 Julie Cailler, David Delahaye et Michel Leclere

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 le 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 servir de vos tactiques préférées !

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 nécessaires 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 essayez de faire une preuve. Elle se sépare 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 buts 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`.

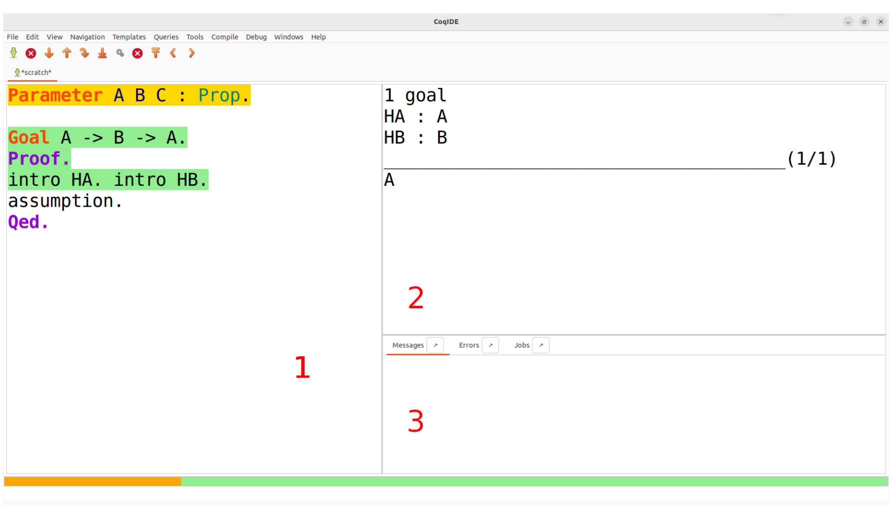

FIGURE 1 – Interface de CoqIDE

1.2 Syntaxe et structure d'un script Coq

Coq est basé sur la logique intuitionniste, c'est à dire LJ. Un script Coq est composé de plusieurs éléments. Tout d'abord, la déclaration 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 de la propriété à prouver 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 utiliser 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 terme *)
```

— **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 supposée 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 définies de façon inductive.

```
Definition plus_un (n : nat) : nat := S n.     (* la déclaration du type du résultat
                                                  n'est pas obligatoire           *)
Fixpoint plus_un_rec (n : nat) : nat :=
    match n with
    | O => 0
    | S n' => S (plus_un_rec n')
end.
```

— **Relations inductives** : une relation prend une paire d'éléments (possiblement des ensembles) et décrit le comportement que doit 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)
.
```

— **Ensemble définit par induction** : le mot-clé `inductive` permet également de définir des ensemble par induction, en spécifiant le(s) cas de base et le(s) cas inductif(s). Par exemple, les entiers naturels peuvent être définis de la façon suivante :

```
Inductive N : Set =
    | o : N
    | s : N -> N.
.
```

— **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 *)
```

— **Théorème, 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 *)
```

— **Print** : La commande `Print` affiche dans la partie *messages* les informations sur l'objet passé en paramètre.

```
Print nat.

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

Inductive nat : Set :=  O : nat | S : nat -> nat.
Arguments S _%nat_scope
```

— **Compute** : La commande `Compute` évalue l'expression passé en paramètre et affiche le résultat dans la partie *messages*.

```
Compute plus_un 3.

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

4
```

Une ligne peut comporter plusieurs instructions, mais le retour à la ligne améliore la lisibilité. Finalement, étant donné que les preuves Coq s'effectuent dans la calcul des séquents intuitionniste, les connecteurs et quantificateurs peuvent s'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 nativement mais peuvent être implémentées à l'aide 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 formule $A \Rightarrow A$.

