CS720

Logical Foundations of Computer Science

Lecture 2: A proof primer

Tiago Cogumbreiro

Programers program every day

Programers program every day

- **There are no tests**, so no way to invest time later.
- You have a **weekly** load of work, don't let it build up.
- To master Coq, you must practice every day.
- Once you master Coq, the course is accessible.

On studying effectively for this course

- **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 cover certain details of a difficult exercise.
- **Use the office hours and use Discord:** Coq is a unusual programming language, so you will get stuck simply because you are not familiar with the IDE or with a quirk of the language.

On studying effectively for this course

Setup

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

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` (confirm this with Gradescope)
4. Make sure you solve all manually-graded exercises (Gradescope won't notify you of this)

Compound types

Compound types

Enumerated types are very simple. You can think of them as a typed collection of constants. We call each enumerated value a **constructor**.

```
Inductive rgb : Type :=
  | red : rgb
  | green : rgb
  | blue : rgb.
```

Compound types

Enumerated types are very simple. You can think of them as a typed collection of constants. We call each enumerated value a **constructor**.

```
Inductive rgb : Type :=
  | red : rgb
  | green : rgb
  | blue : rgb.
```

A **compound type** builds on other existing types. Their constructors accept ***multiple parameters***, like functions do.

```
Inductive color : Type :=
  | black : color
  | white : color
  | primary : rgb → color.
```

Manipulating compound values

```
Definition monochrome (c : color) : bool :=
  match c with
  | black ⇒ true
  | white ⇒ true
  | primary p ⇒ false
  end.
```

Manipulating compound values

```
Definition monochrome (c : color) : bool :=
  match c with
  | black ⇒ true
  | white ⇒ true
  | primary p ⇒ false
  end.
```

We can use the place-holder keyword `_` to mean a variable we do not mean to use.

```
Definition monochrome (c : color) : bool :=
  match c with
  | black ⇒ true
  | white ⇒ true
  | primary _ ⇒ false
  end.
```

Compound types

Allows you to: type-tag, fixed-number of values

Inductive types

Inductive types

How do we describe arbitrarily large/composed values?

Inductive types

How do we describe arbitrarily large/composed values?

Here's the definition of natural numbers, as found in the standard library:

```
Inductive nat : Type :=
  | O : nat
  | S : nat → nat.
```

- `O` is a constructor of type `nat`.
  Think of the numeral `0`.
- If `n` is an expression of type `nat`, then `S n` is also an expression of type `nat`.
  Think of expression `n + 1`.

> What's the difference between `nat` and `uint32`?

Example

Let us implement `is_zero`

Recursive functions

Recursive functions

Recursive functions are declared differently with `Fixpoint`, rather than `Definition`.

Let us implement addition.

Recursive functions

Recursive functions are declared differently with `Fixpoint`, rather than `Definition`.

Let us implement addition.

```
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.
```

Using `Definition` instead of `Fixpoint` will throw the following error:

`The reference eqb was not found in the current environment.`

Not all recursive functions can be described. Coq has to understand that one value is getting "smaller."

All functions must be total: all inputs must produce one output. ***All functions must terminate.***

Back to proofs

An example

```
Example plus_0_4 : 0 + 5 = 4.
Proof.
```

How do we prove this?

An example

```
Example plus_0_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_O_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_0_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`)

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?

Computing equality of naturals

Computing equality of naturals

```
Fixpoint eqb (n1:nat) (n2:nat) : bool :=
  match n1 with
  | O =>
    match n2 with
    | O => true
    _ => false
    end
  | S n1' =>
    match n2 with
    | O => false
    | S n2' => eqb n1 n2
    end
  end.
```

How do we prove this example?

```
Require Import Nat.
Theorem plus_1_neq_0_firsttry : forall n : nat,
  eqb (plus n 1) 0 = false.
Proof.
  intros n.
```

yields

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

How do we prove this example?

```
Require Import Nat.
Theorem plus_1_neq_0_firsttry : forall n : nat,
  eqb (plus n 1) 0 = false.
Proof.
  intros n.
```

yields

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

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

```
In environment
n : nat
Unable to unify "false" with "eqb (plus n 1) 0".
```

Why does simpl fail?

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

Why does simpl fail?

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

A: `eqb` 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 `eqb (n + 1)` be simplified? (Hint: inspect its definition.)

A: `eqb` 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 `eqb (n + 1)` be simplified? (Hint: inspect its definition.)

A: `eqb` 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)
eqb (0 + 1) 0 = false
______________________________________(2/2)
eqb (S n' + 1) 0 = false
```

Now we have two goals to prove!

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

Use another bullet (-).

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

And prove the goal above as well.

Why can this goal be simplified to `false = false`?

```
________________________________________(1/1)
eqb (S n' + 1) 0 = false
```

Why can this goal be simplified to `false = false`?

```
_____________________________________(1/1)
eqb (S n' + 1) 0 = false
```

1. because `S n' + 1 = S (n' + 1)` (follows the second branch of `plus`)
2. because `eqb (S ...) 0 = false` (follows the second branch of `eqb`)