

# Preuves formelles en Coq

LOÏC POTTIER

Notes de cours<sup>1</sup> du DEA de mathématiques,  
Université de Nice-Sophia Antipolis,  
janvier 2003

---

1. Ces notes sont en évolution constante. Elles sont inspirées du cours de DEA de Gilles Dowek et du cours de DEA de Christine Paulin-Mohring et Benjamin Werner. Elles ont été écrites à l'aide du logiciel  $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ ([www.texmacs.org](http://www.texmacs.org)).



# Table des matières

<b>1 Le langage de Coq.</b>	4
1.1 Les objets et la syntaxe.	4
1.1.1 Les types.	4
1.1.2 Les sortes.	5
1.1.3 Les fonctions.	5
1.1.4 Les formules logiques, leurs preuves, et l'isomorphisme de Curry-Howard.	5
1.2 Le typage.	6
1.2.1 Règles de typage informelles.	6
1.2.2 Les règles de typage.	7
1.2.2.1 Les contextes.	7
1.2.2.2 Les jugements de typage.	7
1.2.2.3 Les règles.	7
1.2.2.4 Exemple.	8
1.2.3 Quelques remarques sur les règles de typage des produits.	8
1.3 Le calcul.	8
1.3.1 La $\beta$ -réduction	8
1.3.2 La consistance.	9
1.3.3 Logique intuitionniste et mathématiques constructives.	10
1.3.4 Les définitions et la $\delta$ -réduction.	11
1.3.5 La conversion.	12
1.3.6 Exemple: les codages imprédictifs.	12
1.4 Les types inductifs.	13
1.4.1 Exemples.	14
1.4.1.1 Les entiers naturels.	14
1.4.1.2 L'égalité de Leibniz.	14
1.4.2 Raisonnement et calcul par cas: la construction Cases.	15
1.4.3 La $\iota$ -réduction.	15
1.4.4 Fonctions récursives: la construction Fix.	16
1.4.4.1 Un exemple.	16
1.4.4.2 Un exemple de récursion mutuelle.	16
1.4.4.3 Le cas général.	17
1.4.5 La $\iota$ -réduction, bis.	17
1.4.6 Typage des Cases et des Fix.	18
<b>2 Introduction au système Coq.</b>	19
2.1 Commandes élémentaires.	19
2.1.1 Lancer coq depuis un shell	19
2.1.2 Obtenir le type d'un terme	19
2.1.3 Définir une constante	19
2.1.4 Obtenir la valeur d'une constante	20
2.1.5 Calculer.	20
2.1.5.1 Fonctions récursives.	20
2.1.6 Démontrer.	21
2.1.6.1 $\forall A, A \rightarrow A$	21
2.1.6.2 $\forall A, A \rightarrow \neg\neg A$	21
2.1.6.3 Calcul des propositions.	22
2.1.6.4 Calcul des prédicats.	25
<b>3 Exemples</b>	28
3.1 L'ensemble des parties est plus gros que l'ensemble.	28

# Chapitre 1

## Le langage de Coq.

### 1.1 Les objets et la syntaxe.

Coq permet de représenter des objets mathématiques, des formules décrivant leurs propriétés, ainsi que des démonstrations de ces formules. Il permet aussi de calculer sur ces objets. Ces objets, formules, preuves sont appelés des *termes*. Leur ensemble constitue le *calcul des constructions inductives* (CCI en abrégé). Les termes du CCI sont peu différents des expressions mathématiques courantes, dans leur contenu et dans leur syntaxe. Mais ils sont représentés avec une syntaxe plutôt exotique en Coq, en raison de son interface ASCII. Ils ont les formes suivantes:

- les *constantes* et les *variables*, désignées par des noms:  $x, y, \dots, \text{id}, \text{nat}, \dots$   
ce sont des mots commençant par une lettre, comprenant des lettres, chiffres, et les caractères `_` et `'`
- les *applications* de fonctions à des arguments:  $f(a_1, \dots, a_n)$  sera noté  $(f a_1 \dots a_n)$  en Coq
- les définitions de fonctions, ou *abstractions*:  
la fonction  $x \mapsto t(x)$ , où  $x$  est dans  $A$ , sera notée  $\lambda x: A. t(x)$  dans le CCI,  
et  $[x: A]t(x)$  en Coq.
- les *produits*. Ce sont
  - des produits d'ensembles indexés par un ensemble  $I$ :  $\prod_{i \in I} E_i$ , noté  $(i: I)E$  dans Coq.  
Si  $i$  n'apparaît pas dans l'expression  $E$ , on note aussi ce produit  $I \rightarrow E$
  - des quantifications universelles:  $\forall x: A, P$ , noté  $(x: A)P$  dans Coq.  
De même, si  $x$  n'apparaît pas dans  $P$ , on note aussi ce produit  $A \rightarrow P$

En fait, pour le CCI et Coq, il n'y a pas de différence entre ces deux types de produits, et ils sont notés de la même façon:

on les notera avec la quantification universelle  $\forall x: A, P$  pour le CCI, et avec  $(x: A)P$  pour Coq.