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 variables et des prédicats, ils permettent de représenter des formules et des propriétés en logique du premier ordre. Il faut cependant noter que toutes les variables en Coq doivent être typées par un `Set`. par exemple $\forall x.\ P(x)$ doit s'écrire `forall x : E, P(x)` où `E` est un `Set` typant `x`. Ainsi, la formule $\forall x. P(x) \Rightarrow \exists y.\ P(y)$ 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 Peano

En réalité, dans Coq, les entiers sont représentés à la manière des entiers de Peano (définis par la constante de base O et la fonction successeur S). Ainsi, 2 est en réalité `S(S(O))`. On peut ainsi définir les axiomes de l'arithmétique Peano 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)
```

Notons que dans cet exemple, les variables `x` et `y` sont quantifiées universellement et réutilisée dans chaque axiome. Il est également possible des les inclure directement dans la définition de l'axiome, par exemple :

```
Axiom ax3 : forall x y : nat, (s x) = (s y) -> x = y.
```

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, si 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 axioms

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

Les tactiques correspondent à des applications spécifiques de règles du calcul des séquents ou a des regroupements d'enchaînement d'applications de règles. Elles permettent même, dans certains cas, de deviner quelles sont les règles que l'on peut utiliser. Par exemple, la tactique `intros` va appliquer autant de fois qu'elle le peut les règles d'introduction correspondant au connecteur $\Rightarrow$ et quantificateur $\forall$ sur le but à prouver. Les règles d'introduction sont les règles « droite » du calcul des séquents et s'appliquent sur le but et entraînent un passage d'une (sous-)formule du but en hypothèse. Les règles « gauche » quant à elles sont appelées règles d'élimination et s'appliquent sur une formule en hypothèse et entrainent un passage en but d'une sous-formule de cette hypothèse.

Cette section décrit les différentes tactiques disponibles dans Coq.

2.1 Mémo

Le tableau suivant récapitule les règles principales. Le détail des tactiques est disponible plus loin. Les parties en gris 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	Hypothèses (g)	But (d)
$\Rightarrow$	`apply` $\langle H \rangle$	`intro(s)`
$\forall$	`apply` $\langle H \rangle$	`intro(s)`
$\neg$	`destruct` $\langle H \rangle$	`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`
$\bot$	`elim` $\langle H \rangle$ `(ou exfalso)`	–
$t_1 = t_2$	`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 racine 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 les cas $\Rightarrow_d$ et $\neg_d$ et par le nom de la variable dans le cas $\forall_d$.

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 le but obtenu. Le nommage est facultatif (par défaut, les hypothèses sont nommées `H`, `H0`, `H1`, …) dans le cas $\Rightarrow_d$ et par le nom de la variable dans le cas $\forall_d$. Attention, cette tactique ne déclenche pas la règle $\neg_d$.

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 `ax` du calcul des séquents. 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 : $\top_d$

Description : La tactique `trivial` permet de terminer une preuve dont le but est `True` correspondant de fait à l'application de la règle $\top_d$. Mais cette tactique est en fait une stratégie automatique de preuve qui essaie d'abord de résoudre le but courant avec la tactique `assumption` et, si cela ne marche pas, applique la tactique `intros` puis essaie de résoudre le but obtenu avec l'une des règles correspondant au symbole racine du but : $\top_d$ si le but est $\top$, `reflexivity` si le but est une égalité, ... ou avec `assumption` si le but est dans les hypothèses. La tactique `auto` étend cette stratégie à des buts conjonctifs.

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 <Hyp>`

Règles LJ associées : $\Rightarrow_g$ (sous-certaines conditions), $\forall_g$

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`. Cette tactique peut également être utilisée en cas de conjonction dans une hypothèse, par exemple avec `A -> ( B -> (C /\ D))` et un but `D`. Dans ce cas, `apply` générera deux nouveau sous-buts à prouver : `A` et `B` (en effet, si de `A` et `B` on peut déduire `C /\ D`, alors a fortiori on peut déduire `D`).

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.
```

Script

```
Parameters A B C D : Prop.

Goal A -> B -> (A -> (B -> ((C /\ D))))
     -> D.
Proof.
    intros HA HB H.
    apply H.
    ...
Qed.
```

Obligation de preuve

```
HA : A
HB : B
H : A -> B -> C /\ D
________________ (1/1)
D

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

HA : A
HB : B
H : A -> B -> C /\ D
________________ (1/2)
A
________________ (2/2)
B
```

