Propositions and Predicates

Pierre Castéran and Pierre Letouzey

Paris, November 2011
Shanghai, July 2012

In this class, we shall present how *Coq*'s type system allows us to express properties of programs and/or mathematical objects.
We will try to show the great expressive power of this formalism, mostly by examples.

Some very basic Propositions

Let e and e' be two expressions of the same type. We can build a proposition which expresses the equality between e and e'.

```
Check 1+1 = 2.
```

Some very basic Propositions

Let e and e' be two expressions of the same type. We can build a proposition which expresses the equality between e and e'.

```
Check 1+1 = 2.
```

1+1 = 2 : Prop

Some very basic Propositions

Let e and e' be two expressions of the same type. We can build a proposition which expresses the equality between e and e'.

```
Check 1+1 = 2.
```

1+1 = 2 : Prop

```
Check 2 = 3.
```

2 = 3 : Prop

Some very basic Propositions

Let e and e' be two expressions of the same type. We can build a proposition which expresses the equality between e and e'.

```
Check 1+1 = 2.
```

1+1 = 2 : Prop

```
Check 2 = 3.
```

2 = 3 : Prop

```
Check negb (negb true) = true.
```

negb (negb true) = true : Prop

Building Propositions from Predicates

A predicate is a function returning a proposition.

```
Check lt.
```

$lt : nat \to nat \to Prop$

Building Propositions from Predicates

A predicate is a function returning a proposition.

```
Check lt.
```

lt : nat → nat → Prop

```
Check lt 0 6.
```

$0 < 6 : Prop$

Building Propositions from Predicates

A predicate is a function returning a proposition.

```
Check lt.
```
lt : nat → nat → Prop
```
Check lt 0 6.
```
0 < 6 : Prop

```
Require Import ZArith. Open Scope Z_scope.
Check Zlt.
```
Zlt : Z → Z → Prop

Building Propositions from Predicates

A predicate is a function returning a proposition.

```
Check lt.
```
lt : nat → nat → Prop
```
Check lt 0 6.
```
0 < 6 : Prop

```
Require Import ZArith. Open Scope Z_scope.
Check Zlt.
```
Zlt : Z → Z → Prop
```
Check Zlt 2 3.
```
2 < 3 : Prop

Propositions vs. boolean values

Don't be mistaken :

A proposition (in Prop) usually cannot be *computed* much, but can be a Coq *statement* that we can (try to) prove.

Propositions vs. boolean values

Don't be mistaken :

A proposition (in Prop) usually cannot be *computed* much, but can be a Coq *statement* that we can (try to) prove.

Example of propositions : `True`, `False`, `1=2`,

...

Propositions vs. boolean values

Don't be mistaken :

A proposition (in Prop) usually cannot be *computed* much, but can be a Coq *statement* that we can (try to) prove.

Example of propositions : `True`, `False`, `1=2`,

...

A boolean (in bool) is a Coq *expression* that can be *computed* to the values `true` or `false`. A boolean can be used in programs but not directly in statements.

Propositions vs. boolean values

```
Check Zlt_bool.
```

Zlt_bool : Z → Z → bool

Propositions vs. boolean values

```
Check Zlt_bool.
```

Zlt_bool : Z → Z → bool

```
Check Zlt_bool 2 3.
```

Zlt_bool 2 3 : bool

Propositions vs. boolean values

```
Check Zlt_bool.
```

Zlt_bool : Z → Z → bool

```
Check Zlt_bool 2 3.
```

Zlt_bool 2 3 : bool

```
Compute Zlt_bool 2 3.
```

= true
: bool

Propositions vs. boolean values

```
Check Zlt_bool.
```
Zlt_bool : Z → Z → bool

```
Check Zlt_bool 2 3.
```
Zlt_bool 2 3 : bool

```
Compute Zlt_bool 2 3.
```
= true
: bool

```
Compute 2 < 3.
```
= (3 <= 3)%nat
: Prop

