Introduction to Coq

SoC TaeYoung Kim

KAIST, 수학문제연구회

November 16, 2020

What is Coq?

Coq is

- Interactive Theorem Prover
- Formal Proof System
- Computer-aided proof
- Inductive Programming Language
- Purely functional dependent type programming language

How we prove something?

1. Define properties, functions, elements
2. State proposition
3. Apply tactics to prove goal given hypothesis
    - rewrite equality
    - apply proved theorems
    - induction
    - case analysis
4. Qed

Proposition as type

To Prove

True	Single element 1
False	No proof object (empty type)
$A \to B$	Function from type A to type B Given proof of A, construct proof of B
$\forall x : T, A(x)$	Function from type T to type $A(x)$ Given element x, construct proof of $A(x)$
$\exists x : T, A(x)$	Pair of element $x : T$ and type $A(x)$
$\neg A$	Function from type A to type False If some element in type A exists, no function from A to False

This is called Curry-Howard Correspondence (Or Isomorphism)

True is a Singleton

True is type with single element 1. You can prove true by apply I.

$$True := 1.$$

$$True \to P = P$$

True has unique element, and gives no effectiveness.

False is Empty

False is type with no element. You can not prove false. I will denote false as $\perp$.

$$False := .$$

We denote negation of proposition as $\neg P = P \rightarrow False$, so proving negation should construct such function, which proves that P is also empty.

$$(False \rightarrow P) = True$$

If False is in hypothesis, we can eliminate current goal by various tactics, like discriminate, inversion, absurd.

Imply is a Function

To prove Implication, $P \to Q$, we introduce P as a hypothesis, and Q as a conclusion. We do this with tactic intro P. In specific, we need to construct proof of Q from proof of P, or constructing function from type P to type Q.
To use impliance, we need two hypothesis $HPQ : P \to Q$ and $HP : P$. Then tactic apply HPQ in HP gives the answer, or for conclusion Q, apply HPQ makes conclusion to P.

And is a Product

To prove conjunction, $P \wedge Q$, we need to prove both P and Q. We do this with tactic split. In specific, this is a function that pairs proof of P and proof of Q to proof of $P \wedge Q$. And this construction is called product.

$$P \to Q \to (P \wedge Q)$$

$$P \wedge Q \to R = P \to Q \to R$$

Given proof of conjunction $H : P \wedge Q$, we can extract proof of both P and Q. This is done by tactic destruct H as [HP HQ]. This is a function of a type

$$P \wedge Q \to P \times Q$$

Or is a Disjoint Union

To prove disjunction, $P \vee Q$, we need to prove one of P and Q, and need to specify which one really holds. We do this with tactic left/right. And this construction is called coproduct, or sum.

$$\text{left} : P \to P \vee Q$$
$$\text{right} : Q \to P \vee Q$$

$$P \vee Q \to R = (P \to R) \wedge (Q \to R)$$

Given proof of disjunction $H : P \vee Q$, we either have proof of P or proof of Q, and what is the case. This is done by tactic destruct H as [HA—HB], which will generate two goals. This is a function of a type

$$(P \to R) \to (Q \to R) \to (P \vee Q \to R)$$

Forall is a Dependent function

$$\forall a : A, P(a)$$

To prove forall, we need to give proof of $P(a)$ indexed by $a : A$. This is proved using tactic intro a, which introduces new variable $a : A$. This type is called dependent product, $\prod_{a:A} P(a)$.

$$\forall a : A, P(a) \to Q$$

to use forall, we can simply apply this hypothesis, to any of $P(a)$ of $a : A$.

Exists is a Dependent pair

$$\exists a : A, P(a)$$

To prove exists, we need to give explicit $a : A$ and the proof of $P(a)$. This is proved using tactic exists a, which explicitly give the element. This type is called dependent sum, $\sum_{a:A} P(a)$

$$\exists a : A, P(a) \to Q$$

To use existential quantifier, we need to extract what element a satisfies $P(a)$. We do this by destruct H as [a Ha].