2.7 Elim

Nom : `elim`

Règle LJ/LK associée : aucune

Description : La tactique `elim` permet de faire passer une hypothèse dont le connecteur racine est 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 : aucune

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`

Règles LJ/LK associées : $\neg_g$, $\wedge_g$, $\Leftrightarrow_g$, $\vee_g$, $\exists_g$

Description : La tactique `destruct` permet d'éliminer des négations, conjonctions, disjonctions et existentiels en hypothèse. L'hypothèse en question est détruite. Pour le cas de la négation, le nouveau but devient l'ancienne hypothèse sans la négation (l'ancien but est perdu). Pour la conjonction, deux nouvelles hypothèses apparaissent (les deux sous-formules de la conjonction). Pour l'équivalence, deux nouvelles implications apparaissent en hypothèse (correspondant aux deux sens de l'équivalence). Pour la disjonction, deux nouvelles branches sont créées, chacune possédant sa propre hypothèse (les deux sous-formules de la disjonction). Pour la règle $\exists_g$, deux nouvelles hypothèses apparaissent, l'une introduit la variable et l'autre la formule. Les hypothèses 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`

Règles LJ associées : $\vee_{d_1}$, $\vee_{d_2}$

Description : Les tactiques `left` et `right` permet d'éliminer une disjonction en conclusion. En déduction 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 : `apply NNPP`

Règle LJ$_{\texttt{em}}$ associée : *em*

Description : La tactique `apply` permet d'appliquer la règle du tiers exclu *em*. Cette règle remplace le but courant B par $\neg\neg B$. Cette règle du tiers exclu appelée `NNPP` dans Coq doit auparavant être importée via la commande d'import suivante.

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$_{\texttt{EQ}}$/LK$_{\texttt{EQ}}$ associée : $refl$

Description : La tactique `reflexivity` vous permet de fermer une branche quand le but que vous voulez prouver est une égalité entre deux éléments syntaxiquement égaux.

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 [->|<-] Hyp [in Hyp']`

Règles LJ_{EQ}/LK_{EQ} associées : $=_{d_1}$, $=_{d_2}$, $=_{g_1}$, $=_{g_2}$

Description : Pour une hypothèse d'égalité `Hyp` de la forme `A = B` donnée en axiome ou en hypothèse, la tactique `rewrite` vous permet de remplacer dans le but (ou une autre hypothèse, dans ce cas il faut préciser laquelle par : `in Hyp'`) un des éléments de l'égalité par l'autre ; `rewrite -> Hyp` (ou simplement `rewrite Hyp`) remplace l'élément de gauche par celui de droite (ici A par B) et à l'inverse `rewrite <- Hyp` fait le contraire.

Import(s) requis : Aucun

Script

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

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 : aucune

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 O n) = n

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

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

2.17 Unfold

Nom : `unfold`

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

Description : La tactique `unfold` permet de remplacer un terme 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`

Règles associées : règles liées à l'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 arithmétiques 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`

Règles associées : 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 des 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 Peano : `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

```
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.
```

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

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

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

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

No more goals.
```

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.
```

3.3 Fonctions, relations et schémas d'induction

Nous allons travailler avec la fonction `sum_n` qui fait la somme des `n` premiers entiers. Cette fonction est récursive, nous devons donc la déclarer à l'aide du mot-clé `Fixpoint`. Sa définition en `Coq` est la suivante :

```
Fixpoint sum_n (n : nat) : nat :=
    match n with
    | O => 0
    | S n' => n + (sum_n n')
end.
```

Cette définition indique que la fonction prend un entier naturel `n` et retourne un entier naturel. Nous définissons son cas de base et son équation d'induction :

— Base : `O => 0`, car la somme des 0 premiers entiers est égale à 0.

— Induction : `S n' => n + (sum_n n')`, on considère `n` de la forme `S n'`, c'est à dire que `n` est le successeur d'un autre entier `n'`, et nous exprimons la somme en fonction de `n'`. La somme des `n` premiers entiers peut alors s'exprimer comme `n + (sum_ n n')`, avec `n = S n'` (autrement dit, `n' = n - 1`).

