Coq par l'exemple

Pierre Castéran, Université Bordeaux 1 et LaBRI

28 janvier 2010

coq.inria.fr
www.labri.fr/~casteran/CoqArt

Définir des constantes, faire des calculs

```
Definition trois := 3.
```

Définir des constantes, faire des calculs

```
Definition trois := 3.
```
trois is defined

```
Print trois.
```

Définir des constantes, faire des calculs

```
Definition trois := 3.
```
trois is defined

```
Print trois.
```
trois = 3 : nat

```
Eval compute in 10 * trois * trois.
```

Définir des constantes, faire des calculs

```
Definition trois := 3.
```
trois is defined

```
Print trois.
```
trois = 3 : nat

```
Eval compute in 10 * trois * trois.
```
= 90 : nat

Définir des fonctions simples

```
Definition carre (n:nat) := n * n.

Definition somme_carres (a b:nat) := carre a + carre  b.

Check somme_carres.
```

Définir des fonctions simples

```
Definition carre (n:nat) := n * n.

Definition somme_carres (a b:nat) := carre a + carre  b.

Check somme_carres.
somme_carres  : nat -> nat -> nat

Eval compute in somme_carres 3 4.
```

Définir des fonctions simples

```
Definition carre (n:nat) := n * n.

Definition somme_carres (a b:nat) := carre a + carre  b.

Check somme_carres.
somme_carres  : nat -> nat -> nat

Eval compute in somme_carres 3 4.
  = 25  : nat
```

Définitions récursives structurelles (arithmétique de Peano)

```
Fixpoint fact n :=
  match n with 0 => 1
             | S p => n * fact p
  end.

Eval compute in fact 6.
```

Définitions récursives structurelles (arithmétique de Peano)

```
Fixpoint fact n :=
  match n with 0 => 1
               | S p => n * fact p
  end.

Eval compute in fact 6.
 = 720 : nat
```

L'arithmétique de Peano joue ici un rôle surtout pédagogique et fondamental (récurrence). Il existe des représentations plus compactes des nombres entiers !

Fonctionnelles et fonctions anonymes

```
Definition compose_nat (f g:nat -> nat)  :=
    fun x : nat => g (f x).

Definition f0 := compose_nat S carre.

Eval compute in f0 8.
```

Fonctionnelles et fonctions anonymes

```
Definition compose_nat (f g:nat -> nat)  :=
    fun x : nat => g (f x).

Definition f0 := compose_nat S carre.

Eval compute in f0 8.
  = 81 : nat

Check fun f : nat -> nat => compose_nat f f.
```

Fonctionnelles et fonctions anonymes

```
Definition compose_nat (f g:nat -> nat)  :=
    fun x : nat => g (f x).

Definition f0 := compose_nat S carre.

Eval compute in f0 8.
  = 81 : nat

Check fun f : nat -> nat => compose_nat f f.
fun f : nat -> nat => compose_nat f f
    : (nat -> nat) -> nat -> nat
```

Polymorphisme

```
Definition compose (A B C:Type)
                   (f : A -> B)(g : B -> C) :=
           fun x => g (f x).

Fixpoint iterate (A:Type)(f : A -> A)(z:A)(n:nat) :=
  match n with 0 => z
             | S p => iterate A  f (f z) p
  end.
```

```
Require Import ZArith.
Open Scope Z_scope.

Definition exp2 := iterate Z (Zmult 2) 1.

Eval compute in exp2 55.
```

```
Require Import ZArith.
Open Scope Z_scope.

Definition exp2 := iterate Z (Zmult 2) 1.

Eval compute in exp2 55.
    = 36028797018963968 : Z

Definition fibonacci n :=
  fst (iterate _  (fun p:Z*Z => (snd p, fst p + snd p))
      (1,1)
       n).

Eval compute in fibonacci 20.
```

```
Require Import ZArith.
Open Scope Z_scope.

Definition exp2 := iterate Z (Zmult 2) 1.

Eval compute in exp2 55.
    = 36028797018963968 : Z

Definition fibonacci n :=
  fst (iterate _  (fun p:Z*Z => (snd p, fst p + snd p))
      (1,1)
       n).

Eval compute in fibonacci 20.
10946 : Z
```

