Coq

Logiciel pour l'enseignement
23 mai 2016
leo.brunswic.fr/teaching/

Léo Brunswic
leo.brunswic@univ-avignon.fr

Coq est un langage de preuve assisté par ordinateur. Ce langage créé en France par un laboratoire de l'INRIA à des applications qui vont des preuves mathématiques à la certification de programme. Il n'est ni la première initiative de ce genre ni la dernière mais a un certains nombre de succès qui méritent d'attirer l'attention. Tout d'abord,

L'équipe de George Gonthier est un exemple du premier, elle a démontré en 2012 le théorème de Feit-Thomson :

Théorème 1. *Tout groupe simple est de cardinal premier ou pair.*

Ce théorème très complexe a été prouvé une première fois ("à la main") dans les années 1950. C'est un exemple de la force de coq vis à vis de ses prédécesseurs. Le langage arrive à passer à l'échelle, c'est à dire n'est pas limité à la logique la plus élémentaire. L'équipe de Sylvie Boldot est plutot orienté vers la certification de programme, c'est à dire une preuve informatique (vérifiable par ordinateur) qu'un programme donné fait bien ce qu'on attend de lui. La certification programme est un enjeux industriel majeur du XXIe car la complexité des programmes rend leur vérification à la main impossible. Je vous invite à visionner l'excellente conférence : "Pourquoi mon pourquoi mon ordinateur calcule faux ?" .

Le langage Coq est fondé sur le "calcul des constructions" qui est une forme de lambda calcul typé. Sans rentrer dans les détails théoriques, cela a quelques implications pratiques :

- La construction des mathématiques utilisé est intuitionniste, c'est à dire qu'il n'y a pas de tiers-exclu (!) ;
- La fondation des mathématiques n'est pas le concept d'ensemble mais plutot le concept de fonction ou de programme ;
- La récurrence est un élément fondamental.

Pour utiliser Coq, nous allons utiliser le logiciel CoqIDE qui est installé sur vos machines. L'objectif de ce TP est de découvrir des facettes élémentaires des preuves de mathématiques sur ordinateur.

1 Premiers exemples de preuve en Coq

1.1 Un Exemple

Théorème 2 (Déduction)**.**
$\forall A, B,$

$$A \Rightarrow (A \Rightarrow B) \Rightarrow B$$

Démonstration.
Soit A
Soit B
Supposons $(H_1) : A$ vrai.
Supposons $(H_2) : A \Rightarrow B$ vrai.
Appliquer (H_2) à (H_1).
C'est bien ce qu'on cherchait.
CQFD □

```
Theorem deduction :
  forall A B : Prop
       A->(A->B)->B.
Proof.
  intro A.
  intro B.
  intro H1.
  intro H2.
  apply H2 in H1.
  trivial.
Qed.
```

1.2 Comprendre cette preuve

Regardons la preuve de gauche. Nous avons l'habitude de penser cette comme une rédaction correcte mais on peut également la comprendre de la manière suivante. Dans une applications de ce théorème, il faut d'abord que je me donne A et B deux propositions qui permettent d'instancier le résultat dans un cas particulier. Puis je sais qu'auparavant j'ai prouvé A et que j'ai aussi prouvé $A \Rightarrow B$ et donc je peux utiliser l'implication pour prouver B. Lorsque l'on écrit $A \Rightarrow B$, la proposition A n'est ni vraie ni fausse, de même pour B. Par contre on peut voir cette affirmation comme une fonction qui a une preuve de A va me donner une preuve de B.

C'est exactement de cette manière que Coq fonctionne : il voit tout objet comme une fonction. La commande `Theorem` définit une fonction `deduction` qui prend argument deux propositions "A" et "B", puis un troisième argument qui est une preuve de "A" et un quatrième argument qui est une preuve de "$A \Rightarrow B$". La fonction construite doit alors renvoyer une preuve de B. La preuve est elle même est une fonction, `forall A B : Prop` revient à dire que la fonction que l'on veut construire prend deux arguments "A" et "B" de *type* "Proposition". Puis, `A->(A->B)->B.`

La commande `intro A.` prend le premier argument attendu et le met dans le "contexte" en l'appelant "A". Le second `intro B.` fait de même avec le second argument et l'appelle "B". De même, "intro H1" et "intro H2" et prennent les troisième et quatrième arguments, le met dans le contexte et l'appelle "H1".

La commande "apply" demande à coq d'appliquer la fonction $H_2 : A \to B$ à H_1. L'opération est fait "en place", c'est à dire que l'image de H_1 par H_2 est encore appelée H_1. La commande `trivial.` applique des tactiques de décisions que l'on va appeler terminales, c'est à dire qui achèvent la preuve. Ici on pourrait la remplacer par "assumption" qui vérifie que la conclusion voulue est dans le contexte.

