Introduction à Coq

C. Dubois, J.-C. Léchenet

Avant de commencer

(Véritable introduction la semaine prochaine)

L'UE Prog1 est divisée en 2 modules :

- Sémantique des Langages (ISL)
- Preuve Formelle Mécanisée (PFM)
  - Initiation à Coq
  - Lien entre logique et programmation (preuves et programmes)
  - Déduction automatique
  - Évaluation : projet Coq + contrôle avec documents

Ce cours : introduction à Coq du point de vue pratique

Plan

Plan

Besoin

- Écrire des preuves complètement formelles est fastidieux
- Idée : utiliser un ordinateur pour vérifier les raisonnements
- Les logiciels permettant cette interaction entre l'homme et l'ordinateur sont appelés des assistants de preuve (*proof assistant*/*interactive theorem prover* en anglais)

Quelques exemples

Plusieurs outils existent :

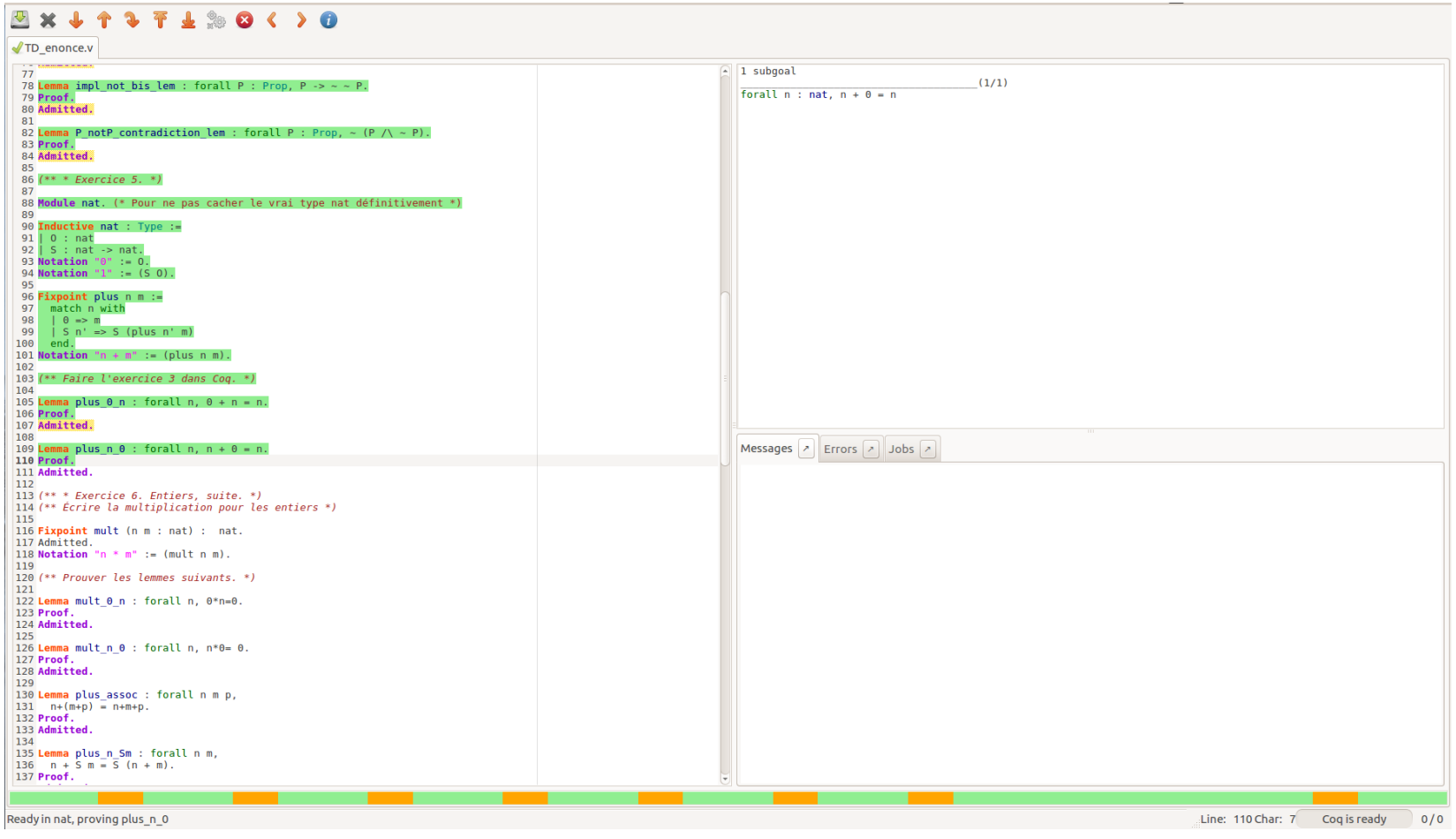
- **Coq**
- Isabelle/HOL
- Agda
- Mizar
- PVS
- ...

