Homework 3

Functional Programming (ITI0212)

due: 2023-04-17

Place your solutions in a module named `Homework3` in a file with path `homework/Homework3.idr` within a repository called `iti0212-2023` on the TalTech GitLab server (`https://gitlab.cs.ttu.ee/`). Submit only your Idris source file. Do not include any temporary files or build artifacts, such as `.swp`, `.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
Write a named semigroup instance for lists whose element type is a monoid, such that the combining is done pointwise.

```
implementation [pointwise] Monoid a => Semigroup (List a)  where
```

Your combining operation should behave as follows:

```
Homework3> (<+>)@{pointwise} ["hello ", "goodbye ", "tere!"] ["world", "friend"]
["hello world", "goodbye friend", "tere!"]
Homework3> (<+>)@{pointwise} [Just 1, Nothing, Nothing] [Nothing, Just 2]
[Just 1, Just 2, Nothing]
```

Note: unfortunately, I don't know of a way to specify a named instance of an infix operator without writing it prefix.

Problem 2
Write the *update* function for lists, returning `Nothing` if the index `i` is out-of-bounds for the argument list, and `Just` the list with `new` replacing the element at index `i` otherwise:

```
update_list  :  (new : a) -> (i : Nat) -> List a -> Maybe (List a)
```

Hint: you can shorten your definition by using the fact that `Maybe` is a `Functor`.

Problem 3
Now write the *update* function for vectors so that the index is guaranteed to be in-bounds, and therefore no `Maybe` is required in the result type:

```
update_vect  :  (new : a) -> (i : ?index_type) -> Vect n a -> Vect n a
```

Problem 4
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:

```
Homework3> monadify (+) (Just 1) (Just 2)
Just 3
Homework3> monadify (+) (the (List _) [1,2]) [3,4,5]
[4, 5, 6, 5, 6, 7]
Homework3> :exec monadify (++) getLine getLine >>= printLn
hello_
world
"hello_world"
```

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

Problem 5
Recall the type family of tuples of arbitrary arity from lecture 9:

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

Write the concatenation function for tuple types, so that for example:

```
Homework3> concat_tuple ["hello" , 42] [True , id]
["hello", 42, True, id]
Homework3> concat_tuple ["hello" , 42] []
["hello", 42]
Homework3> concat_tuple [] [True , id]
[True, id]
```

Writing the definition of this function is easy (indeed, Idris can write it for you), it's figuring out the type that might be tricky.

Problem 6
Write the function `as_tuple`, which returns a *tuple* with the same elements in the same order as the argument *vector*:

```
Homework3> as_tuple []
[]
Homework3> as_tuple [1, 2, 3]
[1, 2, 3]
Homework3> concat_tuple (as_tuple ["hello", "world"]) (as_tuple [1, 2, 3])
["hello", "world", 1, 2, 3]
```

Writing the definition of this function is easy (indeed, Idris can write it for you), it's figuring out the type that might be tricky.

Problem 7
Consider the following type constructor:

```
Either'   :  Type -> Type -> Type
Either' a b  =  DPair Bool (\ x => if x then b else a)
```

For any types `a` and `b`, the types `Either' a b` and `Either a b` are isomorphic. Prove this by writing the following functions:

```
from_either :  Either a b -> Either' a b

to_either   :  Either' a b -> Either a b
```

so that:

```
Homework3> (to_either . from_either) (Left ?x)
Left ?x
Homework3> (to_either . from_either) (Right ?y)
Right ?y
Homework3> (from_either . to_either) (False ** ?z)
(False ** ?z)
Homework3> (from_either . to_either) (True ** ?w)
(True ** ?w)
```

Problem 8
In *dynamically-typed programming languages* expressions are not statically sorted by their types. Instead, they are paired together with a *type tag* containing information about their type that can be used at run-time. We can simulate this behavior in Idris using the following type:

```
Object  :  Type
Object  =  DPair Type id
```

Write the following function that converts elements of any type into an `Object`:

```
wrap  :  {a : Type} -> (x : a) -> Object
```

Next, write the function `unwrap` so that `unwrap (wrap ?x)` evaluates to `?x` for arbitrary `?x`. Note that `unwrap` must be a *dependent function* because the *type* of the result depends on the *value* of the argument.

Hint: like non-dependent `Pair` types, `DPair` types have *projection functions* `fst` and `snd`, but where the type of the second projection depends on the value of the first: `fst : DPair a b -> a` and `snd : (dp : DPair a b) -> b (fst dp)`.

Problem 9
Proponents of dynamic typing claim that this approach has many advantages. Among its disadvantages is that when writing functions we cannot make any assumptions about the type of the data we will receive as function arguments, and must keep checking the tags to find out.

Write the following addition function for untyped terms:

```
(+)  :  Object -> Object -> Object
```

which, when applied to two `Integer`s gives their sum, when applied to two `String`s gives their concatenation, when applied to two `Bool`s gives their conjunction, and when applied to anything else gives an error:

```
data  Error : Type  where
  MkError  :  String -> Error
```

For example:

```
homework3> unwrap $ wrap 2 + wrap 3
5 : Integer
Homework3> unwrap $ wrap "hello " + wrap "world"
"hello world" : String
Homework3> unwrap $ wrap False + wrap True
False : Bool
Homework3> unwrap $ wrap "hello " + wrap 42
MkError "type error" : Error
```