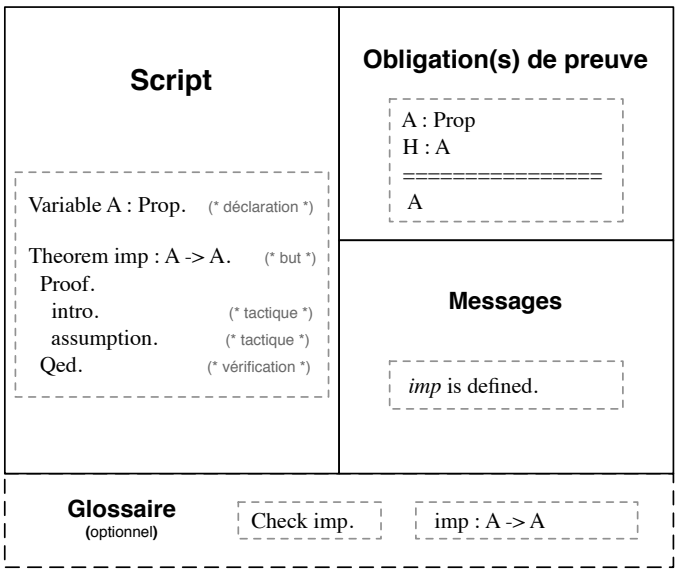
Mini-guide Coq

Interface

On utilise Coq via l'interface graphique CoqIDE (commande `coqide`).
La fenêtre est organisée en trois (ou quatre) zones :



La partie gauche de la fenêtre contient le **script** de preuve, que l'utilisateur peut modifier et sauver dans un fichier.
On peut envoyer progressivement, ligne à ligne, le script à Coq (à l'aide du bouton ⇓) et on peut également revenir en arrière (bouton ⇑) pour changer le script.
Le script est composé de **déclarations** et de théorèmes, aussi appelés **buts**. Chaque théorème est suivi de sa **preuve**, composée d'une suite de **tactiques**. Chaque déclaration, but et tactique se **termine par un point**. On peut aussi écrire des commentaires `(* comme ceci *)`.

En haut à droite, l'**obligation de preuve** (ou **but**) courante est affichée lorsqu'on est en train de prouver un théorème.

La deuxième fenêtre de droite contient les messages d'erreur (si une tactique échoue par exemple) et les messages de succès (si Coq réussit à vérifier une preuve après le `Qed`.).

Enfin, la fenêtre tout en bas (qui n'apparaît pas par défaut) permet d'interroger Coq à propos de l'environnement dans lequel on est en train de développer une preuve. On peut, par exemple, vérifier l'énoncé d'un théorème déjà prouvé ou d'un axiome en utilisant la commande `Check`. On peut faire apparaître cette fenêtre depuis le menu *Queries*.

Syntaxe

Correspondance entre notations papier et Coq

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

Une formule n'est bien formée que si les objets utilisés sont du bon type (les types jouent un rôle analogue à celui des ensembles). Par exemple, on ne peut pas écrire `(1 /\ 1)` ni `(P -> (P+2))`. Coq vérifie les types et souvent les devine tout seul. Sinon, on peut les préciser avec la syntaxe `x : T`.

Quelques types utiles

`nat`	type des entiers naturels (`3 : nat`)
`Prop`	type des propositions logique (`False : Prop`)
`Set`	type des types de termes (`nat : Set`)
`T1 -> T2`	types des fonctions de `T1` dans `T2`
`T -> Prop`	types des prédicats unaires sur `T`

Déclarations

Les déclarations permettent de déclarer à la fois des objets (avec leur types) que l'on souhaite ajouter à l'environnement, des formules que l'on souhaite supposer dans la suite du script (les *axiomes*), des formules que l'on va prouver (les *théorèmes*), et des abréviations que l'on souhaite définir.

Objets

La déclaration : `Variable` ⟨`nom`⟩ `:` ⟨`type`⟩.
introduit dans l'environnement un nouvel objet, nommé ⟨`nom`⟩ et de type ⟨`type`⟩.
Par exemple :

`Variable T : Set.`	déclare un nouveau type `T` de termes.
`Variable x : T.`	déclare un habitant `x` dans le type `T`.
`Variable f : T -> T.`	déclare un habitant `f` dans le type `T -> T`.
`Variable p : T -> Prop.`	déclare un nouveau prédicat `p` portant sur les termes de type `T`.

Axiomes

On peut alors supposer certaines propriétés sur les objets introduits. On donne un nom à ces axiomes pour pouvoir les utiliser dans les preuves par la suite. `Axiom` ⟨`nom`⟩ `:` ⟨`énoncé`⟩.
Par exemple :

```
Axiom x_p_init : p x -> forall y:T, p y.
```

Théorèmes

On déclare les théorèmes de la même manière que les axiomes, mais il faut en fournir la preuve. On peut faire débuter la preuve par le mot-clé `Proof` (facultatif). La commande `Qed` (nécessaire) permet de vérifier la preuve à la fin de son développement. `Theorem` ⟨`nom`⟩ `:` ⟨`énoncé`⟩.
Par exemple :

```
Theorem x_p_init_f : p x -> forall y:T, p (f y).
   Proof.
   ... (tactiques) ...
   Qed.
```

Abréviations

Enfin, pour éviter des notations trop lourdes, on peut introduire une *abréviation* et celle-ci peut (ou non) être paramétrée par des variables. `Definition` ⟨`nom`⟩ [`variables`] `:=` ⟨`corps`⟩.
Par exemple :

```
Definition p_init (z:T) (g:T->T) := p z -> forall y:T, p (g y).
Theorem x_p_init_f_2 : p_init x f.
```

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.

