Guide Coq

L'assistant de preuve Coq est un outil de preuve interactive. Son but est de guider l'utilisateur pas à pas vers la réalisation d'une preuve dans la calcul des séquents, plus précisement dans le système LJ. Ce petit guide est là pour vous aider à débuter avec Coq. Il vous servira également de mémo pour retrouver comment vous sevir de vos tactiques préferées !

Table des matières

1 Premiers pas avec Coq

1.1 CoqIDE

Ce guide se base sur l'environnement de développement CoqIDE. Il est normalement installé sur les machines de la fac. Si vous souhaitez l'installer sur votre machine personnelle, vous pouvez suivre la documentation sur les liens suivants :

— `https://coq.inria.fr/`

— `https://coq.inria.fr/refman/practical-tools/coqide.html`

— `https://opam.ocaml.org/packages/coqide/`

D'autres IDE sont également disponibles (`https://coq.inria.fr/user-interfaces.html`), tels que VSCoq pour VSCode ou Codium par exemple.

La figure 1 montre l'interface de CoqIDE. La fenêtre est séparée en trois parties : le script (1), l'obligation de preuve (2) et les messages (3). Lorsque vous voulez faire une preuve en Coq, vous devez lui fournir la propriété à prouver et lui donner les étapes de preuves nécessaire pour la prouver. Le rôle de coq est de vous aider à savoir quoi prouver et de s'assurer que tous les cas sont traités correctement.

1 - Le script : Le script est ce que vous écrivez. C'est ici que vous allez rentrer vos formules à prouver, vos preuves, vos définitions, tactiques, etc.

2 - L'obligation de preuve : L'obligation de preuve apparaît lorsque vous essayer de faire une preuve. Elle se séparer en deux parties : les hypothèses (au dessus du trait, ce qui est à gauche du $\vdash$ dans un séquent) et le but (sous le trait, ce qui est à droite du $\vdash$ dans un séquent). Un nombre (ici, 1/1) vous indique également combien de but vous devez prouver (ici, nous traitons le premier cas, sur un cas au total). Lorsque vous passez à l'étape suivante dans le script, l'obligation évolue en conséquence pour vous donner l'état actuel de la preuve.

3 - Les messages : C'est ici que s'inscrivent les messages d'erreurs, lors de l'application incorrecte d'une règle ou si un cas a été oublié par exemple.

Dans Coq, après avoir écrit un pas de preuve, il vous faut l'appliquer à l'obligation de preuve. C'est durant cette phrase d'application que Coq va vérifier si votre raisonnement est correct. Pour appliquer un pas de preuve, vous pouvez utiliser le raccourci `ctrl+flèche du bas`. Pour revenir au pas de preuve précédent, utiliser `ctrl+flèche du haut`.

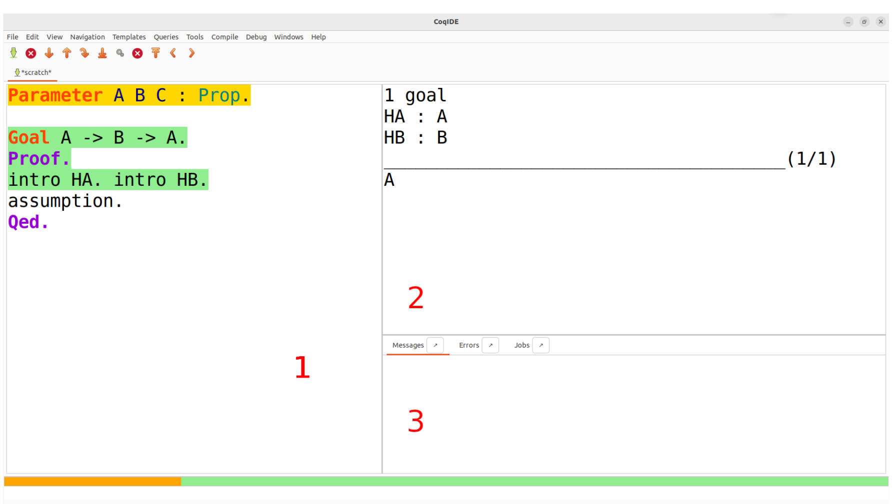

FIGURE 1 – Interface de `CoqIDE`

1.2 Syntaxe et structure d'un script **Coq**

Coq est composé de plusieurs éléments. Tout d'abord, la déclarations des éléments que l'on va manipuler, à savoir les paramètres (propositions, types des éléments) et les axiomes. Ensuite, la partie preuve en elle-même, composée d'une partie déclaration et d'une partie application de tactiques, c'est à dire, les actions que vous pouvez effectuer sur les formules logiques. Coq vous offre également la possibilité de définir des fonctions et des relations. Cette partie vous présente simplement les éléments et leur syntaxe, une explication détaillée de chaque cas est fournie plus loin dans ce document.

