Ingénierie de la preuve

Julien Narboux

Master 1

2010

Plan du cours

Objectifs du cours

1. Découvrir la preuve de programmes et de théorèmes.
2. Aspects pratiques: Coq
3. Aspects théorique: isomorphisme de Curry-Howard, ...

Comment combattre le pire ennemi de l'informaticien: le bug

Logiciels critiques

- Nucleaire
- Transports
- Santé
- ...

Exemple "Ariane Vol 501"

"Because of the different flight path, a data conversion from a 64-bit floating point to 16-bit signed integer value caused a hardware exception (more specifically, an arithmetic overflow, as the floating point number had a value too large to be represented by a 16-bit signed integer). Efficiency considerations had led to the disabling of the software handler (in Ada code) for this error trap, although other conversions of comparable variables in the code remained protected. This caused a cascade of problems, culminating in destruction of the entire flight."

Exemple "Mars Climate Orbiter"

"The 'root cause' of the loss of the spacecraft was the failed translation of English units into metric units in a segment of ground-based, navigation-related mission software."

Un logiciel pour corriger les bugs ?

Problème de l'arrêt

Il n'existe pas de programme permettant de décider si un programme termine ou pas.

Qu'est-ce qu'un bug ?

C'est un programme qui ne respecte pas ses spécifications.

Première idée

Tester !
On écrit une fonction:
Programme + Propriété $\longrightarrow$ Vrai/Faux

Première idée

Tester !
On écrit une fonction:
Programme + Propriété $\longrightarrow$ Vrai/Faux

Mais çà ne suffit pas !

Model Checking

- On réduit la classe des propriétés que l'on va vérifier.

Exemple

- pas d'accés en dehors des tableaux (Airbus A380)

Problème

On a une infinité de cas.

Solution

Réaliser une preuve.

Un raisonnement fini pour traiter une infinité de cas.

Qu'est-ce qu'une preuve ?

- un argument convainquant
- une suite de déductions à partir des axiomes
- un algorithme

Oui mais...

Il peut être difficile de se convaincre qu'une preuve est correcte :

- le nombres de théorèmes

Oui mais...

Il peut être difficile de se convaincre qu'une preuve est correcte :

- le nombres de théorèmes
- présence de calculs

Oui mais...

Il peut être difficile de se convaincre qu'une preuve est correcte :

- le nombres de théorèmes
- présence de calculs
- trop de détails techniques, trop de cas

Oui mais...

Il peut être difficile de se convaincre qu'une preuve est correcte :

- le nombres de théorèmes
- présence de calculs
- trop de détails techniques, trop de cas
- la taille de la preuve

La problème de la vérification d'une preuve

- Le nombres de théorèmes (classification des groupes)
- Présence de calculs
  - Théorème des 4 couleurs (1976 Appel and Haken, formalisé en Coq par Gonthier et Werner)
  - Théorème de Hales (1998, en cours de formalisation en Coq, Isabelle, Hol)
  - ...
- Trop de détails techniques
  - Un compilateur (Compcert: compilateur C prouvé en Coq)
  - Un système d'exploitation (seL4: micro kernel prouvé en Isabelle)
  - Un système de paiement (Gemalto)
- La taille de la preuve (théorème de Fermat-Wiles)
  - Paul Wolfskehl offre 100,000 marks
  - deux conditions: relues par les paires, attendre 2 ans
  - 1907-1908 : 621 tentatives

Une quête de la rigueure

1. Clarifier les *hypothèses*

Une quête de la rigueure

1. Clarifier les *hypothèses*
2. Clarifier ce qu'est une *preuve*

Une quête de la rigueure

1. Clarifier les *hypothèses*
2. Clarifier ce qu'est une *preuve*
3. Etre si précis que l'on a plus besoin de *comprendre* la preuve pour la *vérifier*

Une quête de la rigueure

1. Clarifier les *hypothèses*
2. Clarifier ce qu'est une *preuve*
3. Etre si précis que l'on a plus besoin de *comprendre* la preuve pour la *vérifier*
4. Automatiser des preuves

Par définition vérifier qu'une preuve est correcte est un problème décidable.

On peut donc construire des assistants de preuve.

Exemples

- Coq
- Isabelle
- HOL
-

Qu'est-ce que Coq ?

- un assistant de preuve
- développé et distribué librement par l'INRIA

Il permet de :

- définir des notions mathématiques et/ou des programmes
- démontrer mécaniquement des théorèmes mathématiques mettant en jeu ces définitions

Exemples d'énoncés mathématiques

- Le théorème de pythagore.
- "Si les portes sont ouvertes c'est que la rame est en face d'un quai".