- les *types inductifs*, *traitements par cas* et les *fonctions récursives*: on n'abordera ces objets qu'à la section 1.4, ils sont un peu complexes.

Chaque objet du CCI possède un type. Le fait que  $t$  est de type  $E$  est noté  $t: E$ .

#### Abus de notation:

- les implications imbriquées  $A \rightarrow (B \rightarrow (C \rightarrow D))$  seront notées  $A \rightarrow B \rightarrow C \rightarrow D$ . On dit que  $\rightarrow$  associe à droite.
- la quantifications successives  $\forall x: A, \forall y: A, \forall z: B, P$  seront notées  $\forall x, y: A, z: B, P$ .
- les abstractions successives  $\lambda x: A. \lambda y: A. \lambda z: B. t$  seront notées  $\lambda x, y: A, z: B. t$

#### 1.1.1 Les types.

Les types du CCI sont des termes qu'on peut voir comme des ensembles, et la relation  $t: E$  peut être vue comme  $t \in E$ . Cette intuition est pratique, mais a ses limites, qu'on verra par la suite. Par exemple, l'ensemble des entiers de Peano est représenté par le type `nat` de Coq, et contient <sup>1</sup>: on a  $O: \text{nat}, S(O): \text{nat}$ , etc.

Plus généralement, chaque objet de Coq a un type:

$$O: \text{nat}, \text{nat}: \text{Set}, \text{Set}: \text{Type}_0, \text{Type}_0: \text{Type}_1, \dots$$

### 1.1.2 Les sortes.

Les types qui sont des types de types sont appelés des *sortes*. Les sortes du CCI sont Set et Prop, tous deux de type  $\text{Type}_0$ , lui-même de type  $\text{Type}_1$ , etc.

- Prop représente l'ensemble des propositions logiques.
- on peut voir Set comme le type des ensembles calculatoires (nombres entiers, rationnels, etc).
- quant aux  $\text{Type}_i$ , aussi appelés *univers*, ils peuvent être vus comme les types des ensembles en général.

On a de plus les inclusions suivantes

$$\text{Prop}, \text{Set} \subset \text{Type}_0 \subset \text{Type}_1 \dots$$

La suite  $(\text{Type}_i)_{i \in \mathbb{N}}$  permet d'éviter un paradoxe du type de celui de l'ensemble des ensembles<sup>2</sup> qu'on obtiendrait si on supposait  $\text{Type}: \text{Type}$ . Néanmoins, on peut noter en Coq  $\text{Type}_i$  simplement par  $\text{Type}$ , le système se chargeant de trouver les indices quand cela est possible.

### 1.1.3 Les fonctions.

Les types produits sont des ensembles de fonctions.

Le type produit  $\prod_{i \in I} E_i$  a pour éléments les familles  $(x_i)_{i \in I}$ , où chaque  $x_i$  est de type  $E_i$ , autrement dit les fonctions  $i \mapsto x_i$ . On appelle aussi ces types "types dépendants".

Lorsque les ensembles  $E_i$  ne dépendent pas de  $i$ , ce sont tous un même type  $E$ , et on retrouve les fonctions usuelles, avec un ensemble de départ  $I$  et un ensemble d'arrivée  $E$ . Dans ce cas, le type  $\prod_{i \in I} E_i$  est noté  $I \rightarrow E$ . On appelle ces types "types non dépendants".

### 1.1.4 Les formules logiques, leurs preuves, et l'isomorphisme de Curry-Howard.

Les formules logiques, ou propositions, sont les éléments de la sorte Prop. Ce sont donc des types! En effet, une proposition est vue comme l'ensemble de ses preuves.

Par exemple, la proposition fausse est définie dans le CCI comme un type vide.

- l'implication logique  $A \Rightarrow B$  est alors l'ensemble des fonctions qui, à partir d'une preuve de  $A$ , donne une preuve de  $B$ . C'est pour cela qu'elle est notée  $A \rightarrow B$  dans le CCI.

Par exemple, la fonction identité  $x \mapsto x$  ( $\lambda x.x$  dans le CCI, notée  $[x: A]x$  en Coq), qui est de type  $A \rightarrow A$ , est une preuve de la proposition  $A \rightarrow A$ .

- la quantification universelle  $\forall x: E, P(x)$ , notée  $(x: A)(Px)$  en Coq, est un type produit, et représente l'ensemble des fonctions qui, étant donné un élément  $x$  de  $E$ , donnent un élément de  $P(x)$ , c'est-à-dire une preuve que  $x$  vérifie  $P$ .

