CS720

Logical Foundations of Computer Science

Lecture 2: A proof primer

Tiago Cogumbreiro

On studying effectively for this course

Setup

1. Have CoqIDE available in a computer you have access to
2. Have `vol1.zip` extracted in a directory *you alone* have access to

Caveats

1. **There are no tests**, so no way to invest time later
2. In this course you'll **weekly** load of work, don't let it build up
3. Re-submitting a homework assignment will increase your next-week workload
4. Recall that the lowest grade of your homework assignments is ignored

On studying effectively for this course

Suggestions

- **Read the chapter before the class:**
  This way we can direct the class to specific details of a chapter, rather than a more topical end-to-end description of the chapter.
- **Attempt to write the exercises before the class:**
  We can guide a class to cover certain details of a difficult exercise.
- **Use the office hours and our online forum:** Coq is a unusual programming language, so you will get stuck simply because you are not familiar with the IDE or a quirk of the language

On studying effectively for this course

Homework structure

1. Open the homework file with CoqIDE: that file is the chapter we are covering
2. Read the chapter and fill in any exercise
3. To complete a homework assignment ensure you have 0 occurrences of `Admitted`

This information is available in our online forum.

Today we will...

- cover some proof techniques: rewriting terms, case analysis, and induction
- conclude chapters `Basics.v` and `Induction.v`

Homework 1

`Basic.v` is due September 12, Wednesday, 11:59pm EST

Submit it via email: Tiago.Cogumbreiro@umb.edu

An example

```
Example plus_O_4 : 0 + 5 = 4.
Proof.
```

> How do we prove this?

An example

```
Example plus_O_4 : 0 + 5 = 4.
Proof.
```

> How do we prove this?

- We cannot. This is unprovable, which means we are not able to write a script that proves this statement.
- Coq will **not** tell you that a statement is false.

Another example

```
Example plus_0_5 : 0 + 5 = 5.
Proof.
```

> How do we prove this? We "know" it is true, but why do we know it is true?

Another example

```
Example plus_O_5 : 0 + 5 = 5.
Proof.
```

> How do we prove this? We "know" it is true, but why do we know it is true?

There are two ways:

1. We can think about the definition of plus.
2. We can brute-force and try the tactics we know (`simpl`, `reflexivity`)

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
    | 0 ⇒ m
    | S n' ⇒ S (plus n' m)
  end.

Notation "x + y" := (plus x y) (at level 50, left associativity) : nat_scope.
```

Another example

```
Example plus_0_6 : 0 + 6 = 6.
Proof.
```

> How do we prove this?

Another example

```
Example plus_0_6 : 0 + 6 = 6.
Proof.
```

> How do we prove this?

The same as we proved `plus_0_5`. This result is true for any natural n!

Ranging over all elements of a set

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.
  intros n.
  simpl.
  reflexivity.
Qed.
```

- `Theorem` is just an *alias for* `Example` *and* `Definition`.
- `forall` introduces a variable of a given type, eg `nat`; the logical statement must be true for all elements of the type of that variable.
- Tactic `intros` is the dual of `forall` in the tactics language

Forall example

Given

```
1 subgoal
______________________________________(1/1)
forall n : nat, 0 + n = n
```

and applying `intros n` yields

```
1 subgoal
n : nat
______________________________________(1/1)
0 + n = n
```

The `n` is a variable name of your choosing.

> Try replacing `intros n` by `intros m`

simpl and reflexivity work under forall

```
1 subgoal
______________________________________(1/1)
forall n : nat, 0 + n = n
```

Applying `simpl` yields

```
1 subgoal
______________________________________(1/1)
forall n : nat, n = n
```

Applying `reflexivity` yields

```
No more subgoals.
```

`reflexivity` also simplifies terms

```
1 subgoal
______________________________________(1/1)
forall n : nat, 0 + n = n
```

Applying `reflexivity` yields

```
No more subgoals.
```

Summary

- `simpl` and `reflexivity` work under `forall` binders
- `simpl` only unfolds definitions of the *goal*; does not conclude a proof
- `reflexivity` concludes proofs and simplifies

Multiple pre-conditions in a lemma

```
Theorem plus_id_example : forall n m:nat,
  n = m ->
  n + n = m + m.
Proof.
  intros n.
  intros m.
```

Multiple pre-conditions in a lemma

```
Theorem plus_id_example : forall n m:nat,
  n = m ->
  n + n = m + m.
Proof.
  intros n.
  intros m.
```

yields

```
1 subgoal
n, m : nat
______________________________________(1/1)
n = m -> n + n = m + m
```

Multiple pre-conditions in a lemma

applying `intros H` yields

```
1 subgoal
n, m : nat
H : n = m
______________________________________(1/1)
n + n = m + m
```

How do we use `H`? **New tactic:** use `rewrite → H` (lhs becomes rhs)

```
1 subgoal
n, m : nat
H : n = m
______________________________________(1/1)
m + m = m + m
```

> How do we conclude? Can you write a `Theorem` that replicates the proof-state above?