Nous définissons ensuite les spécification de la fonction, c'est à dire le contrat que doit remplir la fonction. Dans notre exemple, nous voulons que la fonction renvoie la somme des `n` premiers entiers. Nous définissons donc une spécification qui prend deux entiers `n` et `s` en paramètre, et qui vérifie que `s` est bien la somme des `n` premiers entiers. Une spécification est définie grâce au mot-clé `Inductive`.

```
Inductive is_sum_n : nat -> nat -> Prop :=
    | is_sum_n_O : is_sum_n O 0
    | is_sum_n_S : forall n s, is_sum_n n s -> is_sum_n (S n) ((S n) + s)
.
```

Nous explicitons encore une fois le cas de base et l'équation d'induction :

— Base : `is_sum_n O 0`, car 0 est bien la somme des 0 premiers entiers.

— Induction : `forall n s, is_sum_n n s -> is_sum_n (S n) ((S n) + s)`. On pose l'hypothèse que `s` est la somme des `n` premiers entiers et on dit que la somme des `n + 1` (i.e. `(S n)`) premiers entiers correspond à `(S n) + s`.

Nous pouvons maintenant nous servir des éléments précédents pour prouver que la fonction correspond bien à sa spécification. Pour cela, nous pouvons prouver la correction, c'est à dire que si `sum_n(n)` retourne un résultat alors ce résultat est bien la somme des `n` premier entiers :

$$\forall n, s.\ \texttt{sum\_n}(n) = s \Rightarrow \texttt{is\_sum\_n}(n, s)$$

Nous pouvons aussi prouver la complétude, c'est à dire que si la relation `is_sum_n n s` existe pour deux entiers alors `s` est bien la somme des `n` premiers entiers :

$$\forall n, s.\ \texttt{is\_sum\_n}(n, s) - > \texttt{sum\_n}(n) = s$$

Pour voir la différence entre induction structurelle et induction fonctionnelle, nous allons montrer la correction de deux façons différentes, l'une en utilisant le schéma d'induction structurelle sur les entiers (qu'on appelle aussi récurrence), l'autre en utilisant le schéma d'induction fonctionnelle. Nous ferons également une preuve de complétude avec une induction structurelle sur la relation.

3.3.1 Induction structurelle sur les entiers

Démontrer la correction de la fonction `sum_n` par induction structurelle sur `n`. `n` étant un entier, cette induction se fait en considérant le cas de base `n = 0` et le cas inductif `n = S n'`. Pour cela, nous avons besoin de la tactique `induction`, qui s'utilise de la façon suivante :

```
induction n as [| n' IHn'].
```

Cette tactique prend comme premier paramètre l'élément sur lequel nous souhaitons faire l'induction (ici, `n`). Ensuite, elle nous permet de nommer les hypothèse dans le cas de base et le cas inductif. Dans notre exemple, nous ne nommons rien dans le cas `n = 0`, et pour le cas d'induction nous appelons notre paramètre `n'` et l'hypothèse `IHn'`.

```
Theorem sum_n_correct : forall n s, sum_n(n) = s -> is_sum_n n s.
Proof.
  induction n as [| n' IHn'].
  - (* n = 0 *) intros s H. simpl in H. rewrite <- H. apply is_sum_n_O.
  - (* n = S n' *)  intros s H. simpl in H. rewrite <- H. apply is_sum_n_S.
                    apply IHn'. reflexivity.
Qed.
```

3.3.2 Induction fonctionnelle

L'induction structurelle a été spécialement créée pour prouver des théorèmes sur des fonctions définies inductivement. Ce type de schéma est une induction sur la *structure de la fonction*. L'idée de l'induction fonctionnelle n'est il pas de voir la fonction comme une relation inductive entre ses paramètres et son résultat. Par exemple, si nous souhaitions donner informellement le schéma d'induction fonctionnelle de la fonction `sum_n`, nous pourrions le faire de la manière suivante :

```
P[0, 0] -> (pour tout n s, P[n, s] -> P[n + 1, s + (n + 1)]) -> pour tout n, P[n, sum_n(n)]
```