Propositions vs. boolean values

```
Definition Zmax n p := if n < p then p else n.
```

Propositions vs. boolean values

```
Definition Zmax n p := if n < p then p else n.
```

(Error : the term ‘‘ n < p ‘‘ has type ‘‘Prop’’ ... *)*

Propositions vs. boolean values

```
Definition Zmax n p := if n < p then p else n.
(*  Error : the term " n < p " has type "Prop" ... *)

Definition Zmax n p := if Zlt_bool n p then p else n.
```

Propositions vs. boolean values

```
Definition Zmax n p := if n < p then p else n.
(*  Error : the term `` n < p `` has type ``Prop'' ... *)

Definition Zmax n p := if Zlt_bool n p then p else n.

Lemma not_a_statement : Zlt_bool 2 3.
```

Propositions vs. boolean values

```
Definition Zmax n p := if n < p then p else n.
(*  Error : the term '' n < p '' has type ''Prop'' ... *)

Definition Zmax n p := if Zlt_bool n p then p else n.

Lemma not_a_statement : Zlt_bool 2 3.
(* Error: The term "Zlt_bool 2 3" has type "bool"
   which should be Set, Prop or Type. *)
```

Propositions vs. boolean values

```
Definition Zmax n p := if n < p then p else n.
(*  Error : the term `` n < p `` has type ``Prop'' ... *)

Definition Zmax n p := if Zlt_bool n p then p else n.

Lemma not_a_statement : Zlt_bool 2 3.
(* Error: The term "Zlt_bool 2 3" has type "bool"
   which should be Set, Prop or Type. *)
```

Propositions vs. boolean values

The examples below are well formed propositions :

```
Zlt_bool 2 3 = true
```

Propositions vs. boolean values

The examples below are well formed propositions :

```
Zlt_bool 2 3 = true

Zlt_bool 2 3 = false
```

Propositions vs. boolean values

The examples below are well formed propositions :

```
Zlt_bool 2 3 = true

Zlt_bool 2 3 = false

Zeq_bool (6*6) (9*4) = true
```

Propositions vs. boolean values

The examples below are well formed propositions :

```
Zlt_bool 2 3 = true

Zlt_bool 2 3 = false

Zeq_bool (6*6) (9*4) = true

6*6 = 9*4
```

Propositions vs. boolean values

The examples below are well formed propositions :

```
Zlt_bool 2 3 = true

Zlt_bool 2 3 = false

Zeq_bool (6*6) (9*4) = true

6*6 = 9*4

45 <= Zmax 34 45
```

Quantifiers and Connectives

The following propositions are well formed :

```
(* The square of any integer is greater or equal than 0 *)
forall n:Z, 0 <= n * n

(* There exists at least some integer whose square is 4 *)
exists n:Z, n * n = 4

(* Z is unbounded *)
forall n:Z, exists p:Z, n < p

(* A well-formed, unprovable proposition *)
forall n:Z, n^2 <= 2^n
```

Negation (not, $\sim$)

```
(* Zlt is irreflexive *)

Check Zlt_irrefl.
```

Negation (`not`, $\sim$)

```
(* Zlt is irreflexive *)

Check Zlt_irrefl.
```

Zlt_irrefl : forall n : Z, $\sim$ *n* $<$ *n*

Negation (not, ∼)

```
(* Zlt is irreflexive *)

Check Zlt_irrefl.
```

Zlt_irrefl : forall n : Z, ∼ n < n

```
Check forall n : Z, ~ n < n.
```

Negation (`not`, $\sim$)

```
(* Zlt is irreflexive *)

Check Zlt_irrefl.
```
Zlt_irrefl : forall n : Z, ~ n < n
```
Check forall n : Z, ~ n < n.
```
forall n : Z, ~ n < n : Prop