How do we prove this example?

```
Theorem plus_1_neq_0_firsttry : forall n : nat,
  beq_nat (plus n 1) 0 = false.
Proof.
  intros n.
```

yields

```
1 subgoal
n : nat
______________________________________(1/1)
beq_nat (plus n 1) 0 = false
```

How do we prove this example?

```
Theorem plus_1_neq_0_firsttry : forall n : nat,
  beq_nat (plus n 1) 0 = false.
Proof.
  intros n.
```

yields

```
1 subgoal
n : nat
______________________________________(1/1)
beq_nat (plus n 1) 0 = false
```

Apply `simpl` and it does nothing. Apply `reflexivity`:

```
In environment
n : nat
```

Why does simpl fail?

Q: Why can't `beq_nat (n + 1)` be simplified? (Hint: inspect its definition.)

Why does simpl fail?

Q: Why can't `beq_nat (n + 1)` be simplified? (Hint: inspect its definition.)

A: `beq_nat` expects the first parameter to be either `0` or `S ?n`, but we have an expression `n + 1` (or `plus n 1`).

Why does simpl fail?

Q: Why can't `beq_nat (n + 1)` be simplified? (Hint: inspect its definition.)

A: `beq_nat` expects the first parameter to be either `0` or `S ?n`, but we have an expression `n + 1` (or `plus n 1`).

Q: Can we simplify `plus n 1`?

Why does simpl fail?

Q: Why can't `beq_nat (n + 1)` be simplified? (Hint: inspect its definition.)

A: `beq_nat` expects the first parameter to be either `0` or `S ?n`, but we have an expression `n + 1` (or `plus n 1`).

Q: Can we simplify `plus n 1`?

A: No because `plus` decreases on the first parameter, not on the second!

Case analysis (1/3)

Let us try to inspect value `n`. Use: `destruct n as [| n']`.

```
2 subgoals
______________________________________(1/2)
beq_nat (0 + 1) 0 = false
______________________________________(2/2)
beq_nat (S n' + 1) 0 = false
```

Now we have two goals to prove!

```
1 subgoal
______________________________________(1/1)
beq_nat (0 + 1) 0 = false
```

How do we prove this?

Case analysis (2/3)

After we conclude the first goal we get:

```
This subproof is complete, but there are some unfocused goals:

______________________________________(1/1)
beq_nat (S n' + 1) 0 = false
```

Use another bullet (-).

```
1 subgoal
n' : nat
______________________________________(1/1)
beq_nat (S n' + 1) 0 = false
```

And prove the goal above as well.

Case analysis (3/3)

- Use: `destruct n as [| n']` when you want to explicitly name the variables being introduced
- Otherwise, use: `destruct n` and let Coq automatically name the variables.

> Using automatically generated variable names makes the proofs more brittle to change.

Basic.v

- New syntax: `forall` to range over all values of a type
- New syntax: `Theorem` and its relation with `Definition` and `Example`
- New tactic: `intros`
- Learn: interplay between `forall`, `simpl`, and `reflexivity`
- New syntax: `->` to represent implication
- New tactic: `rewrite` to replace terms using equality
- New tactic: `destruct` to perform case analysis
- New tactic: bullets (`-`, `*`, and `+`) and scopes (`{` and `}`)

Compile `Basic.v`

CoqIDE:

- Open Basics.v. In the "Compile" menu, click on "Compile Buffer".

Console:

- `make Basics.vo`

Induction.v

Example: prove this lemma (1/4)

```
Theorem plus_n_O : forall n:nat,
  n = n + 0.
Proof.
```

Example: prove this lemma (1/4)

```
Theorem plus_n_O : forall n:nat,
  n = n + 0.
Proof.
```

Tactic `simpl` does nothing.