Pense-bête

Symbole	Introduction	Elimination
$\forall$	`intro` ou `intros`	`apply` ⟨`Hyp`⟩
$\Rightarrow$	`intro` ou `intros`	`apply` ⟨`Hyp`⟩
$\neg$	`intro`	`destruct` ⟨`Hyp`⟩
$\wedge$	`split`	`destruct` ⟨`Hyp`⟩ `as (H1,H2)`
$\Leftrightarrow$	`split`	`destruct` ⟨`Hyp`⟩ `as (H1,H2)`
$\vee$	`left` ou `right`	`destruct` ⟨`Hyp`⟩ `as [H1|H2]`
$\exists$	`exists` ⟨`term`⟩	`destruct` ⟨`Hyp`⟩ `as (x,H)`
$\top$	`trivial`	$--$
$\bot$	$--$	`exfalso` ou `destruct` ⟨`Hyp`⟩
$t_1 = t_2$	`reflexivity`	`rewrite <-` ⟨`Hyp1`⟩ `in` ⟨`Hyp2`⟩

Les parties en gris sont optionnelles et permettent de nommer les hypothèses que l'on va introduire dans l'environnement. Dans le cas de la réécriture d'une égalité, la flèche vers la gauche permet d'inverser le sens de réécriture et le `in` permet de réécrire l'égalité dans une hypothèse.

Concept	Tactique	Utilisation
hypothèse	`assumption`	permet de conclure quand la conclusion du but courant est également une hypothèse de ce but.
étape	`assert` ⟨`Form`⟩ `as H`	permet d'introduire un résultat intermédiaire ⟨`Form`⟩ qu'on devra prouver avant de l'avoir comme nouvelle hypothèse.
définition	`unfold` ⟨`def`⟩ `in` ⟨`Hyp`⟩	déplie la définition ⟨`def`⟩.
calcul	`simpl in` ⟨`Hyp`⟩	permet d'effectuer un calcul (addition, concaténation...)
inductif	`apply` ⟨`règle`⟩	introduction correspondant à une règle de construction
	`induction` ⟨`Hyp`⟩	application du principe d'induction
	`inversion` ⟨`Hyp`⟩	raisonnement par cas
arithmétique	`omega`	résolution d'(in)équations entre entiers.

Détails des tactiques

— intros

La tactique `intros` applique la règle d'introduction correspondant au connecteur ou quantificateur de la conclusion du but courant. Et elle recommencera, autant de fois que possible, sur la conclusion du but obtenu. Par exemple, si la conclusion du but courant est : $\forall x, p(x) \Rightarrow \exists y, p(y)$, a tactique `intros` va introduire la variable `x` ainsi que l'hypothèse `p x`.

```
...                                          intros.      x : nat
=========================================    ------>      H : p x
 forall x:nat, p x -> exists y:nat, p y                   ============================
                                                           exists y:nat, p y
```

— exists

Pour prouver une formule quantifiée existentiellement comme `exists y:nat, p y` , l'utilisateur doit fournir à Coq à la fois le *témoin* et la preuve que ce témoin vérifie le prédicat `p` (ce qui correspond à la règle d'introduction

de ∃). La tactique `exists` permet de faire cela. Dans le but obtenu précédemment, on peut instancier `y` par `x` dans la formule que l'on cherche à prouver à l'aide de la commande `exists x`. Il restera alors à montrer `p x`.

```
x : nat                                          exists x.          x : nat
H : p x                                          ⟶                  H : p x
============================                                        ============================
 exists y:nat, p y                                                   p x
```

— assumption

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.

```
x : nat
H : p x                                          assumption.
============================                     ⟶              Proof completed.
 p x
```

— apply

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 `Q` et que l'on a en hypothèse une formule `P -> Q` (nommée `H`), alors on peut appliquer cette hypothèse grâce à la commande `apply H`. Il restera alors à prouver `P`.

```
H : P -> Q                                       apply H.           H : P -> Q
============================                     ⟶                  ============================
 Q                                                                   P
```

D'une manière plus générale, la tactique apply peut s'utiliser sur toute hypothèse `H` de la forme $\forall x_1 \dots \forall x_k,\ P_1 \Rightarrow \dots P_n \Rightarrow Q'$. Il est parfois nécessaire de préciser avec quelles valeurs il faut instancier les différentes variables `x_i`.
Ceci peut se faire à l'aide de la commande `apply H with (x_1 := y_1)` (...) `(x_n := y_n)`.

— destruct

La tactique `destruct` permet d'éliminer des conjonctions, disjonctions, négations, contradictions et existentiels en hypothèse. Si c'est une conjonction `H:P1 /\ P2` on se retrouve avec deux hypothèses au lieu d'une seule : une pour `P1` et l'autre pour `P2`.

```
H : P1 /\ P2                                     destruct H.        H1 : P1
============================                     ⟶                  H2 : P2
 Q                                                                  ============================
                                                                     Q
```

Si c'est une disjonction `P1 \/ P2`, on obtient deux sous-buts avec uniquement `P1` ou `P2` en hypothèse.

```
H : P1 \/ P2          destruct H.        H : P1                          H : P2
================      ⟶                  ================      +         ================
 Q                                       Q                               Q
```