Homework 4

ITI0212

Due: 05/10/2024

Place your solutions in a module named `Homework4` in a file with path `homework/Homework4.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 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.

Note: you should to add `%default total` to the beginning of your script. This will tell Idris that all the functions we are defining should be total, so that we do not accidentally fall into the case in which we can prove any statement by simply giving a function that does not terminate.

Note: the latter part of this assignment references the encoding of the connectives of first-order logic covered in Lecture 11. You will need to add the following definitions from Lecture 11 to your Idris file:

```
Or : Type -> Type -> Type
Or = Either

And : Type -> Type -> Type
And = Pair

Implies : Type -> Type -> Type
Implies a b = a -> b

Exists : (a : Type) -> (p : a -> Type) -> Type
Exists = DPair

Forall : (a : Type) -> (p : a -> Type) -> Type
Forall a p = (x : a) -> (p x)
```

Problem 1. Recall the type `<=` from Lecture 10:

```
data (<=) : (p : Nat) -> (n : Nat) -> Type where
  LeZ : 0 <= n
  LeS : p <= n -> S p <= S n
```

and define the type `IsSorted` that tells if a list of natural numbers is sorted:

```
data IsSorted : List Nat -> Type where
  NilSort : IsSorted []
  SinglSort : IsSorted [x]
  ConsSort : IsSorted (y :: ys) -> x <= y -> IsSorted (x :: y :: ys)
```

Prove that the list `[0, 1, 2, 3]` is sorted, that is, build a function of type:

```
is_sorted_0123 : IsSorted [0, 1, 2, 3]
```

Problem 2. Prove that if a list is sorted, then adding one to every element will result in a list that is again sorted. That is, write a function with the following type:

```
is_sorted_succ : (xs : List Nat) -> IsSorted xs -> IsSorted (map S xs)
```

Problem 3. Recall that the function `replicate : Nat -> a -> List a` takes a natural number `n : Nat`, an element `x : a`, and returns the constant list with `n` times the value `x`. Prove that such a list is always sorted, that is, construct a function of the following type:

```
is_sorted_cst : (lg, val : Nat) -> IsSorted (replicate lg val)
```

Hint: recall from Lecture 10 that less or equal is reflexive, and do not forget to add `import Data.List` at the beginning of your file to access the function `replicate`.

Problem 4 (De Morgan's laws)**.** As you might already be familiar from your experience in other programming languages, the "and" and "or" operators can be switched into one another using negation. These equivalences between the two operators are also known as De Morgan's laws: let's prove some of them in Idris. First, show that, if the "and" of two statements holds, then it is impossible for either one of the two statements to be false:

```
and_not_or  : (a `And` b) `Implies` Not (Not a `Or` Not b)
```

Similarly, if the "or" of two statements holds, then it must be impossible for both statements to be false at the same time.

```
or_not_and : (a `Or` b) `Implies` Not (Not a `And` Not b)
```

Finally, if the "or" of two statements is false, then both statements must be false at the same time. Prove it along with the converse statement:

```
not_or   :  Not (a `Or` b) `Implies` (Not a `And` Not b)
not_or'  :  (Not a `And` Not b) `Implies` Not (a `Or` b)
```

(Interestingly, the converse of `and_not_or` and `or_not_and` are equivalent to the law of excluded middle, so they are once again unprovable in Idris. For the adventurous reader: can you prove `not_and` along with its converse?)

Problem 5 (Universals and disjunctions)**.** Someone tells you that every element of type `t` satisfies `p`, but another person tells you that they actually satisfy `q`. If one of them is right, then you certainly know that every element satisfies either `p` or `q`. Prove this in Idris:

```
forall_or : (Forall t p `Or` Forall t q)
  `Implies` Forall t (\x => p x `Or` q x)
```

If you can find an element such that `p` and an element such that `Not . p`, then you cannot prove that either `Not . p` always holds or that `p` always holds. Show it in Idris:

```
exist_p_not_p  :  ((Exists a p) `And` (Exists a (Not . p)))
               ->  Not (Forall a p `Or` Forall a (Not . p))
```

Problem 6 (Fun with the *law of excluded middle*)**.** Prove that, for a specific proposition `a`, if we assume the *law of excluded middle* (or more simply, LEM) then we can also prove *double negation elimination* (or more simply, DNE).

```
lem_to_dne  :  a `Or` Not a
            ->  Not (Not a) `Implies` a
```

Despite us not being able to show in Idris that LEM holds, we can prove something quite close to it. Prove that it must be impossible for LEM to be false:

```
not_not_lem  :  Not (Not (a `Or` Not a))
```

From this, you can clearly see that if we had DNE we would be able to remove the two negations and recover LEM, and thus give an algorithm to decide truth or falsity of any mathematical theorem!

