CS420

Introduction to the Theory of Computation

Lecture 14: A primer on the Coq programming language

Tiago Cogumbreiro
On studying effectively for this content

Setup

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

Textbook

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 an 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 content

Exercises structure

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

A primer on the programming language Coq

We will learn the core principles behind Coq
Enumerated type

A data type where the user specifies the various distinct values that inhabit the type.

Examples?
Enumerated type

A data type where the user specifies the various distinct values that inhabit the type.

Examples?

- boolean
- 4 suits of cards
- byte
- int32
- int64
Inductive day : Type :=
  | monday : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.

- Inductive defines an (enumerated) type by cases.
- The type is named day and declared as a : Type (Line 1).
- Enumerated types are delimited by the assignment operator (:=) and a dot (.).
- Type day consists of 7 cases, each of which is is tagged with the type (day).
Printing to the standard output

Compute prints the result of an expression (terminated with dot):

```
Compute monday.
```

prints

```
= tuesday
: day
```
Interacting with the outside world

- Programming in Coq is different from most popular programming paradigms.
- Programming is an **interactive** development process.
- The IDE is very helpful: workflow similar to using a debugger.
- It's a REPL on steroids!
- Compute evaluates an expression, similar to `printf`.

A primer on the Coq programming language

Lecture 14

Tiago Cogumbreiro
Inspecting an enumerated type

```coq
match d with
| monday  ⇒ tuesday
| tuesday ⇒ wednesday
| wednesday ⇒ thursday
| thursday ⇒ friday
| friday  ⇒ monday
| saturday ⇒ monday
| sunday  ⇒ monday
end
```
Inspecting an enumerated type

```coq
match d with
  | monday ⇒ tuesday
  | tuesday ⇒ wednesday
  | wednesday ⇒ thursday
  | thursday ⇒ friday
  | friday ⇒ monday
  | saturday ⇒ monday
  | sunday ⇒ monday
end
```

- match performs **pattern matching** on variable d.
- Each pattern-match is called a **branch**; the branches are delimited by keywords with and end.
- Each *branch* is prefixed by a mid-bar (\(\mid\)) (⇒), a pattern (eg, monday), an arrow (⇒), and a return value
Pattern matching example

Compute match monday with
| monday  ⇒  tuesday
| tuesday ⇒  wednesday
| wednesday  ⇒  thursday
| thursday  ⇒  friday
| friday  ⇒  monday
| saturday  ⇒  monday
| sunday  ⇒  monday
end.
Create a function

**Definition** `next_weekday (d: day) : day :=`

```coq
match d with
| monday ⇒ tuesday
| tuesday ⇒ wednesday
| wednesday ⇒ thursday
| thursday ⇒ friday
| friday ⇒ monday
| saturday ⇒ monday
| sunday ⇒ monday
end.
```
Create a function

```
Definition next_weekday (d: day) : day :=
  match d with
  | monday => tuesday
  | tuesday => wednesday
  | wednesday => thursday
  | thursday => friday
  | friday => monday
  | saturday => monday
  | sunday => monday
  end.
```

- Definition is used to declare a function.
- In this case `next_weekday` has one parameter `d` of type `day` and returns (`:`) a value of type `day`.
- Between the assignment operator (`:=`) and the dot (`.`), we have the body of the function.
Example 2

```
Compute (next_weekday friday).
```

yields (Message pane)

```
  = monday
  : day
```

next_weekday friday is the same as monday (after evaluation)
Example test_next_weekday:
next_weekday (next_weekday saturday) = tuesday.

Proof.
  simpl.  (* simplify left-hand side *)
  reflexivity.  (* use reflexivity since we have tuesday = tuesday *)
Qed.
Example test_next_weekday:
  next_weekday (next_weekday saturday) = tuesday.
Proof.
  simpl. (* simplify left-hand side *)
  reflexivity. (* use reflexivity since we have tuesday = tuesday *)
Qed.

- Example prefixes the name of the proposition we want to prove.
- The return type (:) is a (logical) **proposition** stating that two values are equal (after evaluation).
- The body of function test_next_weekday uses the Ltac proof language.
- The dot (.) after the type puts us in proof mode. (Read as "defined below").
- This is essentially a unit test.
Ltac: Coq's proof language

Ltac is **imperative**! You can step through the state with CoqIDE

Proof begins an ltac-scope, yielding

1 subgoal

```
______________________________________(1/1)
next_weekday (next_weekday saturday) = tuesday
```

Tactic `simpl` evaluates expressions in a goal (normalizes them)
Ltac: Coq's proof language

1 subgoal
______________________________________(1/1)
tuesday = tuesday
  • reflexivity solves a goal with a pattern ?X = ?X

No more subgoals.
  • Qed ends an ltac-scope and ensures nothing is left to prove
Function types

Use Check to print the type of an expression:

```
Check next_weekday.
```

which outputs

```
next_weekday : day -> day
```

Function type `day -> day` takes one value of type `day` and returns a value of type `day`. 
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**.

```coq
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 \rightarrow 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

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:

\[
\text{Inductive } \text{nat} \text{ : Type :=}
\begin{align*}
| \text{O} & : \text{nat} \\
| \text{S} & : \text{nat} \to \text{nat}.
\end{align*}
\]

- \(0\) 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 \text{nat} and \text{uint32}?
Recursive functions

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

```coq
Fixpoint evenb (n:nat) : bool :=
  match n with
  | O ⇒ true
  | S O ⇒ false
  | S (S n') ⇒ evenb n'
end.
```

Using Definition instead of Fixpoint will throw the following error:

The reference evenb 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 proving
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.
  - Because it is unprovable, there is no proof script that can satisfy this claim.

Instead, we can prove the following (later)

Example plus_0_5_not_4 : 0 + 5 <> 4.
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_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 **understand** the definition of plus and use that to our advantage.
2. We **brute-force** and try the tactics we know (simpl, reflexivity)

```coq
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!
Theorem **plus_0_n** : \( \forall n : \text{nat}, 0 + n = n. \)

Proof.

```coq
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, \( \emptyset + n = n \)

Applying simpl yields
1 subgoal
______________________________________(1/1)
forall n : nat, n = n
Applying reflexivity yields
No more subgoals.
reflexivity also simplifies terms

1 subgoal
_____________________________\(1/1\)
\(\forall n : \text{nat}, 0 + n = n\)

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
Basic.v

- New syntax: `Definition` declares a non-recursive function
- New syntax: `Compute` evaluates an expression and outputs the result + type
- New syntax: `Check` prints the type of an expression
- New syntax: `Inductive` defines inductive data structures
- New syntax: `Fixpoint` declares a (possibly) recursive function
- New syntax: `match` performs pattern matching on a value
- New tactic: `simpl` evaluates functions if possible
- New tactic: `reflexivity` concludes a goal \(?X = ?X\)
Ltac vocabulary

- `simpl`
- `reflexivity`