Coq Talk

Ed Morehouse
Outline

1. Introduction
2. The type system
3. Basic commands
4. Proofs and tactics
5. Dependent types in programs
6. Certified programs
7. Program extraction
what is it?

Coq is an interactive programming system based on the *Calculus of Inductive Constructions* (CIC). By the *Curry–Howard isomorphism* it is also an interactive proof assistant.
obtaining and using

- sponsored by *Institut national de recherche en informatique et en automatique* (INRIA) in France
- available on the web at coq.inria.fr
- current version: 8.2.
- interfaces: terminal (coqtop), windowed (CoqIDE), emacs (ProofGeneral), web (ProofWeb)
language(s)

three languages:

**specification language** (Gallina) for defining typed \(\lambda\)-terms in CIC. These can represent terms in the object language as well as propositions about these terms and proofs of these propositions.

**command language** (Vernacular) for manipulating the environment: importing modules, binding/unbinding identifiers, type-checking/evaluating terms, etc.

**tactic language** (Ltac) for building new tactics from old (e.g. sequencing, repeating, attempting tactics)
the type of a type is a sort.

the set of sorts is \( S = \{ \text{Prop} , \text{Set} , \text{Type}(i) \mid i : \mathbb{N} \} \)

\( \text{Prop} \) is the universe of logical propositions. Propositions are themselves the types of their proofs. Thus, \( p : P : \text{Prop} \) expresses that proposition \( P \) is proved by proof \( p \).

\( \text{Set} \) is the universe of program types or specifications. Specifications are themselves the types of program terms. For example, \( 3 : \text{nat} : \text{Set} \)

\( \text{Prop} \, \text{Set} : \text{Type}(0) \) and \( \text{Type}(i) : \text{Type}(i+1) \)
The index of \( \text{Type}(i) \) is not part of the concrete syntax but is inferred by the type checker. This results in a limited form of universe polymorphism.
defining terms and types

- **terms** are defined with the command `Definition`.
  
  ```coq
  Definition plus_two x := S (S x) .
  ```

- **inductive types** are defined with the command `Inductive`.
  
  ```coq
  Inductive list (A : Type): Type :=
  | nil : list A
  | cons : A → list A → list A
  .
  ```

- **recursive functions** are defined with the command `Fixpoint`.
  
  ```coq
  Fixpoint length {A : Type} (l : list A) : nat :=
  match l with
  | nil ⇒ 0
  | cons _ ls ⇒ S (length ls)
  end
  ```
proving propositions

proofs of propositions can be defined the same way:

```coq
Inductive leq : nat → nat → Prop :=
  | leq_n : ∀ n, leq n n
  | leq_S : ∀ n m, leq n m → leq n (S m)

Definition zero_leq_one : leq 0 1 :=
  (leq_S 0 0) (leq_n 0)
```

That is, `zero_leq_one : leq 0 1 : Prop`, so we have constructed a proof of `0 ≤ 1`.

This is seldom done. Instead, the interactive proof system is used.
interlude: some useful commands

**Check**  shows the type of a term (as well as information about context, implicit parameters, etc. that we don’t care about yet)

**Print**  shows the definition of a (transparent) term

**Locate**  shows the term behind a *notation* (for quoted string argument)

**Search**  lists all functions returning (proofs of) the argument type (proposition)

**SearchPattern**  search using wildcards

**Require Import**  loads a library

**Load**  loads a module

**Eval compute in**  compute the normal form of a term
An example:

Lemma S_comb :
\( \forall (A \; B \; C : \text{Prop}) , (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C \).

Proof .
intro A . intro B . intro C .
intro H1 . intro H2 . intro H3 .
apply H1 . assumption .
apply H2 . assumption .
Qed .

