Aula 13 - Lógica em Coq 1

Rafael Castro - rafaelcgs10.github.io/coq

30/05/2018

Revisão dos elementos de lógica vistos

- Implicação (->).
- Quantificação universal (*forall*).
- Operador de igualdade (=).
- Operadores booleanos (*orb*, *andb*).

Proposições em Coq

- Coq é uma linguagem tipada: toda expressão (bem tipada) tem um tipo.
- Proposições são expressões!
- Qual é o tipo de uma proposição como $3 = 3$?
- O tipo *Prop* é o tipo das proposições!

```
Check 3 = 3.
(* ===> Prop *)

Check forall n m : nat, n + m = m + n.
(* ===> Prop *)

Check 3 = 4.
(* ===> Prop *)
```

Proposições são *first class citizens*!

- Proposições são *first class citizens* (valores de primeira classe).
- Podem ser argumentos e retornos de funções.
- A função abaixo retorna uma proposição.

```
Definition plus_fact : Prop := 2 + 2 = 4.
Check plus_fact.
(* ===> plus_fact : Prop *)
```

Proposições parametrizadas

- Podemos escrever funções com argumentos e que retornam uma proposição sobre os mesmos.
- Em Coq, chamamos funções essas funções de propriedades.
- A função abaixo define a propriedade de *f* ser injetiva.

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

O operador de igualdade

- O operador de igualdade (=) também é uma função.

```
Check @eq.
(* ===> forall A : Type, A -> A -> Prop *)
```

Conjunção de proposições

- A conjunção entre duas proprosições *A* e *B* é escrita como `A /\ B`.
- \ *é apenas açúcar sintático para /and*.
- Para provar um objetivo com uma conjunção é necessário provar ambas as proposições.
- A tática *split* separa uma conjunção no objetivo em dois sub-objetivos.

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

Check and.
```

Introdução da conjunção

- Se assumirmos as proposições A e B, então podemos concluir A /\ B.
- Onde você já viu isso antes?

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

Conjunção numa das hipóteses

- Uma conjunção numa das hipóteses pode ser separada em duas hipóteses.
- A tática *destruct* na hipótese faz exatamente isso.

```
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.
```

Conjunção é comutativa

- Algumas vezes pode ser necessário rearranjar a ordem de uma conjunção.

```
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.
```

Disjunção de proposições

- Uma disjunção de proposições A \/ B é verdadeira quando ao menos uma delas é verdadeira.
- \/ é açúcar sintático para *or*.
- Se uma hipótese é uma disjução, então pode-se fazer análise de caso nela por meio de *destruct*.

```
Lemma or_example :
  forall n m : nat, n = 0 \/ m = 0 -> n * m = 0.
Proof.
  intros n m Hnm.
  destruct Hnm as [Hn | Hm].
  - (* Here, [n = 0] *) rewrite Hn. reflexivity.
  - (* Here, [m = 0] *) rewrite Hm. rewrite <- mult_n_O.
    reflexivity.
Qed.
Check or. (* ===> or : Prop -> Prop -> Prop *)
```

O objetivo é uma disjunção

- Para provar uma disjunção é preciso demostrar apenas um lado da mesma.
- As táticas *left* e *right* servem para fazer essa escolha.

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

Exemplo com left e right

- O exemplo abaixo faz uma análise de caso em *n*.

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

Negação e falsidade

- Até o momento provamos coisas serem verdadeiras.
- Em Coq também é possível provar que coisas são falsidades.
- A negação é representada pelo operador unário ~, que é um açucar sintático para *not*.
- Coq define *not* como uma função de Prop para o tipo bottom. O tipo bottom (*False*) é definido como sempre falso, nada é uma prova do mesmo .
- O tipo bottom também pode ser visto como uma contradição *0 = 1*.

```
Module MyNot.

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

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

Do falso tudo se prova

- A teorema abaixo afirma que do falso segue-se que qualquer proposição P é verdade.
- *destruct* pode ser utilizado como *inversion* na hipótese falsa.

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

Exemplo de coisa falsa

- Como bem estabelicido pela ciência: zero e um são diferentes!

```
Theorem zero_not_one : ~(0 = 1).
Proof.
  intros contra. inversion contra.
Qed.
```

Desigualdade em Coq

- Ao invés de negar uma igualdade, pode-se utilizar o operador de desigualdade <>.

```
Check (0 <> 1).
Locate "<>" .
(* ===> Prop *)

Theorem zero_not_one' : 0 <> 1.
Proof.
  intros H. inversion H.
Qed.
```

Negação é apenas uma função para falso!

- Negação é apenas uma função para falso. Podemos utilizar *unfold* em funções.
- Falso implica em Falso:

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

Introdução da dupla negação

- Introdução da dupla negação pode ser provado.
- Tente fazer a eliminação da dupla negação!

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

Tipo verdade em Coq

- O tipo verdade (True) é a constante verdadeira de tipo *Prop*.

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

Print True.
```