CS720

Logical Foundations of Computer Science

Lecture 8: Logical connectives in Coq

Tiago Cogumbreiro

Today we will learn...

- more logic connectives
- constructive logic (and its relation to classical logic)
- building propositions with functions
- building propositions with inductive definitions

Logic connectives

Truth

$$\top$$

Truth

Truth can be encoded in Coq as a proposition that always holds, which can be described as a proposition type with a single constructor with 0-arity.

Truth example

```
Goal True.
```

(Done in class.)

Equivalence

$$P \iff Q$$

Logical equivalence

```
Definition iff A B : Prop = (A → B) /\ (B → A).

(* Notation ↔ *)
```

Split equivalence in goal

```
Goal (1 = 1 <-> True).

Theorem mult_0 :
  forall n m, n * m = 0 <-> n = 0 \/ m = 0.
Admitted.
```

- When induction is required, prove each side by induction independently.
  Split, and prove each side in its own theorem by induction.

Apply equivalence to assumption

```
Goal
  forall x y z,
  x * (y * z) = 0 ->
  x * y = 0 \/ z = 0.
Proof.
Admitted.
```

Interpret equivalence as equality

The `Setoid` library lets you treat an equivalence as an equals:

Tactics `rewrite`, `reflexivity`, and `symmetry` all handle equivalence as well.

```
Require Import Coq.Setoids.Setoid.

Goal
  forall x y z,
  x * (y * z) = 0 <-> x = 0 \/ (y = 0 \/ z = 0).
Proof.
Admitted.
```

Existential quantification

$$\exists x . P$$

Existential quantification

Notation:

```
exists x:A, P x
```

- To conclude a goal `exists x:A, P x` we can use tactics `exist x.` which yields `P x`.
- To use a hypothesis of type `H:exists x:A, P x`, you can use `destruct H as (x,H)`

Use exist for existential in goal

To conclude a goal `exists x:A, P x` we can use tactics `exist x.` which yields `P x`.

```
Goal
    forall y,
    exists x, Nat.beq x y = true.

Goal
    exists x y,
    3 + x = y.
```

- Give the value that satisfies the equality.
- You can play around with exists to figure out what makes sense.

Destruct existential in assumption

```
Goal
    forall n,
    (exists m, n = 4 + m) →
    (exists o, n = 2 + o).
```

Constructive logic

is not classical logic

Constructive logic is not classical logic

- Coq implements a constructive logic
- Every proof consists of evidence that is constructed
- You cannot assume the law of the excluded middle (proofs that appear out of thin air)
- Truth tables may fail you!
  Especially if there are negations involved.

The following are **unprovable** in constructive logic (and therefore in Coq):

```
Goal forall (P:Prop), P \/ ~ P.

Goal forall P Q, ((P -> Q) -> P) -> P.

Goal forall (P Q:Prop), ~(~P\/~Q) -> P \/ Q.
```

Building propositions with functions

Building propositions with functions

```
Fixpoint replicate (P:Prop) (n:nat) :=
  match n with
  | 0 ⇒ True
  | S m ⇒ P /\ replicate P m
  end.

Print replicate (1 = 0) 3.

Goal forall P,
Replicate P 0 <-> True.

Goal forall P n,
P <-> Replicate (S n).
```

List membership example

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

- Computation cannot match on propositions
- Computations destruct types, not propositions

Building propositions with data structures (inductively)

Enumerated propositions

Recall enumerated types?

You can think of true as an enumerated type.

```
Inductive True : Prop :=
| I : True.
```

Many equivalent proofs

```
Inductive Foo : Prop :=
| A : Foo
| B : Foo.
```

Many equivalent proofs

```
Inductive Foo : Prop :=
| A : Foo
| B : Foo.
```

Yet, same as having one

```
Goal
  Foo <-> True.
```

- We can prove `Foo` with `A` or with `B`, we still just have `Foo`
- What happens when we do a case analysis on `Foo`? Show when `A` holds, then show when `B` holds.

Falsehood

Falsehood in Coq is represented by an **empty** type.

```
Inductive False : Prop :=.
```

This explains why case analysis proves the following goal:

```
Goal
    False →
    1 = 0.
```

Composite inductive propositions

Disjunction

```
Inductive or (A B : Prop) : Prop :=
  | or_introl :
    A ->
    or A B
  | or_intror :
    B ->
    or A B
```

Conjunction

```
Inductive and (P Q : Prop) : Prop :=
| conj :
    P →
    Q →
    and P Q.
```

Adding parameters to predicates

```
Inductive Bar : nat → Prop :=
| C : Bar 1
| D : Bar 2.
```

Adding parameters to predicates

```
Inductive Bar : nat → Prop :=
| C : Bar 1
| D : forall n,
    Bar (S n).

Goal forall n,
    Bar n →
    n <> 0.
```

Alternative definition of Bar

```
Definition Bar2 n : Prop := n <> 0.
```

Existential

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
  | exist : forall x : A,
    P x →
    sig A P.
```

Recursive inductive propositions

Defining `In` inductively

```
Inductive In {A:Type} : A → list A → Prop :=
```

Defining In inductively

```
Inductive In {A:Type} : A → list A → Prop :=

| in_eq:
    forall x l,
    In x (x::l)
| in_cons:
    forall x y l,
    In x l →
    In x (y::l).
```

Fixed parameters in inductive propositions

```
Inductive In {A:Type} (x: A) : list A → Prop :=
| in_eq:
    forall x l,
    In x (x::l)
| in_cons:
    forall x y l,
    In x l →
    In x (y::l).
```

Defining even numbers

```
Inductive Even : nat → Prop :=
| even_0 : Even 0
| even_s_s : forall n,
    Even n →
    Even (S (S n)).
Goal forall n,
    Even n →
    exists m, n = 2 * m.
```