Propositions and Predicates

Pierre Castéran

Paris, June 2010

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'.

```
Require Import ZArith.
Open Scope Z_scope.

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'.

```
Require Import ZArith.
Open Scope Z_scope.

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'.

```
Require Import ZArith.
Open Scope Z_scope.

Check 1+1 = 2.
```

1+1 = 2 : Prop

```
Check 2 = 3.
```

2 = 3 : Prop

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

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'.

```
Require Import ZArith.
Open Scope Z_scope.

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

`Check Zlt.`

Building Propositions from Predicates

```
Check Zlt.
```

Zlt : Z -> Z -> Prop

```
Check Zlt 2 3.
```

2 < 3 : Prop

Building Propositions from Predicates

```
Check Zlt.
```
Zlt : Z -> Z -> Prop
```
Check Zlt 2 3.
```
2 < 3 : Prop
```
Check le.
```
le : nat -> nat -> Prop
```
Check le 0%nat 6%nat.
```

Building Propositions from Predicates

```
Check Zlt.
```

Zlt : Z -> Z -> Prop

```
Check Zlt 2 3.
```

2 < 3 : Prop

```
Check le.
```

le : nat -> nat -> Prop

```
Check le 0%nat 6%nat.
```

(0 <= 6)%nat : Prop

Don't be mistaken !

```
Check Zlt_bool 2 3.
```

Zlt_bool 2 3 : bool

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

Don't be mistaken !

```
Check Zlt_bool 2 3.
```

Zlt_bool 2 3 : bool

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

(Error : the term “ n < p “ has type “Prop” ... *)*

Don't be mistaken !

```
Check Zlt_bool 2 3.
```

Zlt_bool 2 3 : bool

```
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.
```

Notice that the following examples are well formed propositions :

```
Ztl_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 are well-formed propositions :

```
(* 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.
```

There exists some useful notations for nested quantifiers, which we shall present in further examples.

Negation (not)

```
(*  Zlt is irreflexive *)

Check Zlt_irrefl.
```

Negation (not)

```
(*  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, $\sim$ *n* $<$ *n*

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

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 ! *)
forall z:Z, ~ In z nil.
~ (exists z:Z, In z nil).
```

Implication (->)

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

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

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)

```
Coq.ZArith.Zbool.Zle_bool :
forall n m : Z, n <= m <-> Zle_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.

Definition is_prime (n:Z) :=
  2 <= n ∧
  forall p q, 0 < p -> 0 < q -> n = p * q ->
               p = n ∨ q = n.
```

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 m preserves
  the elements' multiplicity  *)

Definition preserves_multiplicity
      (m : list Z -> list Z -> list Z) :=
  forall l l' n,
     multiplicity n (m 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
      (m : list Z -> list Z -> list Z) :=
  forall l l', sorted_Zle l -> sorted_Zle l' ->
               sorted_Zle (m l l').

Definition merge_spec (m : list Z -> list Z -> list Z):=
  preserves_sort m ∧ preserves_multiplicity m.
```

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 O ->
 (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) := P ∨ Q ∧ (∼P ∧ ∼ Q).
```

```
Definition or_ex (P Q : 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.
Check decides2 _ Zle Zle_bool.
```

decides2 Z Zle Zle_bool : Prop

```
Require Import Relations.
Print order.

Record order (A : Type) (R : relation A) : Prop :=
Build_order
  {  ord_refl : reflexive A R;
     ord_trans : transitive A R;
     ord_antisym : antisymmetric A R }

Print antisymmetric.
antisymmetric =
fun (A : Type) (R : relation A) =>
   forall x y : A, R x y -> R y x -> x = y
: forall A : Type, relation A -> 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 s:
  forall A:Type,(A->A->bool) -> list A -> list A.
```

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

Variable s:
  forall A:Type,(A->A->bool) -> list A -> list A.

Definition sort_correct :=  (* to improve ! *)
    forall  (A:Type)
            (R : relation A)
            (r : A -> A -> bool),
                 order A R -> decides2 A R r ->
                forall l, let l' := s A r l in
                    sorted A R l' ∧
                     forall a, In a l <-> In a l'.
```