Exemple

On peut par exemple étudier un programme de calcul de la racine carrée sur les entiers:

- Programmation d'une fonction (comme en Ocaml) :

$$\text{sqrt} : \text{int} \longrightarrow \text{int} * \text{int}$$

- Spécification d'un calcul de racine carrée entière

$$\text{sqrt\_prop} : \forall n : \text{int}.0 \leq n \longrightarrow$$

$$\exists s, r : \text{int}.0 \leq s \wedge 0 \leq r \wedge n = s^2 + r \wedge s^2 \leq n < (s+1)^2$$

- Preuve de correction de sqrt vis-à-vis de la spécification sqrt_prop avec n la donnée, $s = \text{fst}\ (\text{sqrt}\ n)$ et $r = \text{snd}\ (\text{sqrt}\ n)$.

Processus de preuve

Les deux étapes du développement d'une démonstration dans Coq sont les suivantes :

- d'abord la construction *interactive* d'une démonstration *par l'utilisateur* ;
- ensuite la vérification *automatique* de la correction de démonstration *par le système*.

L'utilisateur prouve, puis le système vérifie que la preuve est bien correcte.

Ressources disponibles

- Site de l'INRIA consacré à Coq:
  - Téléchargement du logiciel:
    `http://coq.inria.fr/distrib-fra.html`
  - Tutorial Coq:
    `http://coq.inria.fr/doc/tutorial.html`
  - Manuel de référence Coq:
    `http://coq.inria.fr/doc/`
- Livre et recueil d'exercices sur Coq
  - Livre sur Coq par Y. Bertot et P. Castéran
  - Recueil d'exercices (associé au livre) :
    `http://www.labri.fr/perso/casteran/CoqArt/index.html`

Notion de séquent

Les systèmes formels modélisant des logiques utilisent souvent un langage basé sur une structure appelée **séquent**. Il s'agit d'une paire (Γ, F) composée

- d'un multi-ensemble de formules Γ (l'ordre ne compte pas, répétitions possibles) et
- d'une formule F.

Traditionnellement on note cette paire:

$$\Gamma \vdash F$$

Intuitivement, un séquent permet de représenter le fait que des hypothèses Γ, on peut déduire F.

Notation Coq

- En Coq au lieu d'écrire $\{A_1, A_2, \ldots, A_n\} \vdash P$ on écrit:

```
H_1 : A_1
H_2 : A_2

H_n : A_n
______________________________________ (1/1)
P
```

- Parenthésage: $(A \to B \to C)$ signifie $(A \to (B \to C))$.

Déduction naturelle

- On utilise des séquents.
- On ne manipule pas les hypothèses.

Règles pour la logique minimale

$$\frac{}{\Gamma \vdash A} \text{ si } A \in \Gamma$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \text{ Intro } \to \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \text{ Elim } \to$$

Preuve de la formule K

$$\dfrac{\dfrac{A, B \vdash A}{A \vdash B \to A}\,\text{Intro} \to}{\vdash A \to B \to A}\,\text{Intro} \to$$

Preuve de la formule S

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A \to B \to C, A \to B, A \vdash A \to B \to C \qquad A \to B \to C, A \to B, A \vdash A}{A \to B \to C, A \to B, A \vdash B \to C}\,\text{MP} \qquad \ldots\ X\ \ldots}{A \to B \to C, A \to B, A \vdash C}\,\text{MP}}{A \to B \to C, A \to B \vdash A \to C}\,\text{Intro} \to}{A \to B \to C \vdash (A \to B) \to A \to C}\,\text{Intro} \to}{\vdash (A \to B \to C) \to (A \to B) \to A \to C}\,\text{Intro} \to$$

X:

$$\dfrac{A \to B \to C, A \to B, A \vdash A \to B \qquad A \to B \to C, A \to B, A \vdash A}{A \to B \to C, A \to B, A \vdash B}\,\text{MP}$$

Règles du $\wedge$

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ \text{Intro}\ \wedge$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P}\ \text{Elim}\ \wedge\text{g}$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q}\ \text{Elim}\ \wedge\text{d}$$

Règles du $\vee$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q}\ \text{Intro } \vee\text{g}$$

$$\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}\ \text{Intro } \vee\text{d}$$

$$\frac{\Gamma \vdash P \vee Q \qquad \Gamma, P \vdash R \qquad \Gamma, Q \vdash R}{\Gamma \vdash R}\ \text{Elim } \vee$$

Règle du $\neg$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \text{Intro}\ \neg$$

$$\frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ \text{Elim}\ \neg$$

Remarque

On peut voir $\neg A$ comme $A \rightarrow \bot$.

