Logic Foundations
Logic: Logic in Coq

熊英飞 胡振江

信息学院计算机系

2021年3月30日/4月2日

Proposition Type

Logic Claim

Any statement we might try to prove in Coq has a type, namely Prop, the type of propositions.

```
Check 3 = 3 : Prop.

Check forall n m : nat, n + m = m + n : Prop.

Check 2 = 2 : Prop.

Check 3 = 2 : Prop.
```

Proposition Definition

```
Definition plus_claim : Prop := 2 + 2 = 4.

Check plus_claim : Prop.

Theorem plus_claim_is_true :
  plus_claim.
Proof. reflexivity.  Qed.
```

Predicate/Property Definition

```
Definition is_three (n : nat) : Prop :=
  n = 3.
Check is_three : nat -> Prop.

Definition injective {A B} (f : A -> B) :=
  forall x y : A, f x = f y -> x = y.

Lemma succ_inj : injective S.
Proof.
  intros n m H. injection H as H1. apply H1.
Qed.
```

Logic Connectives

Conjunction (Logic And)

Split on the goal:

```
Example and_example : 3 + 4 = 7 /\ 2 * 2 = 4.
Proof.
 split.
 - (* 3 + 4 = 7 *) reflexivity.
 - (* 2 * 2 = 4 *) reflexivity.
Qed.

Lemma and_intro : forall A B : Prop, A -> B -> A /\ B.
Proof.
 intros A B HA HB. split.
 - apply HA.
 - apply HB.
Qed.
```

Conjunction (Logic And)

Destruct on the hypothesis:

```
Lemma and_example2 :
 forall n m : nat, n = 0 /\ m = 0 -> n + m = 0.
Proof.
 intros n m H.
 destruct H as [Hn Hm].
 rewrite Hn. rewrite Hm.
 reflexivity.
Qed.

Lemma and_example2' :
 forall n m : nat, n = 0 /\ m = 0 -> n + m = 0.
Proof.
 intros n m [Hn Hm].
 rewrite Hn. rewrite Hm.
 reflexivity.
Qed.
```

Conjunction (Logic And)

```
Lemma proj1 : forall P Q : Prop,
 P /\ Q -> P.
Proof.
 intros P Q HPQ.
 destruct HPQ as [HP _].
 apply HP.  Qed.

Theorem and_commut : forall P Q : Prop,
 P /\ Q -> Q /\ P.
Proof.
 intros P Q [HP HQ].
 split.
  - (* left *) apply HQ.
  - (* right *) apply HP.  Qed.
```

Disjunction (Logic Or)

```
Lemma eq_mult_o :
 forall n m : nat, n = o \/ m = o -> n * m = o.
Proof.
 intros n m [Hn | Hm].
 - (* Here, [n = o] *)
   rewrite Hn. reflexivity.
 - (* Here, [m = o] *)
   rewrite Hm. rewrite <- mult_n_O.
   reflexivity.
Qed.

Lemma or_intro_l : forall A B : Prop, A -> A \/ B.
Proof.
 intros A B HA.
 left.
 apply HA.
Qed.
```

Disjunction (Logic Or)

```
Lemma zero_or_succ :
 forall n : nat, n = 0 \/ n = S (pred n).
Proof.
 intros [|n'].
 - left. reflexivity.
 - right. reflexivity.
Qed.
```

Falsehood and Negation

```
Definition not (P:Prop) := P -> False.

Notation "~ x" := (not x) : type_scope.

Check not : Prop -> Prop.
```

False is a specific contradictory proposition defined in the standard library.

Falsehood and Negation

Principle of Explosion

```
Theorem ex_falso_quodlibet : forall (P:Prop),
  False -> P.
Proof.
  intros P contra.
  destruct contra.  Qed.

Notation "x <> y" := (~(x = y)).

Theorem zero_not_one : 0 <> 1.
Proof.
  unfold not.
  intros contra.
  discriminate contra.
Qed.
```

Falsehood and Negation

```
Theorem not_False :
  ~ False.
Proof.
  unfold not. intros H. destruct H. Qed.

Theorem contradiction_implies_anything : forall P Q : Prop,
  (P /\ ~P) -> Q.
Proof.
  intros P Q [HP HNA]. unfold not in HNA.
  apply HNA in HP. destruct HP.  Qed.

Theorem double_neg : forall P : Prop,
  P -> ~~P.
Proof.
  intros P H. unfold not. intros G. apply G. apply H.  Qed.
```

Falsehood and Negation

```
Theorem not_true_is_false : forall b : bool,
 b <> true -> b = false.
Proof.
 intros b H.
 destruct b eqn:HE.
 - (* b = true *)
  unfold not in H.
  apply ex_falso_quodlibet.
  apply H. reflexivity.
 - (* b = false *)
  reflexivity.
Qed.
```

exfalso.