— **Commentaires** : vous pouvez ajouter des commentaires dans votre code Coq comme ceci :

```
(* Un commentaire en Coq *)
```

— **Point** : chaque expression doit finir par un point `.` et chaque point doit être suivi d'un espace ou d'un saut de ligne.

```
Goal A -> A -> A.
intro. intro.
```

— **Indentation** : vous pouvez indenter votre code à l'aide de symboles tels que `*`, `-`, `+`, … afin d'en améliorer la lisibilité et de ne pas vous perdre lors du traitement de plusieurs branches.

```
...
split. (* Ou n'importe quelle instruction qui génère plusieurs branches *)
- (* Premier sous-but *)
    + (* sous-sous-but 1 *)
    + (* sous-sous-but 2 *)
- (* Second sous-but *)
```

— **Paramètres** : les paramètres représentent les éléments, les types ou les fonctions que vous allez utilisez dans vos preuves. Ces éléments se déclarent à l'aide du mot-clé `Parameter` ou `Parameters` en cas de définitions multiples.

```
Parameter A : Prop.          (* Un élément A de type Prop*)
Parameters B C : Prop.       (* Deux éléments B et C de type Prop*)
Parameter T : Set.           (* Un élément T qui est un type de termes *)
```

— **Variables** : de manière analogue aux paramètres, il est possible de déclarer les variables que vous allez utiliser dans votre code. Ces éléments se déclarent à l'aide du mot-clé `Variable` ou `Variables` en cas de définitions multiples.

```
Parameter T : Set.           (* Un élément T qui est un type de termes *)
Variable x : T.              (* Une variable x de type T *)
```

— **Axiomes** : un axiome est une propriété assumée vraie par Coq. Elle est définie grâce au mot-clé `Axiom`.

```
Parameter N : Set.                  (* Un élément N qui est un type de termes *)
Parameter o : N.                    (* Un élément o de type N *)
Parameter s : N -> N.               (* Une fonction s de N -> N *)
Parameter plus : N -> N -> N.       (* Une fonction  plus de (N -> N) -> N *)
Axiom ax : (plus x o) = x.          (* Un axiome ax indiquant que (plus x o) = x *)
```

— **Fonctions** : une fonction prend un élément ou un ensemble d'éléments et retourne un élément ou un ensemble d'éléments. Elle est définie grâce au mot-clé `Definition`, ou `Fixpoint` pour les fonctions définie de façon inductive.

```
Definition plus_un (n : nat) := n+1.

Fixpoint plus_un (n : nat) : nat :=
    match n with
    | O => 1
    | S n' => n' + 1
end.
```

— **Relations inductives** : une relation prend une pair d'éléments (possiblement des ensembles) et décrit le comportement que dois respecter la relation (pour le(s) cas de base et le(s) cas inductif(s)). Elle est définie grâce au mot-clé `Inductive`.

```
Inductive is_plus_un : nat -> nat -> Prop :=
    | is_plus_un_O : is_plus_un O 1
    | is_plus_un_S : forall n m, is_plus_un n m -> is_plus_un (S n) (S m)
.
```

— **Goal** : le mot-clé `Goal` précède la propriété que vous voulez prouver.

```
Parameter A : Prop.      (* Un élément A de type Prop *)
Goal A -> A.             (* Un but à prouver : A -> A *)
```

— **Proof** : le mot-clé `Proof` indique le début de la preuve.

```
Parameter A : Prop.      (* Un élément A de type Prop *)
Goal A -> A.             (* Un but à prouver : A -> A *)
Proof.                   (* Le début de la preuve *)
```

— **Qed** : le mot-clé `Qed` indique la fin de la preuve.

```
Parameter A : Prop.      (* Un élément A de type Prop *)
Goal A -> A.             (* Un but à prouver : A -> A *)
Proof.                   (* Le début de la preuve *)
...
Qed.                     (* La fin de la preuve *)
```

— **Abort** : le mot-clé `Abort` vous permet d'abandonner une preuve sans la terminer.

```
Parameter A : Prop.      (* Un élément A de type Prop *)
Goal A -> A.             (* Un but à prouver : A -> A *)
Proof.                   (* Le début de la preuve *)
...
Abort.                   (* Abandon de la preuve *)
```

— **Admitted** : le mot-clé `Admitted` vous permet d'admettre une propriété sans la prouver.

```
Parameter A : Prop.      (* Un élément A de type Prop *)
Goal A -> A.             (* Un but à prouver : A -> A *)
Proof.                   (* Le début de la preuve *)
...
Admitted.                (* La propriété est admise, sans preuve *)
```

— **Théoreme, lemmes** : les mots-clés `Theorem` et `Lemma` peuvent remplacer `Goal` si vous souhaitez sauvegarder la preuve pour une utilisation future.

```
Parameter A : Prop.              (* Un élément A de type Prop *)
Lemma a_imp_a : A -> A.          (* Un but à prouver : A -> A *)
Proof.                           (* Le début de la preuve *)
...
Qed.                             (* La fin de la preuve *)
```

Une ligne peut comporter plusieurs instruction, mais le retour à la ligne peut améliorer la lisibilité. Finalement, étant donné que les preuves Coq s'effectuent dans la calcul des séquents, les connecteurs et quantificateurs peuvent d'exprimer en Coq, comme l'indique le tableau suivant :

Papier	$\bot$	$\top$	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$	$\exists x.\ P$	$\forall x.\ P$
Coq	`False`	`True`	`~P`	`P /\ Q`	`P \/ Q`	`P -> Q`	`P <-> Q`	`exists x, P`	`forall x, P`

1.3 Types de base et théories

Coq possède plusieurs types de base, que vous pouvez manipuler avec des méthodes associées. Vous devez toujours déclarer les symboles que vous voulez utiliser avant de vous en servir, afin que Coq puisse les reconnaître. Certaines théories ne sont pas définies de base mas peuvent être implémentées va des axiomes.