Coq est développé par l'Inria depuis 1984.

Réussites :

- En mathématiques : théorème des quatre couleurs (2004), théorème de Feit et Thomson (2012)
- En informatique : CompCert un compilateur certifié pour le langage C

Outil interactif



Fondements logiques

- Coq est basé sur le Calcul des Constructions Inductives
- Cette présentation ne présente pas cette théorie (trop complexe)
- Elle s'attache surtout à apporter les bases pour se servir de Coq comme un outil

Possibilités attendues du langage

On voudrait pouvoir :

1. Définir des objets
2. Énoncer des théorèmes
3. Les prouver

Possibilités attendues du langage

On voudrait pouvoir :

1. Définir des objets
2. Énoncer des théorèmes
3. Les prouver

Des langages différents (en première approximation) :

1. Un langage fonctionnel pur appelé Gallina
2. Une logique d'ordre supérieure
3. Un ensemble de tactiques

Plan

Type algébrique de données (1/2)

- Un type défini par les différents cas possibles, les *constructeurs*
- Les constructeurs sont des objets à part entière
- Existe en OCaml, $\approx$ type enum en C

Type algébrique de données (1/2)

- Un type défini par les différents cas possibles, les *constructeurs*
- Les constructeurs sont des objets à part entière
- Existe en OCaml, $\approx$ type enum en C

Un type route

```
Inductive route : Type :=
| departementale : route
| nationale : route
| autoroute : route.
```

Type algébrique de données (1/2)

- Un type défini par les différents cas possibles, les *constructeurs*
- Les constructeurs sont des objets à part entière
- Existe en OCaml, $\approx$ type enum en C

Un type route

```
Inductive route : Type :=
| departementale : route
| nationale : route
| autoroute : route.
```

```
Check route. (* route : Type *)
Check autoroute.
(* autoroute : route *)
```

Type algébrique de données (1/2)

- Un type défini par les différents cas possibles, les *constructeurs*
- Les constructeurs sont des objets à part entière
- Existe en OCaml, ≈ type enum en C

Un type route

```
Inductive route : Type :=
| departementale : route
| nationale : route
| autoroute : route.
```

```
Check route. (* route : Type *)
Check autoroute.
(* autoroute : route *)
```

- Pour examiner un terme dont le type est inductif, on utilise le pattern-matching (filtrage)

```
Definition agrandir (r : route) := match r with
  | departementale ⇒ nationale
  | nationale ⇒ autoroute
  | autoroute ⇒ autoroute
  end.
```

Type algébrique de données (1/2)

- Un type défini par les différents cas possibles, les *constructeurs*
- Les constructeurs sont des objets à part entière
- Existe en OCaml, ≈ type enum en C

Un type route

```
Inductive route : Type :=
| departementale : route
| nationale : route
| autoroute : route.
```

```
Check route. (* route : Type *)
Check autoroute.
(* autoroute : route *)
```

- Pour examiner un terme dont le type est inductif, on utilise le pattern-matching (filtrage)

```
Definition agrandir (r : route) := match r with
  | departementale ⇒ nationale
  | nationale ⇒ autoroute
  | autoroute ⇒ autoroute
  end.
```

```
Check agrandir.
(*
   agrandir : route → route
*)
```

Type algébrique de données (2/2)

Les constructeurs peuvent prendre des arguments.

```
Inductive terrain : Type :=
| t_terre : terrain
| t_route : route → terrain
| t_batiment : terrain.
Check (t_route nationale). (* terrain *)

Definition agrandir_terrain (t : terrain) := match t with
  | t_terre ⇒ t_batiment
  | t_route r ⇒ t_route (agrandir r)
  | t_batiment ⇒ t_batiment
  end.
Check agrandir_terrain. (* : terrain → terrain *)
```

Contraintes sur le pattern-matching

Exhaustif et non-redondant.