Useful trick: If you are trying to prove a goal that is nonsensical, apply ex_falso_quodlibet to change the goal to False.

Truth

```
Lemma True_is_true : True.
Proof. apply I. Qed.
```

I : True: a predefined constant

Logic Equivalence

```
Definition iff (P Q : Prop) := (P -> Q) /\ (Q -> P).

Notation "P <-> Q" := (iff P Q)  (at level 95, no associativity): type_scope.

Theorem iff_sym : forall P Q : Prop,  (P <-> Q) -> (Q <-> P).
Proof.
  intros P Q [HAB HBA].
  split.
  - (* -> *) apply HBA.
  - (* <- *) apply HAB.  Qed.

Lemma not_true_iff_false : forall b, b <> true <-> b = false.
Proof.
  intros b. split.
  - (* -> *) apply not_true_is_false.
  - (* <- *)
    intros H. rewrite H. intros H'. discriminate H'. Qed.
```

Setoids and Logical Equivalence

A "setoid" is a set equipped with an equivalence relation.

```
Lemma mult_0 : forall n m, n * m = 0 <-> n = 0 \/ m = 0.
Proof.
  split.
  - apply mult_eq_0.
  - apply eq_mult_0. Qed.

Theorem or_assoc : forall P Q R : Prop, P \/ (Q \/ R) <-> (P \/ Q) \/ R.
Proof.
  intros P Q R. split.
  - intros [H | [H | H]].
    + left. left. apply H.
    + left. right. apply H.
    + right. apply H.
  - intros [[H | H] | H].
    + left. apply H.
    + right. left. apply H.
    + right. right. apply H. Qed.
```

Setoids and Logical Equivalence

A "setoid" is a set equipped with an equivalence relation.

```
Lemma mult_0_3 :
  forall n m p, n * m * p = 0 <-> n = 0 \/ m = 0 \/ p = 0.
Proof.
  intros n m p.
  rewrite mult_0. rewrite mult_0. rewrite or_assoc.
  reflexivity.
Qed.

Lemma apply_iff_example :
  forall n m : nat, n * m = 0 -> n = 0 \/ m = 0.
Proof.
  intros n m H. apply mult_0. apply H.
Qed.
```

Existential Quantification

```
Definition even x := exists n : nat, x = double n.

Lemma four_is_even : even 4.
Proof.
 unfold even. exists 2. reflexivity.
Qed.

Theorem exists_example_2 : forall n,
 (exists m, n = 4 + m) ->
 (exists o, n = 2 + o).
Proof.
 intros n [m Hm].
 exists (2 + m).
 apply Hm.  Qed.
```

Programming with Propositions

False and True

```
Inductive False : Prop :=

Inductive True : Prop :=
  I : True
```

Recursive Proposition

```
Fixpoint In {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] => False
  | x' :: l' => x' = x \/ In x l'
  end.

Example In_example_1 : In 4 [1; 2; 3; 4; 5].
Proof.
   simpl. right. right. right. left. reflexivity.
Qed.

Example In_example_2 : forall n, In n [2; 4] -> exists n', n = 2 * n'.
Proof.
  simpl.
  intros n [H | [H | []]].
  - exists 1. rewrite <- H. reflexivity.
  - exists 2. rewrite <- H. reflexivity.
Qed.
```

Proof of Generic/Higher-Order Lemmas

```
Theorem In_map :
 forall (A B : Type) (f : A -> B) (l : list A) (x : A),
  In x l ->
  In (f x) (map f l).
Proof.
 intros A B f l x.
 induction l as [|x' l' IHl'].
 - (* l = nil, contradiction *)
  simpl. intros [].
 - (* l = x' :: l' *)
  simpl. intros [H | H]. (* \/ *)
  + rewrite H. left. reflexivity.
  + right. apply IHl'. apply H.
Qed.
```

Applying Theorems to Arguments

Proofs as First-Class Objects

Proof Object

Check plus_comm : forall n m : nat, n + m = m + n.

A proof object represents a logic derivation establishing of the truth of the statement

if we have an object of type **n = m → n + n = m + m** and we provide it an "argument" of type **n = m**, we can derive **n + n = m + m**.

Using Theorems like Functions

```
Lemma plus_comm3_take3 :
  forall x y z, x + (y + z) = (z + y) + x.
Proof.
  intros x y z.
  rewrite plus_comm.
  rewrite (plus_comm y z).
  reflexivity.
Qed.
```

Using Theorems like Functions

```
Theorem in_not_nil :
  forall A (x : A) (l : list A), In x l -> l <> [].
Proof.
  intros A x l H. unfold not. intro Hl.
  rewrite Hl in H.
  simpl in H.
  apply H.
Qed.
```

```
Lemma in_not_nil_42_take4 :
  forall l : list nat, In 42 l -> l <> [].
Proof.
  intros l H.
  apply (in_not_nil nat 42).
  apply H.
Qed.
```

```
Lemma in_not_nil_42_take5 :
  forall l : list nat, In 42 l -> l <> [].
Proof.
  intros l H.
  apply (in_not_nil _ _ _ H).
Qed.
```

