Ingénierie de la preuve

Julien Narboux et Nicolas Magaud

Master 1

2013

Plan du cours I

Plan du cours II

- Framework Map-Reduce

10 L'envers du décor

Objectifs du cours

1. Mieux comprendre ce qu'est une preuve, apprendre à raisonner rigoureusement.
2. Découvrir la preuve de programmes et de théorèmes.
3. Aspects pratiques: Coq
4. Aspects théorique: correspondance de Curry-Howard, ...

Remerciements

Ce support de cours est largement inspiré d'idées, d'exemples ou de cours produits par mes collègues/ancien professeurs (les erreurs sont les miennes):

- Yves Bertot
- Gilles Dowek
- Hugo Herbelin
- Pierre Lescanne
- David Pichardie
- Benjamin Werner
- ...

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

Logiciels critiques

- Nucléaire
- 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."

Coût : $327 600 000.

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 $\rightarrow$ Vrai/Faux

Première idée

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

Mais çà ne suffit pas !

Interprétation abstraite

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

Exemple

2003 Astrée vérifie l'absence d'erreur d'execution (RTE) dans le système de controle de vol de l'Airbus A340, (132,000 lignes de C).

2004 Astrée vérifie l'absence d'erreur d'execution (RTE) dans des systèmes pour l'Airbus A380.

2008 Astrée vérifie l'absence d'erreur d'execution (RTE) dans le système d'accrochage du véhicule automatique de transfert européen (ATV).

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 (correspondance de Curry-Howard)

Oui mais...

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

- le nombre de théorèmes

Oui mais...

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

- le nombre de théorèmes
- présence de calculs

Oui mais...

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

- le nombre 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 nombre 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

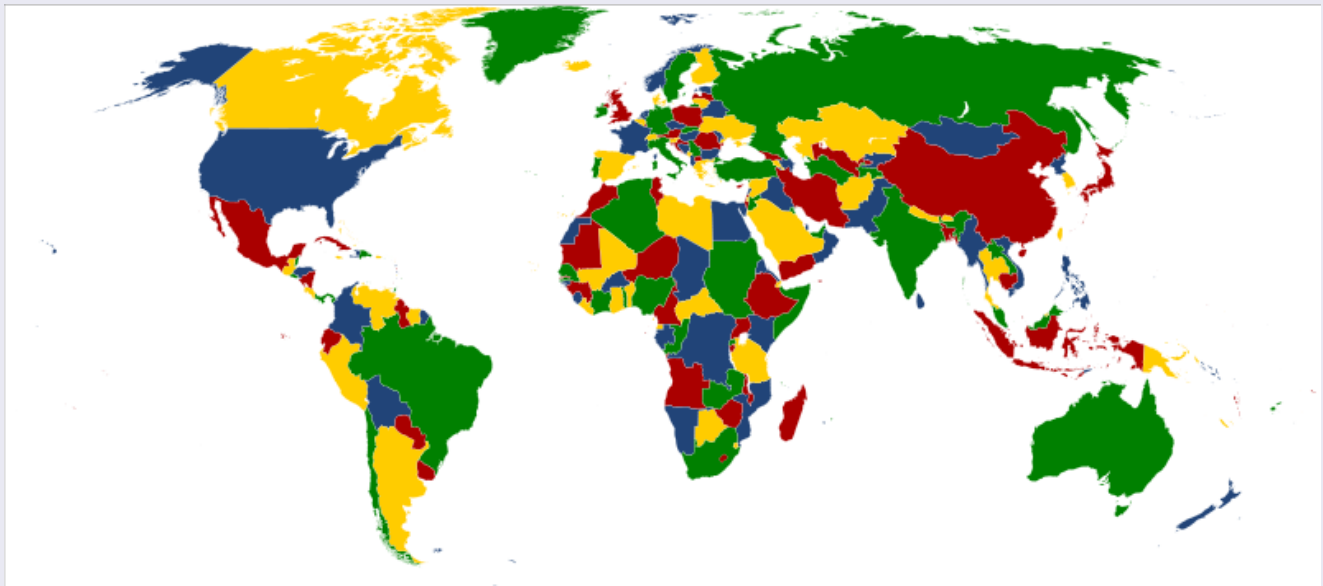
Présence de calculs

Théorème des 4 couleurs

Quatre couleurs suffisent pour colorier une carte géographique *plane* sans que deux pays ayant une *frontière* en commun ne soient de la même couleur.

