CS420

Introduction to the Theory of Computation

Lecture 4: Manipulating theorems; data-structures

Tiago Cogumbreiro

Today we will learn...

1. More on the assert tactic
2. Defining data-structures in Coq

More on `assert`

Exercise 1

```
Lemma zero_in_middle:
  forall n m, n + 0 + m = n + m.
Proof.
  intros.
```

Exercise 1

```
Lemma zero_in_middle:
  forall n m, n + 0 + m = n + m.
Proof.
  intros.
```

1. Using intermediate results: `plus_n_O`
2. Passing parameters to theorems: `add_assoc`

Exercise 1: Solution 1

1. Using intermediate results: `plus_n_O`

Exercise 1: Solution 1

1. Using intermediate results: `plus_n_O`

```
Lemma zero_in_middle:
  forall n m, n + 0 + m = n + m.
Proof.
  intros.
  assert (n + 0 = n). {
    rewrite plus_n_O.
    reflexivity.
  }
  rewrite H.
  reflexivity.
Qed.
```

Exercise 2: add is associative

```
Lemma add_assoc:
  forall n m o,
  (n + m) + o = n + (m + o).
```

Exercise 2: add is associative

```
Lemma add_assoc:
  forall n m o,
  (n + m) + o = n + (m + o).

Proof.
  intros.
  induction n. {
    simpl.
    reflexivity.
  }
  simpl.
  rewrite IHn.
  reflexivity.
Qed.
```

Exercise 1: Solution 2

2. Passing parameters to theorems: `add_assoc`

```
Lemma zero_in_middle:
  forall n m, n + 0 + m = n + m.
Proof.
```

Exercise 1: Solution 2

2. Passing parameters to theorems: `add_assoc`

```
Lemma zero_in_middle:
  forall n m, n + 0 + m = n + m.
Proof.

  intros.
  assert (Hx := add_assoc n 0 m).
  rewrite Hx.
  simpl.
  reflexivity.
Qed.
```

Exercise 1: Solution 2

```
Lemma zero_in_middle_2:
  forall n m, n + (0 + m) = n + m.
Proof.
```

Exercise 1: Solution 2

```
Lemma zero_in_middle_2:
  forall n m, n + (0 + m) = n + m.
Proof.
```

You are now ready to conclude HW1

How do we define a data structure that holds two nats?

A pair of nats

```
Inductive natprod : Type :=
| pair : nat → nat → natprod.

Notation "( x , y )" := (pair x y).
```

> Explicit vs implicit: be cautious when declaring notations, they make your code harder to understand.

How do we read the contents of a pair?

Accessors of a pair

Accessors of a pair

```
Definition fst (p : natprod) : nat :=
```

Accessors of a pair

```
Definition fst (p : natprod) : nat :=
  match p with
  | pair x y ⇒ x
  end.

Definition snd (p : natprod) : nat :=
  match p with
  | (x, y) ⇒ y (* using notations in a pattern to be matched *)
   end.
```

How do we prove the correctness of our accessors?

(What do we expect fst/snd to do?)

Proving the correctness of our accessors:

```
Theorem surjective_pairing : forall (p : natprod),
  p = (fst p, snd p).
Proof.
  intros p.
```

```
1 subgoal
p : natprod
______________________________________(1/1)
p = (fst p, snd p)
```

> Does `simpl` work? Does `reflexivity` work? Does `destruct` work? What about `induction`?

How do we define a list of nats?

A list of nats

```
Inductive natlist : Type :=
  | nil : natlist
  | cons : nat → natlist → natlist.

(* You don't need to learn notations, just be aware of its existence:*)

Notation "x :: l" := (cons x l) (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).

Compute cons 1 (cons 2 (cons 3 nil)).
```

outputs:

```
= [1; 2; 3]
: list nat
```

How do we concatenate two lists?

Concatenating two lists

```
Fixpoint app (l1 l2 : natlist) : natlist :=
 match l1 with
 | nil ⇒ l2
 | h :: t ⇒ h :: (app t l2)
 end.

Notation "x ++ y" := (app x y) (right associativity, at level 60).
```

Proving results on list concatenation

```
Theorem nil_app_l : forall l:natlist,
  [] ++ l = l.
Proof.
  intros l.
```

- Can we prove this with `reflexivity`? Why?

Proving results on list concatenation

```
Theorem nil_app_l : forall l:natlist,
  [] ++ l = l.
Proof.
  intros l.
```

Can we prove this with `reflexivity`? Why?

```
  reflexivity.
Qed.
```

Nil is a neutral element wrt app

```
Theorem nil_app_l : forall l:natlist,
  l ++ [] = l.
Proof.
  intros l.
```

Can we prove this with `reflexivity`? Why?

Nil is a neutral element wrt app

```
Theorem nil_app_l : forall l:natlist,
  l ++ [] = l.
Proof.
  intros l.
```

Can we prove this with `reflexivity`? Why?

```
In environment
l : natlist
Unable to unify "l" with "l ++ [ ]".
```

How can we prove this result?

We need an induction principle of `natlist`

For some property `P` we want to prove.

- Show that $P([])$ holds.
- Given the induction hypothesis $P(l)$ and some number n, show that $P(n :: l)$ holds.

Conclude that $P(l)$ holds for all l.

> How do we know this principle? Hint: compare `natlist` with `nat`.

How do we know the induction principle?

Use search

```
Search natlist.
```

which outputs

```
 nil: natlist
 cons: nat → natlist → natlist
(* trimmed output *)
natlist_ind:
   forall P : natlist → Prop,
   P [] →
   (forall (n : nat) (l : natlist), P l → P (n::l)) → forall n : natlist, P n
```

Nil is neutral on the right (1/2)

```
Theorem nil_app_r : forall l:natlist,
  l ++ [] = l.
Proof.
  intros l.
  induction l.
  - reflexivity.
  -
```

yields

```
1 subgoal
n : nat
l : natlist
IHl : l ++ [ ] = l
______________________________________(1/1)
(n :: l) ++ [ ] = n :: l
```

Nil is neutral on the right (2/2)

```
1 subgoal
n : nat
l : natlist
IHl : l ++ [ ] = l
______________________________________(1/1)
(n :: l) ++ [ ] = n :: l
```

Nil is neutral on the right (2/2)

```
1 subgoal
n : nat
l : natlist
IHl : l ++ [ ] = l
______________________________________(1/1)
(n :: l) ++ [ ] = n :: l
```

```
simpl.          (* app (n::l) [] = n :: (app l []) *)
rewrite → IHl.  (*   n :: (app l []) = n :: l *)
                (*        ^^^^^^^^^         ^ *)
reflexivity.    (* conclude *)
```

> Can we apply rewrite directly without simplifying?
> Hint: before and after stepping through a tactic show/hide notations.
> How do we state a theorem that leads to the same proof state (without ltac)?