...which makes much more sense when seen interactively...
### Tactics for First-Order Constructive Logic

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>intro</code></td>
<td>$(\rightarrow i, \forall i)$ Adds binder/antecedent of current goal to premises. New goal becomes body/consequent of old goal.</td>
</tr>
<tr>
<td><code>split</code></td>
<td>$(\land i)$ Replaces a conjunction goal with a pair of new goals corresponding to the conjuncts.</td>
</tr>
<tr>
<td><code>left</code> / <code>right</code></td>
<td>$(\lor i)$ Replaces a disjunction goal with its first / second disjunct.</td>
</tr>
<tr>
<td><code>exists</code></td>
<td>$(\exists i)$ Replaces an existential goal with the specified instantiation.</td>
</tr>
<tr>
<td><code>apply</code></td>
<td>$(\rightarrow e, \forall e)$ Matches the current goal with the conclusion / body of a premise. A new goal is created for each antecedent / binder of the premise.</td>
</tr>
<tr>
<td><code>elim</code></td>
<td>$(\land e, \lor e, \exists e)$ Creates a new goal for each constructor of its argument premise. Each new goal is an implication whose antecedent is the argument of the respective constructor.</td>
</tr>
</tbody>
</table>

*(actually, split, left, right and exists are all special cases of tactic constructor)*
examples of predicate logic proofs

- conjunction is commutative:

  \[
  \text{Lemma and\_comm :}
  \forall (A \, B : \text{Prop}) , A \land B \rightarrow B \land A .
  \]

- properties of relations:

  \[
  \begin{align*}
  \text{Variable } D &: \text{Set} . \\
  \text{Variable } R &: D \rightarrow D \rightarrow \text{Prop} . \\
  \text{Hypothesis } R\text{-symmetric :} \\
  &\forall x \, y : D , R \, x \, y \rightarrow R \, y \, x . \\
  \text{Hypothesis } R\text{-transitive :} \\
  &\forall x \, y \, z : D , R \, x \, y \rightarrow R \, y \, z \rightarrow R \, x \, z . \\
  \text{Lemma refl\_if :} \\
  &\forall x : D , (\exists y : D , R \, x \, y) \rightarrow R \, x \, x .
  \end{align*}
  \]
negation and classical logic

**tactics:**

- **absurd** ($\bot e$) Replaces the current goal two goals: argument and its negation:

\[
\begin{align*}
P & \quad P \rightarrow \bot \\
\bot & \\
A
\end{align*}
\]

- **classical_left / classical_right** like left / right but adds negation of the other to hypotheses (Module Classical)

**examples:**

- law of the excluded middle:

```
Lemma LEM : \forall P , P \lor \neg P .
```

- not not elim:

```
Lemma NNE : \forall P , \neg \neg P \rightarrow P .
```
a larger example

the drinker’s paradox  At any non-empty bar, there is someone who, if he is drinking, then everyone is drinking:

```
Variable Person : Set .
Variable Harry : Person .
Variable Drinks : Person → Prop .

Lemma drinker :
  ∃ x : Person , Drinks x → ∀ (y : Person) , Drinks y .
```
tactics for equality

reflexivity  Eliminates the goal $x = x$.

symmetry  Replaces goal $u = v$ with goal $v = u$.

transitivity  Takes a term argument, $t$, and replaces goal $u = v$ with the two goals $u = t, t = v$.

rewrite (→ / ⇐)  Takes as argument a hypothesis equality and rewrites the goal by this equality in the direction specified.

replace ... with ...  Replaces all free occurrences of the first argument with the second in the current goal and adds as a new goal that the two are equal.
examples of equality proofs

Variable f : nat -> nat.

Hypothesis h : f 0 = 0.

Hypothesis f_1_0 : f 1 = f 0.

Lemma L2 : f (f 1) = 0.
Coq automatically defines induction rules for all inductive types defined by the user, as well as for the logical constants, $\land$, $\lor$, $\exists$, which are also defined inductively. There are several tactics for induction and case analysis. For now, we can get by with just the elim rule mentioned earlier. Note that this elimination rule is dual to the introduction rule constructor (of which our logical constant intro rules are instances).

example:

```
Lemma plus_n_0 : \forall (n : nat), n = n + 0 .
```

```
Lemma plus_n_S : \forall (n m : nat), S (n + m) = n + S m .
```

```
Lemma plus_comm : \forall (n m : nat), n + m = m + n .
```

Ed Morehouse
Coq Talk
existence, inductively defined

It’s worth having a look at how ∃ is defined as an inductive predicate (on predicates):

```
Inductive ex {A : Type} (P : A → Prop) : Prop :=
ex_intro : ∀ (x : A), P x → ex P
```

this is just the dependent product type, Σ, which has introduction rule

```
\frac{a: A \quad b: P(a)}{(a, b): \exists_A. P} \exists_i
```

As an inductive type, the elimination rule for ∃ is its induction rule.

example:

```
Lemma ∃_leq :
  ∀ (n m : nat), (∃ x : nat, n + x = m) → n ≤ m .
```
An *inversion strategy* looks at an instance of the conclusion of an inference rule and infers the instances of the premises necessary for the conclusion to have been derived by this rule.

