# Petit guide de survie en Coq

**Alexandre Miquel**

*La documentation de la version courante du système (V8.0) se trouve en ligne sur:*

`http://coq.inria.fr/doc/main.html`

## Table des matières


1 Généralités
- 1.1 Démarrage du système
- 1.2 Commandes et tactiques
- 1.3 Élements de syntaxe
- 1.4 Scripts de preuve

2 Les commandes essentielles
- 2.1 `Abort`
- 2.2 `Axiom, Conjecture`
- 2.3 `Check`
- 2.4 `Definition`
- 2.5 `Extraction`
- 2.6 `Parameter, Parameters`
- 2.7 `Print`
- 2.8 `Proof`
- 2.9 `Theorem, Lemma, Fact, Remark`
- 2.10 `Qed`
- 2.11 `Quit`
- 2.12 `Restart`
- 2.13 `Show`
- 2.14 `Undo`

3 Les tactiques essentielles
- 3.1 `apply`
- 3.2 `assert`
- 3.3 `assumption`
- 3.4 `change`
- 3.5 `destruct`
- 3.6 `elim`
- 3.7 `exact`
- 3.8 `exists`
- 3.9 `induction`
- 3.10 `intro, intros`
- 3.11 `reflexivity`
- 3.12 `rewrite`
- 3.13 `simpl`
- 3.14 `split`
- 3.15 `symmetry`
- 3.16 `transitivity`
- 3.17 `unfold`
- 3.18 Composition de tactiques

# 1 Généralités

## 1.1 Démarrage du système

Le sytème interactif Coq (le *toplevel*) est lancé à l'aide de la commande `coqtop`, dont l'invite "`Coq <`" signale que le système attend une commande.

Il est possible de lancer Coq avec l'interface graphique CoqIDE à l'aide de la commande `coqide`.

**Note:** Pour afficher les connecteurs et les quantificateurs à l'aide de caractères utf-8 sous CoqIDE, il peut être nécessaire d'inclure dans le fichier les commandes:

```
Add LoadPath "/usr/local/lib/coq/ide".
Require Export utf8.
```

## 1.2 Commandes et tactiques

#### Structure des commandes

En Coq, une commande est formée d'un nom de commande (commençant par une majuscule), suivie de zéro, un ou plusieurs arguments, et terminée par un point. Par exemple:

```
Quit.
Check 2 + 2 = 5.
Print nat.
Definition id := fun (A : Set) (x : A) => x.
Eval compute id nat 7.
```

Comme en Caml, les commentaires s'écrivent entre `(*` et `*)`.

#### Passage du mode commande au mode preuve

L'utilisateur déclare son intention d'effectuer une preuve avec une commande de la forme

```
Lemma and_commut :
  forall A B : Prop, A /\ B <-> B /\ A.
```

Cette commande donne un nom (`and_commut`) au lemme, qui permettra de le référencer par la suite. (On peut remplacer `Lemma` par `Theorem`, `Fact` ou `Remark`.) Une fois cette déclaration effectuée, le système passe en mode preuve (signalé par l'invite "`and_commut <`").

#### Sous-buts et tactiques

En mode preuve, le système affiche en permanence un ou plusieurs sous-buts (*subgoals*) qu'il s'agit de démontrer. Ces sous-buts sont essentiellement des séquents de la déduction naturelle écrits verticalement: les hypothèses (nommées) sont en haut, et la conclusion figure en bas sous un double trait. Dans la partie haute figurent également des déclarations de variables.

```
          H1 : forall x : T, A x -> B x
          H2 : forall x : T, B x -> C x
          x : T
          H : A x
          ============================
           C x

        barbara <
```

La preuve se fait à l'aide de *tactiques* (distinguées des commandes par une minuscule initiale), qui effectuent des transformations plus ou moins complexes sur le but courant. Les tactiques sont susceptibles d'engendrer de nouveaux sous-buts (correspondant aux prémisses), ou au contraire de faire disparaître le but courant (lorsque celui-ci est résolu).

La preuve est terminée lorsqu'il n'y a plus de sous-but à démontrer. On repasse alors au mode commande avec la commande `Qed`.[1]

## 1.3 Élements de syntaxe

La correspondance entre la syntaxe du calcul des prédicats du premier ordre et celle de Coq est récapitulée dans le tableau ci-dessous.

| | |
|---|---|
| $\top$ | `True` |
| $\bot$ | `False` |
| $\neg A$ | `~ A` |
| $A \wedge B$ | `A /\ B` |
| $A \vee B$ | `A \/ B` |
| $A \Rightarrow B$ | `A -> B` |
| $A \Leftrightarrow B$ | `A <-> B` |
| $\forall x \in D\ A(x)$ | `forall x : D, A x` |
| $\exists x \in D\ A(x)$ | `exists x : D, A x` |
| | |

(On notera qu'en Coq, toutes les quantifications sont bornées par un type de données.)