Negation (not, ∼)

```
(* Zlt is irreflexive *)

Check Zlt_irrefl.
```
Zlt_irrefl : forall n : Z, ∼ n < n
```
Check forall n : Z, ∼ n < n.
```
forall n : Z, ∼ n < n : Prop
```
(* There is no integer square root of 2 *)
Check ∼(exists n:Z, n*n = 2).
```

Negation (not, $\sim$)

```
(* Zlt is irreflexive *)

Check Zlt_irrefl.
Zlt_irrefl : forall n : Z, ~ n < n
Check forall n : Z, ~ n < n.
forall n : Z, ~ n < n : Prop

(* There is no integer square root of 2 *)
Check ~(exists n:Z, n*n = 2).


Require Import List.
(* No number in the empty list of integers ! *)
```

Negation (not , ∼)

```
(* Zlt is irreflexive *)

Check Zlt_irrefl.
Zlt_irrefl : forall n : Z, ~ n < n
Check forall n : Z, ~ n < n.
forall n : Z, ~ n < n : Prop

(* There is no integer square root of 2 *)
Check ~(exists n:Z, n*n = 2).


Require Import List.
(* No number in the empty list of integers ! *)
```

Implication (→, -> in ascii)

```
(*  Zle_trans *)
forall n m p : Z, n <= m → m <= p → n <= p.

(*  Zlt_asym *)
forall n p:Z,  n < p →  ~ p < n.
```

Implication (→, -> in ascii)

```
(*  Zle_trans *)
forall n m p : Z, n <= m → m <= p → n <= p.

(*  Zlt_asym *)
forall n p:Z,  n < p →  ~ p < n.
```

Notice that in *Coq*, negation is defined in terms of implication and falsehood :

```
Definition not (A:Prop) := A → False.
```

Beware of associativity !

Coq considers $\rightarrow$ as a right-associative binary operator :
A proposition written $A \rightarrow B \rightarrow C$ must be read as $A \rightarrow (B \rightarrow C)$ and *not* $(A \rightarrow B) \rightarrow C$.

Beware of associativity !

Coq considers $\rightarrow$ as a right-associative binary operator :
A proposition written $A{\rightarrow}B{\rightarrow}C$ must be read as $A{\rightarrow}(B{\rightarrow}C)$ and *not* $(A{\rightarrow}B){\rightarrow}C$.

Notice that $A{\rightarrow}B{\rightarrow}C$ is logically equivalent to $A \wedge B{\rightarrow}C$.

Beware of associativity !

Coq considers $\rightarrow$ as a right-associative binary operator :
A proposition written $A \rightarrow B \rightarrow C$ must be read as $A \rightarrow (B \rightarrow C)$ and *not* $(A \rightarrow B) \rightarrow C$.

Notice that $A \rightarrow B \rightarrow C$ is logically equivalent to $A \wedge B \rightarrow C$.

This remark can be generalized to n implications :
$A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n \rightarrow B$ is logically equivalent to
$A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow B$.

Disjunction (or, \/)

```
forall n:Z, 0 <= n \/ n < 0.

forall n p : Z, n < p \/ p <= n.

forall n p : Z, n < p \/ p = n \/ p < n.

(forall n : nat, n = 0 \/ exists p:nat, p < n)%nat.

forall l:list Z,
   l = nil \/ exists a, exists l', l = a::l'.
```

Conjonction (and, /\)

```
let (q,r) :=  Zdiv_eucl 456 37 in
                   456 = 37 * q + r /\
                   0 <= r < 37. (*  0 <= r /\ r < 37 *)

forall a b q r: Z, 0 < b →
                   a = b * q + r →
                   0 <= r < b →
                   q = a / b /\ r = a mod b.
```

Logical Equivalence (`iff`, $\leftrightarrow$, `<->` in ascii)

```
(* Zlt_is_lt_bool *)
forall n m : Z, n < m ↔ Zlt_bool n m = true

forall l1 l2 : list Z,
   (forall z:Z, In z (l1 ++ l2)  ↔
    In z l1 \/ In z l2).
```