1976 Appel and Hake (1478 configurations, 1200 heures de calcul)

2004 Formalisation en Coq par Gonthier et Werner



Conjecture de Kepler/Théorème de Hales

Pour un empilement de sphères égales, la densité maximale est atteinte pour un empilement cubique à faces centrées.

1998 Preuve par Thomas Hales

2004 - ? Projet Flyspeck: formalisation du théorème en cours en HOL-light avec des contributions en Coq et Isabelle (plus de 300000 lignes)

Photo par Robert Cudmore

Robert MacPherson, éditeur, écrit que:

> *"The news from the referees is bad, from my perspective. They have not been able to certify the correctness of the proof, and will not be able to certify it in the future, because they have run out of energy to devote to the problem. This is not what I had hoped for. The referees put a level of energy into this that is, in my experience, unprecedented. "*

Le problème de la vérification d'une preuve I

La taille de la preuve

Théorème de Feit-Thompson

```
Theorem Feit_Thompson (gT:finGroupType) (G:{group gT}):
odd ##|G| -> solvable G.
```

Preuve en Coq par Georges Gonthier et son équipe (septembre 2012)[a]:
170 000 lignes, 15 000 définitions, 4 200 théorèmes

[a] http://ssr2.msr-inria.inria.fr/~jenkins/current/progress.html

Théorème de Fermat-Wiles

Il n'existe pas de nombres entiers non nuls , et tels que :

$$x^n + y^n = z^n$$

dès que est un entier strictement supérieur à 2.

- Paul Wolfskehl offre 100,000 marks
- deux conditions: relues par les pairs, attendre 2 ans
- 1907-1908 : 621 tentatives

La problème de la vérification d'une preuve

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)
- Système de pilotage d'une ligne de métro (ligne D à Lyon, ligne 14 à Paris)

Une quête de la rigueur

1. Clarifier les *hypothèses*

Une quête de la rigueur

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

Une quête de la rigueur

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

Une quête de la rigueur

1. Clarifier les *hypothèses*
2. Clarifier ce qu'est une *preuve*
3. Être 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
- PVS
- 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".

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`
  - Manuel de référence Coq:

    `http://coq.inria.fr/doc/`
- Livre et recueil d'exercices sur Coq:
  - *Coq'Art* par Y. Bertot et P. Castéran (en français en ligne, en anglais ou en chinois à la bibliothèque)

    `http://www.labri.fr/perso/casteran/CoqArt/coqartF.pdf`
  - *Software Foundations* par Benjamin C. Pierce, Chris Casinghino, Michael Greenberg,Vilhelm Sjöberg, Brent Yorgey

    `http://www.cis.upenn.edu/~bcpierce/sf/`
  - D'autres cours en ligne sont disponibles sur moodle.

