Introduction to the Theory of Computation

Lecture 2: Pattern matching; reflexivity

Tiago Cogumbreiro
Today we will learn...

- Compound types
- Pattern matching
- Inductive types
- Recursive functions
- Proofs with forall

Chapter: Basics.v
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

(demo)
Back learning the basics
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 \rightarrow day
```

Function type \( \text{day} \rightarrow \text{day} \) takes one value of type \text{day} and returns a value of type \text{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**.

```ocaml
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**  \( \text{monochrome} \ (c : \text{color}) : \text{bool} := \)
\[
\text{match } c \text{ with}
\]
\[
| \text{black} \Rightarrow \text{true} \\
| \text{white} \Rightarrow \text{true} \\
| \text{primary} \ p \Rightarrow \text{false}
\]
\[
\text{end.}
\]

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

**Definition**  \( \text{monochrome} \ (c : \text{color}) : \text{bool} := \)
\[
\text{match } c \text{ with}
\]
\[
| \text{black} \Rightarrow \text{true} \\
| \text{white} \Rightarrow \text{true} \\
| \text{primary} \ _ \Rightarrow \text{false}
\]
\[
\text{end.}
\]
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:

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

- 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 nat and uint32?
Recursive functions

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

```coq
Fixpoint evenb (n:nat) : bool :=
  match n with
  | 0 => true
  | S 0 => 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.
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.

(* See Nat.add *)
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 $\text{plus}_06 : 0 + 6 = 6$.
Proof.

How do we prove this?

The same as we proved $\text{plus}_05$. This result is true for any natural $n$!
Ranging over all elements of a set

Theorem \(\text{plus}_0\_n\) : \(\forall n : \text{nat}, \ 0 + n = n\).
Proof.

\begin{verbatim}
intros n.
simpl.
reflexivity.
Qed.
\end{verbatim}

- 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

forall n : nat, 0 + n = n

Applying simpl yields

1 subgoal

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