Building new Predicates

```
Definition is_square_root (n r : Z) :=
    r * r <= n < (r+1)*(r+1).

Check is_square_root 9 3.
```

The predicate `is_square_root` can be used to *specify* a square root function : If you build a `sqrt` function, you'll want to prove the following proposition :

```
forall n, 0<=n → is_square_root n (sqrt n)
```

Building new Predicates

```
Definition is_prime (n:Z) :=
  2 <= n /\
  forall p q, 0 < p → 0 < q → n = p * q →
              p = n \/ q = n.
```

Building new Predicates

Predicates can be built either directly, or inductively, or recursively. For instance, given a type `A`, membership in a `(list A)` can be written :

```
Fixpoint In (a:A) (l:list A) : Prop :=
    match l with
      | nil => False
      | b :: m => b = a \/ In a m
    end.
```

Building new Predicates

```
(* number of occurences of n in l *)
Fixpoint multiplicity (n:Z)(l:list Z) : nat :=
  match l with
    nil => 0%nat
  | a::l' => if Zeq_bool n a
             then S (multiplicity n l')
             else multiplicity n l'
  end.

(* l' is a permutation of l *)
Definition is_perm (l l':list Z) :=
    forall n, multiplicity n l = multiplicity n l'.
```

Specifying a merge function

```
(* The binary function f preserves
  the elements' multiplicity  *)

Definition preserves_multiplicity
      (f : list Z → list Z → list Z) :=
  forall l l' n,
     multiplicity n (f l l') =
    (multiplicity n l + multiplicity n l')%nat.
```

Specifying a merge function (2)

```
(* let's assume the following predicate "to be sorted"
  is defined *)
Parameter sorted_Zle : list Z → Prop.

Definition preserves_sort
      (f : list Z → list Z → list Z) :=
  forall l l', sorted_Zle l → sorted_Zle l' →
               sorted_Zle (f l l').

Definition merge_spec (f : list Z → list Z → list Z):=
  preserves_sort f /\ preserves_multiplicity f.
```

Quantifying over propositions and predicates

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

Quantifying over propositions and predicates

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

forall P : Prop, ~ P ↔ P → False.
```

Quantifying over propositions and predicates

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

forall P : Prop, ~ P ↔ P → False.

forall P Q R:Prop, (P /\ Q → R) ↔ (P → Q → R).
```

Quantifying over propositions and predicates

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

forall P : Prop, ~ P ↔ P → False.

forall P Q R:Prop, (P /\ Q → R) ↔ (P → Q → R).

forall P Q, P \/ Q → Q \/ P.
```

Quantifying over propositions and predicates

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

forall P : Prop, ~ P <-> P -> False.

forall P Q R:Prop, (P /\ Q -> R) <-> (P -> Q -> R).

forall P Q, P \/ Q -> Q \/ P.

False_ind: forall P : Prop, False -> P
```

Quantifying over propositions and predicates

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

forall P : Prop, ~ P ↔ P → False.

forall P Q R:Prop, (P /\ Q → R) ↔ (P → Q → R).

forall P Q, P \/ Q → Q \/ P.

False_ind: forall P : Prop, False → P

absurd: forall A C : Prop, A → ~ A → C
```

```
forall P : nat → Prop, ~ (exists n, P n) →
                        forall n, ~ P n.
```

```
forall P : nat → Prop, ~ (exists n, P n) →
                        forall n, ~ P n.

nat_ind: forall P : nat → Prop,
 P 0 →
 (forall n:nat, P n → P (S n)) →
 forall n:nat, P n.
```

```
forall P : nat → Prop, ∼ (exists n, P n) →
                        forall n, ∼ P n.

nat_ind: forall P : nat → Prop,
 P 0 →
 (forall n:nat, P n → P (S n)) →
 forall n:nat, P n.