Induction-1

Many of Coq properties are constructed using induction.

```
Inductive Nat : Type :=
| 0 : Nat
| S (n : nat) : Nat.

Inductive List (A : Type) : Type :=
| Nil : List A
| Cons (a : A) (l : List A) : List A.

Inductive Even (n : nat) : Prop :=
| Even_0 : Even 0
| Even_SS (n : nat) (H : Even n) :
    Even (S (S n)).
```

Induction-2

To construct inductive element, we create finite tree, as

```
Definition three : Nat := S (S (S 0)).
```

or

```
Definition list123 : List Nat :=
  Cons 1 (Cons 2 (Cons 3 (Nil))).
```

or

```
Definition even4 : even 4 :=
  Even_SS 2 (Even_SS 0 Even_0).
```

Induction-3

To use something inductive, we use induction tactic, induction n or induction H.

```
Theorem even_spec :
  forall n, even n -> exists k, n = 2 * k.
Proof.
  intros n Heven. induction Heven.
 - exists 0. simpl. reflexivity.
 - destruct IHHeven as [k Hk].
   rewrite Hk. exists (S k).
   simpl. apply eq_S. rewrite <- plus_n_O.
   apply plus_n_Sm.
 Qed.
```

Induction is a Primitive Recursive function

Then what is induction? Induction is simply a function of type

$$P(0) \to \prod_{n:N}(P(n) \to P(S(n))) \to \prod_{n:N} P(n)$$

which naturally exists. In fact, we define inductive type as a type that such function exists.
And if you are familar to programming, this is actually a recursion(We generate result from subresults). In specific, tactic fix gives strong induction, which is actually recursion.

Recursion-1

Majority of Coq functions are recursive, like addition.

```
Fixpoint add (a b : nat) : nat :=
  match a with
  | 0 => b
  | S a' => S (add a' b)
  end.
```

And it requires termination. Why?

Recursion-2

Consider following function.

```
Fail Fixpoint false_proof (n : nat) : False :=
  false_proof n.
```

This function calls itself, which does not terminate. And if such function exists, we have proof of False!
So at least one of element should decrease in recursive call. Or, you can give some function on arguments that strictly decrease.

CoInduction

CoInduction is dual of Induction. Inductive type is formally defined as an initial algebra of endofunctor, and Coinductive type is formally defined as a final coalgebra of endofunctor.
Inductive type is constructed as finite tree, where Coinductive type is constructed as infinite tree. To formalize infinite sequence, infinite tree, we need coinduction, coinductive type, cofixpoint functions. Similar to Fixpoint, there is guard on CoFixpoint that makes function computable.

```
CoInductive Seq : Type :=
| Next (n : nat) (s : Seq) : Seq.

CoFixpoint const (n : nat) : Seq :=
  Next n (const n).
```

Computability

The guard gives what functions we allow. We only allow functions that computers can give answer in finite time. Or in recursion theory sense, primitive-recursive functions.
This eliminates AoC, which is highly uncomputable. Moreover, you can't assert excluded middle

$$P \vee \neg P$$

which is not computable unless P is decidable. This makes Coq constructive mathematics.
The proposition P is decidable if and only if $P \vee \neg P$ is provable.

Equality-1

One of critical point of Coq is equality. We have $p = q$ for $p, q : A$, $eq1 = eq2$ for $eq1, eq2 : p = q$, $eq3 = eq4$ for $eq3, eq4 : eq1 = eq2$, and so on. If you are familar to algebraic topology, this is higher homotopy group. The main construction of equality is reflective, symmetric, transitive. And each of these corresponds to constant loop, reverse of loop, concatenation of loop.
Why this matters?

Equality-2

Suppose we have a group G. The definition of subgroup is given as

```
Record subgroup_prop_els (G : group) (H : subgroup_
  subgr_p_g :> G; subgr_p_H : H subgr_p_g }.
```

Then we have two different element if we have two different proof for subgr_p_H. You may use proof irrelevance axiom or weakening the definition using boolean equality.