Des types plus communs (1/2)

Booléens - le type bool est prédéfini de la façon suivante :

```
Inductive bool : Type :=
| true : bool
| false : bool.

Check true. (* true : bool *)
Check false. (* false : bool *)

Definition negb b :=
  match b with
  | true ⇒ false
  | false ⇒ true
  end.
Check negb. (* negb : bool → bool *)
```

Des types plus communs (2/2)

On peut définir des types récursifs.

Des types plus communs (2/2)

On peut définir des types récursifs.

Les entiers (de Peano)

```
Inductive nat : Type :=
| O : nat (* lettre O, interprété comme 0 *)
| S : nat → nat (* successeur *).
Check O. (* O : nat *)
Check S. (* S : nat → nat *)
```

Des types plus communs (2/2)

On peut définir des types récursifs.

Les entiers (de Peano)

```
Inductive nat : Type :=
| O : nat (* lettre O, interprété comme 0 *)
| S : nat → nat (* successeur *).
Check O. (* O : nat *)
Check S. (* S : nat → nat *)
```

Les listes de booléens

```
Inductive list_bool : Type :=
| nil : list_bool
| cons : bool → list_bool → list_bool.
Check nil. (* nil : list_bool *)
Check cons. (* cons : bool → list_bool → list_bool *)
```

Fonctions récursives (1/2)

On peut définir des fonctions récursives sur des types récursifs.

```
Fixpoint plus n m :=
  match n with
  | O ⇒ m
  | S n' ⇒ S (plus n' m)
  end.
(*
   plus is defined
   plus is recursively defined (decreasing on 1st argument)
*)
```

Fonctions récursives (1/2)

On peut définir des fonctions récursives sur des types récursifs.

```
Fixpoint plus n m :=
  match n with
  | O ⇒ m
  | S n' ⇒ S (plus n' m)
  end.
(*
   plus is defined
   plus is recursively defined (decreasing on 1st argument)
*)
```

Terminaison

La fonction doit terminer !

Cas simple détecté automatiquement par Coq : récurrence structurelle

Fonctions récursives (2/2)

Concaténation de listes de booléens

```
Fixpoint app l1 l2 :=
  match l1 with
  | nil ⇒ l2
  | cons a l ⇒ cons a (app l l2)
  end.
(* app is recursively defined (decreasing on 1st argument) *)
Check app. (* list_bool → list_bool → list_bool *)
```

Fonctions récursives (2/2)

Concaténation de listes de booléens

```
Fixpoint app l1 l2 :=
  match l1 with
  | nil ⇒ l2
  | cons a l ⇒ cons a (app l l2)
  end.
(* app is recursively defined (decreasing on 1st argument) *)
Check app. (* list_bool → list_bool → list_bool *)
```

Définition incorrecte

```
Fixpoint app l1 l2 :=
  match l1 with
  | nil ⇒ l2
  | cons a _ ⇒ cons a (app l1 l2)
  end.
(* Error: Cannot guess decreasing argument of fix. *)
```

Autres remarques sur les fonctions

- Fonctions comme valeurs de première classe

```
Definition apply_neg (f:_→ _→ bool) b1 b2 :=
  f (negb b1) (negb b2).
Check apply_neg. (* apply_neg : (bool → bool → bool) →
  bool → bool → bool *)
```

- Applications partielles

```
Definition nor := apply_neg andb.
Check nor. (* nor : bool → bool → bool *)
```

- Fonctions anonymes

```
Definition idb (b:bool) := b.
Definition idb : bool → bool := fun b ⇒ b.
```

Type polymorphique

Les types peuvent être paramétrés par d'autres types.