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
```

Syntaxe

Logique	Coq
$\bot$	`False`
$\top$	`True`
$a = b$	`a = b`
$a \neq b$	`a <> b`
$\neg A$	`~ A`
$A \vee B$	`A \/ B`
$A \wedge B$	`A /\ B`
$A \Rightarrow B$	`A -> B`
$A \Leftrightarrow B$	`A <-> B`
$f(x, y, z)$	`(f x y z)`
$\forall xy, P(x, y)$	`forall (x y:A), P x y`
$\exists xy, P(x, y)$	`exists (x:A) (y:B), P x y`

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 $A \to \bot$.

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 $\exists$

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

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

Convention de nommage

Remarque

Les règles sont nommées en les lisant *de haut en bas*.

Résumé

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

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q}\ \text{Intro} \vee\text{g} \qquad \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}\ \text{Intro} \vee\text{d} \qquad \frac{\Gamma \vdash P \vee Q \qquad \Gamma, P \vdash R}{\Gamma \vdash R}$$

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

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

$$\frac{\Gamma \vdash \exists x\, A(x) \qquad \Gamma, A(x) \vdash B}{\Gamma \vdash B}\ \exists elim\ x \notin FV(\Gamma) \cup FV(B) \qquad \frac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x\, A}\ \exists\ \text{intro}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x\, A}\ \forall\ \text{intro}\ (x \notin FV(\Gamma)) \qquad \frac{\Gamma \vdash \forall x\, A}{\Gamma \vdash A[x \leftarrow t]}\ \forall\ \text{elim}$$

Liens entre les tactiques Coq et les règles d'inférence

Etant donnée une règle d'inférence R comportant n prémisses de la forme:

$$\frac{\Gamma_1 \vdash P_1 \quad \ldots \quad \Gamma_n \vdash P_n}{\Gamma \vdash G}\ \mathrm{R}$$

L'application de la tactique Coq correpondante consiste à transformer le but:

$$\Gamma \vdash G$$

en n sous buts:

$$\Gamma_1 \vdash P_1$$

$$\ldots$$

$$\Gamma_n \vdash P_n$$

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.
```

Variante Règle Elim $\rightarrow$

$$\frac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}$$

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

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

```
assert (A).
```

Variante Elim $\rightarrow$

$$\frac{\Gamma, H : A \rightarrow B \vdash A}{\Gamma, H : A \rightarrow B \vdash B}$$

```
H: A -> B                          H: A -> B
======================             ======================
B                                  A
```

```
apply H.
```

Variante Elim $\rightarrow$

$$\frac{\Gamma, H : H_1 \rightarrow H_2 \rightarrow B \vdash A}{\Gamma, H : A \rightarrow B \vdash B}$$

```
H: H1 -> H2 -> B
======================
B
```

```
H: H1 -> H2 -> B
======================
H1
```

```
H: H1 -> H2 -> B
======================
H2
```

```
apply H.
```

Règle Intro ∧

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \text{Intro} \wedge$$

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

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

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

```
split.
```

Règle Elim $\wedge$ g

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \text{Elim}\ \wedge\text{g}$$

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

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

Règle Intro $\vee$ d

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \text{Intro } \vee\text{d}$$

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

```
right.
```

Règle Intro $\vee$ g

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \text{Intro } \vee\text{g}$$

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

```
left.
```

Règle Elim $\vee$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash G \qquad \Gamma, B \vdash G}{\Gamma \vdash G}\ \text{Elim } \vee$$

$$\frac{\Gamma, H : A \vee B, A \vdash G \qquad \Gamma, H : A \vee B, B \vdash G}{\Gamma, H : A \vee B \vdash G}$$

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

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

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

elim H;intro. ou bien destruct H ou bien decompose [or] H.

Variante Elim $\wedge$

$$\frac{\Gamma, H : A \wedge B, H_0 : A, H_1 : B \vdash G}{\Gamma, H : A \wedge B \vdash G}$$

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

```
H : A /\ B
H0 : A
H1 : B
======================
G
```

`elim H;intro.` ou bien `destruct H` ou bien `decompose [and] H.`

Règle Elim $\bot$

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

```
....                          ....
======================        ======================
P                             False
```

```
exfalso.
```

Règle Elim $\bot$

$$\dfrac{\dfrac{}{\Gamma, H : \bot \vdash \bot}\ \text{Ax}}{\Gamma, H : \bot \vdash P}\ \text{Elim } \bot$$