Equality-3

```
Record subgroup_bool_els (G : group) (H : subgroup_
  subgr_b_g :> G; subgr_b_H : H subgr_b_g = true }.
```

Instead of prop, bool is decidable. Moreover, equality of boolean is unique so we don't need proof irrelevance.

Equality-4

One of paradox is that following statement is not provable.

$$\forall f, g : A \to B, (\forall a : A, fa = ga) \to f = g$$

This is called functional extensionality, and can not be proven due to non-inductiveness of function. This also holds for dependent function.
We can safely add some axioms, as following

```
Axiom functional_extensionailty :
  forall (A B : Type) (f g : A -> B),
  (forall a : A, f a = g a) -> f = g.

Axiom proof_irrelevance :
  forall (P : Prop) (H1 H2 : P), H1 = H2.
```

Set, Type

Coq is not set theory based, but type Set exists. What is it?
There is a theorem that collection of every sets is not a set. Than we can create

- Collection of Sets (Universe zero)
- Collection of Collection of Sets (Universe one)
- and goes on

In Coq, we call Universe as type, and type zero as set.
And Set is one that is computationally relevant, unlike Prop. This construction is due to resolve Girard's paradox, which is analogy of Russel's paradox in set theory.

Proof as a data-1

The lift operator of cpo to dcpo is given as

```
CoInductive Stream (D : cpo) :=
| Eps : Stream D -> Stream D
| Val : D -> Stream D.
```

If this stream is finite, we can get the value. But without using the proof that stream is infinite, we can't define such function.

Proof as a data-2

```
Class finite_evidence (D : cpo) (d : Stream D) :=
  {pred_n : nat; pred_d' : D; pred : pred_nth d pred_n = Val D pred_d'}.

Lemma eps_finite_finite (D : cpo) :
  forall d, Finite D (Eps D d) -> Finite D d.
Proof.
  intros. inversion H; subst. apply H1. Defined.

Fixpoint extract_evidence (D : cpo) (d : Stream D) (H : Finite D d) :
  finite_evidence D d.
Proof.
  destruct d.
- apply eps_finite_finite in H. apply extract_evidence in H.
  destruct H.
  exists (S pred_n0) (pred_d'0). simpl. apply pred0.
- exists 0 t. reflexivity.
Defined.
```

Difference with other mathematics

- Purely constructive
- Not set theory, but type theory.
- Proof is also a mathematical object.
- Many proofs rely on (structural) induction, or coinduction.

Difference with other programming language

- Purely functional
- Dependent Types
  In specific, dependent product type and dependent pair type.
- All functions always terminate.
  If system can't ensure this, you should prove this.
- Deterministic computation
  No random, input.

Underlying Theories

- Dependent Type Theory
  Type theory, but type may depend on argument.
  Required to define first order propositions.
- Proof Theory
  A proposition is true iff there exists proof object of that type.
- Constructive Mathematics
  We need to construct object to prove existence.
  We can't say "Suppose there is no ..."
- Curry-Howard Correspondence
  There is correspondence between logic and computation.

Curry-Howard Isomorphism

There is a isomorphism between logic and computation.

Logic	Computation
Implication	Lambda Calculus
Conjunction	Product
Disjunction	Sum
Inductive logic	Algebraic data type

Applications-mathematics

Theorem (Feit-Thompson)

Every finite group of odd order is solvable.

Theorem (4-color)

Every planar graph is 4 colorable.

These two theorem's proof is formalized by Coq.

Example (MathComp)

Coq library for formalization of mathematical theory

Applications-mathematics

Why is (finite) group theory and graph theory uses Coq a lot?
Because everything is decidable in finite space!
If you are familar to analysis, everything is infinite in analysis. That's why analysis is *very* hard. But, there exists analysis formalization with computable analysis, computable probability theory, etc.

