Introduction to Coq
Part 1: the calculus of inductive constructions
and inductive types

Yves Bertot

September 2023
A tutorial about Coq

Objectives of Coq session

- Write mathematical statements
- Mark some of these statements as “proved”
- Record the proofs for later analysis
- Perform some guaranteed computations
Bare metal and library extensions

- User interfaces: jscoq, coq-lsp, vscoq, coqide, emacs
  - Follow download instructions from https://coq.inria.fr
  - In a hurry, use https://coq.vercel.app/
  - For a clean sheet, https://coq.vercel.app/scratchpad.html

- The most basic commands
  - Check: just verify that a formula is well formed
  - Compute: force the computation

- Working with knowledge that has already been formalized:
  - loading libraries
    - Loading elementary arithmetic       Require Import Arith.
    - More advanced arithmetic           Require Import ZArith.
    - Some datastructures                Require Import List
Expressions are made of functions applied to arguments

Variables receive their value a function application, forever
  - There is no assignment construct that can change the value of a variable

Anonymous functions can be written by the user for immediate use

A point of syntax: parenthesis are not used to represent function application

Some predefined functions have an infix syntax
Function usage

Check Nat.add 3 5. (* the result shows predefined notation. *)

Check (fun x => 3 + x). (* temporary use of x *)

Check (fun x => 3 + x) 5. (* at execution x receives 5 *)

Compute (fun x => 3 + x) 5.

Fail Check x. (* x only exists inside the scope of the function *)

Note the syntax to write a function applied to two arguments
Parentheses are not needed to represent function application
The command Check not only verifies that an expression is correctly written, it also give its type

A function with two arguments of type $\text{nat}$ returning a value of type $\text{nat}$ has the following type

$$\text{nat} \to \text{nat} \to \text{nat}$$

The arrow $\to$ is not associative, but implicit parenthesis are as follows:

$$\text{nat} \to (\text{nat} \to \text{nat})$$
Sorts and Families of types

- Some types can a number parameter
  - The type of vectors of a given size
  - The type of numbers under a given bound
- These types are represented by functions whose output is in a type of types
- Three types of types are given Set, Type, and Prop
  - Types of types are called sort
- For instance nat has type Set
- A type of vectors could have type Type -> nat -> Type
- A type of bounded numbers could have type nat -> Set
Dependent types

Let’s assume the existence of a type \( \text{vector} : \text{Type} \to \text{nat} \to \text{Type} \)

What would be the type of a function that takes as input a natural number \( n \) and returns a vector of zeros, of length \( n \)?

\[ \text{mk0vector} : \forall n : \text{nat}, \text{vector} \text{nat} n \]

If the zeros are taken in an existing field type \( K \), the type would be:

\[ \text{mk0vector'} : \forall n : \text{nat}, \text{vector} K n \]

\( \forall \) is often used instead of \( \forall \), theoretical lecture also calls this a product type, using \( \Pi \) as notation.
Dependent types

▶ Let’s assume the existence of a type \(\text{vector} : \text{Type} \rightarrow \text{nat} \rightarrow \text{Type}\)

▶ What would be the type of a function that takes as input a natural number \(n\) and returns a vector of zeros, of length \(n\)?

▶ \(\text{mk0vector} : \forall n : \text{nat}, \text{vector} \text{nat} n\)

▶ If the zeros are taken in an existing field type \(K\), the type would be:

\(\text{mk0vector}' : \forall n : \text{nat}, \text{vector} K n\)

▶ \(\forall\) is often used instead of \(\text{forall}\), theoretical lecture also calls this a product type, using \(\prod\) as notation
The logic of dependent types

- A universally quantified theorem is a function that yields baby theorems for every inputs

- **If** $T_1$ is the theorem that says that every natural number can be decomposed uniquely into a product of prime numbers, **then** $T_1 \ 24$ is a theorem that says that 24 can be decomposed . . .

- In this way, $\forall$ can really be read as a logical universal quantification

- This relies on the fact that the theorem statement is understood as a type

- The sort $\text{Prop}$ is especially dedicated to types that are used to denote mathematical statements
Inductive Types

- New types can be defined by providing constructors and deducing a destructor by a minimality argument
- Running example a set of three elements

Inductive mod3 : Type := Zero | One | Two.

Check Zero.

Definition mod3_to_nat (x : mod3) : nat :=
    match x with Zero => 0 | One => 1 | Two => 2 end.

Definition mod3_succ (x : mod3) : mod3 :=
    match x with Zero => One | One => Two | Two => Zero end.

- The minimality principle is in the match construct
- Only required closes are Zero, One, and Two
Proofs, the bare metal way

Definition le_2_2 : 2 <= 2 := le_n 2.