Propositions

Le type `Prop` est associé aux *propositions logiques* (ne pas confondre avec le type `bool`).

```
Check 3 < 5.
```

3 < 5 : Prop

Autres exemples :

- `forall n:nat, 4 < n -> carre n < exp2 n.`
- `forall n p:nat, n+p = p+n.`
- `~exists x:Q, x ^ 2 == 2#1.`
- `forall P Q : Prop, (P -> Q) <-> (~Q -> ~P).`
- ```
  forall (A:Type)(P : A -> Prop),
      ~(exists x, P x) -> forall x, ~ P x.
  ```
  ```

Un exemple de preuve : la somme des n premiers nombres impairs

```
Fixpoint sum_odd (n : nat) : nat :=
 match n with 0 => 0
              | S p => sum_odd p + 1 + 2 * p
 end.
Eval compute in sum_odd 4.
```

= 16 : nat

```
Lemma sum_odd_square : forall n:nat, sum_odd n = n * n.
Proof.
```

1 subgoal

==========

*forall n : nat, sum_odd n = n * n*

```
induction n.
```

1 subgoal

==========

*forall n : nat, sum_odd n = n * n*

```
induction n.
```

2 subgoals

==========

*sum_odd 0 = 0 * 0*

subgoal 2 is:

*sum_odd (S n) = S n * S n*

*sum_odd 0 = 0 * 0*

```
simpl.
```

*sum_odd 0 = 0 * 0*

```
simpl.
```

0 = 0

```
trivial.
```

```
  sum_odd 0 = 0 * 0
simpl.
0 = 0
trivial.
1 subgoal
  n : nat
  IHn : sum_odd n = n * n
  ============================
   sum_odd (S n) = S n * S n
simpl.
```

```
  n : nat
  IHn : sum_odd n = n * n
  ============================
   sum_odd n + 1 + (n + (n + 0)) = S (n + n * S n)
rewrite IHn.
```

$n : nat$
$IHn : sum\_odd\ n = n * n$
$=========================$
$sum\_odd\ n + 1 + (n + (n + 0)) = S\ (n + n * S\ n)$

```
rewrite IHn.
```

$=========================$
$n * n + 1 + (n + (n + 0)) = S\ (n + n * S\ n)$

```
ring.
```

```
  n : nat
  IHn : sum_odd n = n * n
  ============================
   sum_odd n + 1 + (n + (n + 0)) = S (n + n * S n)
rewrite IHn.
 ============================
   n * n + 1 + (n + (n + 0)) = S (n + n * S n)
ring.
Proof completed
```

```
Qed.
```

```
Qed.
```
sum_odd_square is defined

```
Check sum_odd_square 45.
```

```
Qed.
```
sum_odd_square is defined

```
Check sum_odd_square 45.
```
sum_odd_square 45
*: sum_odd 45 = 45 * 45*

Entiers De Peano

```
Check 0.
0 : nat
Check S.
S: nat -> nat
Check S (S (S 0)).
3 : nat
Check plus.
plus : nat -> nat -> nat
```

```
Lemma L1 : forall n, 0 + n = n.
Proof.
 intro n.
 n : nat
========
0 + n = n
 simpl.
 n : nat
========
n = n
reflexivity.
Qed.
```

```
Lemma L2 : forall n, n + 0 = n.
Proof.
 intro n;simpl.
```

$n : nat$

$==========$

$n + 0 = n$

```
 induction n.
```

```
Lemma L2 : forall n, n + 0 = n.
Proof.
 intro n;simpl.
```

n : nat

==========

n + 0 = n

```
 induction n.
```

2 subgoals

==========

0 + 0 = 0

subgoal 2 is:

S n + 0 = S n

```
reflexivity.
```

```
 reflexivity.
```
$n : nat$
$IHn : n + 0 = n$
$=========================$
$S\ n + 0 = S\ n$
```
simpl.
```

```
 reflexivity.
```

$n : nat$

$IHn : n + 0 = n$

$============================$

$S\ n + 0 = S\ n$