1.3.1 Les propositions

Le type `Prop` permet de déclarer des symboles propositionnels. Ils permettent de représenter des formules et des propriétés en logique des propositions. Dans l'exemple suivant, nous créons un élément `A` de type `Prop`, puis indiquons à Coq que nous souhaitons prouver la formula $A \Rightarrow A$.

Script

```
Parameter A : Prop.

Goal A -> A.
```

Obligation de preuve

```
1 goal
________________(1/1)
A -> A
```

1.3.2 Les prédicats et les éléments

Le type `Set` permet de déclarer un type de termes. Combiner à des prédicats, ils permettent de représenter des formules et des propriétés en logique du premier ordre. Par exemple, la formula $\forall x.\ P(x) \Rightarrow \exists y.\ P(y)$ peut être représentée en Coq de la façon suivante :

Script

```
Parameter E : Set.
Parameter P : E -> Prop.

Goal forall x : E, P(x) -> exists y: E, P(y).
```

Obligation de preuve

```
1 goal
__________________________(1/1)
forall x : E, P x
    -> exists y : E, P y
```

1.3.3 Les entiers naturels

Les entiers naturels (0, 1, 2, ...) sont nativement définis dans Coq. Par exemple, la formule $\forall (x : \texttt{int}).\ x + 0 = x$ peut être représentée en Coq de la façon suivante :

Script

```
Goal forall x : nat, x + 0 = 1.
```

Obligation de preuve

```
1 goal
____________________________(1/1)
forall x : nat, x + 0 = 1
```

1.3.4 Les entiers de Péano

En réalité, dans Coq, les entiers sont représentés à la manière des enters de Péano (définis par le symbol de base O et l'opération successeur S). Ainsi, 2 est en réalité `S(S(O))`. On peut définir les axiomes de Péano de la manière suivante, ainsi que la formule $1 + 1 = 2$:

Script

```
Section Peano.

Parameter N : Set.
Parameter o : N.
Parameter s : N -> N.
Parameters plus mult : N -> N -> N.
Variables x y : N.
Axiom ax1 : ~ ((s x) = o).
Axiom ax2 : exists z, ~(x = o) -> (s z) = x.
Axiom ax3 : (s x) = (s y) -> x = y.
Axiom ax4 : (plus x o) = x.
Axiom ax5 : (plus x (s y)) = s (plus x y).
Axiom ax6 : (mult x o) = o.
Axiom ax7 : (mult x (s y)) = (plus (mult x y) x).

End Peano.

Goal (plus (s o) (s o)) = (s (s o)).
```

Obligation de preuve

```
1 goal
________________________(1/1)
plus (s o) (s o) = s (s o)
```

1.3.5 Isomorphisme de types

En Coq, la théorie équationnelle correspondant aux isomorphismes de types dans les CCC peut être implémentée grâce aux axioms suivants. Nous pouvons par exemple représenter en Coq la formula $\forall A, B, C.\ (A \times B = B \times C) \Rightarrow A = C$:

Script

```
Open Scope type_scope.
Section iso_axioms.

Variables A B C : Set.

Axiom Com : A * B = B * A.
Axiom Ass : A * (B * C) = A * B * C.
Axiom Cur : (A * B -> C) = (A -> B -> C).
Axiom Dis : (A -> B * C) = (A -> B) * (A -> C).
Axiom P_unit : A * unit = A.
Axiom AR_unit : (A -> unit) = unit.
Axiom AL_unit : (unit -> A) = A.

End iso_axioms.

Goal forall A B C: Set, (A * B = B * C) -> A = C.
```

Obligation de preuve

```
1 goal
_____________________(1/1)
forall A B C : Set,
A * B = B * C -> A = C
```

1.4 Vos premières preuves avec Coq !

1.4.1 Logique propositionnelle

Commençons par prouver que $A \Rightarrow A$. Il s'agit d'une preuve en logique propositionnelle. Dans cette preuve, nous déclarons une proposition A, et nous indiquons à Coq que nous souhaitons prouver $A \Rightarrow A$ grâce au mot-clé `Goal`. Par la suite, nous débutons la preuve grâce à l'instruction `Proof` et appliquons deux tactiques (`intro` et `assumption`, détaillées dans la prochaine section). Une fois arrivée à la fin de la preuve, nous indiquons à Coq que la preuve est finie grâce à l'instruction `Qed`.

Script

```
(* Ma première preuve Coq !*)
Parameter A : Prop.

Goal A -> A.
Proof.
    intro HA.
    assumption.
Qed.
```

Obligation de preuve

```
1 goal
__________________________________(1/1)
A -> A

######################

HA : A
__________________________________(1/1)
A

######################

No more goals.
```

1.4.2 Logique du premier ordre

Montrons à présent que $\forall x.\ P(x) \Rightarrow \exists y.\ P(y)$. Il s'agit d'une preuve en logique du premier ordre, nous devons donc déclarer un type E qui sera le type de tous les éléments de ma preuve, et un prédicat P qui portera sur des éléments de type E. Ces déclarations sont effectuées sur les deux premières lignes. Ensuite, nous indiquons la propriété à prouver, à savoir $\forall x.\ P(x) \Rightarrow \exists y.\ P(y)$. Nous explicitons donc que pour chaque élément x de type E, s la propriété $P(x)$ est vraie, alors il existe un élément y de type E tel que $P(y)$ est vrai. Nous appliquons ensuite trois tactiques (`intro`, `exists` et `assumption`, détaillées dans la prochaine section) et concluons la preuve avec `Qed`.