Règles du $\bot$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}\ \text{Elim}\ \bot$$

Règles du $\forall$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x\, A}\ \forall \text{ intro } (x \text{ n'est pas libre dans } \Gamma)$$

$$\frac{\Gamma \vdash \forall x\, A}{\Gamma \vdash A[x \leftarrow t]}\ \forall \text{ elim}$$

Règles du $\exists$

$$\frac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x\, A}\ \exists \text{ intro}$$

$$\frac{\Gamma \vdash \exists x\, A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}\ \exists \text{ elim } (x \text{ n'est pas libre dans } \Gamma \text{ ni } B)$$

Règle Axiome

$$\frac{}{\Gamma \vdash A} \text{ si } A \in \Gamma$$

```
  X : A
============
  A
```

Proof completed.

`assumption` ou `apply X`

Règle Intro $\rightarrow$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\ \text{Intro} \rightarrow$$

```
....                    X : A
=====================   =====================
A -> B                  B
```

`intro X` ou `intros` pour en faire plusieurs.

Règle Elim $\rightarrow$

$$\frac{\Gamma \vdash A \rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \text{Elim} \rightarrow$$

```
                                    ======================
                                    A -> B

....
======================
B

                                    ======================
                                    A

                cut A
```

Règle Intro ∧

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ \text{Intro} \wedge$$

```
....
======================
A /\ B
```

```
======================
A
```

```
======================
B
```

```
split
```

Règle Elim $\wedge$ g

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P}\ \text{Elim } \wedge\text{g}$$

```
....
=====================          =====================
A                              A /\ B
```

```
assert (T: A /  B);[idtac|elim T;intros;assumption].
```

Règle Intro $\vee$ d

$$\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}\ \text{Intro } \vee\text{d}$$

```
....
======================          ======================
A \/ B                          B

                    right
```

Règle Intro $\vee$ g

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q}\ \text{Intro } \vee\text{g}$$

```
....
======================          ======================
A \/ B                          A

                    left
```

Règle Elim $\vee$

$$\frac{\Gamma \vdash P \vee Q \qquad \Gamma, P \vdash R \qquad \Gamma, Q \vdash R}{\Gamma \vdash R}\ \text{Elim} \vee$$

```
....
H : A \/ B
======================
G
```

```
H : A \/ B
H0 : A
======================
G
```

```
H : A \/ B
H0 : B
======================
G
```

```
elim H;intro
```

Règle Elim $\bot$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}\ \text{Elim } \bot$$

```
....
H : False
======================
P
```

Proof completed.

```
elim H
```

Règle Intro $\bot$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \text{Intro}\ \neg$$

```
....                          H : A
======================        ======================
~ A                           False
```

```
intro
```

Règle Elim $\bot$

$$\frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ \text{Elim } \neg$$

```
....
======================
G
```

```
...
======================
A
```

```
...
======================
~ A
```

```
absurd A
```

Quelques tactiques Coq

- `intro`
- `assert`
- `apply`
- `exists`
- `decompose [ex] H`
- `decompose [and] H`
- `decompose [or] H`

Logique classique vs intuitionniste

Les formules suivantes sont des formules valides en logique classique:

tiers exclu $P \vee \neg P$

élimination de la double négation $\neg\neg P \rightarrow P$

loi de Peirce $((P \rightarrow Q) \rightarrow P) \rightarrow P$

Ces propositions impliquent qu'il y a des démonstrations qui ne construisent pas l'objet satisfaisant la proposition prouvée.
Certains mathématiciens ont refusé ces propositions:

- Brouwer,
- Heyting, ...

On dit qu'une preuve est constructive si elle n'utilise pas le tiers exclu.

Propriétés de la logique intuitioniste

Propriété de la disjonction

D'une preuve de $A \vee B$ on peut extraire une preuve de A ou une preuve de B

Propriété du témoin

D'une preuve de $\exists x, A(x)$ on peut extraire un témoin t et une preuve de $A(t)$.

Exemple de preuve classique

Montrons que :

$$\exists x, y \notin \mathbb{Q}, x^y \in \mathbb{Q}$$

Exemple de preuve classique

Montrons que :

$$\exists x, y \notin \mathbb{Q}, x^y \in \mathbb{Q}$$

Considérons $\sqrt{2}^{\sqrt{2}}$.

a. Si $\sqrt{2}^{\sqrt{2}} \in \mathbb{Q}$.
On choisit $x = \sqrt{2}$ et $y = \sqrt{2}$.

b. Sinon $\sqrt{2}^{\sqrt{2}} \notin \mathbb{Q}$.
On choisit $x = \sqrt{2}^{\sqrt{2}}$ et $y = \sqrt{2}$.

$$x^y = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\times\sqrt{2}} = \sqrt{2}^2 = 2 \in \mathbb{Q}$$

En fait, un théorème d'analyse affirme que $\sqrt{2}^{\sqrt{2}}$ est irrationnel et que c'est le cas "a" qu'il faut choisir, mais la démonstration fondée sur le tiers exclu **ne le dit pas**.