(*Hint*: there is only one way of proving this. Keep in mind that when the goal looks like a function, you should introduce a lambda-abstraction, and when the goal looks like an `Either` you can simply try `Left` or `Right` as argument and see which one works! Another hint: when proving this negation, you should use your hypothesis twice.)

Problem 7. Show that if two vectors are equal then so are their lengths:

```
same_length  :  {xs : Vect m a} -> {ys : Vect n a} ->
    xs = ys -> length xs = length ys
```

Now show that if two vectors are equal then they have the same element at each index:

```
same_elments  :  {xs , ys : Vect n a} ->
    xs = ys -> index i xs = index i ys
```

Finally, show that two vectors with different head elements are different:

```
different_heads  :  {xs , ys : Vect n a} ->
    Not (x = y) -> Not (x :: xs = y :: ys)
```

Problem 8 (`List` is a functor)**.** We presented the `Functor` interface in lecture 5 for type constructors with a higher-order `map` function. Such a `map` function is supposed to preserve identity functions and function composition. These properties can be encoded (pointwise) using the following interface:

```
interface Functor t => FunctorV (t : Type -> Type) where
  pres_idty : (xs : t a) -> map Prelude.id xs = xs
  pres_comp : (f : a -> b) -> (g : b -> c) -> (xs : t a) ->
    map (g . f) xs = (map g . map f) xs
```

Prove that `List` is a functor (pointwise) by instantiating the above interface for `List`.

```
FunctorV List where
  pres_idty xs = ?identity_preservation
  pres_comp f g xs = ?composition_preservation
```

Problem 9 (Reversing a list is an involution)**.** Here is the definition of a function that reverses lists:

```
rev : List a -> List a
rev [] = []
rev (x :: xs) = rev xs ++ [x]
```

We want you to prove that reversing a list twice yields the original list. The goal of this problem is the following:

```
rev_inv : (xs : List a) -> rev (rev xs) = xs
```

In a similar way to what we did for commutativity of addition in lecture 12, one way to prove this proceeds by observing how `rev` acts on constructors. In particular, it will be useful to begin by proving the following lemma (you just have to fill in the preorder reasoning steps):

```
rev_inv_cons : (x : a) -> (xs : List a) ->
  rev (xs ++ [x]) = x :: rev xs
rev_inv_cons x [] = ?rev_inv_cons_0
rev_inv_cons x (y :: xs) =
  let
    IH = rev_inv_cons x xs
  in Calc $
    |~ rev ((y :: xs) ++ [x])
    ~~ rev (xs ++ [x]) ++ [y]  ...( ?rev_inv_cons_1 )
    ~~ (x :: rev xs) ++ [y]    ...( ?rev_inv_cons_2 )
    ~~ x :: (rev xs ++ [y])    ...( ?rev_inv_cons_3 )
    ~~ x :: rev (y :: xs)      ...( ?rev_inv_cons_4 )
```

After this, you can prove the main theorem by induction on the list. You are invited to try and do this from scratch, however, we also include a proof sketch to help you:

```
rev_inv [] = ?rev_inv_0
rev_inv (x :: xs) =
  let
    IH = rev_inv xs
  in Calc $
    |~ rev (rev (x :: xs))
    ~~ rev (rev xs ++ [x])  ...( ?rev_inv_1 )
    ~~ x :: rev (rev xs)    ...( ?rev_inv_2 )
    ~~ x :: xs              ...( ?rev_inv_3 )
```

Hint: don't forget to look for an opportunity to use the lemma `rev_inv_cons`!