Exemple le plus classique : les listes.

```
Inductive list (A:Type) : Type :=
| nil : list A
| cons : A → list A → list A.
Check list. (* list : Type → Type *)
Check nil. (* nil : forall A : Type, list A *)
Check cons. (* cons : forall A : Type, A → list A → list A *)
```

Fonctions polymorphiques

Les fonctions peuvent aussi être paramétrés par des types.
Un exemple un peu artificiel.

```
Definition id (A:Type) (x:A) := x.
Check id. (* id : forall A : Type, A → A *)
```

Un exemple sur les listes.

```
Fixpoint length (A:Type) (l:list A) :=
  match l with
  | nil _ ⇒ 0
  | cons _ _ t ⇒ S (length A t)
  end.
```

Retour sur un ancien exemple.

```
Definition apply_neg (f:_→ _→ bool) b1 b2 :=
  f (negb b1) (negb b2).
```

Pourquoi se limiter à `bool` ?

Fonctions polymorphiques

Les fonctions peuvent aussi être paramétrés par des types.

Un exemple un peu artificiel.

```
Definition id (A:Type) (x:A) := x.
Check id. (* id : forall A : Type, A → A *)
```

Un exemple sur les listes.

```
Fixpoint length (A:Type) (l:list A) :=
  match l with
  | nil _ ⇒ 0
  | cons _ _ t ⇒ S (length A t)
  end.
```

Retour sur un ancien exemple.

```
Definition apply_neg (A:Type) (f:_→ _→ A) b1 b2 :=
  f (negb b1) (negb b2).
Check apply_neg. (* apply_neg : forall A : Type,
  (bool → bool → A) → bool → bool → A *)
```

Lisibilité

On peut améliorer la lisibilité des termes manipulés.

- Avec des paramètres implicites
  Si un argument peut toujours être inféré à partir d'un autre, on peut le rendre implicite.

```
Definition apply_neg {A:Type} (f:_→ _→ A) b1 b2 :=
  f (negb b1) (negb b2).
Definition nor := apply_neg andb.
(* au lieu de apply_neg bool andb
            ou apply_neg _ andb *)
```

- Avec des notations

```
Notation "x = y" := ...
Notation "[ ]" := nil.
```

Plan

Un Type

Précédemment, tous nos objets étaient dans `Type`. Les énoncés sont dans un autre type,

`Prop`

Énoncé d'un théorème

- Un premier théorème

```
Theorem negb_true_false : negb true = false.
```

Énoncé d'un théorème

- Un premier théorème

  ```
  Theorem negb_true_false : negb true = false.
  ```

- Plus intéressant ?

  ```
  Theorem app_nil_l : forall {A : Type} (l : list A), app [] l = l.
  ```

Énoncé d'un théorème

- Un premier théorème

  ```
  Theorem negb_true_false : negb true = false.
  ```

- Plus intéressant ?

  ```
  Theorem app_nil_l : forall {A : Type} (l : list A), app [] l = l.
  ```

- Mais peut-on exprimer autre chose que des égalités ?

Constantes, connecteurs, quantificateurs

- Constantes : `True`, `False`
- Connecteurs : $\sim$ (non), $\wedge$ (et), $\vee$ (ou), $\rightarrow$ (implique)
- Quantificateurs : `forall`, `exists`

```
Check True. (* True : Prop *)
Check True → False. (* True → False : Prop *)
```

Constantes, connecteurs, quantificateurs

- Constantes : `True`, `False`
- Connecteurs : $\sim$ (non), $\wedge$ (et), $\vee$ (ou), $\rightarrow$ (implique)
- Quantificateurs : `forall`, `exists`

```
Check True. (* True : Prop *)
Check True → False. (* True → False : Prop *)
```

Logique intuitionniste

$p \rightarrow q \not\equiv \sim p \vee q$

Constantes, connecteurs, quantificateurs

- Constantes : `True`, `False`
- Connecteurs : $\sim$ (non), $\wedge$ (et), $\vee$ (ou), $\rightarrow$ (implique)
- Quantificateurs : `forall`, `exists`

```
Check True. (* True : Prop *)
Check True → False. (* True → False : Prop *)
```

