FCPLCD – Exercise 2

1 Resources

Coq

- Docs
- Online editor via the Coq homepage
- Coq Tactics Cheatsheet

Lean

- Install Lean or use the web editor.
- Join the Lean forum to ask questions or search for answers.
- Read sections 1.{1, 2, 3, 5, 6} of Functional Programming in Lean for an introduction to Lean's syntax.
- Read Theorem Proving in Lean Chapter 5 for an introduction to proof tactics.

2 Inductive Data Types

In the last exercise you defined natural numbers as well as addition and multiplication. You can do the same in Coq and Lean. Just as in the last exercise, natural numbers are constructed from either a basic element (zero) or inductively as the successor of an existing number.

2.1 Natural Numbers

1. For the first exercise start by defining a type `MyNat` representing the natural numbers. Also define `one`, `two`, `three` and `four` based on this new type – we will use them later.

2. In Presburger Arithmetic, the axioms for addition are an integral part of defining natural numbers. With Peano numbers, we can define addition as an independent function. So for this exercise, define an `add` function on your type of natural numbers. Test it by computing an example addition (e.g. `add one two`) using your theorem prover's evaluation command. For convenience in later exercises, define the "`+`" notation for your addition function.

2.2 Lists

You can define lists in a similar fashion to natural numbers, namely as inductive types. For example, a list of natural numbers can be defined as:

$$\frac{}{\texttt{nil} : \texttt{NatList}} \qquad \frac{\texttt{head} : \texttt{Nat} \quad \texttt{tail} : \texttt{NatList}}{\texttt{cons head tail} : \texttt{NatList}}$$

For this exercise, define a *polymorphic* list type `PolyList` by using a type parameter. Then define functions on `PolyList` to:

1. calculate the `length` of a list.
2. `append` two lists.
3. `reverse` a list.

Define the infix notations `::` for `cons` and `++` for `append`.

3 Tactics

In the previous exercise sheet you also constructed proofs about relations over natural numbers. For example, you proved that $3 < 4$. The basis for those proofs was the formalization of equality - namely the axioms:

$$\frac{}{a = a} \qquad \frac{b = a}{a = b} \qquad \frac{a = b \quad b = c}{a = c}$$

Coq and Lean come equipped with formalizations of many predicates like `=`, $\wedge$, `<`, as well as theorems about these predicates. To prove a theorem in these systems, we need to build a proof tree as in Holbert. But instead of visually, we do it line by line using helper functions called "tactics". For example, in Lean, the statement and proof of a simple theorem could look as follows:

```
theorem nat_eq_trans : ∀ x y z : MyNat, (x = y ∧ y = z) → x = z := by
  intro x y z h
  apply Eq.trans
  · have hx := h.left
    exact hx
  · have hy := h.right
    exact hy
```

3.1 First Proofs

Prove theorems analogous to `nat_eq_trans` but for symmetry and reflexivity. Also, try having `x`, `y` and `z` be of *any* type, not just `MyNat`.

4 Inductive Proofs

Now we have everything together to practice inductive proofs. Remember the principle structure of such proofs is:

1. prove your theorem for the base case(s) of your data type
2. show that, assuming your theorem holds for a given instance of your data type, it also holds for immediately "larger" instances (instances constructed using the "smaller" instance as argument)

4.1 Induction on Natural Numbers

Commutativity and associativity are interesting properties of functions, so let's prove them for addition on natural numbers. Again we use our self defined type from the first exercise and the `+` operator represents the infix notation for addition on `MyNat`.

4.1.1 Associativity

Prove the following theorem by induction. Note that both Coq and Lean have dedicated tactics for performing induction.

$$\forall\ x\ y\ z : \texttt{MyNat},\ x + (y + z) = (x + y) + z$$

4.1.2 Commutativity

Prove commutativity of addition by induction:

$$\forall\ x\ y : \texttt{MyNat},\ x + y = y + x$$

The proof requires two lemmas, whose exact statements depend on how you've defined your addition function. The first one is either $a + 0 = a$ or $0 + a = a$, and the second one either $a + \mathit{succ}\, b = \mathit{succ}\, (a + b)$ or $\mathit{succ}\, a + b = \mathit{succ}\, (a + b)$. You will probably notice which one you need when trying to prove commutativity.

4.2 Induction on Lists

As the final exercise for this week, prove that reversing a list is self-inverse:

$$\forall\ (\alpha : \texttt{Type})\ (l : \texttt{PolyList}\ \alpha),\ \texttt{reverse}\,(\texttt{reverse}\ l) = l$$

There are different ways to prove this theorem, but let's go through one path of argumentation together the "pen and paper" way:

Proof Sketch We prove the theorem by induction on l.

Base Case: `l = nil`

Thus, our goal is `nil = reverse (reverse nil)`.
We can easily show this from the definition of `reverse`.

Induction Step: `l = hd :: tl`

Thus, our goal is `hd :: tl = reverse (reverse (hd :: tl)` and our induction hypothesis is `tl = reverse (reverse tl)`.
We can start using the definition of `reverse` to simplify the goal to:

```
hd :: tl = reverse (reverse tl ++ [hd])
```

Assuming we had a lemma for distributivity of `reverse` over `append`, we get:

```
hd :: tl = reverse [hd] ++ reverse (reverse tl)
```

Assuming we had a lemma showing `reverse [x] = [x]`, we get:

```
hd :: tl = [hd] ++ reverse (reverse tl)
```

Using the induction hypothesis, we get:

```
hd :: tl = [hd] ++ tl
```

By unfolding the definition of `append` we can easily show our goal.

□

You see we assumed two lemmata in this proof sketch. Start by proving them, before you prove the full theorem.