Example: prove this lemma (1/4)

```
Theorem plus_n_O : forall n:nat,
  n = n + 0.
Proof.
```

Tactic `simpl` does nothing. Tactic `reflxivity` fails.

Example: prove this lemma (1/4)

```
Theorem plus_n_O : forall n:nat,
  n = n + 0.
Proof.
```

Tactic `simpl` does nothing. Tactic `reflxivity` fails. Apply `destruct n`.

```
2 subgoals
______________________________________(1/2)
0 = 0 + 0
______________________________________(2/2)
S n = S n + 0
```

Example: prove this lemma (2/4)

After proving the first, we get

```
1 subgoal
n : nat
______________________________________(1/1)
S n = S n + 0
```

Applying `simpl` yields:

```
1 subgoal
n : nat
______________________________________(1/1)
S n = S (n + 0)
```

Example: prove this lemma (2/4)

After proving the first, we get

```
1 subgoal
n : nat
______________________________________(1/1)
S n = S n + 0
```

Applying `simpl` yields:

```
1 subgoal
n : nat
______________________________________(1/1)
S n = S (n + 0)
```

Tactic `reflexivity` fails and there is nothing to rewrite.

We need an induction principle of `nat`

For some property `P` we want to prove.

- Show that $P(0)$ holds.
- Given the induction hypothesis $P(n)$, show that $P(n + 1)$ holds.

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

Example: prove this lemma (3/4)

Apply `induction n.`

```
2 subgoals
______________________________________(1/2)
0 = 0 + 0
______________________________________(2/2)
S n = S n + 0
```

> How do we prove the first goal?
> Compare `induction n` with `destruct n.`

Example: prove this lemma (4/4)

After proving the first goal we get

```
1 subgoal
n : nat
IHn : n = n + 0
______________________________________(1/1)
S n = S n + 0
```

applying `simpl` yields

```
1 subgoal
n : nat
IHn : n = n + 0
______________________________________(1/1)
S n = S (n + 0)
```

> How do we conclude this proof?

Intermediary results

```
Theorem mult_0_plus' : forall n m : nat,
  (0 + n) * m = n * m.
Proof.
  intros n m.
  assert (H: 0 + n = n). { reflexivity. }
  rewrite -> H.
  reflexivity. Qed.
```

- `H` is a variable name, you can pick whichever you like.
- Your intermediary result will capture all of the existing hypothesis.
- It may include `forall`.
- We use braces { and } to prove a sub-goal.

Formal versus informal proofs

- The objective of a mechanical (formal) proofs is to appease the proof checker.
- The objective of an informal proof is to convince (logically) the reader.
- `ltac` proofs are imperative, assume the reader can step through
- In informal proofs we want to help the reader reconstruct the proof state.

An example of an `ltac` proof

```
Theorem plus_assoc : forall n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p. induction n as [| n' IHn'].
  - reflexivity.
  - simpl. rewrite -> IHn'. reflexivity. Qed.
```

1. The proof follows by induction on n.

An example of an `ltac` proof

```
Theorem plus_assoc : forall n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p. induction n as [| n' IHn'].
  - reflexivity.
  - simpl. rewrite -> IHn'. reflexivity. Qed.
```

1. The proof follows by induction on n.
2. In the base case, we have that $n = 0$. We need to show $0 + (m + p) = 0 + m + p$, which follows by the definition of $+$.

An example of an `ltac` proof

```
Theorem plus_assoc : forall n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p. induction n as [| n' IHn'].
  - reflexivity.
  - simpl. rewrite -> IHn'. reflexivity. Qed.
```

1. The proof follows by induction on n.
2. In the base case, we have that $n = 0$. We need to show $0 + (m + p) = 0 + m + p$, which follows by the definition of $+$.
3. In the inductive case, we have $n = \texttt{S}\ n'$ and must show $Sn' + (m + p) = Sn' + m + p$. From the definition of $+$ it follows that $\texttt{S}\ (n' + (m + p)) = \texttt{S}\ (n' + m + p)$. The proof concludes by applying the induction hypothesis $n' + (m + p) = n' + m + p$.

Induction.v

- Learn: how to compile `Basic.v`
- Learn: induction principle for natural numbers.
- New tactic: `induction`
- New tactic: `assert`
- Learn: formal vs informal proofs

Ltac vocabulary

- `simpl`
- `reflexivity`
- `intros`
- `rewrite`
- `destruct`
- `induction`
- `assert`