Les tactiques qui implémentent les règles de la déduction naturelle ainsi que certaines règles gauches du calcul des séquents sont les suivantes:

| **Tactique** | **Règle(s) de déduction** |
|---|---|
| `assumption` | Axiome |
| `intro`, `intros` | $\Rightarrow$-intro, $\forall$-intro, $\neg$-intro |
| `apply` | $\Rightarrow$-élim, $\forall$-élim, $\neg$-élim |
| `split` | $\wedge$-intro, $\top$-intro |
| `left`, `right` | $\vee$-intro$_1$, $\vee$-intro$_2$ |
| `exists` | $\exists$-intro |
| `destruct` | $\wedge$-gauche, $\vee$-gauche, $\bot$-gauche, $\top$-gauche, $\exists$-gauche |

## 1.4 Scripts de preuve

Il est d'usage de regrouper les preuves dans des *scripts* de preuves (stockés dans des fichiers portant l'extension "`.v`"[2]) comme le suivant:

```
Lemma and_commut :
  forall A B : Prop, A /\ B <-> B /\ A.

Proof.
intros; split; intros.
 destruct H; split; assumption.
 destruct H; split; assumption.
Qed.
```

(La commande `Proof` ne fait rien; elle ne sert qu'à rendre les scripts de preuve plus jolis.)

Ces preuves peuvent être ensuite rejouées avec la commande `coqc mes_preuves.v`, qui produit un fichier `mes_preuves.vo` (version compilée des preuves) en cas de succès — c'est-à-dire lorsque les preuves qu'il contient sont correctes.

# 2 Les commandes essentielles

## 2.1 `Abort`

La commande `Abort` abandonne la preuve en cours et repasse en mode commande.

## 2.2 `Axiom, Conjecture`

La commande `Axiom id : A` ajoute à l'environnement courant un nouvel axiome désigné par l'identificateur `id`, et dont l'énoncé est la proposition `A`. Attention: l'ajout d'un axiome peut mettre en danger la cohérence logique du système.

La commande `Conjecture` est synonyme d'`Axiom`.

## 2.3 `Check`

La commande `Check M` affiche le type du terme `M`, tel qu'il est inféré par le système. (La commande échoue si `M` est mal typé.) Cette commande est souvent utilisée pour vérifier si une constante est déclarée dans l'environnement courant, et pour retrouver son type.

## 2.4 `Definition`

La commande `Definition id := M` définit dans l'environnement courant une nouvelle constante `id` égale à `M`, sous réserve que le terme `M` est bien typé. La constante `id` hérite du type de `M`, tel qu'il est inféré par le système.

**Variantes:**

1. `Definition id : T := M` définit dans l'environnement courant une nouvelle constante `id` de type `T` égale à `M`, sous réserve que le type du terme `M` (tel qu'il est inféré par le système) est convertible avec le type `T`.
2. `Definition id (x_1 : T_n) ... (x_n : T_n) := M` est synonyme de

   ```
   Definition id := fun (x_1 : T_n) ... (x_n : T_n) => M
   ```

3. `Definition id (x_1 : T_n) ... (x_n : T_n) : U := M` est synonyme de

   ```
   Definition id : forall (x_1 : T_n) ... (x_n : T_n), U :=
     fun (x_1 : T_n) ... (x_n : T_n) => M
   ```

## 2.5 `Extraction`

La commande `Extraction id` extrait un programme Caml à partir de la constante Coq `id`.

**Variantes:**

1. `Recursive Extraction id1 ... idn` extrait un programme Caml à partir des constantes `id1`... `idn` ainsi que toutes les constantes dont elles dépendent.
2. `Extraction "fichier" id1 ... idn`. Envoie le programme extrait dans le fichier `fichier`.
3. `Recursive Extraction "fichier" id1 ... idn`. Idem.

## 2.6 `Parameter, Parameters`

La commande `Parameter id : T` ajoute à l'environnement courant une nouvelle constante `id` non définie de type `T`. Attention: cette commande peut rendre le système incohérent, typiquement dans le cas où le type `T` est vide.

On peut utiliser la commande `Parameter` pour déclarer plusieurs constantes:

```
Parameter id_1 ... id_n : T
Parameter (id_1,1 ... id_1,n : T_1) ... (id_k,1 ... id_k,n_k : T_k)
```

La commande `Parameters` est synonyme de `Parameter`.