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

Proof completed.

```
elim H.
```

Règle Intro ¬

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

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

```
intro.
```

Règle Elim $\bot$

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

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

....
======================

G

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

```
absurd A.
```

Quelques tactiques Coq

- `intro` (introduire dans le context des hypothèses)
- `assert` (prétendre qu'une proposition est vraie)
- `apply` (applique un théorème)
- `exists` (fournir un témoin pour prouver un 'exists')
- `decompose [ex] H` (exhibe les témoins des 'exists' dans l'hypothèse H)
- `decompose [and] H` (coupe les 'et' de H)
- `decompose [or] H` (raisonne par cas sur les 'ou' de H)
- `unfold t` déplie une définition
- `simpl`
- `reflexivity`, `symmetry`, `transitivity`
- `rewrite`, replace ... with

Composition de tactiques

`tac0; tac1` applique la tactique `tac0` sur le but courant, puis la tactique `tac1` sur les n sous-buts engendrés par la tactique `tac0`

Quelques notations

Logique	Coq
$\wedge$	`/\`
$\vee$	`\/`
$\neg$	`~`
$\Rightarrow$	`->`
$\Leftrightarrow$	`<->`
$\forall$	`forall`
$\exists$	`exists`

Parenthèses: L'implique est parenthèsé à droite par défaut: $(A \to B \to C)$ signifie $(A \to (B \to C))$.

Utilisation de ∀

Quantifier sur les termes

```
forall n : nat, n >= 0
```

Utilisation de ∀

Quantifier sur les termes

```
forall n : nat, n >= 0
```

Quantifier sur les propositions

```
forall P Q : Prop, P \/ Q -> Q \/ P
```

Utilisation de ∀

Quantifier sur les termes

```
forall n : nat, n >= 0
```

Quantifier sur les propositions

```
forall P Q : Prop, P \/ Q -> Q \/ P
```

Quantifier sur les prédicats

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

Utilisation de ∀

Quantifier sur les termes

```
forall n : nat, n >= 0
```

Quantifier sur les propositions

```
forall P Q : Prop, P \/ Q -> Q \/ P
```

Quantifier sur les prédicats

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

Quantifier sur les types

```
eq_trans : forall (A : Type) (x y z : A), x = y /\ y = z -> x=
fst : forall A B : Type, A * B -> A
```

Prop vs bool

Prop est le type des propositions que l'on prouve.
Exemples: `True`, `False`, `2<3`, `forall x, x=x`, `leb 2 3 = true`.

bool est le type des booléens utilisables dans des programmes (dans des `if`).
Exemples: `true`, `false`, `leb 2 3`

```
Compute (leb 2 3).
     = true
     : bool
Compute (2 <= 3).
     = 2 <= 3
     : Prop
```

On peut prouver que: `forall b: bool, b=true \/ b=false`.
Mais on n'admet pas toujours que: `forall P, P \/ ~P`

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 $\neg\neg P \to P$

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

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 "b" qu'il faut choisir, mais la démonstration fondée sur le tiers exclu **ne le dit pas**.

On rajoute la règle:

Réduction à l'absurde

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

Remarque

On aurait pu écrire:

$$\frac{\Gamma \vdash \neg\neg P}{\Gamma \vdash P}\ \text{RAA'}$$

Exercice

Montrer que cette règle préserve la validité classique.

Exercices

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

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

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

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

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

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 $A \to B$ et H' est de type A
alors
$H\ H'$ est de type B.

Exemple

Exemple du point de vue type:

Si H est de type $A \to B$ et H' est de type A
alors
$H\ H'$ est de type B.

Exemple du point de vue preuve:

Si H est une preuve de $A \to B$ et H' est une preuve de A
alors
$H\ H'$ est une preuve de B.

