Lab 10

Functional Programming

2024-04-19

This week we are learning about the *propositions as types* interpretation of logic. Under this interpretation we can regard a *type* as a *logical proposition*, and regard an *element* of such a type as a *proof* of the corresponding proposition. This allows us to represent propositions and proofs as data within our programs, which we can analyze and manipulate, just like any other data.

An important tool for proving propositions about inductively-defined data is *proof by induction*. Under the propositions as types interpretation this corresponds to the familiar notion of a *recursive function*. In order for such an inductive proof to be valid the corresponding recursive function must be *total*. We can enforce totality checking for a file using the Idris directive `%default total`.

One strategy that is often helpful when writing proofs in Idris is to *explicitly bind implicit arguments* in the clauses of a definition, either on the left or on the right of the defining `=`. This is useful when Idris makes a bad choice for naming a variable in an implicit argument (for example, by *shadowing* another variable), or when Idris doesn't have enough information to correctly infer the type of a goal that we are working on. Once a proof is complete we may choose to erase such bindings if they can be inferred by Idris because the names no longer matter.

To complete this lab you should import the module `Lecture10` that we developed interactively in class. You will also need to import `Data.Nat` in order to use the type constructor `LTE`.

LTE and Addition

In class we proved that `LTE` is a reflexive and transitive relation on the natural numbers. Now we will explore how this relation interacts with the operation of addition.

Task 1
As a warm-up, prove that every natural number is less than or equal to its own successor:

```
succ_larger  :  {n : Nat} -> LTE n (S n)
```

You should do this by induction on the natural number `n`, which you can bring into scope by explicitly binding the implicit argument using the notation `{n = n}` on the left side of a clause.

Task 2
Use the fact that you proved in task 1 together with the *transitivity* of LTE, which was proved in class, in order to prove the following "weakening lemmas" about LTE:

```
lte_weaken_right :  {m , n : Nat} -> LTE m n -> LTE m (S n)
lte_weaken_left  :  {m , n : Nat} -> LTE (S m) n -> LTE m n
```

Task 3
Now prove the following facts about adding zero on the right or on the left:

```
zero_plus_right  :  (m , n : Nat) -> LTE (m + 0) (m + n)
zero_plus_left   :  (m , n : Nat) -> LTE (0 + n) (m + n)
```

As you examine the intermediate proof states, recall that addition of natural numbers is defined recursively on the first argument (`:printdef plus`), so that as far as Idris is concerned `0 + n` and `n` are interchangeable, and likewise, `S m + n` and `S (m + n)` are interchangeable.

Task 4
Next, prove the following facts about adding a successor on the right or on the left:

```
succ_plus_right :  (m , n : Nat) -> LTE (m + n) (m + S n)
succ_plus_left  :  (m , n : Nat) -> LTE (m + n) (S m + n)
```

Exponentiating an Even Number

In class we proved that the sum and product of two even natural numbers is even. Here we consider exponentiation. Without thinking too hard, we might believe that for any n, if m is even then the exponential m^n is even too. This is almost true, except when n is 0.

We begin by defining a predicate for positive numbers as a `Nat`-indexed type:

```
data  Positive : Nat -> Type  where
  One_positive  :  Positive 1
  S_positive    :  Positive n -> Positive (S n)
```

which we can think of as follows:

$$\begin{tikzcd}
 & \texttt{One\_positive} & \texttt{S\_positive One\_positive} & \texttt{S\_positive (S\_positive One\_positive)} & & \texttt{: Positive n} \\
\texttt{0} \arrow[r, mapsto, "\texttt{S}"'] & \texttt{1} \arrow[r, mapsto, "\texttt{S}"'] & \texttt{2} \arrow[r, mapsto, "\texttt{S}"'] & \texttt{3} & \texttt{...} & \texttt{: (n : Nat)}
\end{tikzcd}$$

The type `Positive 0` is empty, the type `Positive 1` is a singleton containing the element `One_positive`, and every type `Positive (S (S n))` is a singleton containing the result of applying the function `S_positive` to the sole inhabitant of the type `Positive (S n)`.

We can use this type together with `Even` to express the proposition that we wish to prove: if m is even and n is positive then m^n is even. This should go smoothly, except for one wrinkle.

Task 5
Because we don't yet know how to tell Idris about equality, we will need the following obvious lemma, which you should now prove:

```
even_times_one  :  Even n -> Even (n * 1)
```

We are now ready to prove that a positive power of an even number is even. Usually the easiest way to prove a property about a recursively defined object is to try to follow its recursive structure in the structure of your proof. Examine the recursive structure of the exponentiation function on natural numbers with `:printdef power`. On which argument is it recursive?

Task 6
Write a proof of the theorem that a positive power of an even number is even by induction on the assumption corresponding to the recursive argument in the function `power`.

```
pow_even_pos  :  Even m -> Positive n -> Even (power m n)
```