---

1.  $O, S(O), S(S(O)), \dots$

2. Soit  $E$  l'ensemble de tous les ensembles. On a donc  $E \in E$ . Soit maintenant  $X$  l'ensemble des ensembles qui ne s'appartiennent pas:

$$X = \{A \in E \mid A \notin A\}$$

alors ni  $X \in X$ , ni  $X \notin X$  ne sont vraies! L'ensemble de tous les ensembles n'est donc pas un ensemble, ou du moins pas un ensemble de même type que ses éléments. Ce paradoxe est dû à Russell, mathématicien du début du siècle dernier, qui inventa la théorie des types.

De manière plus imagée:

«Dans un village, le barbier est l'homme qui rase les hommes qui ne se rasent pas eux-mêmes.

Le barbier se rase-t-il?»

Par exemple, si  $P$  est la fonction

$$\begin{aligned} P: \text{nat} &\rightarrow \text{Prop} \\ x &\mapsto x \text{ est pair} \end{aligned}$$

la fonction  $x \mapsto p(x)$ , où  $p(x)$  est une preuve que  $x$  est pair (et donc  $p(x): P(x)$ ), est de type  $\forall x: E, P(x)$ , c'est donc une preuve de  $\forall x: E, P(x)$ .

Cette correspondance entre formules logiques et types d'une part, et entre preuves et éléments des types d'autre part, est appelée *l'isomorphisme de Curry-Howard*. C'est elle qui permet de représenter les preuves dans le CCI.

## 1.2 Le typage.

La relation de typage " $t$  est de type  $E$ ", ou  $t: E$ , est essentielle dans Coq. Outre le fait qu'elle permet d'éviter des incohérences, elle permet de déterminer si une preuve d'un énoncé est correcte. En effet, cette relation est *décidable*, ce qui veut dire qu'il existe un algorithme déterminant si oui ou non un terme donné est d'un type donné, ce qui revient, avec l'isomorphisme de Curry-Howard, à savoir si une preuve prouve bien un énoncé.

Cet algorithme est le cœur du système Coq, et c'est grâce à lui que Coq peut être utilisé comme un vérificateur de preuves formelles (ce n'est pas son seul usage, il permet aussi de construire pas-à-pas des preuves, ainsi que de calculer, et parfois trouve lui-même des preuves).

### 1.2.1 Règles de typage informelles.

La relation de typage du CCI est définie formellement par des règles.

Avant de les donner toutes, on voici informellement les principales:

- typage des applications:

$$\frac{f: \forall x: A, B(x) \quad a: A}{f(a): B(a)}$$

cette règle se lit

"si  $f$  est de type  $\forall x: A, B(x)$  et si  $a$  est de type  $A$ , alors  $f(a)$  est de type  $B(a)$ ".

dans le cas non dépendant, cela donne

$$\frac{f: A \rightarrow B \quad a: A}{f(a): B}$$

- typage des fonctions:

$$\frac{x: A \vdash t(x): B(x)}{\lambda x: A. t(x): \forall x: A. B(x)}$$

qui se lit

"si, lorsque  $x$  est de type  $A$ ,  $t(x)$  est de type  $B(x)$ , alors  $\lambda x: A. t(x)$  est de type  $\forall x: A. B(x)$ "

dans le cas non dépendant:

$$\frac{x: A \vdash t(x): B}{\lambda x: A. t(x): A \rightarrow B}$$

Par exemple, pour montrer que  $\lambda x: A. \lambda y: A \rightarrow B. (yx)$  est bien une preuve de  $A \rightarrow ((A \rightarrow B) \rightarrow B)$ , on applique ces règles ainsi:

$$\frac{x: A, y: A \rightarrow B \vdash y: A \rightarrow B \quad x: A, y: A \rightarrow B \vdash x: A}{\frac{x: A, y: A \rightarrow B \vdash (yx): B}{x: A \vdash \lambda y: A \rightarrow B. (yx): (A \rightarrow B) \rightarrow B}}{\lambda x: A. \lambda y: A \rightarrow B. (yx): A \rightarrow ((A \rightarrow B) \rightarrow B)}$$

En haut se trouvent deux assertions triviales, puis on applique la règle de typage des applications, et deux fois celle de typage des fonctions.

On peut aussi utiliser les règles de typage pour déterminer le type d'un terme: c'est l'inférence de type, elle aussi décidable dans le CCI.

### 1.2.2 Les règles de typage.

Par soucis de simplicité, on a omis la notion d'environnement présente dans les règles du CCI, qui permet de faire intervenir les définitions de constantes dans le typage. Cette notion n'est pas essentielle, et plutôt d'un intérêt pratique. Les règles complètes peuvent être trouvées par exemple dans le manuel du système Coq.

#### 1.2.2.1 Les contextes.

Un contexte est un ensemble d'hypothèses de types pour des variables distinctes. Plus précisément c'est une suite finie  $\Gamma = (x_1:T_1, \dots, x_n:T_n)$ , où les  $x_i$  sont deux à deux distincts.