Correspondance de Curry-Howard I

logique	programmation
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$	$\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\text{fun } x : A \mapsto t) : A \to B}$
$\dfrac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B}$

Correspondance de Curry-Howard I

logique	programmation
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$	$\dfrac{\Gamma, \quad A \vdash \quad B}{\Gamma \vdash \qquad\qquad A \to B}$
$\dfrac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash \quad A \to B \quad \Gamma \vdash \quad A}{\Gamma \vdash \quad B}$

Correspondance de Curry-Howard I

logique	programmation
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$	$\dfrac{\Gamma, \quad A \vdash \quad B}{\Gamma \vdash \qquad\qquad A \to B}$
$\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash \quad A \to B \quad \Gamma \vdash \quad A}{\Gamma \vdash \quad\quad B}$

Correspondance de Curry-Howard II

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

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

Correspondance de Curry-Howard II

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

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

Correspondance de Curry-Howard II

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

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_1(t) : A}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_2(t) : B}$$

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

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

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

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

$$\frac{\Gamma \vdash m : A \vee B \quad \Gamma, x : A \vdash t : C \quad \Gamma, x : B \vdash u : C}{\Gamma \vdash \mathit{case}\ m\ \mathit{of}\ \mathit{inl}(a) => t \mid \mathit{inr}(a) => u : C}$$

Règles de simplification

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

Correspondance de Curry-Howard

Logique	**λ-calcul/programmation**
formule	type
preuve	terme/programme
vérification d'une démonstration	vérification de type
normalisation des preuves	β-réduction

Extraction

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

Exemple:

```
Inductive list (A : Type) :=
  nil : list A
| cons : A -> list A -> list A

Definition length (A : Type) :=
 fix length l :=
 match l with
  | nil => O
  | _ :: l' => S (length l')
 end.
```

```
type 'a list =
| Nil
| Cons of 'a * 'a list


let rec length = function
| Nil -> O
| Cons (y, l') ->
            S (length l')
```

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.
```

On obtient:

```
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
```

```
Lemma uncurry_prop : forall A B C : Type,
(A -> B -> C) -> (A * B -> C).
Proof.
intros A B C f p.
elim p.
intros a b.
apply f.
apply a.
apply b.
Defined.

Extraction uncurry_prop.
```

On obtient:

```
 let uncurry_prop f = function
| x,x0 -> f x x0
```

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

On se donne:

```
and_ind   : forall A B P : Prop,
              (A -> B -> P) -> A /\ B -> P
conj      : forall A B : Prop, A -> B -> A /\ B
or_ind    : forall A B P : Prop,
              (A -> P) -> (B -> P) -> A \/ B -> P
or_introl : forall A B : Prop, A -> A \/ B
or_intror : forall A B : Prop, B -> A \/ B
```

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. $A \wedge B \to A \vee B$
5. $A \vee B \to B \vee A$

Types inductifs

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

Les définitions inductives consistent à:

- se donner des éléments de base
- et des règles pour construire de nouveaux éléments à partir d'éléments déjà connus.

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
```

$$\rightarrow \textit{simpl} \rightarrow$$

```
  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
```

Autre exemple:

```
Inductive bool : Type :=
  | true : bool
  | false : bool.
```

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.
```

Listes polymorphes

```
Inductive list (A:Set) : Set :=
  nil  : list A
| cons : A -> list A -> list A.
```

`nil` et `cons` ont un *paramètre* supplémentaire : A:Set.
Pour définir la liste `[1;2]` on obtient:

```
cons nat 1 (cons nat 2 (nil nat )).
```

Coq autorise les *arguments implicites*:

```
Implicit Arguments nil.
Implicit Arguments cons.
```

On peut alors écrire:

```
 cons 1 (cons 2 nil ).
```

Et avec les notations:

```
 1::2::nil.
```

Prédicats inductifs

Un prédicat inductif est une propriété définie inductivement.

Exemple: parité

Définition inductive

```
Inductive even : nat -> Prop :=
  | even_O : pair 0
  | even_SS : forall m : nat, even m -> even (S (S m)).
```

Définition classique

```
Definition even n := div2 n = div2 (S n)
```

Définitions mutuellement récursives

Exemple

```
Inductive even : nat -> Prop :=
  | even_O : even 0
  | even_S : forall n, odd n -> even (S n)
with odd : nat -> Prop :=
    odd_S : forall n, even n -> odd (S n).
```

Exemple: $\leq$

Définition inductive

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

Définition classique

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

Raisonnement par cas

inversion