Logique intuitionniste

$p \rightarrow q \not\equiv \sim p \vee q$

- Logique d'ordre supérieur : on peut quantifier sur des fonctions et des prédicats

```
Lemma bool_trivial : forall (P : bool → Prop) b, P b → P b.
```

Remarque importante

Comme pour les fonctions qui reçoivent leurs arguments un par un et non un couple, les énoncés à plusieurs hypothèses utilisent plusieurs implications, plutôt que la conjonction.

```
Lemma bool_ext : forall (P : bool → Prop), P true ∧ P false →
  forall b, P b.
```

s'écrira en fait

```
Lemma bool_ext : forall (P : bool → Prop), P true → P false →
  forall b, P b.
```

Les prédicats inductifs (1/2)

On a vu comment définir des types inductifs. De la même manière, on peut définir des prédicats inductifs (à la Prolog).

Les prédicats inductifs (1/2)

On a vu comment définir des types inductifs. De la même manière, on peut définir des prédicats inductifs (à la Prolog).

```
Inductive even : nat → Prop :=
| even_0 : even 0 (* 0 est pair *)
| even_SS : forall n, even n → even (S (S n)).
  (* si n est pair, n+2 est pair *)
```

Les prédicats inductifs (1/2)

On a vu comment définir des types inductifs. De la même manière, on peut définir des prédicats inductifs (à la Prolog).

```
Inductive even : nat → Prop :=
| even_0 : even 0 (* 0 est pair *)
| even_SS : forall n, even n → even (S (S n)).
  (* si n est pair, n+2 est pair *)
```

On peut alors exprimer d'autres théorèmes.

```
Theorem multiple_2_even : forall n, even (2 * n).
```

Les prédicats inductifs (2/2)

Autre exemple.

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

Les prédicats inductifs (2/2)

Autre exemple.

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

Lemma In_app : forall {A : Type} (x : A) (l1 l2 : list A),
  In x (app l1 l2) → In x l1 ∨ In x l2.
```

Les prédicats inductifs (2/2)

Autre exemple.

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

Lemma In_app : forall {A : Type} (x : A) (l1 l2 : list A),
  In x (app l1 l2) → In x l1 ∨ In x l2.
```

Paramètre et indice

Dans la famille de types `In A x l` :

- `A` est un paramètre, il est global aux constructeurs
- `x` et `l` sont des indices, ils peuvent être instanciés différemment dans chaque constructeur

Plan

Plan

Enjeux

Comment écrire la preuve d'un théorème ?

Enjeux

Comment écrire la preuve d'un théorème ?

- De manière interactive !

Enjeux

Comment écrire la preuve d'un théorème ?

- De manière interactive !
- En s'appuyant sur Ltac, un langage de tactiques permettant de transformer successivement le but actuel en un ou des buts plus simples à résoudre

Enjeux

Comment écrire la preuve d'un théorème ?

- De manière interactive !
- En s'appuyant sur Ltac, un langage de tactiques permettant de transformer successivement le but actuel en un ou des buts plus simples à résoudre
- Certaines tactiques de base ont une correspondance évidente dans le calcul des séquents (par ex. `intros`, `split`)

Plan

Exemple de preuve (1/2)

```
Lemma andb_prop : forall b1 b2,
andb b1 b2 = true → b1 = true ∧ b2 = true.
Proof.
...
Qed.
```

Exemple de preuve (2/2)

```
1 subgoal
______________________________________(1/1)
forall b1 b2 : bool, andb b1 b2 = true → b1 = true ∧ b2 = true
```

Exemple de preuve (2/2)

```
1 subgoal
______________________________________(1/1)
forall b1 b2 : bool, andb b1 b2 = true → b1 = true ∧ b2 = true

> intros b1 b2 H.
```

Exemple de preuve (2/2)

```
1 subgoal
______________________________________(1/1)
forall b1 b2 : bool, andb b1 b2 = true → b1 = true ∧ b2 = true

> intros b1 b2 H.

1 subgoal
b1, b2 : bool
H : andb b1 b2 = true
______________________________________(1/1)
b1 = true ∧ b2 = true
```

Exemple de preuve (2/2)

```
1 subgoal
b1, b2 : bool
H : andb b1 b2 = true
______________________________________(1/1)
b1 = true ∧ b2 = true
```