## 2.7 `Print`

La commande `Print id` affiche la définition de l'identificateur `id` dans l'environnement courant. Cette commande n'a un sens que si l'identificateur `id` a été défini au préalable dans l'environnement, typiquement à l'aide d'une des commandes `Definition`, `Fixpoint` ou `Inductive`.

## 2.8 `Proof`

La commande `Proof` ne fait rien, sinon enjoliver les scripts de preuve où il est d'usage de l'invoquer avant de commencer une démonstration.

## 2.9 `Theorem, Lemma, Fact, Remark`

La commande `Theorem id : A` fait passer le système du mode commande au mode preuve, la proposition `A` devenant alors le but courant. Une fois la démonstration terminée, la preuve est stockée dans l'identificateur `id`, qui peut être utilisé plus tard pour faire appel au théorème.

Les commandes `Lemma`, `Fact` et `Remark` sont des synonymes de `Theorem`.

## 2.10 `Qed`

La commande `Qed` permet de sortir du mode preuve et de revenir au mode commande (uniquement lorsque tous les sous-buts ont été résolus).

## 2.11 `Quit`

La commande `Quit` met fin à la session Coq.

### 2.12 `Restart`

La commande `Restart` abandonne la preuve en cours, et relance la machine de preuve sur le but initial.

### 2.13 `Show`

La commande `Show` affiche le but courant. La commande `Show n` (où `n` est un entier) affiche le but numéro `n`.

### 2.14 `Undo`

La commande `Undo` annule l'effet de la dernière commande de tactique.

## 3 Les tactiques essentielles

### 3.1 `apply`

La tactique `apply` (où `H` est le nom d'une hypothèse du contexte) applique une combinaison des règles ⇒-élim, ¬-élim et ∀-élim suivant la forme de l'hypothèse désignée par `H`.

1. Si l'hypothèse désignée par `H` est de la forme `A -> B` et si le but courant est `B` (ou toute autre proposition convertible), alors la tactique `apply H` remplace le but courant par `A`.
2. Si l'hypothèse désignée par `H` est de la forme `~ A` et si le but courant est `False` (ou toute autre proposition convertible), alors la tactique `apply H` remplace le but courant par `A`.
3. Si l'hypothèse désignée par `H` est de la forme `forall x : T, A x` et si le but courant est `B`, alors la tactique `apply H` essaie d'unifier la proposition `A x` avec `B` (en trouvant la valeur de `x` qui convient), et résout le but courant.

En pratique, la tactique `apply H` enchaîne ces opérations. Par exemple, si `H` désigne l'hypothèse

```
H : forall x y : nat, P x -> Q y -> R x y
```

et si le but est `R 2 3`, alors `apply H` remplace le but courant par les deux sous-buts `P 2` et `Q 3`.

Dans la mesure où l'algorithme d'unification utilisé par `apply` est incomplet, la tactique `apply H` peut échouer à trouver une valeur correcte pour tout ou partie des variables quantifiées universellement dans la proposition désignée par `H`. Dans ce cas, il est préférable d'utiliser la variante

```
apply H with (x1 := M1) ... (xn := Mn)
```

qui instancie explicitement les variables `x1`... `xn` avec les termes `M1`... `Mn`, respectivement.

### 3.2 `assert`

La tactique `assert A` introduit une coupure dans la démonstration du but courant `B` en remplaçant celui-ci par deux sous-buts: 1. la proposition `A` dans le contexte courant, et 2. la proposition `B` dans le contexte courant étendu avec l'hypothèse `A`.

**Variante:**

La tactique `assert H : A` donne explictement le nom de l'hypothèse ajoutée au contexte courant dans le deuxième sous-but engendré.

## 3.3 `assumption`

Cette tactique implémente la règle d'axiome de la déduction naturelle. Elle résout le but courant lorsque celui-ci figure parmi les hypothèses.

## 3.4 `change`

La tactique `change A` remplace le but courant par la proposition `A`, sous réserve que celle-ci est convertible (au sens des règles de calcul du système) avec le but courant.

**Variantes:**

`change A in H` fait la même chose, mais dans l'hypothèse désignée par `H` plutôt que dans le but courant. La proposition `A` doit être convertible avec la proposition désignée par l'hypothèse `H`.

## 3.5 `destruct`

La tactique `destruct H` (où `H` est le nom d'une hypothèse du contexte) applique parmi les règles $\wedge$-gauche, $\Rightarrow$-gauche, $\top$-gauche, $\bot$-gauche et $\exists$-gauche du calcul des séquents celle qui correspond au connecteur (ou au quantificateur) principal de l'hypothèse désignée par `H`.