```
Lemma toto : forall n,
le n 1 -> n = 1 \/ n =0.
Proof.
intros.
inversion H.
left.
reflexivity.
inversion H1.
right.
reflexivity.
Qed.
```

Deux cas:

```
 2 subgoals
n : nat
H : le n 1
H0 : n = 1
______________________________________(1/2)
1 = 1 \/ 1 = 0

n : nat
H : le n 1
m : nat
H1 : le n 0
H0 : m = 0
______________________________________(2/2)
n = 1 \/ n = 0
```

inversion_clear

```
Lemma toto : forall n,
le n 1 -> n = 1 \/ n =0.
Proof.
intros.
inversion_clear H.
left.
reflexivity.
inversion_clear H0.
right.
reflexivity.
Qed.
```

Conseils

- Quand un argument peut être représenté sous forme d'un paramètre, il vaut mieux le faire cela rend les inductions plus simples.

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

vs

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

- Il est parfois utile d'avoir plusieurs versions d'une même définition inductive, cela permet d'avoir plusieurs schémas de preuve possibles.
- Tester vos prédicats sur des cas positifs et négatifs.

Exercice: clôture transitive-réflexive d'une relation R

Première Définition

$$\frac{}{a\ R^*\ a} \qquad \frac{a\ R\ b}{a\ R^*\ b} \qquad \frac{a\ R^*\ b \qquad b\ R^*\ c}{a\ R^*\ c}$$

Seconde Définition

$$\frac{}{a\ R^{\#}\ a} \qquad \frac{a\ R\ b \qquad b\ R^{\#}\ c}{a\ R^{\#}\ c}$$

1. Donner des prédicats inductifs correspondants à ces 2 définitions.
2. Montrer que si R est symétrique alors R^* aussi.
3. On veut montrer que les deux définitions sont équivalentes.
   1. Montrer que $\forall a\ b,\ a\ R^{\#}\ b \rightarrow a\ R^*\ b$.
   2. Montrer que $\forall a\ b,\ a\ R\ b \rightarrow a\ R^{\#}\ b$.
   3. Montrer que $R^{\#}$ est transitive.
   4. En déduire que $\forall a\ b,\ a\ R^*\ b \rightarrow a\ R^{\#}\ b$.

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).
```

Terminaison

En Coq les fonctions doivent toujours terminer !

L'annotation {`struct n`} indique l'argument qui décroît structurellement à chaque appel récursif. Les appels récursifs se font sur des sous-termes stricts. Cela permet de 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.
```

Sinon on aurait...

...une contradiction !

```
Fixpoint toto (b :bool) : bool := negb (toto b).
```

On a `toto true = negb (toto true)`

- Si `toto true = false` alors `toto true = true`.
- Si `toto false = true` alors `toto false = false`.

Totale

En Coq les fonctions doivent toujours être totales !

Solution

Sinon on peut utiliser le type `option`:

```
Inductive option (A:Type) : Type :=
  | Some : A -> option A
  | None : option A.
```

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 p) as p1) => fib p + fib p1
 end.
```

Évaluation 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`, `vm_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.

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.
```

Terminaison

Merge sort

```
Fixpoint merge (l1:list nat) (l2: list nat) :=
 match (l1,l2) with
| (nil,_) => l2
| (_,nil) => l1
|(a::m1,b::m2) => if (le_lt_dec a b) then a::merge m1 l2
                  else  b::merge l1 m2
end.
```

Cette définition n'est pas acceptée.

Error: Cannot guess decreasing argument of fix.

Première solution

On utilise un compteur:

```
Fixpoint merge (l1:list nat) (l2: list nat) (n:nat):=
 match (l1,l2,n) with
| (nil,_,_) => l2
| (_,nil,_) => l1
|(_,_,0) => nil
|(a::m1,b::m2,S p) => if (le_lt_dec a b) then a::merge m1 l2
                else  b::merge l1 m2 p
end.
```

Inconvenients

- Il faut calculer la longueur des listes.
- Les preuves utilisant la fonction doivent prouver que le compteur a bien été initialisé.

Deuxième solution

On utilise Function:

```
Require Export Recdef.