Logique propositionnelle classique

On rajoute la règle:

Réduction à l'absurde

$$\frac{\Gamma, \neg P \vdash \bot}{\Gamma \vdash P}\ \text{RAA}$$

Exercice

Montrer que cette règle préserve la validité.

Exercices

1. $\vdash \neg\neg A \rightarrow A$
2. $\vdash A \vee \neg A$
3. $\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg\neg A, \neg A \vdash \neg\neg A} \quad \dfrac{}{\neg\neg A, \neg A \vdash \neg A}}{\neg\neg A, \neg A \vdash \bot}\ \text{elim } \neg}{\neg\neg A \vdash A}\ \text{RAA}}{\vdash \neg\neg A \to A}\ \text{intro } \to$$

$$\dfrac{\neg(A \vee \neg A) \vdash \neg(A \vee \neg A) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \vee \neg A), A \vdash A}}{\neg(A \vee \neg A), A \vdash A \vee \neg A}\ \text{intro } \vee\text{g} \qquad \dfrac{}{\neg(A \vee \neg A), A \vdash \neg(A \vee \neg A)}}{\neg(A \vee \neg A), A \vdash \bot}\ \text{elim } \bot}{\neg(A \vee \neg A) \vdash \neg A}\ \text{intro } \bot}{\neg(A \vee \neg A) \vdash A \vee \neg A}\ \text{intro } \vee\text{d}}{\neg(A \vee \neg A) \vdash \bot}}{\vdash A \vee \neg A}\ \text{RAA}$$

$$\dfrac{\dfrac{\dfrac{}{(A \to B) \to A, \neg A \vdash \neg A} \quad \dfrac{\dfrac{}{(A \to B) \to A, \neg A \vdash (A \to B) \to A} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \neg A, A \vdash A} \quad \dfrac{}{\Gamma, \neg A, A \vdash \neg A}}{(A \to B) \to A, \neg A, A \vdash \bot}\ \text{elim } \neg}{(A \to B) \to A, \neg A, A \vdash B}\ \text{elim } \bot}{(A \to B) \to A, \neg A \vdash A \to B}\ \text{intro } \to}{(A \to B) \to A, \neg A \vdash A}\ \text{elim } \to}{(A \to B) \to A, \neg A \vdash \bot}\ \text{elim } \bot$$

$$\dfrac{\dfrac{(A \to B) \to A, \neg A \vdash \bot}{(A \to B) \to A \vdash A}\ \text{RAA}}{\vdash ((A \to B) \to A) \to A}\ \text{Intro } \to$$

Coq est intuitionniste

Par défaut Coq travaille en logique intuitionniste.
Pour utiliser le tiers-exclu il faut rajouter l'axiome avec la commande:

```
Require Export Classical.
```

Exercice

Montrer que la formule suivante est valide / prouvable :

$$(A \to (B \to C)) \leftrightarrow (A \wedge B \to C)$$

Remarque:
$\to$ est associatif à droite, on aurait pu écrire:

$$(A \to B \to C) \leftrightarrow (A \wedge B \to C)$$

Curryfication

Définition

La curryfication consiste à transformer une fonction qui prend plusieurs arguments en une fonction qui prend un seul argument et qui retourne une fonction qui prend en arguments les arguments restants.

Remarque: cette opération porte le nom de Haskell Curry (1900-1982).

Exemple

En Caml

Au lieu d'écrire:

```
# let f(x,y) = x + y;;
val f : int * int -> int = <fun>
```

On écrit:

```
# let f x = fun y -> x + y;;
val f : int -> int -> int = <fun>
```

ou bien

```
# let f x y = x + y;;
val f : int -> int -> int = <fun>
```

Exemple

En Coq

On utilise la curryfication quand on écrit des fonctions ou des théorèmes.
On écrira plutôt:

```
Lemma toto: forall p q : R, p > 0 -> q > 0 -> p*q > 0.
```

que:

```
Lemma toto: forall p q : R, p > 0 /\ q > 0 -> p*q > 0.
```

Pourquoi ?

Pourquoi utiliser la curryfication ?

Pourquoi ?

Pourquoi utiliser la curryfication ? Afin de pouvoir réaliser des *applications partielles* plus facilement.

Sémantique de Heyting-Kolmogorov

La sémantique de Heyting-Kolmogorov consiste à donner une interprétation fonctionnelle des démonstrations.

- Une preuve de $A \to B$ est une *fonction* qui, à partir d'une preuve de A donne une preuve de B.
- Une preuve de $A \wedge B$ est une *paire* composée d'une preuve de A et d'une preuve de B.
- Une preuve de $A \vee B$ est une paire (i, p) avec ($i = 0$ et p une preuve de A) ou ($i = 1$ et p une preuve de B).
- Une preuve de $\forall x.A$ est une fonction qui, pour chaque objet t construit un objet de type $A[x := t]$.