On notera  $\Gamma, x:T$  la réunion  $\Gamma \cup \{x:T\}$ , et  $()$  le contexte vide. On écrira  $x \in \Gamma$  le fait que la variable  $x$  apparaît dans le contexte  $\Gamma$ .

#### 1.2.2.2 Les jugements de typage.

Un jugement de typage est un énoncé donnant un type à un terme dans un certain contexte. Il a la forme  $\Gamma \vdash t:T$ , qui se lit " $t$  a le type  $T$  dans le contexte  $\Gamma$ ".

#### 1.2.2.3 Les règles.

Les règles de typage permettent de construire des contextes corrects et de faire des déductions sur les jugements de typage. On note  $\mathcal{S}$  l'ensemble des sortes, et  $\text{WF}(\Gamma)$  le fait qu'un contexte est correct.

Les règles suivantes sont une version simplifiée de celles de Coq: on a omis les définitions de constantes.

**W-E**

$$\overline{\text{WF}(\Gamma)}$$

**W-s**

$$\frac{\text{WF}(\Gamma) \quad s \in \mathcal{S} \quad \Gamma \vdash T:s \quad x \notin \Gamma}{\text{WF}(\Gamma, x:T)}$$

**Ax**

$$\frac{\text{WF}(\Gamma)}{\Gamma \vdash \text{Prop}:\text{Type}_i} \quad \frac{\text{WF}(\Gamma)}{\Gamma \vdash \text{Set}:\text{Type}_i} \quad \frac{\text{WF}(\Gamma) \quad i < j}{\Gamma \vdash \text{Type}_i:\text{Type}_j}$$

**Var**

$$\frac{\text{WF}(\Gamma) \quad (x:T) \in \Gamma}{\Gamma \vdash x:T}$$

**Prod**

$$\frac{\Gamma \vdash T:s_1 \quad \Gamma, x:T \vdash U:s_2 \quad s_1 \in \{\text{Prop}, \text{Set}\} \text{ ou } s_2 \in \{\text{Prop}, \text{Set}\}}{\Gamma \vdash \forall x:T, U:s_2}$$

$$\frac{\Gamma \vdash T:\text{Type}_i \quad \Gamma, x:T \vdash U:\text{Type}_j \quad i, j \leq k}{\Gamma \vdash \forall x:T, U:\text{Type}_k}$$

**Lam**

$$\frac{\Gamma \vdash \forall x:T, U:s \quad \Gamma, x:T \vdash t:U \quad s \in \mathcal{S}}{\Gamma \vdash \lambda x:T. t:\forall x:T, U}$$

**App**

$$\frac{\Gamma \vdash t:\forall x:T, U \quad \Gamma \vdash u:T}{\Gamma \vdash (tu):U[x \leftarrow u]}$$

On dira qu'un terme  $t$  est de type  $T$  dans le contexte  $\Gamma$  si on peut montrer à l'aide de ces règles le jugement  $\Gamma \vdash t : T$ .

#### 1.2.2.4 Exemple.

Montrons par exemple que le terme  $\forall A : \text{Prop}, A \rightarrow A$  est de type  $\text{Prop}$  dans le contexte vide:

$$\frac{\frac{\frac{\text{WF}(\text{()})}{\text{() } \vdash \text{Prop} : \text{Type}_0} \quad \frac{\frac{\text{WF}(A : \text{Prop}) \quad A : \text{Prop} \vdash A : \text{Prop}}{\text{WF}(A : \text{Prop}, x : A)}}{A : \text{Prop}, x : A \vdash A : \text{Prop}}}{A : \text{Prop} \vdash A \rightarrow A : \text{Prop}}}{\text{() } \vdash \forall A : \text{Prop}, A \rightarrow A : \text{Prop}}$$

On a indiqué en bleu les jugements qui se répètent dans l'arbre: ils sont déjà prouvés ailleurs (en rouge, et sur leur gauche).

**Exercice 1.1.** Montrer

$$\text{() } \vdash \lambda A : \text{Prop}. \lambda x : A. (\lambda y : A. y)(x) : \forall A : \text{Prop}. A \rightarrow A$$

### 1.2.3 Quelques remarques sur les règles de typage des produits.

La première règle de typage des produits concerne les sortes  $\text{Prop}$  et  $\text{Set}$ . Elle indique en particulier que si dans un produit  $\forall x : T, U$ , la cible  $U$  est une proposition, alors le produit lui-même est une proposition, quel que soit le type  $T$  : on dit que la sorte  $\text{Prop}$  est *imprédicative*. Il en est de même pour la sorte  $\text{Set}$ .

