr1  :  Monoid a => a
expr2  :  Applicative t => (t a -> b) -> a -> t b
expr3  :  Monad t => (a -> t b) -> (b -> t c) -> a -> t c
```

Problem 6
Write the `Functor` instance for `Tree`s:

```
implementation  Functor Tree  where
```

Problem 7
Write the function

```
monadify  :  Monad m => (a -> b -> c) -> m a -> m b -> m c
```

which transforms any two-argument function into its monadic version:

```
Homework4> monadify (+) (Just 1) (Just 2)
Just 3
Homework4> the (List _) $ monadify (+) [1,2] [3,4,5]
[4, 5, 6, 5, 6, 7]
Homework4> :exec monadify (++) getLine getLine >>= printLn
hello_
world
"hello_world"
```

Hint: using `do`-notation and incrementally refining your goal may be helpful.

Problem 8
The *tensor product* of an m-dimensional column vector $\vec{x}$ with an n-dimensional row vector $\vec{y}$, written "$\vec{x} \otimes \vec{y}$", is an $(m \times n)$-matrix whose entry at the intersection of row i and column j is the product of the ith entry of $\vec{x}$ and the jth entry of $\vec{y}$:

$$(\vec{x} \otimes \vec{y})_{i,j} \quad = \quad \vec{x}_i \times \vec{y}_j$$

or

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{bmatrix} \otimes \begin{bmatrix} y_1 & y_2 & y_3 & \cdots \end{bmatrix} \quad = \quad \begin{bmatrix} x_1y_1 & x_1y_2 & x_1y_3 & \cdots \\ x_2y_1 & x_2y_2 & x_2y_3 & \cdots \\ x_3y_1 & x_3y_2 & x_3y_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

We can represent an n-dimensional (column or row) vector as a `List` of length n, and an $(m \times n)$-matrix as a length m `List` of length n `List`s, where each inner list represents a row of the matrix.

Write the function that computes the tensor product of vectors, as described above:

```
tensor  :  Num a => List a -> List a -> List (List a)
```

Note: You can (and should!) write this function as a short one-liner, using the fact that `List` is a `Functor`.

For example:

```
Homework4> tensor [1,2,3] [10,20,30]
[[10, 20, 30], [20, 40, 60], [30, 60, 90]]
```