Script

```
Parameter E : Set.
Parameter P : E -> Prop.

Goal forall x : E, P(x) -> exists y: E, P(y).
Proof.
  intro x.
  intro HPx.
  exists x.
  assumption.
Qed.
```

Obligation de preuve

```
1 goal
______________________________(1/1)
forall x : E, P x ->
            exists y : E, P y

######################

x : E
______________________________(1/1)
P x -> exists y : E, P y

######################

x : E
HPx : P x
______________________________(1/1)
exists y : E, P y

######################

x : E
HPx : P x
______________________________(1/1)
P x

######################

No more goals.
```

1.4.3 Preuve avec des axiomes

Présentons maintenant une preuve qui utilise des axiomes. Dans cet exemple, nous définissons les axioms de l'arithmétique de Péano et les utilisons afin de prouver que $1 + 1 = 2$ (ou, avec l'arithmétique de Péano, que $S(O) + S(O) = S(S(0))$).

Script

```
Section Peano.

Parameter N : Set.
Parameter o : N.
Parameter s : N -> N.
Parameters plus mult : N -> N -> N.
Variables x y : N.
Axiom ax1 : ~ ((s x) = o).
Axiom ax2 : exists z, ~(x = o) -> (s z) = x.
Axiom ax3 : (s x) = (s y) -> x = y.
Axiom ax4 : (plus x o) = x.
Axiom ax5 : (plus x (s y)) = s (plus x y).
Axiom ax6 : (mult x o) = o.
Axiom ax7 : (mult x (s y)) = (plus (mult x y) x).

End Peano.

Goal (plus (s o) (s o)) = (s (s o)).
Proof.
rewrite -> ax5.
rewrite -> ax4.
reflexivity.
Qed.
```

Obligation de preuve

```
1 goal
________________________(1/1)
plus (s o) (s o) = s (s o)

######################

________________________(1/1)
s (plus (s o) o) = s (s o)

######################

________________________(1/1)
s (s o) = s (s o)

######################

No more goals.
```

2 Les tactiques

Coq est basé sur la déduction naturelle, mais heureusement, l'utilisateur n'est pas toujours obligé de préciser, une à une, les règles de déduction utilisées pour construire une preuve. Il peut aussi s'aider des tactiques. Celles-ci permettent de regrouper un enchaînement d'applications de règles de déduction, ou même deviner quelles sont les règles que l'on peut utiliser dans certains cas. Par exemple, la tactique `intros` va appliquer autant de fois qu'elle le peut la règle d'introduction correspondant au connecteur $\Rightarrow$ ou quantificateur $\forall$ de la formule que l'on souhaite prouver.

Cette section décrit les différentes tactiques disponibles dans Coq. Les tactiques correspondent à des règles dans le calcul des séquents. Des règles différentes s'appliquent selon s la formules est présentes à gauche ou à droite du $\vdash$ (par exemple, $\Rightarrow_g$ et $\Rightarrow_d$), ce qui correspond en Coq à des formules situées respectivement dans les hypothèses ou dans la conclusion. On peut également retrouver les termes « introduction » pour signifier que l'on passe une formule de la droite du thèse vers la gauche, et « élimination » pour le sens contraire.

2.1 Mémo

Le tableau suivant récapitule les règles principales. Le détail des tactiques st disponible plus loin. Les parties en grises sont optionnelles et permettent de nommer certains éléments ou de préciser le cas d'application d'une règle. Plusieurs tactiques peuvent avoir des résultats similaires, mais nous vous encourageons à utiliser celles qui vous sont présentées ici afin de comprendre le comportement de vos preuves.

Symbole	Hypothèse (g)	Conclusion (d)
$\Rightarrow$	`apply` $\langle H \rangle$	`intro(s)`
$\forall$	`apply` $\langle H \rangle$	`intro(s)`
$\neg$	`elim` $\langle H \rangle$	`intro`
$\wedge$	`destruct` $\langle H \rangle$ `[H1 H2]`	`split`
$\vee$	`destruct` $\langle H \rangle$ `[H1 | H2]`	`left` ou `right`
$\exists$	`destruct` $\langle H \rangle$ `[x H]`	`exists` $\langle term \rangle$
$\top$	–	`trivial`
$\bot$	`trivial`	–
$t_1 = t_2$	`rewrite <-` $\langle H1 \rangle$ `in` $\langle H2 \rangle$	`reflexivity`

2.2 Intro

Nom : `intro`

Règle LJ/LK associée : $\Rightarrow_d$, $\forall_d$, $\neg_d$

Description : La tactique `intro` applique la règle d'introduction correspondant au connecteur ou quantificateur de la conclusion du but courant. En fonction des cas, elle passe les prémisses de la règle dans les hypothèses ou introduit une variable dans les hypothèses. Le nommage est facultatif (par défaut, les hypothèses sont nommées `H`, `H0`, `H1`, …) dans le cas $\Rightarrow$ et par le nom de la variable dans le cas $\forall$.