La deuxième règle de typage des produits concerne les univers  $\text{Type}_i$ , qui eux sont *prédicatifs*. En effet, par exemple, le type de  $\forall A : \text{Type}_0, A$ , est  $\text{Type}_1$ , alors qu'il serait  $\text{Type}_0$  si  $\text{Type}_0$  était imprédicatif.

**Exercice 1.2.** Montrer

$$\text{() } \vdash \forall A : \text{Type}_0, A : \text{Type}_1$$

## 1.3 Le calcul.

On détaille ici les deux premières règles de calcul de Coq, les (deux) autres seront abordées plus tard.

### 1.3.1 La $\beta$ -réduction

La règle de calcul de base de Coq est la  $\beta$ -réduction du  $\lambda$ -calcul:

$$((\lambda x. t) a) \rightarrow_{\beta} t[x \leftarrow a]$$

Par exemple,  $((\lambda x : A. \lambda y : A \rightarrow B. (yx)) a) f \rightarrow_{\beta} ((\lambda y : A \rightarrow B. (ya)) f) \rightarrow_{\beta} (fa)$ .

**Théorème 1.1.** *La  $\beta$ -réduction est confluente<sup>3</sup> et termine<sup>4</sup> sur les termes typables du CCI. On dit qu'elle est fortement normalisante.*

<sup>3</sup> la confluence d'une relation  $\rightarrow$  est définie par:

$\forall x, y_1, y_2, x \rightarrow \dots \rightarrow y_1$  et  $x \rightarrow \dots \rightarrow y_2 \Rightarrow \exists z, y_1 \rightarrow \dots \rightarrow z$  et  $y_2 \rightarrow \dots \rightarrow z$

<sup>4</sup> on dit qu'une relation  $\rightarrow$  termine ssi sa relation symétrique  $\leftarrow$  est bien fondée, i.e. il n'existe pas de suite infinie  $x_1 \rightarrow x_2 \rightarrow \dots$

Ce théorème, difficile à démontrer, garantit la terminaison des calculs, ainsi que leur unicité.

**Théorème 1.2.** *La  $\beta$ -réduction est compatible avec le typage:  $t: A, t \rightarrow_{\beta} u \Rightarrow u: A$ .*

Ceci garantit la compatibilité du typage avec la  $\beta$ -conversion, qui la plus petite relation d'équivalence contenant la  $\beta$ -réduction.

Dans le  $\lambda$ -calcul pur, la  $\beta$ -réduction est confluente, mais ne termine pas toujours.<sup>5</sup>

Dans le  $\lambda$ -calcul simplement typé, la  $\beta$ -réduction est fortement normalisante, mais les types ne représente qu'une logique très faible: le calcul des propositions avec uniquement des implications.

Dans le CCI, elle garde cette propriété de normalisation forte, mais maintenant les types représentent une logique beaucoup plus forte, en fait suffisante pour exprimer les mathématiques: le calcul des prédicats d'ordre supérieur.

### 1.3.2 La consistance.

La normalisation forte de la  $\beta$ -réduction et sa compatibilité avec le typage permettent de montrer la consistance logique du CCI: tous les types ne sont pas habités, autrement dit toutes les formules ne sont pas démontrables. En particulier, on va montrer qu'il n'existe pas de preuve de  $\forall X: \text{Prop}, X$ .

**Lemme 1.3.** *Soit  $t$  un terme irréductible tel que  $() \vdash t: \forall x: T, U$ . Alors  $t$  est une abstraction:  $t = \lambda x: T, u$ .*

**Démonstration.** Par récurrence sur la taille<sup>6</sup> de la preuve de  $() \vdash t: \forall x: T, U$ . Ce jugement ne peut être obtenu que par les règles App et Lam.

S'il est obtenu par App, alors  $t = (t_1 u)$  et en hypothèse de la règle on a un jugement  $() \vdash t_1: \forall x: T_1, U_1$ . Or  $t$  est irréductible, donc  $t_1$  aussi, et par hypothèse de récurrence,  $t_1$  est une abstraction. Donc  $t$  est réductible: contradiction.

Donc  $() \vdash t: \forall x: T, U$  est obtenu par la règle Lam, et  $t$  est une abstraction.  $\square$

**Théorème 1.4.** *Il n'y a pas de terme irréductible  $t$  tel que  $() \vdash t: \forall X: \text{Prop}, X$ .*

**Démonstration.** Par l'absurde. Supposons qu'un tel terme  $t$  existe. Alors, par le lemme précédent,  $t$  est une abstraction:  $t = \lambda x: X, u$ . Le jugement  $() \vdash t: \forall X: \text{Prop}, X$  est donc obtenu par la règle Lam. On a alors  $X: \text{Prop} \vdash u: X$  en hypothèse de cette règle.