1. Dans le cas où l'hypothèse est de la forme `A /\ B`, la tactique `destruct H` remplace dans le contexte courant cette hypothèse par deux nouvelles hypothèses `A` et `B`.
2. Dans le cas où l'hypothèse est de la forme `A \/ B`, la tactique `destruct H` remplace le sous-but courant par deux nouveaux sous-buts, dans lesquels l'hypothèse `A \/ B` est remplacée par l'hypothèse `A` et par l'hypothèse `B` respectivement.
3. Dans le cas où l'hypothèse est `True`, la tactique `destruct H` fait disparaître cette hypothèse.
4. Dans le cas où l'hypothèse est `False`, la tactique `destruct H` résout le but courant.
5. Dans le cas où l'hypothèse est de la forme `exists x : T, A x`, la tactique `destruct H` introduit dans le contexte une déclaration de la forme `x : T` et remplace l'hypothèse désignée par `H` par une nouvelle hypothèse de la forme `A x`. Au cours de cette opération, il se peut que la variable `x` soit renommée pour éviter une collision avec un autre objet déclaré avec le même nom.

**Variantes:**

1. `destruct H as (H1, H2)`, dans le cas où `H` désigne une hypothèse de la forme `A /\ B`, donne explicitement le nom des deux hypothèses qui remplacent l'hypothèse désignée par `H` dans le sous-but courant.
2. `destruct H as [H1 H2]`, dans le cas où `H` désigne une hypothèse de la forme `A \/ B`, donne explicitement les noms des hypothèses qui remplacent l'hypothèse désignée par `H` dans chacun des deux sous-buts créés.
3. `destruct H as (x0, H0)`, dans le cas où `H` est une hypothèse de la forme `exists x : T, A x`, donne explicitement le nom du témoin et de l'hypothèse qui remplacent l'hypothèse désignée par `H` dans le sous-but courant.

## 3.6 `elim`

La tactique `elim H` (où `H` est le nom d'une hypothèse du contexte) applique parmi les règles ∧-élim, ⇒-élim, ⊤-élim, ⊥-élim et ∃-élim de la déduction naturelle celle qui correspond au connecteur (ou au quantificateur) principal de l'hypothèse désignée par `H`. (Voir la documentation pour plus de détails.)

En général, on lui préfère la tactique `destruct H`, qui est plus facile d'utilisation.

## 3.7 `exact`

La tactique `exact M` (où `M` est un terme de preuve de CCI) résout le but courant à l'aide du terme de preuve donné en argument. C'est la tactique de plus bas niveau du système, puisqu'elle se contente d'appeler le vérificateur de types/preuves, qui constitue le noyau de Coq.

Comme toute preuve en Coq est de manière ultime un terme de CCI (construit incrémentalement à l'aide du système de tactiques), la tactique `exact` est complète, et peut résoudre n'importe quel but. Cependant, on évite généralement d'y avoir recours dans la mesure où les termes de preuve de CCI tiennent rarement dans la marge...

## 3.8 `exists`

Cette tactique implémente la règle ∃-intro de la déduction naturelle. Lorsque le but est de la forme `exists x : T, A x`, la tactique `exists M` (où `M` est un terme de type `T` destiné à jouer le rôle du témoin) remplace le but courant par un but de la forme `A M`.

## 3.9 `induction`

La tactique `induction id` (où `id` est un identificateur déclaré le contexte) applique sur le but courant le principe de récurrence associé au type de `id`. Cette tactique n'est valable que dans le cas où le type de `id` est un type inductif défini auparavant avec la commande `Inductive`.

Par exemple, si le but courant est de la forme:

```
n : nat
============================
 P n
```

la tactique `induction n` remplace ce but par les deux sous buts

```
============================
 P 0
```

et

```
n : nat
IHn : P n
============================
 P (S n)
```

La tactique `induction id` fonctionne également lorsque l'identificateur `id` n'est pas déclaré dans le contexte, mais est quantifié universellement dans la conclusion. Dans ce cas, la tactique `induction id` commence par effectuer les introductions nécessaires avant d'appliquer le principe de récurrence correspondant.

## 3.10 `intro, intros`

Suivant la forme du but courant, la tactique `intro` applique la règle ⇒-intro, ¬-intro, ou ∀-intro.