Definition length_sum (c:list nat *  list nat) :=
   length (fst c) + length (snd c).

Function merge' (c:list nat*list nat)
                {measure length_sum} : list nat :=
  match c with
 | (nil,l2) => l2
 | (l1,nil) => l1
 |(a::m1 as l1,b::m2 as l2) => if (le_lt_dec a b)
                then a::merge' (m1,l2)
                else  b::merge' (l1,m2)
end.
```

Obligations de preuve

Il faut prouver que:

```
length_sum (l0, l3) < length_sum (a :: l0, b :: l3)
```

La preuve:

```
Proof.
intros.
unfold length_sum.
simpl.
omega.
intros.
unfold length_sum.
simpl.
omega.
Qed.
```

Et quand on ne sait pas si çà termine ?

On peut utiliser un *prédicat* pour modéliser la fonction (comme en Prolog):

Exemple: Siracuse

$$U_0 = N \qquad U_{n+1} = \begin{cases} \frac{U_n}{2} & \text{si } U_n \text{ est pair} \\ 3U_n + 1 & \text{si } U_n \text{ est impair} \end{cases}$$

```
 Inductive syracuse (N:nat) : nat -> nat -> Prop :=
  done : syracuse N 0 N
| even_case : forall n p,even p ->
   syracuse N n p -> syracuse N (S n) (div2 p)
| odd_case : forall n p , odd p ->
   syracuse N n p -> syracuse N (S n) (S(p+p+p)).
```

Siracuse :utilisation

```
Lemma essai: syracuse 15 1 46.
replace (46) with (S(15+15+15)) by reflexivity.
apply odd_case.
repeat constructor.
constructor.
Qed.
```

Quelques fonctions incontournables sur les listes

1. filter
2. map
3. fold/reduce

Filter

La fonction `filter` prend en argument un prédicat et une liste et renvoie la liste des éléments de la liste qui vérifie le prédicat.

```
Fixpoint filter {X:Type}(test: X->bool)(l:list X)
                                          :(list X) :=
  match l with
  | [] => []
  | h :: t => if test h then h :: (filter test t)
                        else filter test t
  end.
```

Exemple