```
simpl.
```

$n : nat$

$IHn : n + 0 = n$

$============================$

$S\ (n + 0) = S\ n$

```
rewrite IHn;reflexivity.
Qed.
```

```
 Lemma plus_comm : forall n p, n + p = p + n.
Proof.
 induction n.
  ============================
  forall p : nat, 0 + p = p + 0
intro p; simpl;rewrite L2;trivial.
```

```
 Lemma plus_comm : forall n p, n + p = p + n.
Proof.
 induction n.
```

==========

forall p : nat, 0 + p = p + 0

```
intro p; simpl;rewrite L2;trivial.
```

n : nat

IHn : forall p : nat, n + p = p + n

==========

forall p : nat, S n + p = p + S n

```
intro p; simpl; rewrite (IHn p).
```

```
 Lemma plus_comm : forall n p, n + p = p + n.
Proof.
 induction n.
```

==========

forall p : nat, 0 + p = p + 0

```
intro p; simpl;rewrite L2;trivial.
```

n : nat

IHn : forall p : nat, n + p = p + n

==========

forall p : nat, S n + p = p + S n

```
intro p; simpl; rewrite (IHn p).
```

...

p : nat

==========

S (p + n) = p + S n

```
induction p.
```

```
induction p.
```

$$==========$$

$$S\ (0 + n) = 0 + S\ n$$

```
simpl;trivial.
```

```
induction p.
```

```
  ============================
   S (0 + n) = 0 + S n
```

```
simpl;trivial.
```

```
  n : nat
  IHn : forall p : nat, n + p = p + n
  p : nat
  IHp : S (p + n) = p + S n
  ============================
   S (S p + n) = S p + S n
```

```
induction p.
```

```
  ============================
   S (0 + n) = 0 + S n
```

```
simpl;trivial.
```

```
  n : nat
  IHn : forall p : nat, n + p = p + n
  p : nat
  IHp : S (p + n) = p + S n
  ============================
   S (S p + n) = S p + S n
```

```
   simpl; rewrite <- IHp;trivial.
Qed.
```

```
Lemma nat_double_ind :
 forall (P:nat->Prop),
 P 0 ->
 P 1 ->
 (forall i, P i -> P (S i) -> P(S (S i))) ->
 forall n, P n.
Proof.
intros.
```

```
Lemma nat_double_ind :
 forall (P:nat->Prop),
 P 0 ->
 P 1 ->
 (forall i, P i -> P (S i) -> P(S (S i))) ->
 forall n, P n.
Proof.
intros.
```

P : nat -> Prop
H : P 0
H0 : P 1
H1 : forall i : nat, P i -> P (S i) -> P (S (S i))
n : nat
==========
P n

```
assert (P n ∧ P (S n)).
```

```
assert (P n ∧ P (S n)).
```

2 subgoals

P : nat -> Prop
H : P 0
H0 : P 1
H1 : forall i : nat, P i -> P (S i) -> P (S (S i))
n : nat
==========
P n ∧ P (S n)

subgoal 2 is:
P n

```
induction n;auto.
```

```
induction n;auto.
```

H1 : forall i : nat, P i -> P (S i) -> P (S (S i))

n : nat

IHn : P n ∧ P (S n)

==========

P (S n) ∧ P (S (S n))

```
induction n;auto.
```
H1 : forall i : nat, P i -> P (S i) -> P (S (S i))
n : nat
IHn : P n ∧ P (S n)
==========
P (S n) ∧ P (S (S n))
```
destruct IHn;firstorder.
```

```
induction n;auto.
```

H1 : forall i : nat, P i -> P (S i) -> P (S (S i))

n : nat

IHn : P n ∧ P (S n)

==========

P (S n) ∧ P (S (S n))

```
destruct IHn;firstorder.
```

H2 : P n ∧ P (S n)

==========

P n

```
tauto.
Qed.
```

Exemple d'application

```
Fixpoint fib (n : nat) : nat :=
 match n with 0 => 1
              | 1 => 1
              | S (S p as q) => fib q + fib p
end.