(forall P:Prop, P \/ ∼ P) ↔
(forall P:Prop, ∼ ∼ P → P).
```

```
Definition or_ex (P Q : Prop) : Prop :=
    (P \/ Q) /\ ~(P /\ Q).
```

```
Definition or_ex (P Q : Prop) : Prop :=
    (P \/ Q) /\ ~(P /\ Q).

Lemma or_ex_not_iff : forall P Q, or_ex P Q →
     ~ (P ↔ Q).
```

Quantification over types

```
SearchRewrite (rev (rev _)).
rev_involutive:
   forall (A : Type) (l : list A), rev (rev l) = l
```

Quantification over types

```
SearchRewrite (rev (rev _)).
rev_involutive:
   forall (A : Type) (l : list A), rev (rev l) = l

forall (A:Type)(P:A→Prop), ~(exists x, P x ) →
                            forall x, ~ P x.
```

Quantification over types

```
SearchRewrite (rev (rev _)).
rev_involutive:
   forall (A : Type) (l : list A), rev (rev l) = l

forall (A:Type)(P:A→Prop), ~(exists x, P x ) →
                            forall x, ~ P x.

forall (A:Type)(x y z:A), x = y → y = z → x = z.
```

Quantification over types

```
SearchRewrite (rev (rev _)).
rev_involutive:
   forall (A : Type) (l : list A), rev (rev l) = l

forall (A:Type)(P:A→Prop), ~(exists x, P x ) →
                                forall x, ~ P x.

forall (A:Type)(x y z:A), x = y → y = z → x = z.

forall (A B:Type)(a:A)(b:B), fst (a,b) = a.
```

Quantification over types

```
SearchRewrite (rev (rev _)).
rev_involutive:
   forall (A : Type) (l : list A), rev (rev l) = l

forall (A:Type)(P:A→Prop), ~(exists x, P x ) →
                                forall x, ~ P x.

forall (A:Type)(x y z:A), x = y → y = z → x = z.

forall (A B:Type)(a:A)(b:B), fst (a,b) = a.

forall (A B : Type)(p:A*B), p = (fst p, snd p).
```

A Little Case Study

```
(*  Compatibility between a predicate and a
boolean function *)

Definition decides (A:Type)(P:A→Prop)(p : A → bool) :=
  forall a:A, P a ↔ (p a)=true.
```

A Little Case Study

```
(*  Compatibility between a predicate and a
boolean function *)

Definition decides (A:Type)(P:A→Prop)(p : A → bool) :=
  forall a:A, P a ↔ (p a)=true.

Definition decides2
     (A:Type)(P:A→A→Prop)(p : A → A → bool) :=
  forall a b :A , P a b ↔ p a b = true.
```

A Little Case Study

```
(*  Compatibility between a predicate and a
boolean function *)

Definition decides (A:Type)(P:A→Prop)(p : A → bool) :=
  forall a:A, P a ↔ (p a)=true.

Definition decides2
     (A:Type)(P:A→A→Prop)(p : A → A → bool) :=
  forall a b :A , P a b ↔ p a b = true.

Check decides2 _ Zle Zle_bool.
```

decides2 Z Zle Zle_bool : Prop

```
Section sort_spec.
Parameter sorted :
  forall (A:Type), relation A → list A → Prop.
```

```
Section sort_spec.
Parameter sorted :
  forall (A:Type), relation A → list A → Prop.

Variable sort:
  forall A:Type,(A→A→bool) → list A → list A.
```

```
Section sort_spec.
Parameter sorted :
  forall (A:Type), relation A → list A → Prop.

Variable sort:
  forall A:Type,(A→A→bool) → list A → list A.

Definition sort_correct :=
 forall (A:Type)
        (R : relation A)
        (r : A → A → bool),
  decides2 A R r →
  forall l, let l' := sort A r l in
    sorted A R l' /\
    forall a, In a l ↔ In a l'.
```