```
Eval simpl in (filter (fun x => leb x 3) (1::20::3::nil)).
 = 1 :: 3 :: nil : list nat
```

Map

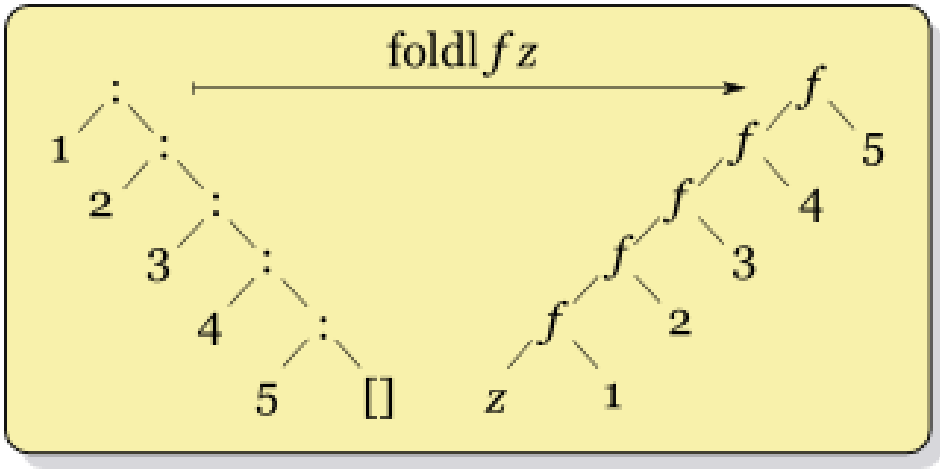
La fonction `map` applique une fonction à chaque élément d'une liste:

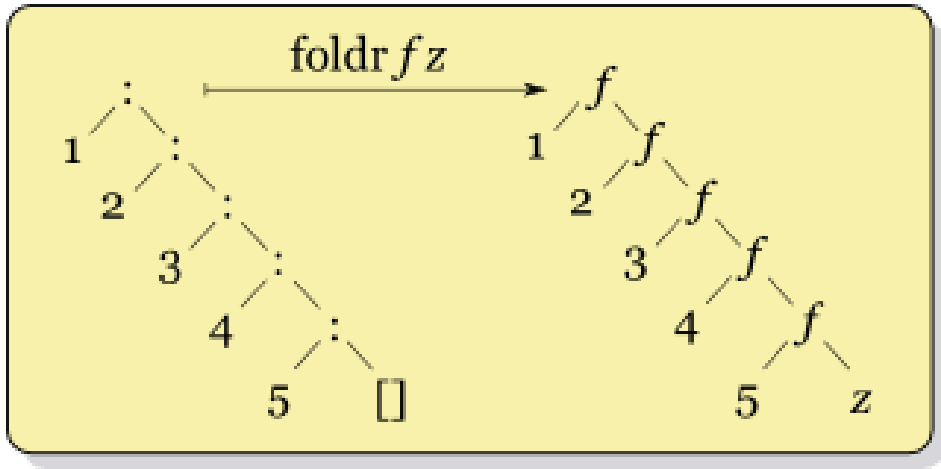
```
Fixpoint map {X Y:Type} (f:X->Y) (l:list X):(list Y):=
  match l with
  | [] => []
  | h :: t => (f h) :: (map f t)
  end.
```

Exemple

```
Eval simpl in (map (plus 2) (1::2::3::4::nil)).
 = 3 :: 4 :: 5 :: 6 :: nil : list nat
```

Fold





Intuitivement, la fonction `fold` consiste à insérer un opérateur binaire entre chaque paire d'éléments consécutifs d'une liste.

```
Fixpoint foldr {X Y:Type} (f: X->Y->Y) (l:list X) (b:Y):Y :=
  match l with
  | nil => b
  | h :: t => f h (foldr f t b)
  end.
```

Fold

Exemple

```
Eval simpl in (fold plus (1::2::3::4::nil) 0).
 = 10 : nat
```

Aparté: Framework Map-Reduce

Ces fonctions ont inspirées le framework map/reduce de Google ou Hadoop de la fondation Apache:

> *"Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical record in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately."*
>
> *Jeffrey Dean and Sanjay Ghemawat*

Exemple I

Compter le nombre d'occurences d'une chaîne de caractère s dans un ensemble de documents:

```
Definition sum l := fold_left plus l 0.

Definition compte_occs s docs :=
        sum (map (compte_occ s) docs).
```

En Coq, certains connecteurs logiques sont primitifs et d'autres sont définis par des types inductifs:

Définis:

- $\top$
- $\bot$
- $\wedge$
- $\vee$
- $\exists$
- $\neg A \equiv A \Rightarrow \bot$

Primitifs:

- $\forall$
- $\Rightarrow$

Vrai

```
Inductive True : Prop :=
  I : True.

True_ind
     : forall P : Prop, P -> True -> P
```

Vrai

```
Inductive True : Prop :=
  I : True.

True_ind
     : forall P : Prop, P -> True -> P
```

Inutile...

Faux

```
Inductive False : Prop := .

False_ind
     : forall P : Prop, False -> P
```

Exemple

```
Theorem ex_falso_quodlibet : forall (P:Prop),
   False -> P.
Proof.
  intros P F.
  inversion F.
Qed.
```

Et

```
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
```

Ou

```
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
```

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
```

Égalité

```
Inductive eq (A:Type) (x:A) : A -> Prop :=
   refl_equal : eq A x x.


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