Using Theorems like Functions

```
Example lemma_application_ex :
  forall {n : nat} {ns : list nat},
    In n (map (fun m => m * 0) ns) ->
    n = 0.
Proof.
  intros n ns H.
  destruct (proj1 _ _ (In_map_iff _ _ _ _ _) H)
        as [m [Hm _]].
  rewrite mult_0_r in Hm. rewrite <- Hm.
  reflexivity.
Qed.
```

```
proj1
    : forall P Q : Prop,
      P /\ Q -> P
In_map_iff
    : forall
        (A : Type@{In_map_iff.u0})
        (B : Type@{In_map_iff.u1})
        (f : A -> B)
        (l : list A)
        (y : B),
      In y (map f l) <->
      (exists x : A,
         f x = y /\
         In x l)
```


北京大学 PEKING UNIVERSITY

Coq vs. Set Theory

Calculus of Inductive Constructions

Functional Extensionality

- Functional extensionality is not part of Coq's built-in logic; it is not provable.

```
Axiom functional_extensionality : forall {X Y: Type} {f g : X -> Y},
 (forall (x:X), f x = g x) -> f = g.

Example function_equality_ex2 :
 (fun x => plus x 1) = (fun x => plus 1 x).
Proof.
 apply functional_extensionality. intros x.
 apply plus_comm.
Qed.
```

Propositions vs. Booleans

- We have two different ways of expressing logical claims in Coq: with Booleans (of type bool), and with propositions (of type Prop).

```
Example even_42_bool : evenb 42 = true.
Proof. reflexivity. Qed.

Example even_42_prop : even 42.
Proof. unfold even. exists 21. reflexivity. Qed.
```

Propositions vs. Booleans

- Correspondence

```
Lemma evenb_double : forall k, evenb (double k) = true.

Lemma evenb_double_conv : forall n, exists k,
 n = if evenb n then double k else S (double k).

Theorem even_bool_prop : forall n,
 evenb n = true <-> even n.
Proof.
 intros n. split.
 - intros H. destruct (evenb_double_conv n) as [k Hk].
  rewrite Hk. rewrite H. exists k. reflexivity.
 - intros [k Hk]. rewrite Hk. apply evenb_double.
Qed.
```

Proof by Reflection

- Enable some proof automation through computation with Coq terms.

```
Example even_1000 : even 1000.
Proof. unfold even. exists 500. reflexivity. Qed.

Example even_1000' : evenb 1000 = true.
Proof. reflexivity. Qed.

Example even_1000'' : even 1000.
Proof. apply even_bool_prop. reflexivity. Qed.
```

难自动化

易自动化

易自动化

The famous 4-color theorem uses reflection to reduce the analysis of hundreds of different cases to a Boolean computation.

Proof by Reflection

The negation of a "Boolean fact" is straightforward to state and prove.

```
Example not_even_1001 : evenb 1001 = false.
Proof.
 reflexivity.
Qed.

Example not_even_1001' : ~(even 1001).
Proof.
 rewrite <- even_bool_prop.
 unfold not.
 simpl.
 intro H.
 discriminate H.
Qed.
```

Proof by Reflection

Equality is sometimes easier to work in the propositional world (by rewriting).

```
Lemma plus_eqb_example : forall n m p : nat,
  n =? m = true -> n + p =? m + p = true.
Proof.
 intros n m p H.
 rewrite eqb_eq in H.
 rewrite H.
 rewrite eqb_eq.
 reflexivity.
Qed.
```

```
eqb_eq
   : forall n1 n2 : nat,
    (n1 =? n2) = true <->
    n1 = n2
```

Classical vs. Constructive Logic

The following intuitive reasoning principle is not derivable in Coq:

```
Definition excluded_middle := forall P : Prop,
  P \/ ~ P.
```

We don't have enough information to choose which of left or right to apply.

Logics like Coq's, which do not assume the excluded middle, are referred to as constructive logics.

Classical vs. Constructive Logic

If we happen to know that P is restricted in some Boolean term b, then knowing whether it holds or not is trivial: we just have to check the value of b.

```
Theorem restricted_excluded_middle : forall P b,
 (P <-> b = true) -> P \/ ~ P.
Proof.
 intros P [] H.
 - left. rewrite H. reflexivity.
 - right. rewrite H. intros contra. discriminate contra.
Qed.

Theorem restricted_excluded_middle_eq : forall (n m : nat),
 n = m \/ n <> m.
Proof.
 intros n m.
 apply (restricted_excluded_middle (n = m) (n =? m)).
 symmetry.
 apply eqb_eq.
Qed.
```

作业

- 完成Logic.v中的至少10个练习题。