Exemple de preuve (2/2)

```
1 subgoal
b1, b2 : bool
H : andb b1 b2 = true
______________________________________(1/1)
b1 = true ∧ b2 = true

> split.
```

Exemple de preuve (2/2)

```
1 subgoal
b1, b2 : bool
H : andb b1 b2 = true
______________________________________(1/1)
b1 = true ∧ b2 = true

> split.

2 subgoals
b1, b2 : bool
H : andb b1 b2 = true
______________________________________(1/2)
b1 = true
______________________________________(2/2)
b2 = true
```

Exemple de preuve (2/2)

```
2 subgoals
b1, b2 : bool
H : andb b1 b2 = true
______________________________________(1/2)
b1 = true
______________________________________(2/2)
b2 = true
```

Exemple de preuve (2/2)

```
2 subgoals
b1, b2 : bool
H : andb b1 b2 = true
______________________________________(1/2)
b1 = true
______________________________________(2/2)
b2 = true

> destruct b1.
```

Exemple de preuve (2/2)

```
2 subgoals
b1, b2 : bool
H : andb b1 b2 = true
_____________________________________(1/2)
b1 = true
_____________________________________(2/2)
b2 = true

> destruct b1.

3 subgoals
b2 : bool
H : andb true b2 = true
_____________________________________(1/3)
true = true
_____________________________________(2/3)
false = true
                                     (3/3)
```

Exemple de preuve (2/2)

```
3 subgoals
b2 : bool
H : andb true b2 = true
_____________________________________(1/3)
true = true
_____________________________________(2/3)
false = true
_____________________________________(3/3)
b2 = true
```

Exemple de preuve (2/2)

```
3 subgoals
b2 : bool
H : andb true b2 = true
_____________________________________(1/3)
true = true
_____________________________________(2/3)
false = true
_____________________________________(3/3)
b2 = true

> reflexivity.
```

Exemple de preuve (2/2)

```
3 subgoals
b2 : bool
H : andb true b2 = true
______________________________________(1/3)
true = true
______________________________________(2/3)
false = true
______________________________________(3/3)
b2 = true

> reflexivity.

2 subgoals
b2 : bool
H : andb false b2 = true
______________________________________(1/2)
false = true
                                      (2/2)
```

Exemple de preuve (2/2)

```
2 subgoals
b2 : bool
H : andb false b2 = true
______________________________________(1/2)
false = true
______________________________________(2/2)
b2 = true
```

Exemple de preuve (2/2)

```
2 subgoals
b2 : bool
H : andb false b2 = true
______________________________________(1/2)
false = true
______________________________________(2/2)
b2 = true
```

```
simpl in H. discriminate H.
```

Exemple de preuve (2/2)

```
2 subgoals
b2 : bool
H : andb false b2 = true
______________________________________(1/2)
false = true
______________________________________(2/2)
b2 = true

simpl in H. discriminate H.

1 subgoal
b1, b2 : bool
H : andb b1 b2 = true
______________________________________(1/1)
b2 = true
```

Exemple de preuve (2/2)

```
1 subgoal
b1, b2 : bool
H : andb b1 b2 = true
______________________________________(1/1)
b2 = true
```

Exemple de preuve (2/2)

```
1 subgoal
b1, b2 : bool
H : andb b1 b2 = true
______________________________________(1/1)
b2 = true
```

```
destruct b2.
```

Exemple de preuve (2/2)

```
1 subgoal
b1, b2 : bool
H : andb b1 b2 = true
_____________________________________(1/1)
b2 = true

destruct b2.

2 subgoals
b1 : bool
H : andb b1 true = true
_____________________________________(1/2)
true = true
_____________________________________(2/2)
false = true
```

Exemple de preuve (2/2)

```
2 subgoals
b1 : bool
H : andb b1 true = true
______________________________________(1/2)
true = true
______________________________________(2/2)
false = true
```

Exemple de preuve (2/2)

```
2 subgoals
b1 : bool
H : andb b1 true = true
______________________________________(1/2)
true = true
______________________________________(2/2)
false = true

reflexivity.
```