In Coq, we can use the tactic, `inversion` on inductive predicates to infer the arguments of their constructors.

example:

```coq
Inductive even : nat → Prop :=
| Even0 : even 0
| EvenSS : ∀ n, even n → even (S (S n))

Lemma even_pp : ∀ n : nat, even (S (S n)) → even n.
```
Coq comes with a number of automated tactics implementing decision procedures and semi-decision procedures. These can be augmented by the user by writing new tactics in Ltac or in some other language (e.g. OCaml). Before accepting a proof, Coq extracts a proof term from the alleged proof and type checks it, thereby rejecting any bogus proofs potentially created by a tactic (c.f. DeBruijn criterion).

Some notable automated tactics:

- **auto / eauto**: use PROLOG-style resolution and a hints database to solve atomic goals
- **tauto / intuition**: implements a decision procedure for constructive propositional logic. (intuition is able to translate subgoals into equivalent ones that tauto can solve)
- **omega**: fully automated procedure for propositional logic and Presburger arithmetic (equalities and inequalities over \( \mathbb{N} \) and \( \mathbb{Z} \))
- **ring**: solves equations in polynomials over a ring
So far we have seen dependent types only through the Curry-Howard lens in quantifiers. But dependent types don’t have to live in \textit{Prop}:

```coq
Inductive vector (A : Type) : nat -> Type :=
  | vnil : vector A 0
  | vcons : \forall n, A -> vector A n -> vector A (S n).
```

This allows us to write head and tail functions that don’t go wrong:

```coq
Definition vhead : \forall \{A n\} , vector A (S n) -> A .
```

```coq
Definition vtail : \forall \{A n\} , vector A (S n) -> vector A n .
```
Σ types

We have already seen how Σ types are used to define the existential quantifier in Prop. Ξ has a doppelganger sig that lives in Type:

\[
\text{Inductive sig } \{A : \text{Type}\} (P : A \rightarrow \text{Prop}) : \text{Type} := \\
\text{exist} : \forall x : A, P x \rightarrow \text{sig } P
\]

sig is very useful for writing certified programs, that is, programs that return both a result and a proof that that result satisfies a certain predicate. If the predicate is sufficiently strong, the proof can guarantee the (denotational) correctness of the program.

sig is accompanied by a very useful notation for comprehension:

"{ x | P }" := sig (fun x \Rightarrow P)
Recall that in Coq, all functions must be \emph{total} (this is why there is no head or tail for lists).

Consider the predecessor function for natural numbers. If we type it with \texttt{pred : nat \rightarrow nat}, then we have a problem:

\begin{verbatim}
Definition pred_weak_0 (n : nat) : nat :=
  match n with
  | 0 => 0 (* gotta put something here *)
  | S n' => n'
  end.
\end{verbatim}

namely, the predecessor of 0 is not 0.
example: predecessor

We can use dependent types to write a version that only accepts successors:

```coq
Definition pred_weak_1 (n : nat) : 0 ≠ n → nat :=
    match n return (0 ≠ n → nat) with
    | 0 ⇒ fun pf : 0 ≠ 0 ⇒ match nn_z_eq_z pf with end
    | S n' ⇒ fun _ ⇒ n'
    end
.
```

This version requires an additional argument: a proof that the first argument is not zero.

Notice in the first branch of the match, `nn_z_eq_z pf : ⊥`, but `⊥` has no constructors, hence the nested match has no cases. The `⊥-elim` rule allows us to infer any type we want here since no such `pf` could exist.
example: predecessor

We can rewrite this to look a bit more natural using Σ types and \textit{comprehension} notation:

```coq
Definition pred_weak_2 (s : \{n : nat \mid 0 \neq n\}) : nat :=
match s with
| exist 0 pf \Rightarrow match nn_z_eq_z pf with end
| exist (S n') _ \Rightarrow n'
end.
```

(notice how a negative Σ occurrence is isomorphic to a positive Π occurrence – c.f. Curry’s isomorphism)

This version is correct, but to know that, we have to reason about the definition ourselves.
example: strong predecessor