Si ce dernier jugement est obtenu par la règle App,  $u$  est une application, et a donc la forme  $u = (fa_1 \dots a_n)$  avec  $n \neq 0$  et  $f$  n'étant pas une application. Comme  $t$  est irréductible,  $u$  l'est aussi, et  $f$  n'est donc pas une abstraction. Comme  $(fa_1)$  est typable,  $f$  n'est pas un produit. Donc  $f$  est une variable. Comme les règles de typage ne peuvent typer que des termes dont les variables libres sont dans le contexte du jugement, on en déduit que  $f = X$ . Or  $X$  est de type Prop, donc le terme  $(fa_1)$  n'est pas typable, et par suite  $u$  non plus: contradiction.

$X$  est une variable, donc la seule règle autre que App qui peut produire le jugement  $X: \text{Prop} \vdash u: X$  est la règle Var. Mais  $u: X$  n'est pas dans le contexte  $X: \text{Prop}$ .

Donc le jugement  $X: \text{Prop} \vdash u: X$  ne peut provenir d'aucune règle: contradiction.  $\square$

**Corollaire 1.5.** *Il n'y a pas de terme  $t$  tel que  $() \vdash t: \forall X: \text{Prop}, X$*

**Démonstration.** Si un tel terme existait, sa forme normale serait de même type.  $\square$

On peut donc raisonnablement définir la proposition fausse par  $\forall X: \text{Prop}, X$ , puisqu'elle n'a pas de preuve. De même, l'ensemble vide peut-être défini par  $\forall X: \text{Set}, X$ , ou bien  $\forall X: \text{Type}, X$ .

5. si  $\omega = \lambda x. x(x)$ , on a  $\omega(\omega) \rightarrow_{\beta} \omega(\omega) \rightarrow_{\beta} \dots$

6. cette taille est le nombre de règles de typage qui la compose.

Remarquons que la consistance est ici montrée par une démonstration syntaxique, et non par la construction d'un modèle (i.e. une interprétation ensembliste des types et fonctionnelle des autres termes).

### 1.3.3 Logique intuitionniste et mathématiques constructives.

Une particularité du CCI est qu'il permet naturellement de raisonner en logique intuitionniste, c'est-à-dire que, si  $-$  dénote  $\forall X: \text{Prop}, X$ , et  $\neg A$  dénote  $A \rightarrow -$ ,

$$\forall A: \text{Prop}, \neg \neg A \rightarrow A$$

n'est pas prouvable. Ceci est équivalent au fait que le tiers exclus

$$\forall A: \text{Prop}, A \vee \neg A$$

n'est pas prouvable.

En conséquence,  $\exists x, P(x)$  n'est pas équivalent à  $\neg(\forall x, \neg P(x))$ :

$$\neg(\forall x, \neg P(x)) \not\rightarrow \exists x, P(x)$$

Ceci exclut certaines démonstrations par l'absurde, en particulier celles qui permettent de démontrer l'existence d'un objet sans le construire effectivement.

L'origine de ce comportement intuitionniste réside dans le fait qu'une démonstration d'existence dans le CCI permet de dériver un algorithme de construction (puisque une telle démonstration est un  $\lambda$ -terme, donc un algorithme).

Mais rien n'empêche de faire des mathématiques classiques avec le CCI: on peut supposer le tiers exclus en axiome, la théorie reste consistante<sup>7</sup> (on ne peut prouver la proposition fausse, ni trouver d'élément dans l'ensemble vide).

**Théorème 1.6.** *Il n'y a pas de terme  $t$  tel que  $\vdash t: \forall A: \text{Prop}, \neg \neg A \rightarrow A$ .*

**Démonstration.** Par l'absurde. Supposons  $\vdash t: \forall A: \text{Prop}, \neg \neg A \rightarrow A$ . On peut supposer  $t$  irréductible. Par le lemme 1.3,  $t$  est alors une abstraction.

On a donc  $t = \lambda A. a$ , et  $A: \text{Prop} \vdash a: \neg \neg A \rightarrow A$ .

Montrons que  $a$  est une abstraction.

- si  $a$  est une variable, alors  $a$  est dans le contexte, et donc  $a = A$  ainsi que  $\text{Prop} = \neg \neg A \rightarrow A$ , ce qui est manifestement faux.
- si  $a$  est une application: écrivons  $a = (x a_1 \dots a_n)$  où  $x$  n'est pas une application. Comme  $a$  est irréductible, puisque  $t$  l'est,  $x$  n'est pas une abstraction;  $x$  ne pouvant être un produit<sup>8</sup>, c'est donc une variable. Elle doit se trouver dans le contexte, donc  $x = A$ . Mais alors  $(x a_1)$  est mal typé, puisque le type de  $A$  n'est pas un produit: contradiction.
- comme  $a$  ne peut être un produit, puisque son type n'est pas une sorte,  $a$  est une abstraction.

Posons  $a = \lambda x. b$ . On a alors  $A: \text{Prop}, x: \neg \neg A \vdash b: A$ .