Cette interprétation consiste à *calculer avec des preuves*. Cela paraît très proche de la programmation fonctionnelle et du λ-calcul.

Exemple

Exemple du point de vue type:

Si H est de type $A \longrightarrow B$ et H_2 est de type A
alors
$H\ H_2$ est de type B.

Exemple

Exemple du point de vue type:

Si H est de type $A \to B$ et H_2 est de type A

alors

$H\ H_2$ est de type B.

Exemple du point de vue preuve:

Si H est une preuve de $A \to B$ et H_2 est une preuve de A

alors

$H\ H_2$ est une preuve de B.

Correspondance de Curry-Howard I

logique	λ-calcul
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$	$\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\text{fun } x : A \mapsto t) : A \to B}$
$\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash t : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash (t\ a) : B}$

Correspondance de Curry-Howard II

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathit{fst}\ t : A}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathit{snd}\ t : B}$$

Correspondance de Curry-Howard III

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \quad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathit{inl}\ a : A + B}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \quad \frac{\Gamma \vdash b : B}{\Gamma \vdash \mathit{inr}\ b : A + B}$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

$$\frac{\Gamma \vdash m : A \vee B \quad \Gamma, x : A \vdash t : C \quad \Gamma, x : B \vdash u : C}{\Gamma \vdash \mathit{case}\ m\ \mathit{of}\ x\ \mathit{in}\ t \mid u : A + B}$$

Règles de simplification

$$
\begin{array}{rcl}
\mathit{fst}(A, B) & = & A \\
\mathit{snd}(A, B) & = & B \\
\mathit{case}\ (\mathit{inl}\ m)\ \mathit{of}\ x\ \mathit{in}\ t|u & = & t[x := m] \\
\mathit{case}\ (\mathit{inr}\ m)\ \mathit{of}\ x\ \mathit{in}\ t|u & = & u[x := m]
\end{array}
$$

Correspondance de Curry-Howard

Logique	λ-calcul
formule	type
preuve	terme
vérification d'une démonstration	vérification de type
normalisation des preuves	β-reduction

Extraction

Le mécanisme d'extraction transforme un λ-terme Coq en un objet Caml.

Retour sur la correspondance de Curry-Howard à travers la curryfication

```
Definition curry A B C (f:(A * B) -> C)
                                (x:A) (y:B) := f(x,y).
```

```
Lemma curry_prop: forall A B C : Type,
          (A * B -> C) -> A -> B -> C.
Proof.
 intros A B C f x y.
 apply f.
 split.
 apply x.
 apply y.
Defined.
```

```
Print curry_prop.
curry_prop =
fun (A B C : Type) (f : A * B -> C)
                   (x : A) (y : B) => f (x, y)
     : forall A B C : Type,
               (A * B -> C) -> A -> B -> C
```

Exemple de construction d'un terme de preuve

Construire une preuve, sous forme d'un terme de la formule suivante: $A \to (A \to B) \to B$.

- En faire une formule close, à savoir $\forall AB : Prop, A \to (A \to B) \to B$.

Exemple de construction d'un terme de preuve

Construire une preuve, sous forme d'un terme de la formule suivante: $A \to (A \to B) \to B$.

- En faire une formule close, à savoir $\forall AB : Prop, A \to (A \to B) \to B$.
- On ne reste plus qu'à construire un terme du λ-calcul dont le type est le suivant : $\forall A\ B : Prop, A \to (A \to B) \to B$.

Exemple de construction d'un terme de preuve

Construire une preuve, sous forme d'un terme de la formule suivante: $A \to (A \to B) \to B$.

- En faire une formule close, à savoir $\forall AB : Prop, A \to (A \to B) \to B$.
- On ne reste plus qu'à construire un terme du λ-calcul dont le type est le suivant : $\forall A\ B : Prop, A \to (A \to B) \to B$.
  - Ce sera une fonction avec pour arguments A, B, H_1 et H_2 et pour corps un terme de type B construit à partir de A, B, H_1 et H_2.
    `fun (A:Prop) (B:Prop) (H1:A) (H2:A->B) => ...:B`

Exemple de construction d'un terme de preuve

Construire une preuve, sous forme d'un terme de la formule suivante: $A \to (A \to B) \to B$.