1. Lorsque le but courant est de la forme `A -> B`, la tactique `intro` charge dans le contexte une hypothèse supplémentaire `A`, et remplace le but courant par `B`.
2. Lorsque le but courant est de la forme `~ A`, la tactique `intro` charge dans le contexte une hypothèse supplémentaire `A`, et remplace le but courant par `False`.
3. Lorsque le but courant est de la forme `forall x : T, A x`, la tactique `intro` charge dans le contexte une nouvelle déclaration de la forme `x : T` et remplace le but courant par `A x`. Au cours de cette opération, il se peut que la variable `x` soit renommée pour éviter une collision avec un autre objet déclaré avec le même nom.

**Variantes:**

1. `intro H` (ou `intro x`) donne explicitement le nom de l'hypothèse (ou de la variable) chargée dans le contexte.
2. `intros` applique la tactique `intro` autant de fois qu'il est possible de le faire.
3. `intros H1 ... Hn` applique `n` fois la tactique `intro`, en donnant explicitement les noms de chacune des hypothèses (ou des variables) déclarées dans le contexte.

## 3.11 `reflexivity`

Cette tactique résout tout but de la forme `M1 = M2`, où `M1` et `M2` sont deux termes convertibles au sens des règles de calcul de Coq, comme par exemple les termes `2 + 2` et `4`. En particulier, elle résout tous les buts de la forme `M = M`.

## 3.12 `rewrite`

La tactique `rewrite H` (où `H` désigne une hypothèse de la forme `M1 = M2`) remplace dans le but courant toutes les occurrences du terme `M1` par le terme `M2`.

**Variantes:**

1. `rewrite <- H` utilise l'égalité `M1 = M2` dans l'autre sens, en remplaçant `M2` par `M1`.
2. `rewrite H in H0` fait la même chose que `rewrite H`, mais dans l'hypothèse désignée par `H0` plutôt que dans le but courant.
3. `rewrite <- H in H0` fait la même chose que `rewrite <- H`, mais dans l'hypothèse désignée par `H0` plutôt que dans le but courant.

## 3.13 `simpl`

La tactique `simpl` réduit (au sens des règles de calcul du système) l'expression qui constitue le sous-but courant. Ce qui ne la simplifie pas toujours.

**Variantes:**

`simpl in H` fait la même chose, mais dans l'hypothèse désignée par `H` plutôt que dans le but courant.

## 3.14 `split`

Suivant la forme du but courant, cette tactique applique la règle ∧-intro ou la règle ⊤-intro.

1. Lorsque le but courant est de la forme `A /\ B`, la tactique `split` remplace le but courant par deux nouveaux sous-buts `A` et `B` (dans un contexte inchangé).
2. Lorsque le but courant est `True`, `split` le résout.

### 3.15 `symmetry`

La tactique `symmetry` remplace tout but courant de la forme `M1 = M2` par le but `M2 = M1`.

**Variante:**

`symmetry in H` fait la même chose dans une hypothèse `H` (de la forme `M1 = M2`).

### 3.16 `transitivity`

La tactique `transitivity M` remplace tout but courant de la forme `M1 = M2` par deux sous-buts de la forme `M1 = M` et `M = M2`. (Les trois termes `M`, `M1` et `M2` doivent avoir le même type.)

### 3.17 `unfold`

La tactique `unfold id` remplace dans le but courant toutes les occurrences de l'identificateur `id` par le corps de sa définition dans l'environnement courant.

**Variante:**

`unfold id in H` fait la même chose dans une hypothèse `H`.

## 3.18 Composition de tactiques

Il est possible de composer deux tactiques `tac1` et `tac2` à l'aide de l'opérateur "`;`" (point-virgule). La tactique `tac1; tac2` ainsi définie applique d'abord la tactique `tac1` sur le but courant, puis la tactique `tac2` sur chacun des sous-buts engendrés par `tac1`. (En particulier, lorsque la tactique `tac1` résout le but courant, la tactique `tac2` est ignorée.) La construction `tac1; tac2` n'est donc pas équivalente à la suite "`tac1. tac2.`", sauf dans le cas où la tactique `tac1` engendre 1 sous-but exactement.

**Variante:**

`tac0; [tac1 | ... | tacn]` applique la tactique `tac0` sur le but courant, puis les tactiques `tac1`, ..., `tacn` sur les `n` sous-buts engendrés par la tactique `tac0`. Cette tactique échoue si le nombre de sous-buts engendrés par `tac0` est différent de `n`.

---

1 *Quod erat demonstrandum*, le "CQFD" latin.

2

“`.v`” comme “vernaculaire”.

---

*Ce document a été traduit de $L^AT_EX$ par HEVEA*