Exemple de preuve (2/2)

```
2 subgoals
b1 : bool
H : andb b1 true = true
______________________________________(1/2)
true = true
______________________________________(2/2)
false = true

reflexivity.

1 subgoal
b1 : bool
H : andb b1 false = true
______________________________________(1/1)
false = true
```

Exemple de preuve (2/2)

```
1 subgoal
b1 : bool
H : andb b1 false = true
_____________________________________(1/1)
false = true
```

Exemple de preuve (2/2)

```
1 subgoal
b1 : bool
H : andb b1 false = true
_____________________________________(1/1)
false = true

destruct b1.
```

Exemple de preuve (2/2)

```
1 subgoal
b1 : bool
H : andb b1 false = true
_____________________________________(1/1)
false = true

destruct b1.

2 subgoals
H : andb true false = true
_____________________________________(1/2)
false = true
_____________________________________(2/2)
false = true
```

Exemple de preuve (2/2)

```
2 subgoals
H : andb true false = true
_____________________________________(1/2)
false = true
_____________________________________(2/2)
false = true
```

Exemple de preuve (2/2)

```
2 subgoals
H : andb true false = true
______________________________________(1/2)
false = true
______________________________________(2/2)
false = true
```

```
> simpl in H. discriminate H.
```

Exemple de preuve (2/2)

```
2 subgoals
H : andb true false = true
______________________________________(1/2)
false = true
______________________________________(2/2)
false = true

> simpl in H. discriminate H.

1 subgoal
H : andb false false = true
______________________________________(1/1)
false = true
```

Exemple de preuve (2/2)

```
1 subgoal
H : andb false false = true
_____________________________________(1/1)
false = true
```

Exemple de preuve (2/2)

```
1 subgoal
H : andb false false = true
_____________________________________(1/1)
false = true

> simpl in H. discriminate H.
```

Exemple de preuve (2/2)

```
1 subgoal
H : andb false false = true
______________________________________(1/1)
false = true

> simpl in H. discriminate H.

No more subgoals.
```

Exemple de preuve (2/2)

```
No more subgoals.
```

Exemple de preuve (2/2)

```
No more subgoals.

> Qed.
```

Exemple de preuve (2/2)

```
No more subgoals.

> Qed.

andb_prop is defined
```

Exemple de preuve (2/2)

Résumé :

```
Lemma andb_prop : forall b1 b2,
  andb b1 b2 = true → b1 = true ∧ b2 = true.
Proof.
  intros b1 b2 H. (* introduit les hypothèses *)
  split. (* sépare le but en deux sous-buts *)
  - destruct b1. (* raisonnement par cas *)
    + reflexivity. (* true = true *)
    + simpl in H. discriminate H. (* false <> true *)
  - destruct b2. (* raisonnement par cas *)
    + reflexivity. (* true = true *)
    + destruct b1. (* raisonnement par cas *)
      * simpl in H. discriminate H. (* false <> true *)
      * simpl in H. discriminate H. (* false <> true *)
Qed.
```

Exemple de preuve (2/2)

Petite optimisation :

```
Lemma andb_prop : forall b1 b2,
  andb b1 b2 = true → b1 = true ∧ b2 = true.
Proof.
  intros b1 b2 H. (* introduit les hypothèses *)
  split. (* sépare le but en deux sous-buts *)
  - destruct b1. (* raisonnement par cas *)
    + reflexivity. (* true = true *)
    + simpl in H. discriminate H. (* false <> true *)
  - destruct b2. (* raisonnement par cas *)
    + reflexivity. (* true = true *)
    + destruct b1; simpl in H; discriminate H.
Qed.
```

Autre exemple

```
Lemma even_n_or_Sn : forall n, even n ∨ even (S n).
```

Démo !