Import(s) requis : Aucun

Script

```
Parameters A B : Prop.

Goal A -> B.
Proof.
    intro.
    ...
Qed.
```

Obligation de preuve

```
________________ (1/1)
A -> B

######################

H : A
________________ (1/1)
B
```

Script

```
Parameters A B : Prop.

Goal A -> B.
Proof.
    intro HA.
    ...
Qed.
```

Obligation de preuve

```
________________ (1/1)
A -> B

######################

HA : A
________________ (1/1)
B
```

Script

```
Parameter E : Set.
Parameter P : E -> Prop.

Goal forall x : E, P(x).
Proof.
    intro.
    ...
Qed.
```

Obligation de preuve

```
________________ (1/1)
forall x : E, P x

######################

x : E
________________ (1/1)
P x
```

Script

```
Parameter E : Set.
Parameter P : E -> Prop.

Goal forall x : E, P(x).
Proof.
    intro my_var_x.
    ...
Qed.
```

Obligation de preuve

```
________________ (1/1)
forall x : E, P x

######################

my_var_x : E
________________ (1/1)
P my_var_x
```

Script

```
Parameters A :  Prop.

Goal ~A.
Proof.
    intro HA.
    ...
Qed.
```

Obligation de preuve

```
________________ (1/1)
~A

######################

HA : A
________________ (1/1)
False
```

2.3 Intros

Nom : `intros`

Règle LJ/LK associée : $\Rightarrow_d$, $\forall_d$

Description : La tactique `intros` effectue plusieurs `intro` successifs. Et elle recommencera, autant de fois que possible, sur la conclusion du but obtenu. Le nommage est facultatif (par défaut, les hypothèses sont nommées `H`, `H0`, `H1`, ...) dans le cas $\Rightarrow$ et par le nom de la variable dans le cas $\forall$. Attention, cette règle ne permet de faire des introductions multiples sur la négation $\neg$.

Import(s) requis : Aucun

Script

```
Parameters A B C :  Prop.

Goal A -> B -> C.
Proof.
    intros.
    ...
Qed.
```

Obligation de preuve

```
________________ (1/1)
A -> B -> C

######################

H : A
H0 : B
________________ (1/1)
C
```

Script

```
Parameters A B C :  Prop.

Goal A -> B -> C.
Proof.
    intros HA HB.
    ...
Qed.
```

Obligation de preuve

```
________________ (1/1)
A -> B -> C

######################

HA : A
HB : B
________________ (1/1)
C
```

Script

```
Parameter E : Set.
Parameter P : E -> E -> E -> Prop.

Goal forall x y z: E, P x y z.
Proof.
    intros.
    ...
Qed.
```

Obligation de preuve

```
________________ (1/1)
forall x y z : E, P x y z

######################

x, y, z : E
________________ (1/1)
P x y z
```

Script

```
Parameter E : Set.
Parameter P : E -> E -> E -> Prop.

Goal forall x y z: E, P x y z.
Proof.
    intros var_x var_y var_z.
    ...
Qed.
```

Obligation de preuve

```
________________ (1/1)
forall x y z : E, P x y z

######################

var_x, var_y, var_z : E
________________ (1/1)
P var_x var_y var_z
```

2.4 Assumption

Nom : `assumption`

Règle LJ/LK associée : *ax*

Description : La tactique `assumption` correspond à la règle `axiome` de la déduction naturelle. On peut donc l'utiliser pour finir la preuve quand la conclusion du but courant se trouve dans les hypothèses.

Import(s) requis : Aucun

Script

```
Goal A -> A.
Proof.
    intro HA.
    assumption.
Qed.
```

Obligation de preuve

```
HA : A
________________ (1/1)
A

######################

No more goals.
```

2.5 Trivial

Nom : `trivial`

Règle LJ/LK associée : $\top_d$, $\bot_g$

Description : La tactique `trivial` permet de terminer une preuve dont le résultat est trivial, c'est à dire une preuve qui contient `True` ou $\top$ en conclusion ou `False` ou $\bot$ dans les hypothèses.

Import(s) requis : Aucun

Script

```
Goal True.
Proof.
    trivial.
Qed.
```

Obligation de preuve

```
1 goal
________________(1/1)
True

######################

No more goals.
```

Script

```
Goal False -> False.
Proof.
    intro.
    trivial.
Qed.
```

Obligation de preuve

```
H: False
________________(1/1)
False

######################

No more goals.
```

2.6 Apply

Nom : `apply`

Règle LJ/LK associée : $\Rightarrow_g$, $\forall_g$

Description : La tactique `apply` permet d'utiliser une formule que l'on a en hypothèse. Par exemple, si la conclusion du but courant est une formule `B` et que l'on a en hypothèse une formule `A -> B` (nommée `H`), alors on peut appliquer cette hypothèse grâce à la commande `apply H`. Il restera alors à prouver `A`. Pour le cas $\forall_g$, elle permet d'appliquer une formule générale à un cas particulier. Par exemple, si l'on a `forall x: E, P x` en hypothèse, alors en particulier on a `P a` (avec `a` de type `E`).

Import(s) requis : Aucun

Script

```
Parameters A B :  Prop.

Goal (A -> B) -> B.
Proof.
    intro H.
    apply H.
Qed.
```

Obligation de preuve

```
H : A -> B
________________ (1/1)
B

######################

H : A -> B
________________ (1/1)
A
```

Script

```
Parameter E : Set.
Parameters P : E -> Prop.
Parameter a : E.

Goal  (forall x:E, P x) -> P a.
Proof.
  intro.
  apply H.
Qed.
```

Obligation de preuve

```
H : forall x : E, P x
_________________(1/1)
P a

######################

No more goals.
```

2.7 Elim

Nom : `elim`

Règle LJ/LK associée : $\neg_g$

Description : La tactique `elim` permet de faire passer une hypothèse avec une négation dans le but courant, si celui-ci est actuellement `False`. Peut également avoir un comportement similaire à `destruct` mais en conservant l'hypothèse.