Exercice 1. *Démontrer en Coq*

$$(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$$

1.3 Si on a vraiment compris la preuve...

Souvenez-vous bien que $A \Rightarrow B$ en Coq c'est une fonction qui prend en entrée une preuve de A et qui renvoie une preuve de B. Or H_1 est une preuve de A et H_2 est de type $A \Rightarrow B$, c'est donc une fonction ! On se rend compte finalement que tout ce que l'on a fait, c'est construire une fonction qui prend quatre arguments A,B,$H1$ et $H2$ et qui renvoie $H2(H1)$. Je vous invite alors à essayer la preuve suivante

```
Theorem deduction : forall A B : Prop, A->(A->B)-> B.
Proof.
exact( fun A B H1 H2 => H2 H1).
Qed.
```

Définition 1. *Les commandes* `Check x` *et* `Print x` *affichent respectivement le type de* `x` *et la construction de* `x`

Exercice 2. *Tester ces deux commandes sur* `deduction`

1.4 Tactiques classiques, négation et modélisation d'une situation

Les tactiques `left` , `right`, `split` et `destruct` permettent respectivement de prouver les parties gauche et droite d'un "`\/`" dans un but, decouper un "`/\`" ou dans un but en deux buts prouver séparément. Enfin `destruct` permet de casser un `/\` dans le contexte deux.

```
Theorem conjonction : forall A B : Prop, A/\B <-> B/\A.
Proof.
intros A B.
split.
  intro H.
  split.
  destruct H.
  trivial.
  destruct H.
  trivial.
  intro H.
  split.
  destruct H.
  trivial.
  destruct H.
  trivial.
Qed.
```

J'ai utilisé la commande `split` pour casser un `<->`. En effet, `A <-> B` est un raccourci pour `(A -> B)/\(B -> A)`.

Définition 2. *La négation en d'une proposition A en coq s'écrit ˜A. Elle vaut par définition*

$$\tilde{}A := (A \Rightarrow Faux)$$

Si "Faux" est dans le contexte d'une preuve alors le but est automatiquement satisfait, il faut appliquer la tactique "inversion" au "faux" présent dans le contexte.

Exercice 3. *Démontrer les deux propriétés suivantes*

```
forall A : Prop, A -> ~ ~ A

forall A B : Prop, (A \/ ~ B) /\ B -> A
```

Exercice 4. *Il existe en Ecosse un club très fermé qui obéit aux règles suivantes :*

- *Tout membre non écossais porte des chaussettes rouges.*
- *Tout membre porte un kilt ou ne porte pas de chaussettes rouges.*
- *Les membres maries ne sortent pas le dimanche.*
- *Un membre sort le dimanche si et seulement s'il est ecossais.*
- *Tout membre qui porte un kilt est écossais et marié.*
- *Tout membre écossais porte un kilt.*

On souhaite prouver que ce club est si fermé qu'il ne peut accepter personne !

Compléter la définition du modèle et prouver le théorème (c'est plus facile avec le tiers-exclu).

```
Axiom gens : Type.
Axiom membre : gens -> Prop.
Axiom chaussette_rouge : gens -> Prop.
Axiom ecossais : gens -> Prop.
Axiom H1 : forall x : gens,
      membre x -> ~(ecossais x) -> chaussette_rouge x.
Axiom ...

Theorem club_ferme :  forall x : gens, ~(membre x).
........
```

2 Entiers naturels et induction

2.1 Les entiers naturels

Commençons par demander à Coq ce qu'il pense des entiers naturels.

```
Coq <     Print nat.
Inductive nat : Set :=  O : nat | S : nat -> nat
For S: Argument scope is [nat_scope]
```

Je lis : "On définit par récurrence nat de type Set comme étant O qui est de type nat et un constructeur S de type nat → nat". Cette construction correspond exactement aux entiers de Peano.

2.2 Définition d'une fonction récursive et récurrence

On définit l'addition et la relation ≤ par récurrence :

```
Fixpoint plus x y : nat := match x with
| O => y
| S p => S( plus p y)
end.

Inductive le : (nat-> nat -> Prop):=
| le_n : forall x, le x x
| le_S : forall x y, le x y -> le x (S y).
```

Le raisonnement par "récurrence sur x" s'appelle avec la commande `induction x.` Le principe est qu'un entier est un objet qui est obligatoirement construit de la forme `S(S(...S(O))).` Démontrer une propriété par récurrence est un déconstruction de cet entier. Il est soit `O` soit de la forme `S p`.

Exercice 5. *Démontrer que la somme est commutative*

Exercice 6. *Démontrer que tout entier naturel est plus grand que zéro.*

Exercice 7. *Définir la multiplication. Définir "n est pair" de deux manières différentes et démontrer que ces deux définitions sont équivalentes.*