Lemma F1: forall n, fib (S (S n)) = fib (S n) + fib n.
induction n using nat_double_ind;simpl;auto.
Qed.
```

```
Goal forall n, n <=  fib n.
intro n; induction n using nat_double_ind.
simpl;auto.
simpl;auto.
rewrite F1.
```

```
Goal forall n, n <=  fib n.
intro n; induction n using nat_double_ind.
simpl;auto.
simpl;auto.
rewrite F1.
```

n : nat
IHn0 : S n <= fib (S n)
IHn : n <= fib n
==========
S (S n) <= fib (S n) + fib n

```
Goal forall n, n <=  fib n.
intro n; induction n using nat_double_ind.
simpl;auto.
simpl;auto.
rewrite F1.
```

n : nat
IHn0 : S n <= fib (S n)
IHn : n <= fib n
==========
S (S n) <= fib (S n) + fib n

```
destruct n;auto.
```

1 subgoal

n : nat
IHn0 : S (S n) <= fib (S (S n))
IHn : S n <= fib (S n)
==========
S (S (S n)) <= fib (S (S n)) + fib (S n)

1 subgoal

n : nat
IHn0 : S (S n) <= fib (S (S n))
IHn : S n <= fib (S n)
==========
S (S (S n)) <= fib (S (S n)) + fib (S n)

```
Require Import Omega.
omega.
Qed.
```

Ordinaux

```
(** cons a n b represents  omega^a *(S n)  + b *)

Inductive T1 : Set :=
  zero : T1
| cons : T1 -> nat -> T1 -> T1.

Definition omega := cons (cons zero 0 zero) 0 zero.

Definition nat_to_ord (n:nat) : T1 :=
 match n with  0 => zero
             | S p => cons zero p zero
 end.
```

```
Eval compute in omega + omega.
```

```
Eval compute in omega + omega.
```

= cons (cons zero 0 zero) 1 zero

: T1

```
Eval compute in omega + omega.
   = cons (cons zero 0 zero) 1 zero
    : T1
Eval compute in omega * omega.
```

```
Eval compute in omega + omega.
   = cons (cons zero 0 zero) 1 zero
     : T1
Eval compute in omega * omega.
     = cons (cons zero 1 zero) 0 zero
     : T1
```

```
Eval compute in omega + omega.
   = cons (cons zero 0 zero) 1 zero
    : T1
Eval compute in omega * omega.
    = cons (cons zero 1 zero) 0 zero
    : T1
Goal omega + omega * omega = omega * omega.
reflexivity.
Qed.
```

```
Goal  omega * omega <  omega * omega + omega.
Check lt_intro.
```

```
Goal  omega * omega <  omega * omega + omega.
Check lt_intro.
```

lt_intro

: forall a b : T1, compare a b = Lt -> a < b

```
Goal  omega * omega <  omega * omega + omega.
Check lt_intro.
```

lt_intro

: forall a b : T1, compare a b = Lt -> a < b

```
apply lt_intro.
```

```
Goal  omega * omega <  omega * omega + omega.
Check lt_intro.
```

lt_intro

: forall a b : T1, compare a b = Lt -> a < b

```
apply lt_intro.
```

==========================

*compare (omega * omega) (omega * omega + omega) = Lt*

```
Goal  omega * omega <  omega * omega + omega.
Check lt_intro.
```
lt_intro
* : forall a b : T1, compare a b = Lt -> a < b*
```
apply lt_intro.
```
* ============================*
*   compare (omega * omega) (omega * omega + omega) = Lt*
```
reflexivity.
Qed.
```

```
Goal forall n, nat_to_ord n + omega = omega.
induction n;simpl;auto.
Qed.

Goal forall n, nat_to_ord n < omega.
destruct n;apply lt_intro;auto.
Qed.
```

```
Lemma epsilon0_unbounded : forall alpha, exists beta, al-
pha < beta.
Proof.
 intro alpha; exists (succ alpha).
 ...
```

```
Lemma epsilon0_unbounded' :
    ~(exists  alpha, forall beta, beta <= alpha).

Lemma L5 : forall alpha, nf alpha -> alpha < omega ->
            is_finite alpha.
```

$<$ total, bien fondé, victoire d'Hercule, etc.