Import(s) requis : Aucun

Script

```
Parameter A : Prop.

Goal ~ A -> False.
Proof.
  intros.
  elim H.
Qed.
```

Obligation de preuve

```
H : ~A
________________ (1/1)
False

######################

H : ~A
________________ (1/1)
A
```

2.8 Exfalso

Nom : `exfalso`
Règle LJ/LK associée : suppression du but
Description : La tactique `exfalso` permet de supprimer le but courant et de le remplacer par `False`.
Import(s) requis : Aucun

Script

```
Parameter A : Prop.

Goal A -> A.
Proof.
  intros.
  exfalso.
  ...
Qed.
```

Obligation de preuve

```
H : ~A
________________ (1/1)
A

######################

H : ~A
________________ (1/1)
False
```

2.9 Destruct

Nom : `destruct`
Règle LJ/LK associée : $\wedge_g$, $\vee_g$, $\exists_g$
Description : La tactique `destruct` permet d'éliminer des conjonctions, disjonctions et existentiels en hypothèse. L'hypothèse en question est détruite et de nouvelles hypothèse apparaissent. En cas se disjonction, deux nouvelles branches sont créées (à l'aide du symbol `|`), chacune possédant sa propre hypothèse. Pour le cas de la règle $\exists_g$, destruct vous permet de séparer la variable et la formule. Les hypothèse nouvellement créées peuvent être nommées.
Import(s) requis : Aucun

Script

```
Parameter A : Prop.
Parameter B : Prop.

Goal  (A /\ B) -> A.
Proof.
  intro.
  destruct H.
  ...
Qed.
```

Obligation de preuve

```
H : A /\ B
________________ (1/1)
A

######################

H : A
H0 : B
________________ (1/1)
A
```

Script

```
Parameter A : Prop.
Parameter B : Prop.

Goal  (A /\ B) -> A.
Proof.
  intro.
  destruct H as [HA HB].
  ...
Qed.
```

Obligation de preuve

```
H : A /\ B
________________ (1/1)
A

######################

HA : A
HB : B
________________ (1/1)
A
```

Script

```
Parameter A : Prop.
Parameter B : Prop.

Goal  (A \/ B) -> A.
Proof.
  intro.
  destruct H as [HA | HB].
    - (* Branche HA *)
    - (* Branche HB *)
Qed.
```

Obligation de preuve

```
1 goal
H : A \/ B
________________ (1/1)
A

######################

2 goals
H2 : A
________________(1/2)
A
________________(2/2)
A
```

Dans cet exemple, on observe que l'on a à présent deux buts, c'est à dire deux branches au lieu d'une seule. Coq vous montre ainsi les buts à prouver dans les deux branches (ici, `A` dans les deux cas), et les hypothèse de la première branche. Pour accéder aux autres branches, vous pouvez soit prouver la première branche (et Coq passera automatiquement à la suivante), soit structurer votre preuve à l'aide des symboles `-`, `*`, `+`, … pour naviguer entre les branches. Toutes les branches doivent être prouvées pour parvenir à la fin de la preuve.

Script

```
Parameter E : Set.
Parameters P : E -> Prop.

Goal  (exists x: E, P x) -> False.
Proof.
  intro.
  destruct H as [x H'].
  ...
Qed.
```

Obligation de preuve

```
H : exists x : E, P x
_____________________ (1/1)
False

######################

x : E
H' : P x
______________________(1/1)
False
```

2.10 Exists

Nom : `exists`

Règle LJ/LK associée : $\exists_d$

Description : Pour prouver une formule quantifiée existentiellement telle que `exists y : E, P y`, vous devez fournir à Coq à la fois le témoin et la preuve que ce témoin vérifie le prédicat `P`. La tactique `exists` permet de faire cela. Dans le but courant, on peut instancier `y` par `a` dans la formule que l'on cherche à prouver à l'aide de la commande `exists a`. Il restera alors à montrer `P a`.

Import(s) requis : Aucun

Script

```
Parameter E : Set.
Parameter P : E -> Prop.
Parameter a : E.

Goal exists y : E, P y.
Proof.
  exists a.
  ...
Qed.
```

Obligation de preuve

```
_________________(1/1)
exists y : E, P y

######################

________________ (1/1)
P a
```

2.11 Split

Nom : `split`

Règle LJ/LK associée : $\wedge_d$

Description : La tactique `split` permet d'éliminer une conjonction en conclusion. Cette tactique génère deux nouvelles branche, et conserve les hypothèses précédentes. Vous devez prouver les deux côtés de la conjonction (donc les deux branches) pour terminer la preuve.

Import(s) requis : Aucun

Script

```
Parameter A : Prop.
Parameter B : Prop.

Goal  A /\ B.
Proof.
  split.
  + (* Cas A *)
  + (* Cas B *)
...
Qed.
```

Obligation de preuve

```
1 goal
________________ (1/1)
A /\ B

######################

2 goals
________________ (1/2)
A
________________ (2/2)
B
```

2.12 Left et Right

Nom : `left` et `right`

Règle LJ/LK associée : $\vee_d$

Description : Les tactiques `left` et `right` permet d'éliminer une disjonction en conclusion. En deduction naturelle, dans le cas d'un disjonction, il suffit de prouver que l'un des deux cas est vrai pour que la disjonction soit vraie. Cette tactique vous permet ainsi de choisir le côté de la disjonction que vous souhaitez prouver (gauche ou droit). L'autre cas disparaît.