Definition le_1_2 : 1 <= 2 := le_S 1 1 (le_n 1).

Definition le_0_2 : 0 <= 2 := le_S 0 1 (le_S 0 0 (le_n 0)).

Definition mod3_to_nat_le_2 (x : mod3) :
    mod3_to_nat x <= 2 :=
    match x with
    | Zero => le_0_2
    | One => le_1_2
    | Two => le_2_2
    end.
The elimination principle, for proofs

- Proving that a property holds for all elements of an inductive type
- One **only** needs to check that property for every constructor
  - The minimality principle that I mentioned before

Definition mod3_cases (P : mod3 -> Prop) (x : mod3) (h0 : P Zero) (h1 : P One) (h2 : P Two) : P x :=
match x with
| Zero => h0
| One  => h1
| Two  => h2
end.
Inductive types with recursion

- Each constructor may be a function
- Arguments of the function may belong to the type being defined

\[
\text{Inductive list (A : Type) : Type := } \\
\quad \mid \text{nil : list A} \\
\quad \mid \text{cons : A \rightarrow list A \rightarrow list A.}
\]

Check cons nat 3 (cons nat 2 (cons nat 1 (nil nat))).
structural recursion

- elements of inductive types with recursion can contain arbitrary large amounts of information
- Recursive programming can handle all this data in computations
- The command to define a recursive function is called Fixpoint
- Restricted recursion by comparison with conventional functional programming
- Guaranteed termination achieved through a syntactic criterion
- Recursive calls only allowed on subterms obtained by pattern-matching
- The generic reasoning principle (akin to `mod3_cases`) is an induction principle (with induction hypothesis)
Example recursive programming with lists

Fixpoint fold_right (A B : Type) (f : A -> B -> B)(v : B)(l : list A) : B :=
match l with
| nil _ => v
| cons _ x tl => f x (fold_right A B f v tl)
end.

Compute fold_right nat nat nat Nat.add 0
   (cons nat 3 (cons nat 2 (cons nat 1 (nil nat)))).
Matters of productivity and efficiency

- The predefined package of lists is more practical to use than the type shown in these slides
- Notations and implicit arguments make it possible to avoid writing obvious arguments
- Lists are linear representations of data collections, with an access cost that is linear with respect to the amount of stored data
  - conventional programming languages like OCaml provide quasi constant access
  - Other data-structures, like binary search trees or tries, provide much faster access
- Numbers have the same variability in efficiency
  - Binary structures are used to represent integers
  - Addition, multiplication, division are natural to program structurally
  - Other functions require inventiveness
For the record: factorial function with binary numbers

Require Import ZArith.

Fixpoint fact’ (p : positive) (offset : Z) : Z :=
match p with
| xH => (offset + 1)%Z
| xO p => fact’ p offset * (fact’ p (offset + (Zpos p)))
| xI p => (2 * (Zpos p) + 1 + offset) *
          fact’ p offset * fact’ p (offset + (Zpos p))
end.

Definition Zfact (x : Z) : Z :=
match x with | Zpos p => fact’ p 0 | _ => 1%Z end.

Compute Zfact 50.

Computing large factorials will fail in the web-browser, but other
instances of Coq will have no problems
dependent families of inductive types

- The type `list` already presented is actually a family of inductive types.
- The parameter may be a piece of data, and the type may be empty or inhabited depending on the parameter.
- The simplest example: identity

```coq
Inductive eq (A : Type) (x : A) : A -> Prop :=
  eq_refl : eq A x x.
```

- This is how equality is represented in Coq.
- The generic reasoning principle (like `mod3` above) has an important meaning:
  - if `eq A x y` holds, then every property that holds for `x` also holds for `y`
Pervasive use of inductive families of types for logic

- Logical connectives such as conjunction, disjunction, Truth, and Falsehood are described as inductive types or type families
- Existential quantification also
- Equality also
- Constructors give introduction rules, reasoning principles (based on pattern-matching) give elimination rules
- In proofs, this will be made apparent by the use of a single proof command for several behaviors
Existing data structures

- Natural numbers, type `nat`, interpretation by default of arithmetic notations, more functions available after: Require Import Arith.
- Integers, type `Z`, based on a binary encoding, available after: Require Import ZArith. Arithmetic notations can aim to this type after a simple command.
- Rational numbers, type `Q`, available after: Require Import QArith.
- Lists, type `list`, available after: Require Import List.
- Various forms of binary trees, with efficient adding and lookup functions.
- Computation can be performed for recursive functions on these datatypes, using the `Compute` command.
Proofs from the practical side

- Logical statements are types
- When there is an element in the type, the statement is proved
- Making proofs is constructing objects in types
- This can be done by writing programs (as was shown already)
- This is impractical for proofs of reasonable statements
  - It is practical in Agda, but the user-interface has been fine-tuned for that
- In Coq, one resorts to a proof mode, goals, and tactics
Demo: computing 10 digits of PI

Require Import Arith QArith.