A cause du contexte,  $b$  ne peut être une variable (ce serait  $x$  ou  $A$ , mais alors son type ne serait pas correct).  $A$  étant une variable,  $b$  n'est ni une abstraction, ni un produit. C'est donc une application. Écrivons  $b = (y a_1 \dots a_n)$ , avec  $n \neq 0$  et  $y$  pas une application. Comme  $a$  est irréductible,  $b$  aussi. Donc  $y$  n'est pas une abstraction. Ce n'est pas un produit (comme pour  $x$  plus haut), c'est donc une variable. Elle est dans le contexte, ce ne peut être  $A$ , donc  $y = x$ .

Ainsi  $b = (x b_1 \dots a_n)$ .

$x$  est de type  $\neg \neg A$ , c'est-à-dire  $\neg A \rightarrow -$ , ou encore  $(A \rightarrow -) \rightarrow \forall X, X$ .

Donc  $A: \text{Prop}, x: \neg \neg A \vdash b_1: \neg A$ , et  $n = 2$ . De plus  $b_2 = A$ , et ainsi  $b = (x b_1 A)$ . On a donc  $t = \lambda A. \lambda x. (x b_1 A)$ . Et  $b_1$  est irréductible.

<sup>7</sup>. cette consistance peut être montrée en exhibant un modèle ensembliste du CCI.

<sup>8</sup>. si  $x$  est un produit, son type est une sorte, et donc  $(x a_1)$  n'est pas typable: dans une application  $(fu)$ , le type de  $f$  doit être un produit.

On peut alors choisir le terme  $t$  de départ tel que  $b_1$  soit de taille minimale parmi les termes irréductibles  $u$  tels que  $A: \text{Prop}, x: \neg\neg A \vdash u: \neg A$ . En effet, pour un tel terme  $u$ , on obtient  $\vdash \lambda A. \lambda x. (x u A): \forall A: \text{Prop}, \neg\neg A \rightarrow A$ , et  $\lambda A. \lambda x. (x u A)$  est irréductible.

- $b_1$  ne peut être une variable, sinon ce serait une variable du contexte, et son type ne convient pas.
- $b_1$  n'est pas un produit, son type n'étant pas une sorte.
- supposons que  $b_1$  est une application:  $b_1 = (z c_1 \dots c_m)$ ,  $m \neq 0$ ,  $z$  pas une application.  $b$  est irréductible, donc  $b_1$  aussi, et donc  $z$  n'est pas une abstraction. Ce n'est pas non plus un produit, donc  $c$ 'est une variable. On a  $A: \text{Prop}, x: \neg\neg A \vdash b_1: \neg A$ , donc  $z$  est dans le contexte, ce ne peut être  $A$ , donc  $z = x$ . Ainsi  $b_1 = (x c_1 \dots c_m)$ . Comme précédemment avec  $b_1$ , on en déduit que  $c_1$  est irréductible, et de type  $\neg A$  dans le contexte  $A: \text{Prop}, x: \neg\neg A$ . Or  $b_1$  est de taille minimale pour cette propriété, et  $c_1$  de taille strictement plus petite que celle de  $b_1$ : contradiction.
- $b_1$  est donc une abstraction:  $b_1 = \lambda y: A. c$ . Comme  $A: \text{Prop}, x: \neg\neg A \vdash b_1: \neg A$  et  $\neg A = A \rightarrow \forall X, X$  on a alors

$$A: \text{Prop}, x: \neg\neg A, y: A \vdash c: \forall X, X$$

- $c$  n'est pas une variable, sinon elle serait dans le contexte, or son type ne convient pas.
- $c$  n'est pas un produit, son type n'étant pas une sorte.
- supposons que  $c$  est une application:  $c = (v d_1 \dots d_r)$ ,  $r \neq 0$ ,  $v$  pas une application.  $b_1$  est irréductible, donc  $c$  aussi, et donc  $v$  n'est pas une abstraction. Ce n'est pas non plus un produit, donc  $c$ 'est une variable. Ce ne peut être ni  $A$  ni  $y$ , donc  $c$ 'est  $x$ . Ainsi  $c = (x d_1 \dots d_r)$ . Comme pour  $c_1$  précédemment, on en déduit que  $d_1$  est irréductible, et de type  $\neg A$  dans le contexte  $A: \text{Prop}, x: \neg\neg A, y: A$ , donc aussi dans le contexte  $A: \text{Prop}, x: \neg\neg A$ , et comme  $b_1$  est de taille minimale pour cette propriété, on a une contradiction.
- $c$  est donc une abstraction:  $c = \lambda X: \text{Prop}. w$ . Ainsi on a

