Homework 5

Functional Programming

due: 2024-04-24

Place your solutions in a module named `Homework5` in a file with path `homework/Homework5.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. All definitions in your file should be *total*, which you can ensure by using the `%default total` directive.

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
In lecture 8 we wrote the `update` function for `List`s,

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

which replaces the element at a given index with a new element, but which fails if that index is out of bounds. Write the `update` function for `Vect`s, which also replaces the element at the given index with a new element, but which constrains the types so that it cannot fail. For example,

```
Homework5> update_list 'd' 1 ['a', 'b', 'c']
Just ['a', 'd', 'c']
Homework5> update_vect 'd' 1 ['a', 'b', 'c']
['a', 'd', 'c']
Homework5> update_list 'd' 4 ['a', 'b', 'c']
Nothing
Homework5> update_vect 'd' 4 ['a', 'b', 'c']
<Type Error> ...
```

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

Problem 2
Recall the type family of tuples of arbitrary arity from lecture 9:

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

Write the concatenation function for tuples, so that for example:

```
Homework5> concat_tuple ["hello" , 42] [True , id]
["hello", 42, True, id]
```

```
Homework5> concat_tuple ["hello" , 42] []
["hello", 42]
Homework5> 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 may be tricky.

Problem 3
In *dynamically-typed programming languages* values are not statically classified by their types. Instead, they are paired together with a *type tag* containing information about their type that can be used by the run-time environment. We can simulate this behavior in Idris using the following type:

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

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

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

Next, write the function that converts an `Object` back into an element of its original type:

```
unwrap  :  (x : Object) -> ?result_type
```

so that `unwrap (wrap ?x)` evaluates to `?x` for any `?x`. For example:

```
Homework5> unwrap (wrap "hello")
"hello"
Homework5> unwrap (wrap 42)
42
Homework5> unwrap (wrap (the (Tuple _) ["hello" , 42]))
["hello", 42]
```

Note that `unwrap` must be a *dependent function* because the *type* of the result depends on the *value* of the argument.

Hint: Examine the values of some expressions of type `Object`, such as `wrap "hello"`, and think about how to recover the original type.

Problem 4
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 types of the values they will receive as arguments, and so we must keep checking the tags.

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 disjunction, and when applied to anything else gives an error:

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

For example:

```
Homework5> unwrap $ wrap 2 + wrap 3
5 : Integer
Homework5> unwrap $ wrap "hello " + wrap "world"
```

```
"hello world" : String
Homework5> unwrap $ wrap False + wrap True
True : Bool
Homework5> unwrap $ wrap "hello " + wrap 42
MkError "Runtime Error" : Error
```

Note: You don't need the `wrap` and `unwrap` functions from the previous problem to *write* this function, only to conveniently *test* it.

Problem 5
Copy the following indexed type family (which uses `Data.Nat.LTE`) into your file:

```
data  Sorted : List Nat -> Type  where
  -- an empty list is sorted:
  EmptSorted  :  Sorted []
  -- a singleton list is sorted:
  SingSorted  :  Sorted [x]
  -- otherwise a list is sorted if its first two elements are ordered,
  -- and its tail is sorted:
  ConsSorted  :  LTE x y -> Sorted (y :: zs) -> Sorted (x :: y :: zs)
```

We can interpret the *type* `Sorted xs` as the *proposition* that the list `xs` is sorted.

- Prove that the list `[1, 2, 3]` is sorted:

  ```
  one_two_three_sorted  :  Sorted [1, 2, 3]
  ```

- Prove that applying the successor function to each element of a sorted list preserves the sorting:

  ```
  succ_sorted  :  (xs : List Nat) -> Sorted xs -> Sorted (map S xs)
  ```

Problem 6
The following indexed type family, which you should import or copy, is defined in the standard library module `Data.List.Elem`:

```
data  Elem : a -> List a -> Type  where
  Here  :  Elem z (z :: xs)
  There :  Elem z xs -> Elem z (x :: xs)
```

We can interpret the *type* `Elem z xs` as the *proposition* that the element `z` occurs somewhere within the list `xs`.

- Prove that if an element occurs within a given list then it also occurs within the concatenation of that list with any other list:

  ```
  in_left  :  Elem z xs -> (ys : List a) -> Elem z (xs ++ ys)
  ```

- Prove that if an element occurs within a given list then it also occurs within the concatenation of any other list with that list:

  ```
  in_right :  Elem z ys -> (xs : List a) -> Elem z (xs ++ ys)
  ```

Hint: Consider the recursive structure of the list concatenation function `Prelude.List.(++)` and think about which argument to do induction on in order to best follow it.