Coercion Z.of_nat : nat -> Z.

Fixpoint atan_approx (n : nat) (x : Q) :=
  match n with
  | 0%nat => x
  | S p => (-1) ^ S p / (2 * S p + 1 # 1) *
      x ^ (2 * S p + 1) + atan_approx p x
  end.

Definition pi_digits (n m : nat) :=
  let v := 4 * (atan_approx n (1/2) + atan_approx n (1/3)) in
  (Qnum v * 10 ^ m / Zpos (Qden v))%Z.

Time Compute pi_digits 15 10.
(* result in less than 0.02 secs on my machine *)
Proof mode

▶ Entering the mode

Lemma example0 : forall (A : Prop) A -> A.
Proof.

============
forall A : Prop, A -> A

▶ The current goal is the statement we want to prove
▶ The next three commands called tactics will modify the goal
Transforming the goals

Lemma example0 : forall (A : Prop) A -> A.
Proof.
intros A hyp_A.

A : Prop
hyp_A : A
============
A

- The top of the bar is a *context*
  - It contains things that are assumed to exist
  - For instance, \( \text{hyp}_A \) : \( A \) means: “\( \text{hyp}_a \) is a proof of \( A \)”
- The text below the bar is what we need to prove
Lemma example0 : forall (A : Prop) A -> A.
Proof.
intros A hyp_A.
extact hyp_A.

No more goals

- When a solution is found for a goal, it disappears
- If there were several goals, the system displays the next one
- For beginners, this can be puzzling
  - the new goal may look that a transformation of the previous one, even though they are rather unrelated
Finishing a proof

Qed.

- You have to type Qed. at the end of a proof
- Otherwise
  - The theorem is not saved
  - You do not exit proof mode
  - You cannot start another proof
- Other ways to exit proof mode
  - Admitted. The theorem is saved, but recorded as not actually proved
  - Abort. The theorem is not saved
  - Defined. Like Qed., but different on a technicality
A large number of tactics

- Step tactics for basic logical connectives: intros, assert, apply, exact, destruct, split, left, right, exists
- Tactics for equality reasoning: reflexivity, rewrite, replace
- Tactics for defined functions: unfold, fold, change
- Specialized tactics for inductive types: induction, case, discriminate, injection, simpl, cbv
- Automation tactics: auto, tauto, intuition
- Domain specific automated tactics: ring, lia, lra, nia, nra, interval (only loaded upon request)
# A beginner’s tactic table

<table>
<thead>
<tr>
<th>Logical Connective</th>
<th>Hypothesis H</th>
<th>Conclusion</th>
<th>Hypothesis H</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒</td>
<td>apply H</td>
<td>intros H</td>
<td>destruct H as [H1 H2]</td>
<td>destruct H as [H1</td>
</tr>
<tr>
<td>∀</td>
<td>apply H</td>
<td>intros H</td>
<td>split</td>
<td></td>
</tr>
<tr>
<td>∨</td>
<td>destruct H as [x H1]</td>
<td>exists v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>¬</td>
<td>destruct H as [H1</td>
<td>H2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>rewrite H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>rewrite H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>rewrite H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>False</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:**
- `apply H` applies the hypothesis `H` to the conclusion.
- `destruct H` destructs the hypothesis `H`.
- `intros H` introduces the hypothesis `H`.
- `exists v` introduces the existential variable `v`.
- `split` splits the conclusion.
- `reflexivity` and `ring` are specific tactics used in certain logical settings.
Goal handling tactics

▶ exact will solve a goal by providing an assumption from the context that is the same

▶ assert (hyp_name : statement) will create two goals
  ▶ In the first you have to prove statement
  ▶ In the second you have an extra hypothesis hyp_name stating that statement holds

▶ Very useful to state intermediary steps in your proof, to make it more readable
Proofs by induction

- induction e will be available anytime e belongs to an inductive type
- The proof will follow a canonical structure, requiring to check all constructors of the inductive type, providing induction hypotheses when relevant
Demo time
Real numbers

- Real numbers cannot be described by inductive type
- We cannot use the `Compute` command to obtain a “better form” of a real number
- However, we can compute in proof
  - We can verify that two real numbers are equal
  - We can add an hypothesis that states an approximation of value