Nous pouvons par la suite instancier `P[n,s]` par le prédicat de notre choix, tant qu'il vérifie la propriété voulue. Par exemple, en remplaçant `P[n,s]` par "$n * (n + 1)/2 = s$", on obtient :

```
0*(0+1)/2 = 0 ->
(pour tout n s, n*(n+1)/2 = s -> (n + 1)*((n+1)+1)/2 = s + (n + 1)) ->
pour tout n, n*(n+1)/2 = sum_n(n).
```

Il est également possible de remplacer `P[n,s]` par le prédicat suivant : "sum_n(n) = s". En remplaçant `P` dans l'expression initiale, nous obtenons ainsi :

```
sum_n(0) = 0 ->
(pour tout n s, sum_n(n) = s -> sum_n(n + 1) = s + (n + 1)) ->
pour tout n, sum_n(n) = sum_n(n).
```

Nous pouvons ainsi remarquer que `P[n,s]` est en fait le prédicat `is_sum_n`. Ce prédicat est la spécification de la fonction `sum_n` ! C'est grâce à lui qu'on peut montrer que la fonction est conforme à sa spécification.

En Coq, c'est le module `FunInd` qui nous permet d'extraire le schéma d'induction fonctionnelle. Il faut donc importer ce module.

```
Require Import FunInd.
```

Une fois ce module importé, on extrait le schéma d'induction fonctionnelle de `sum_n` de la façon suivante :

```
Functional Scheme sum_n_ind := Induction for sum_n Sort Prop.
```

Pour vérifier le schéma créé, on peut utiliser la commande `Print`. Le début est assez peu compréhensible, mais regardons plus attentivement la fin :

```
Print sum_n_ind.

(sum_n_equation n)
     : forall P : nat -> nat -> Prop,
        (forall n : nat, n = 0 -> P 0 0) ->
        (forall n n' : nat,
        n = S n' -> P n' (sum\_n n') -> P (S n') (n + sum\_n n')) ->
        forall n : nat, P n (sum_n n)
```

On arrive à distinguer le schéma informel décrit plus haut. Il suffit pour cela de remplacer `P` par `is_sum_n`. Nous allons à présent démontrer la correction grâce au schéma d'induction fonctionnelle. Une nouvelle commande va alors être introduite :

```
[functional induction (_) using _.]
```

Cette commande permet d'exécuter une induction fonctionnelle sur l'objet entre parenthèses, en utilisant le schéma donné.

```
Theorem sum_n_correct' : forall n s, sum_n(n) = s -> is_sum_n n s.
Proof.
    (** Contrairement a une induction classique, on doit ici introduire
        [n] pour pouvoir l'utiliser lors du schéma d'induction fonctionnelle. **)
    intros n.

    (** Utilisons la commande décrite plus haut. On veut faire une induction
        sur [sum_n n], en utilisant le schéma [sum_n_ind].
        On remplace donc les [_] dans la ligne donnée, et on obtient : **)
    functional induction (sum_n n) using sum_n_ind.
    - (* n = 0 *) intros s H. rewrite <- H. apply is_sum_n_O.
    - (* n = S n' *) intros s H. rewrite <- H. apply is_sum_n_S. apply IHn0. reflexivity.
Qed.
```

Ici, la preuve avec le schéma d'induction fonctionnelle ressemble énormément à la preuve par induction structurelle. En effet, elle était également assez directe avec l'induction structurelle. Nous verrons en cours la fonction `even`, sur laquelle le schéma d'induction fonctionnelle fait des merveilles pour rendre la preuve beaucoup plus courte.

3.3.3 Induction structurelle sur la relation

Pour ce dernier cas, nous montrons la complétude de la fonction grâce à une induction sur la structure de la *relation*. Nos cas de base et d'induction sont ceux définis par la relation, à savoir `is_sum_n_O` et `is_sum_n_S`.

```
Theorem sum_n_complete : forall n s, is_sum_n n s -> sum_n(n) = s.
Proof.
    intros n s H. induction H as [| n s IH1 IH2].
    - (* is_sum_n_O *) simpl. reflexivity.
    - (* is_sum_n_S *) simpl. rewrite -> IH2. reflexivity.
Qed.
```