We can have Coq guarantee the correctness by using a $\Sigma$ type for the output that includes a proof that the successor of the output (witness) is the input:

```coq
Definition pred_strong_1 (s : {n : nat | 0 \neq n}) : {m : nat | proj1_sig s = S m} :=
    match s return {m : nat | proj1_sig s = S m} with
    | exist 0 pf \Rightarrow match nn_z_eq_z pf with end
    | exist (S n') _ \Rightarrow exist _ n' (refl_equal _)
    end.
```

In one sense this version is optimal since it is guaranteed to correctly implement (denotationally) the predecessor function. However, in at least two respects it is quite sub-optimal:

- it is hard to read (and write)
- it is inconvenient to use, because of all the proof-related baggage
example: strong predecessor

The first problem is nicely solved by *interactive definition*. As with *interactive proof*, we can enter the interactive environment to incrementally construct a term of the output type, availing ourselves to all the tactics and automation mechanisms we are used to.

The problem is that unless the function we are defining is *strongly typed*, the term we construct may not be the one we wanted:

```coq
Definition pred_weak_1_auto : \forall (n : nat) , 0 \neq n \rightarrow nat. 
  auto.
Defined.
```

But with strong enough typing this is not a problem:

```coq
Definition pred_strong_2 : 
  \forall n : nat , n \neq 0 \rightarrow \{v : nat | n = S v\} .
```
The second problem can be solved with *program extraction*, a method by which all the logical parts of a program are *erased*, leaving only the functional parts.

Coq can extract into several functional languages, including Scheme, CaML and Haskell.
extended example: extracting insertion sort

We will sketch the extraction of a certified version of insertion sort for lists of integers.

First we define the sorted predicate:

\[
\text{Inductive sorted : list } \mathbb{Z} \rightarrow \text{Prop :=}
\begin{align*}
| \text{sorted_empty : sorted nil} \\
| \text{sorted_singleton : } \forall z : \mathbb{Z}, \text{sorted (z :: nil)} \\
| \text{sorted_inductive :} \\
\forall (z1, z2 : \mathbb{Z}) (zs : \text{list } \mathbb{Z}), \\
z1 \leq z2 \rightarrow \\
\text{sorted (z2 :: zs)} \rightarrow \\
\text{sorted (z1 :: z2 :: zs)}
\end{align*}
\]

Note the similarity to PROLOG.
extended example: extracting insertion sort

We define a function to count the number of occurrences of each element in the list:

\[
\text{Fixpoint occurrences } (z : \mathbb{Z}) (zs : \text{list } \mathbb{Z}) \{\text{struct } zs\} : \mathbb{N} :=
\]

and a predicate that says one list is a permutation of another if all integers have the same number of occurrence in each list:

\[
\text{Definition permutation } (\text{list1 list2 : list } \mathbb{Z}) : \text{Prop} :=
\]

\[
\forall z : \mathbb{Z}, \text{occurrences } z \text{ list1} = \text{occurrences } z \text{ list2} .
\]

Note the non-effectiveness of this predicate
extended example: extracting insertion sort

We prove that permutation is an equivalence relation, and that it is compatible with consing an element:

```
Lemma cons_compat_perm :
  ∀ (z : Z) (list1 list2 : list Z) ,
  permutation list1 list2 →
  permutation (z :: list1) (z :: list2) .
```

and swapping the head elements:

```
Lemma swap_compat_perm :
  ∀ (a b : Z) (list1 list2 : list Z) ,
  permutation list1 list2 →
  permutation (a :: b :: list1) (b :: a :: list2) .
```
Next we define the `insert` function

```coq
Fixpoint insert (z : Z) (l : list Z) {struct l} : list Z :=
  match l with
  | nil ⇒ z :: nil
  | (a :: l’) ⇒
    match Z_le_gt_dec z a with
    | left _ ⇒ z :: a :: l’
    | right _ ⇒ a :: (insert z l’)
    end
  end
.
```

and prove that inserting an element preserves sorting and results in a permutation of consing.
**extended example: extracting insertion sort**

Finally, we define our main `sort` function.

```
Definition sort :
  \forall l : list Z ,
  \{l' : list Z | permutation l l' \land sorted l'\} .
```

Because it is strongly typed, we can do this interactively and be confident that whatever term we find will be denotationally correct.

From this we can extract insertion sort in OCaml with the `Extraction` vernacular command.
to be continued.