Lab 11

Functional Programming (ITI0212)

2021.04.06

This week we are learning about how to regard a type as a proposition and an element of a type as a proof of the corresponding proposition.

To complete this lab you should import the module `Lecture11` that we developed interactively during lecture. If you are using Idris 2, you will also need to import `Data.Nat` in order to use the type constructor `LTE`.

LTE and Addition

In lecture 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 of the generated clause.

Task 2
Use the fact that you proved in task 1 together with the transitivity of LTE, which was proved in lecture, in order to prove the following two “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)
```

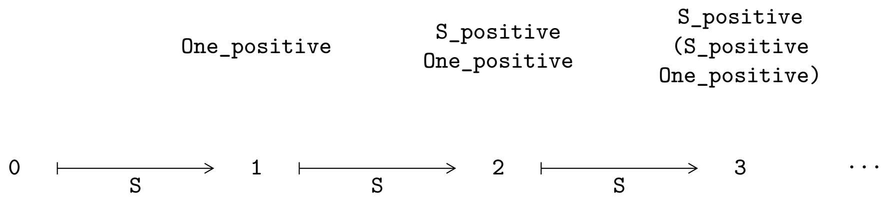
Exponentiating an Even Number

In lecture 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 (S Z)
  S_positive    :  Positive n -> Positive (S n)
```

which we can think of as follows:



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 also 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 easy lemma, which you should now prove:

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

We are now nearly ready to prove that a positive power of an even number is even. Often 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 using induction on the assumption corresponding to the recursive argument in the function `power`.

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