Lab 11

Functional Programming (ITI0212)

2023-04-10

This week we are learning about the *propositions as types* paradigm. It tells us that propositions (the mathematical statements) are in fact the same things as types (the functional programming objects). This is known as the Curry-Howard correspondence. When we have `t : T`, then it can be read both as *`t` is a term of type T* and *`t` is a proof of the proposition T.* We see here that terms becomes proofs, and constructing a term of a type (= proposition) is the exact same thing as constructing a proof of this type (= proposition).

As a side note, you should add, before coding any functions, the following command to your script: `%default total`. This will tell Idris that all the functions we are defining should be total. Indeed, if it were not the case, we could perfectly well construct a partial function of some type, thus giving a partial proof of a proposition.

A remark on implicit vs. explicit arguments

Often when we do theorem proving in functional programming, we like to leave arguments implicit as much as possible. For instance, consider those two functions proving that `<=` is reflexive:

```
leReflExplicit : (n : Nat) -> n <= n
leReflExplicit 0 = LeZ
leReflExplicit (S k) = LeS (leReflExplicit k)
```

and

```
leReflImplicit : {n : Nat} -> n <= n
leReflImplicit {n = 0} = LeZ
leReflImplicit {n = (S k)} = LeS leReflImplicit
```

In the explicit version, it is easier to write the function because the explicit argument automatically appears in the body. In the implicit version, we have to tell Idris that we want to deal with the implicit argument. The process generally goes as follow. First write the type of your function:

```
leReflImplicit : {n : Nat} -> n <= n
```

Then ask Idris (with the key-binding) to write the body of your function:

```
leReflImplicit : {n : Nat} -> n <= n
leReflImplicit = ?leReflImplicit_rhs
```

Here you notice that the `n` is not present in the body, so you cannot pattern-match on it. However, you can manually add it like so:

```
leReflImplicit : {n : Nat} -> n <= n
leReflImplicit {n = n} = ?leReflImplicit_rhs
```

Notice that you could also have chosen to put `{n = anotherNameForMyImplicitArgument}`, it doesn't really matter. The important thing is that the left hand side of the `=` is the same as the name of the implicit argument. Finally, you can ask Idris to pattern-match on it, as usual with the key-binding:

```
leReflImplicit : {n : Nat} -> n <= n
leReflImplicit {n = 0} = ?leReflImplicit_rhs_1
leReflImplicit {n = (S k)} = ?leReflImplicit_rhs_2
```

In the today's lab, we made things easier and all the arguments are made explicit when there is a need to pattern-match on them. However you should try to make them implicit, typically, for the Task 3, you could adapt

```
leWeakRight : (m, n : Nat) -> m <= n -> m <= S n
```

as

```
leWeakRight' : {m, n : Nat} -> m <= n -> m <= S n
```

Warning: when you have implicit arguments that are not explicitly bound (i.e. between `{ }`), Idris will choose names for them, and unfortunately, it (often) chooses the same name for different arguments making things confusing. To avoid this problem, do not hesitate to name your implicit arguments anyway (e.g. `{n = n}`, `{p = p}`, etc.), even if you don't pattern match on them. This is typically what will happen in Task 1: if you don't put names, then Idris will decide to name both `m` and `n` as `p`, and name `p` as `n`, which is... kind of confusing.

Facts about less or equal

As a general hint for this section, do not forget that you can use previous result to prove new ones. Recall the type $\leqslant$ from the lecture:

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

Task 1
Prove that `<=` is transitive, that build a function of the following type:

```
leTrans : m <= n -> n <= p -> m <= p
```

Task 2
State and prove that any integer is less or equal than its successor.

Task 3
Prove the following:

```
leWeakRight : (m, n : Nat) -> m <= n -> m <= S n
```

and

```
leWeakLeft : (m, n : Nat) -> S m <= n -> m <= n
```

Hint: for `leWeakLeft`, if you are stuck, you can use `leTrans`.

Task 4
Prove the following:

```
zeroPlusRight : (m, n : Nat) -> m + 0 <= m + n
```

and

```
zeroPlusLeft : (m, n : Nat) -> 0 + n <= m + n
```

Hint: for `zeroplusLeft`, if you are stuck, you can use `leTrans`.

Task 5
Prove the following:

```
succPlusRight : (m, n : Nat) -> m + n <= m + S n
```

and

```
succPlusLeft : (m, n : Nat) -> m + n <= S m + n
```

On the length of lists

Task 6
State and prove that if two lists `xs, ys` are such that `length xs <= length ys`, then `length (x :: xs) <= length (y :: ys)`.

Recall in the lecture how we define the `IsPrefix` proposition:

```
data IsPrefix : (xs : List a) -> (ys : List a) -> Type where
  IsPrefixNil : IsPrefix [] ys
  IsPrefixCons : IsPrefix xs ys -> IsPrefix (z::xs) (z::ys)
```

Task 7
State and prove that if a list `xs` is the prefix of a list `ys`, then the length of `xs` is smaller or equal than the length of `ys`.

Optional tasks : Induction on lists

Recall from the lecture that the induction principle for natural numbers has the following type:

```
induction : (prop : Nat -> Type) ->
  (base_case : prop 0) ->
  (induction_step : (k : Nat) -> prop k -> prop (S k)) ->
  (n : Nat) -> prop n
```

Task 8
Write the type of the induction principle for the type `List a`. Give a proof that it is true, that is, write a function of the type you just wrote.

Task 9
State (using the type $\leqslant$ as above) that the length of a list `length xs` is always less or equal than `length (x :: xs)`. Now prove it using the induction principle you just defined.