Remarque

Au moins pour certaines preuves qui n'utilisent que des tactiques correspondant à des règles de logique (intuitionniste), on peut construire un arbre de preuve.

```
Lemma test : forall P Q R,
  P ∧ Q ∨ P ∧ R → P.
Proof.
  intros.
  destruct H as [H0|H1].
  – destruct H0.
    assumption.
  – destruct H1.
    assumption.
Qed.
```

$$\dfrac{\dfrac{\dfrac{}{p, q \vdash p}\,\texttt{assumption}}{p \wedge q \vdash p}\,\texttt{destruct H0} \qquad \dfrac{\dfrac{}{p, r \vdash p}\,\texttt{assumption}}{p \wedge r \vdash p}\,\texttt{destruct H1}}{\dfrac{(p \wedge q) \vee (p \wedge r) \vdash p}{\vdash (p \wedge q) \vee (p \wedge r) \Rightarrow p}\,\texttt{intros}}\,\texttt{destruct H}$$

Plan

Remarques

- Ceci est une liste de tactiques fréquemment utilisées
- Plutôt bas niveau
- La classification n'est pas standard et plutôt hasardeuse

Tactiques pour manipuler les types inductifs

- `destruct x` pour une analyse par cas de `x`
- `discriminate` pour utiliser la disjonction des constructeurs
- `injection H` pour utiliser l'injection des constructeurs dans `H`
- `inversion H` $\simeq$ `destruct H`, mais plus pour les prédicats
- `induction H` $\simeq$ `destruct H` avec hypothèse d'induction
- `constructor` pour appliquer un constructeur du type de la conclusion

Pour manipuler la logique

- `intros` pour introduire les variables quantifiées universellement
- `exists x` pour donner la valeur témoin attendue
- `split` pour détruire un but en plusieurs sous-buts
- `left`, `right` pour choisir une branche d'une disjonction
- `reflexivity` pour utiliser la réflexivité de l'égalité
- `symmetry` pour utiliser la symétrie de l'égalité
- `contradiction` si les hypothèses contiennent `False` ou `P` et `~P`

Autres

- `exact H` lorsque l'hypothèse H et le but actuel sont identiques
- `assumption` pour chercher dans les hypothèses
- `apply H` pour appliquer `H` dont la conséquence est la conclusion actuelle
- `rewrite` [$\leftarrow$] `H` pour récrire en utilisant l'égalité `H`
- `replace x with y` pour remplacer `x` par `y` en prouvant `x=y`
- `subst x` pour éliminer `x` en utilisant les égalités disponibles
- `assert H as Ha` pour introduire un lemme intermédiaire `H` pendant la preuve
- `specialize (H x)` pour appliquer partiellement une hypothèse
- `pose proof lem as H` pour ajouter un théorème aux hypothèses
- `simpl` pour simplifier des termes
- `unfold f` pour remplacer `f` par sa définition

Plan

Principales caractéristiques de Coq

- Un langage fonctionnel pur, Gallina
- Un mécanisme pour définir des types (et des prédicats) inductifs très puissant
- Ltac pour écrire des preuves

Définition donnée par le site web

Coq implements a program specification and mathematical higher-level language called Gallina that is based on an expressive formal language called the Calculus of Inductive Constructions that itself combines both a higher-order logic and a richly-typed functional programming language. Through a vernacular language of commands, Coq allows :

- *to define functions or predicates, that can be evaluated efficiently ;*
- *to state mathematical theorems and software specifications ;*
- *to interactively develop formal proofs of these theorems ;*
- *to machine-check these proofs by a relatively small certification "kernel" ;*
- *to extract certified programs to languages like Objective Caml, Haskell or Scheme.*

As a proof development system, Coq provides interactive proof methods, decision and semi-decision algorithms, and a tactic language for letting the user define its own proof methods. Connection with external computer algebra system or theorem provers is available.

As a platform for the formalization of mathematics or the development of programs, Coq provides support for high-level notations, implicit contents and various other useful kinds of macros.

Pour aller plus loin

- Idéal pour commencer (progressif, très reconnu et très utilisé)
  Software Foundations, Benjamin Pierce et al.

  `https://softwarefoundations.cis.upenn.edu/`
- Plus poussé
  Certified Programming with Dependent Types, Adam Chlipala

  `http://adam.chlipala.net/cpdt/`
- Coq' Art, Yves Bertot et Pierre Castéran

  `https://www.labri.fr/perso/casteran/CoqArt/`
- La documentation de Coq

  `https://coq.inria.fr/refman/`

  `https://coq.inria.fr/refman/command-index.html`

  `https://coq.inria.fr/refman/tactic-index.html`