Import(s) requis : Aucun

Script

```
Parameter A : Prop.
Parameter B : Prop.

Goal  A \/ B.
Proof.
  left.
...
Qed.
```

Obligation de preuve

```
1 goal
________________ (1/1)
A \/ B

######################

________________ (1/2)
A
```

Script

```
Parameter A : Prop.
Parameter B : Prop.

Goal  A \/ B.
Proof.
  right.
...
Qed.
```

Obligation de preuve

```
1 goal
________________ (1/1)
A \/ B

######################

________________ (1/2)
B
```

2.13 NNPP

Nom : `NNPP`

Règle LJ/LK associée : Tiers exclu

Description : La tactique `NNPP` vous permet d'utiliser le tiers exclu en déduction naturelle, c'est à dire le fait que $\neg\neg A = A$.

Import(s) requis : `Require Import Classical.`

Script

```
Parameter A : Prop.
Require Import Classical.

Goal A.
Proof.
    apply NNPP.
    ...
Qed.
```

Obligation de preuve

```
________________ (1/1)
A

######################

________________ (1/1)
~~A
```

2.14 Reflexivity

Nom : `reflexivity`

Règle LJ/LK associée : égalité

Description : La tactique `reflexivity` vous permet de fermer une branche quand le but que vous voulez prouver est égal à lui-même.

Import(s) requis : Aucun

Script

```
Parameter A : Prop.

Goal A = A.
Proof.
    reflexivity.
Qed.
```

Obligation de preuve

```
________________ (1/1)
A = A

######################

No more goals.
```

2.15 Rewrite

Nom : `rewrite`

Règle LJ/LK associée : réécriture

Description : Pour une égalité `A = B` donnée en axiome ou en hypothèse, la tactique `rewrite` vous permet de remplacer un élément par l'autre, et vice-versa. Il est possible de préciser l'axiome concerné ou le sens de la réécriture si plusieurs application sont possibles, voire même de réécrire dans les hypothèses.

Import(s) requis : Aucun

Script

```
Axiom add_null : forall (x : nat), x + 0 =

Goal  1 + 0 =  1.
Proof.
  rewrite -> add_null.
  ...
Qed.
```

Obligation de preuve

```
________________ (1/1)
1 + 0 = 1

######################

________________ (1/1)
1 = 1
```

Script

```
Parameter A : Prop.
Parameter B : Prop.

Goal  (A = B) -> (A -> B).
Proof.
  intro HAB.
  intro HA.
  rewrite <- HAB.
  ...
Qed.
```

Obligation de preuve

```
HAB : A = B
HA : A
________________ (1/1)
B

######################

HAB : A = B
HA : A
________________ (1/1)
A
```

Script

```
Parameter A : Prop.
Parameter B : Prop.

Goal  (A = B) -> (A -> B).
Proof.
  intro HAB.
  intro HA.
  rewrite -> HAB in HA.
  ...
Qed.
```

Obligation de preuve

```
HAB : A = B
HA : A
________________ (1/1)
B

######################

HAB : A = B
HA : B
________________ (1/1)
B
```

2.16 Simpl

Nom : `simpl`

Règle LJ/LK associée : simplification

Description : La tactique `simpl` permet de simplifier l'écriture d'une expression, en se basant sur les définitions de base de Coq ou sur vos propres fonctions. Elle peut également être utilisée sur une hypothèse.

Import(s) requis : Aucun

Script

```
Goal  1 + 0 =  1.
Proof.
  simpl.
  ...
Qed.
```

Obligation de preuve

```
________________ (1/1)
1 + 0 = 1

######################

________________ (1/1)
1 = 1
```

Script

```
Goal (1 + 1 = 2) -> False.
Proof.
  intro H.
  simpl in H.
  ...
Qed.
```

Obligation de preuve

```
H : 1 + 1 = 2
________________ (1/1)
False

######################

H : 2 = 2
________________ (1/1)
False
```

Script

```
Fixpoint add (a: nat) (b: nat) : nat :=
  match a with
    | O => b
    | S x => S (add x b)
  end.

Goal  forall n, (add O n) = n.
Proof.
  intros.
  simpl.
  ...
Qed.
```

Obligation de preuve

```
n : nat
________________ (1/1)
(add 0 n) = n

######################

n : nat
________________ (1/1)
n = n
```

2.17 Unfold

Nom : `unfold`

Règle LJ/LK associée : remplacement d'un term par sa définition

Description : La tactique `unfold` permet de remplacer un term par sa définition. Cela peut s'avérer utile si `simpl` ne parvient pas à simplifier une expression. Tout comme cette dernière, `unfold` se base sur les définitions de base de Coq ou sur vos propres fonctions, et peut être utilisée sur une hypothèse.

Import(s) requis : Aucun

Script

```
Definition plus_un (x : nat) : nat :=
  x + 1.

Goal plus_un 0 = 1.
Proof.
  unfold plus_un.
  ...
Qed.
```

Obligation de preuve

```
_______________ (1/1)
plus_un 0 = 1

######################

_______________ (1/1)
0 + 1 = 1
```

2.18 Lia

Nom : `lia`

Règle LJ/LK associée : arithmétique linéaire

Description : La tactique `lia` permet de raisonner efficacement sur des preuves impliquant des entiers. Vous pouvez vous en servir pour simplifier des expressions arithmétique ou fermer des branches.