- En faire une formule close, à savoir $\forall AB : Prop, A \to (A \to B) \to B$.
- On ne reste plus qu'à construire un terme du λ-calcul dont le type est le suivant : $\forall A\ B : Prop, A \to (A \to B) \to B$.
  - Ce sera une fonction avec pour arguments A, B, H_1 et H_2 et pour corps un terme de type B construit à partir de A, B, H_1 et H_2.
    `fun (A:Prop) (B:Prop) (H1:A) (H2:A->B) => ...:B`
  - Une manière de construire un objet de type B est de prendre l'objet H1 de type A et de lui appliquer l'objet (de type fonctionnel) H_2. Cela donne le terme (applicatif) $(H_2\ H_1)$.

Exemple de construction d'un terme de preuve

Construire une preuve, sous forme d'un terme de la formule suivante: $A \to (A \to B) \to B$.

- En faire une formule close, à savoir $\forall AB : Prop, A \to (A \to B) \to B$.
- On ne reste plus qu'à construire un terme du λ-calcul dont le type est le suivant : $\forall A\ B : Prop, A \to (A \to B) \to B$.
  - Ce sera une fonction avec pour arguments A, B, H_1 et H_2 et pour corps un terme de type B construit à partir de A, B, H_1 et H_2.
    `fun (A:Prop) (B:Prop) (H1:A) (H2:A->B) => ...:B`
  - Une manière de construire un objet de type B est de prendre l'objet H1 de type A et de lui appliquer l'objet (de type fonctionnel) H_2. Cela donne le terme (applicatif) $(H_2\ H_1)$.
  - Un terme de preuve possible pour $\forall AB : Prop, A \to (A \to B) \to B$ est donc `fun (A:Prop) (B:Prop) (H1:A) (H2:A->B) => (H2 H1)`.

Exercices

Construire le terme de preuve pour les types suivants:

1. $(A \to B \to C) \to (A \to B) \to A \to C$
2. $(A \to B) \to (B \to C) \to (A \to C)$
3. $A \wedge B \to B \wedge A$
4. $A \vee B \to B \vee A$
5. $A \wedge B \to A \vee B$

Types inductifs

Coq est basé sur un formalisme appelé Calcul des Constructions avec types Inductifs.

Deux points de vue possibles:

- On se donne les entiers.
- On se donne les types inductifs en général (les entiers sont un cas particulier).

Premier exemple (sans récursion)

```
Inductive mois : Set :=
| Janvier   : mois
| Fevrier   : mois
| Mars      : mois
| Avril     : mois
| Mai       : mois
| Juin      : mois
| Juillet   : mois
| Aout      : mois
| Septembre : mois
| Octobre   : mois
| Novembre  : mois
| Decembre  : mois.
```

Définition d'une fonction par filtrage

```
Definition mois_suivant (m:mois) : mois :=
match m with
| Janvier   => Fevrier   | Fevrier   => Mars
| Mars      => Avril     | Avril     => Mai
| Mai       => Juin      | Juin      => Juillet
| Juillet   => Aout      | Aout      => Septembre
| Septembre => Octobre   | Octobre   => Novembre
| Novembre  => Decembre  | Decembre  => Janvier
end.
```

- Comment calculer avec cette fonction ?

- Comment calculer avec cette fonction ?
  `Eval compute in (mois_suivant Mars).`

- Comment calculer avec cette fonction ?
  `Eval compute in (mois_suivant Mars).`
- Comment voir le code de cette fonction ?

- Comment calculer avec cette fonction ?
  `Eval compute in (mois_suivant Mars).`
- Comment voir le code de cette fonction ?
  `Print mois_suivant.`

Définition du mois précédent

```
Definition mois_precedent (m:mois) : mois :=
match m with
| Janvier   => Decembre  | Fevrier   => Janvier
| Mars      => Fevrier   | Avril     => Mars
| Mai       => Avril     | Juin      => Mai
| Juillet   => Juin      | Aout      => Juillet
| Septembre => Aout      | Octobre   => Septembre
| Novembre  => Octobre   | Decembre  => Novembre
end.
```

Premier lemme

```
Eval compute in (mois_precedent (mois_suivant Janvier)).
```

Premier lemme

```
Eval compute in (mois_precedent (mois_suivant Janvier)).

Lemma mois_prec_suiv : forall m:mois,
     mois_precedent (mois_suivant m)=m.
Proof.
intros m.
elim m;
simpl;
reflexivity.
Qed.
```

elim

```
12 subgoals
m : mois
______________________________________(1/12)
mois_precedent (mois_suivant Janvier) = Janvier

______________________________________(2/12)
mois_precedent (mois_suivant Fevrier) = Fevrier

______________________________________(3/12)
mois_precedent (mois_suivant Mars) = Mars

...
______________________________________(12/12)
mois_precedent (mois_suivant Decembre) = Decembre
```

simpl

```
  m : mois
_____________________________________(1/12)
mois_precedent (mois_suivant Janvier) = Janvier
```

$$\longrightarrow \mathit{simpl} \longrightarrow$$

```
  m : mois
_____________________________________
Janvier = Janvier
```