Applications-programming

Example (CompCert)

Fully verified C-compiler

Example (Iris logic)

Logic framework for reasoning on concurrent higher order programs

Example (KAIST Concurrency and Parallelism Laboratory)

Works for designing and **verifying** concurrent program and system

Pros and Cons

Advantages

- Assurance for truth of proof
- Strong automation for proof
  Pattern matching hypothesis and goal, omega, ring, auto, etc.

Disadvantages

- Need to check every detail.
- Every proof is constructive, no law of excluded middle, axiom of choice.
- Every function is computable. But computable functions are very rare, for example computable real numbers are measure zero.

Extension

Hoq is homotopy type thoery based coq implementation. Going deeper to induction, we have higher inductive type which allows induction over equality of types, and equality of equality of types, and, continues.

If you are interested, you may look at Homotopy type thoery itself, which gives good intuitions for inductive type. Or, the implementation of Hoq.

How to study?

This is programming language, so practice is important.

- Formalizing mathematics
  - Graph Theory : Good
  - Algebra : Good
  - Number Theory : Not bad
  - Topology : Hard
  - Analysis : Very hard
- Verifying algorithms (This is what I mostly do)
- Logic

Other theorem prover

- HOL
  Used on deep learning aided proof. See HOList paper.
- Z3
  Automated theorem prover, developed by Microsoft.
- Lean
  Also a interactive theorem prover, relatively younger than Coq. Since it is more fresh, mathematician tries to formalize mathematics using Lean.
- Idris
  Functional programming language. Most modern dependently typed programming language.

Coq is more specialized on programming language theory.

Example-recursive function

```
Fixpoint sum (n : nat) : nat :=
  match n with
  | 0 => 0
  | S n' => n + (sum n')
  end.
```

decreasing on 1st argument n.

$$\text{sum } n = \sum_{i=0}^{n} i$$

Example-Proposition

```
Theorem summation :
  forall n, sum n + sum n = n * (n + 1).
Proof.
  induction n.
  - simpl. reflexivity.
  - simpl. apply eq_S. rewrite <- plus_assoc.
    rewrite <- plus_assoc. apply f_equal2_plus.
    reflexivity. simpl.
    rewrite Nat.add_succ_r. apply eq_S.
    rewrite plus_comm. rewrite <- plus_assoc.
    rewrite IHn. rewrite Nat.mul_succ_r.
    rewrite plus_comm. reflexivity.
  Qed.
```

Example-automation

```
Theorem summation_ring :
  forall n, sum n + sum n = n * (n + 1).
Proof.
  induction n.
  - simpl. reflexivity.
  - simpl.
    replace (n + sum n + S (n + sum n)) with
            ((sum n + sum n) + n + S n).
    rewrite IHn. ring. ring.
   Qed.
```

How to Install

`https://coq.inria.fr/`

Current stable version is 8.12.0.

Don't forget to modify your enviroment variable 'PATH' to contain path to coq executable.

Or try jsCoq! `https://jscoq.github.io/`

How to use

Best : Your favorite IDE + Coq plugin
Good : CoqIde
Bad : coqc
Never : Text editor

Supplementary materials

You can see functional programming related materials here. I recommend you to read ‘First order function’, ‘Type systems’. ‘Parametric Polymorphism’.
`https://hjaem.info/articles/main`
Coq’Art `https://www.labri.fr/perso/casteran/CoqArt/`
‘Programs and Proofs’ `https://ilyasergey.net/pnp/`
‘Software Foundation’ `https://softwarefoundations.cis.upenn.edu/current/index.html`
MathComp Documentation
`https://math-comp.github.io/documentation.html`
‘Coq Workshop’ `https://coq-workshop.gitlab.io/2020/`
‘CoqPL’ `https://popl19.sigplan.org/track/CoqPL-2019`
‘ITP contest’ `https://competition.isabelle.systems/`

Related papers

Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. https://www.irif.fr/~sozeau/research/publications/drafts/Coq_Coq_Correct.pdf
The HoTT Library: A formalization of homotopy type theory in Coq https://arxiv.org/abs/1610.04591
HOList: An Environment for Machine Learning of Higher-Order Theorem Proving https://arxiv.org/abs/1904.03241