Import(s) requis : `Require Import Lia.`

Script

```
Require Import Lia.

Goal 1 + 1 = 2.
    Proof.
    lia.
Qed.
```

Obligation de preuve

```
_______________ (1/1)
1 + 1 = 2

######################

No more goals.
```

2.19 Inversion

Nom : `inversion`

Règle LJ/LK associée : arithmétique linéaire

Description : Parfois, vous serez confronté à des hypothèses qui ne sont vraies que si certaines autres hypothèses le sont aussi. Dans ce genre de cas, vous pouvez utiliser la tactique `inversion` pour découvrir ces hypothèses et tenter des les prouver.

Import(s) requis : Aucun

Script

```
Inductive nat : Set :=
  | O
  | S : nat -> nat.

Goal forall a b, S a = S b -> a = b.
Proof.
  intros a b H.
  inversion H.
  ...
Qed.
```

Obligation de preuve

```
a, b : nat
H : S a = S b
________________ (1/1)
a = b

######################

a, b : nat
H : S a = S b
H0 : a = b
________________ (1/1)
a = b
```

Dans cette exemple, nous voulons prouver que si les successeurs de `a` et `b` sont égaux alors `a` et `b` sont eux-même égaux. Nous supposons que `(S a) = (S b)`. Cependant, par notre construction des entiers, cela ne peut être vrai que si `a` et `b` sont eux-même égaux. Nous utilisons donc `inversion` pour demander à Coq d'analyser la manière dont sont construit `a` et `b` et réaliser que cela implique que `a` et `b` sont égaux, et ainsi ajouter cette hypothèse au contexte.

3 Pour aller plus loin

3.1 Sauvegarde de preuves

Coq vous permet de déclarer des lemmes et de théorèmes afin de pouvoir les réutiliser plus tard. Ces lemmes et théorèmes sont comme des axiomes, sauf que vous devez en fournir la preuve. Par exemple, nous pouvons déclarer le théorème suivant : $\forall x, y \in \mathbb{N}.\ (x+1)+y = (x+y)+1$, ou en version Péano : `plus (s x) y = (s (plus x y))` et le sauvegarder pour une future utilisation.

Script

```
Parameter N : Set.
Parameter o : N.
Parameter s : N -> N.
Parameter plus : N -> N -> N.
...

Theorem add_x_y_un : forall x y : N,
        plus (s x) y = (s (plus x y)).
Proof.
    ...
Qed.
```

Obligation de preuve

```
1 goal
________________ (1/1)
forall x y : N,
    plus (s x) y = s (plus x y)
```

3.2 Création de tactiques

Vous avec la possibilité de créer vous-même vos propres tactiques avec `Ltac`. Cela vous permet par exemple d'automatiser l'utilisation de plusieurs axiomes, lemmes ou théorèmes. Dans cette exemple, on va considérer l'axiome de commutativité de l'addition : $\forall n, m, p \in \mathbb{N}.\ n + (m + p) = n + m + p$. Grâce à cela, on veut prouver que $a + b + c + d = a + (b + c + d)$. On peut tout d'abord faire la preuve classique :

Script	Obligation de preuve

<pre>Axiom assoc : forall n m p : nat,
    n + (m + p) = n + m + p.
 	
Goal forall (a b c d : nat),
(a + b + c) + d = a + (b + c + d).
Proof.
intros a b c d.
rewrite -> assoc.
rewrite -> assoc.
reflexivity.
Qed.</pre>
	<pre>1 goal
	______________ (1/1)
	forall a b c d : nat,
	    a + b + c + d = a + (b + c + d)
	
	######################
	
	a, b, c, d : nat
	______________ (1/1)
	a + b + c + d = a + (b + c + d)
	
	######################
	
	a, b, c, d : nat
	______________ (1/1)
	a + b + c + d = a + (b + c) + d
	
	######################
	
	a, b, c, d : nat
	______________ (1/1)
	a + b + c + d = a + b + c + d
	
	######################
	
	No more goals.</pre>

On peut à présent tenter d'automatiser un peu cette preuve, notamment les application successives de `assoc`. Pour ce faire, on définit la tactique `auto_assoc` qui va répéter `rewrite assoc` autant de fois que possible.

Script

```
Axiom assoc :  forall n m p : nat,
    n + (m + p) = n + m + p.

Ltac auto_assoc :=
    repeat rewrite assoc.

Goal forall (a b c d : nat),
    (a + b + c) + d = a + (b + c + d).
Proof.
    intros a b c d.
    auto_assoc.
    reflexivity.
Qed.
```

Obligation de preuve

```
1 goal
________________ (1/1)
forall a b c d : nat,
    a + b + c + d = a + (b + c + d)

######################

a, b, c, d : nat
________________ (1/1)
a + b + c + d = a + (b + c + d)

######################

a, b, c, d : nat
________________ (1/1)
a + b + c + d = a + b + c + d

######################

No more goals.
```

Il est bien entendu possible d'améliorer cette tactique, par exemple en lui demandant de faire `intros` au début, ou `reflexivity` à la fin.

Script

```
Axiom assoc :  forall n m p : nat,
    n + (m + p) = n + m + p.

Ltac auto_assoc :=
  intros ;
  repeat rewrite associativite;
  try reflexivity.

Goal forall (a b c d : nat),
    (a + b + c) + d = a + (b + c + d).
Proof.
    auto_assoc.
Qed.
```

Obligation de preuve

```
1 goal
________________ (1/1)
forall a b c d : nat,
    a + b + c + d = a + (b + c + d)

######################

No more goals.
```