Raisonnement par cas

```
mois_ind : forall P : mois -> Prop,
       P Janvier -> P Fevrier ->  P Mars ->
       P Avril ->   P Mai ->      P Juin ->
       P Juillet -> P Aout ->     P Septembre ->
       P Octobre -> P Novembre -> P Decembre ->
  forall m : mois, P m
```

Aparté: l'égalité en Coq

```
Check eq.
eq  :  forall A : Type, A -> A -> Prop

Check refl_equal.
refl_equal : forall (A : Type) (x : A), x = x
```

- C'est un type polymorphe `(A:Type)`. Cela signifie que l'on a une relation d'égalité générique dont le premier argument est le type des éléments à comparer.
- L'égalité est une relation réflexive, symétrique et transitive. Ces propriétés sont utilisables grâce aux tactiques `reflexivity`, `symmetry` et `transitivity t`.

Egalité de Leibniz

```
Check eq_ind.
  eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
P x -> forall y : A, x = y -> P y
```

Les tactiques pour l'égalité

- `rewrite`
- `rewrite in`
- `replace with`
- `replace with in`
- `subst`

Aparté: Les sortes

Une sorte est un type pour les types.
Les propositions A, B, etc. sont des types (ceux de leurs termes de preuves).
Ces types sont de type `Prop`.
On dit que A, B, etc. sont de sorte `Prop`.
D'un autre coté, les booléens `bool`, les entiers `nat` sont des types dont le type est `Set`.
`Set` et `Prop` sont de type `Type`.

`Set` là où l'on calcule

`Prop` là où l'on raisonne

Les entiers de Peano

1. L'élément appelé zéro et noté: 0, est un entier naturel.
2. Si n est un entier naturel alors son successeur noté S(n) est un entier naturel.

Autre notation: `P ::= O | S P`

Coq vs OCaml

Coq

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

Caml

```
   type nat =
   O
 | S of nat
```

Récursion/Induction sur nat

```
forall P : nat -> Prop,
       P 0 ->
       (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

- Quantification universelle sur une propriété `P: nat -> Prop`
- Conclusion : `forall n : nat, P n`
- 2 cas:
  1. cas de base
  2. récurrence

```
forall n : nat, P n
```

```
----------------
P 0

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

En Coq, un principe d'induction est automatiquement associé à chaque définition inductive.

Autre exemple: les listes

Coq

```
Inductive list : Set :=
  Nil : list
| Cons : nat -> list -> list.
```

Caml

```
type list =
   Nil
 | Cons of int*list
```

Principe d'induction

```
Check list_ind.
```

```
list_ind
  : forall P : list -> Prop,
    P Nil ->
    (forall (n : nat) (l : list), P l -> P (Cons n l)) ->
    forall l : list, P l
```

Les constructeurs sont distincts

Pour cela, on dispose d'une tactique spécifique `discrimate`.

```
Lemma Mars_Fev : Mars<>Fevrier.
intro H; discriminate.
Qed.
```

```
Lemma zero_succ : forall n:nat, ~(S n)=O.
intros n H; discriminate H.
Qed.
```

Les constructeurs sont injectifs

Pour cela, on dispose d'une tactique spécifique `injection`.

```
Lemma test_injection: forall x y, S x = S y -> x=y.
Proof.
intros.
injection H.
intro.
assumption.
Qed.
```

Définition de fonctions

Maintenant qu'on a des structures de données on veut écrire des fonctions qui les manipulent.

Caml	Coq
let	Definition
let rec	Fixpoint

Fonction non récursive

Définition non récursive par filtrage:

```
Definition is_zero : nat -> bool :=
  fun (n:nat) => match n with
       O   => true
     | S _ => false
  end.
```

ou bien

```
Definition is_zero (n:nat) :=
 match n with
       O   => true
     | S _ => false
 end.
```

Fonction récursive: point fixe

Addition des entiers naturels:

```
Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
  | O   => m
  | S p => S (plus p m)
end.

Eval compute in (plus O 56).
Eval compute in (plus 12 67).
Eval compute in (plus 67 12).
```

En Coq les fonctions doivent toujours terminer !

L'annotation {`struct n`} indique l'argument qui décroit à chaque appel récursif. Cela est nécessaire pour pouvoir garantir la terminaison de la fonction.

Exemple

```
Fixpoint bidon (n:nat) : nat := bidon n.
```

Cette définition est refusée par le système et conduit à une erreur

```
Error: Recursive definition of bidon is ill-formed.
```

La profondeur du filtrage peut être supérieur à 1. On peut par exemple définir une fonction pour tester la parité :

```
Fixpoint pair (n:nat) : Prop :=
match n with O => True
| (S O) => False
| (S (S p)) => pair p
end.

Eval compute in (pair 8789).
Eval compute in (pair 8790).
```

Exemple: Fibonacci

```
Fixpoint fib n {struct n}  :=
  match n with
   O => 1
 | S O => 1
 | S ((S n) as n1) => fib n + fib n1
 end.
```

Evaluation des fonctions

A la suite d'une définition par point-fixe un certain nombre de règles de calcul (une par cas du filtrage) sont ajoutées dans le système.

- tactiques `compute`, `simpl` pour faire des réductions dans le but courant.
- `simpl in H`, pour faire la même chose dans une hypothèse plutôt que pour le but courant.

Exemple:

Les réductions pour `plus`:

$$\texttt{plus O m} \longrightarrow_\iota \texttt{m}$$

$$\texttt{plus (S n) m} \longrightarrow_\iota \texttt{S (plus n m)}$$

Preuve de propriétés de programmes

Associativité de l'opération d'addition:

```
forall n m p : nat, (plus (plus n m) p) =
                    (plus n (plus m p))
```

Croissance de la fonction Fibonacci:

```
forall n : nat, fib n <= fib (S n)
```

Associativité de +

Démonstration par induction: `intros n m p; elim n.`

1. Cas de base : `0 + (m + p) = 0 + m + p`
   - étape de simplification : on utilise les règles de calcul pour les fonctions mises en jeu (ici, uniquement +).
   - conclusion de la démonstration : introductions, puis réflexivité de l'égalité
2. Cas de récurrence.

Fibonacci croissante I

```
Lemma fib_croissante : forall n : nat, fib n <= fib (S n).
Proof.
intros n.

assert (fib n <= fib (S n) /\ fib (S n) <= fib (S (S n))).

induction n.

(* cas de base *)
simpl;auto.

(* récurrence *)
assert (forall n : nat, fib (S (S n)) = fib n + fib (S n)).
   simpl; trivial.
rewrite H.
```

Fibonacci croissante II

```
rewrite H.
split.
decompose [and] IHn; clear IHn.
assumption.
decompose [and] IHn; clear IHn.
apply (plus_le_compat).
assumption.
assumption.
(* fin assert *)
decompose [and] H; clear H.
assumption.
Qed.
```

Exercice

Définir une fonction d'itération telle que:

```
iter mois_suivant 12 m = m
```

Correction

```
Fixpoint iter (f:mois->mois) (n:nat) {struct n}
          : mois -> mois :=
  match n with
   | O     => (fun (m:mois) => m)
   | (S p) => (fun (m:mois) => iter f p (f m))
  end.

Lemma mois_12 : forall m : mois, iter mois_suivant 12 m = m.
intros m; case m; simpl; trivial.
Qed.
```

Le type produit

```
Inductive prod (A: Set) (B: Set) : Set :=
  | Couple : A -> B -> prod A B.

  Definition point := prod nat nat.
```

```
  prod_ind
     : forall (A B : Set) (P : prod A B -> Prop),
(forall (a : A) (b : B), P (Couple A B a b)) ->
forall p : prod A B, P p
```

Le $\wedge$

```
Inductive et (A: Prop) (B: Prop) : Prop :=
  | Conj : A -> B -> et A B.
```

```
et_ind
    : forall A B P : Prop,
  (A -> B -> P) -> et A B -> P
```

Le type somme

```
Inductive somme (A: Set) (B: Set) : Set :=
  | inl : A -> somme A B
  | inr : B -> somme A B.
```

```
somme_ind
     : forall (A B : Set) (P : somme A B -> Prop),
       (forall a : A, P (inl A B a)) ->
       (forall b : B, P (inr A B b)) ->
        forall s : somme A B, P s
```

Le $\vee$

```
  Inductive ou (A: Prop) (B : Prop) : Prop :=
| ouintrog : A -> ou A B
| ouintrod : B -> ou A B.
```

```
ou_ind
     : forall A B P : Prop,
  (A -> P) -> (B -> P) -> ou A B -> P
```

Les booléens

```
Inductive bool (A: Prop) (B: Prop) : Set :=
 | left :  A -> bool A B
 | right : B -> bool A B.
```

Le quantificateur existentiel

```
  Inductive ex (A: Set) (P: A -> Prop) : Prop :=
| ex_intro : forall x : A, P x -> ex A P.
```

```
ex_ind
      : forall (A : Set) (P : A -> Prop) (P0 : Prop),
        (forall x : A, P x -> P0) -> ex A P -> P0
```

à finir. . .

Prédicats inductifs

Exemple

```
Inductive le (n : nat) : nat -> Prop :=
   le_n : n <= n
| le_S : forall m : nat, n <= m -> n <= S m.
```

Exemple

```
Definition le (n m: nat) : Prop := exists p:nat, m=n+p.
```