$$A: \text{Prop}, x: \neg\neg A, y: A, X: \text{Prop} \vdash w: X$$

- $w$  n'est pas une variable, son type n'étant pas dans le contexte.
- $w$  n'est pas un produit, son type n'étant pas une sorte.
- supposons que  $w$  est une application:  $w = (e f_1 \dots f_k)$ ,  $k \neq 0$ ,  $e$  pas une application.  $c$  est irréductible, donc  $w$  aussi, et donc  $e$  n'est pas une abstraction. Ce n'est pas non plus un produit, donc  $c$ 'est une variable. Ce ne peut être ni  $A$ , ni  $y$ , ni  $X$ , donc  $c$ 'est  $x$ . Ainsi  $w = (x f_1 \dots f_r)$ . Comme pour  $d_1$  précédemment, on en déduit que  $f_1$  est irréductible, et de type  $\neg A$  dans le contexte  $A: \text{Prop}, x: \neg\neg A, y: A, X: \text{Prop}$ , donc aussi dans le contexte  $A: \text{Prop}, x: \neg\neg A$ , et comme  $b_1$  est de taille minimale pour cette propriété, on a une contradiction.
- $w$  ne peut être une abstraction, car son type  $X$  est une variable.

Donc le terme  $w$  ne peut exister: on a la contradiction recherchée.

□

### 1.3.4 Les définitions et la $\delta$ -réduction.

Comme en mathématiques, on peut dans le CCI et dans Coq nommer les objets complexes. C'est le processus de définition des constantes: étant donné un terme  $t$ , on crée une nouvelle constante  $c$  du langage, par la relation  $c := t$ . Le type de cette constante sera celui de sa valeur  $t$ .

À la définition des constantes est associée une opération de calcul qui consiste à *déplier* une constante, i.e. la remplacer par sa valeur. C'est la  $\delta$ -réduction, notée  $\rightarrow_\delta$ .

Par exemple,  $\text{id} := \lambda A: \text{Set}. \lambda x: A. x$  est la définition de l'identité polymorphe, et on a

$$\text{id}(\text{nat})(O) \rightarrow_{\delta} (\lambda A: \text{Set}. \lambda x: A. x)(\text{nat})(O) \rightarrow_{\beta} (\lambda x: \text{nat}. x)(O) \rightarrow_{\beta} O$$

**Théorème 1.7.** *La propriété de normalisation forte de la  $\beta$ -réduction est conservée si on ajoute la  $\delta$ -réduction.*

On dira qu'un terme qui ne peut être réduit par les relations de réductions est en *forme normale*.

### 1.3.5 La conversion.

Comme en mathématiques on écrit  $a + 2 = 4$  si  $a$  est défini comme étant égal à 2, on dira que deux termes qui ont la même forme normale sont *convertibles*. Le typage est alors compatible avec cette relation d'équivalence de conversion.

De plus, comme en mathématiques où le nom des variables muettes n'est pas important, on identifiera les termes qui ne diffèrent que par le nom de leurs variables liées (les variables après les  $\lambda$  et les  $\forall$ ).

### 1.3.6 Exemple: les codages imprédicatifs.

Avec ce qu'on a vu du CCI jusqu'à présent on peut déjà définir les objets de base de l'informatique: les nombres entiers et les opérateurs booléens.

**Définition 1.8.** *On définit les connecteurs booléens ainsi:*

$$\begin{aligned} \mathbf{et} &:= \lambda A, B: \text{Prop}. \forall X: \text{Prop}, (A \rightarrow B \rightarrow X) \rightarrow X \\ \mathbf{non} &:= \lambda A: \text{Prop}. \forall X: \text{Prop}, A \rightarrow X \\ \mathbf{faux} &:= \forall X: \text{Prop}, X \end{aligned}$$

– Montrons par exemple

$$\forall A, B: \text{Prop}, \mathbf{et}(A, B) \rightarrow A^9$$

D'après l'isomorphisme de Curry-Howard, il suffit de trouver un terme dont le type est  $\forall A, B: \text{Prop}, \mathbf{et}(A, B) \rightarrow A$ . Réduisons d'abord  $\mathbf{et}(A, B)$ :

$$\mathbf{et}(A, B) \rightarrow_{\delta} \rightarrow_{\beta} \forall X: \text{Prop}, (A \rightarrow B \rightarrow X) \rightarrow X$$

Soit  $f$  une fonction de type  $\forall X: \text{Prop}, (A \rightarrow B \rightarrow X) \rightarrow X$ , il s'agit de construire maintenant avec  $f$  un terme de type  $A$ . Le terme  $f(A)$  est de type  $(A \rightarrow B \rightarrow A) \rightarrow A$ . Il suffit de trouver un terme de type  $A \rightarrow B \rightarrow A$  et le tour est joué: on lui applique  $f(A)$ . Le terme  $\lambda x: A. \lambda y: B. x$  convient. Finalement, le terme

$$\lambda A, B: \text{Prop}. \lambda f: \mathbf{et}(A, B). f(\lambda x: A. \lambda y: B. x)$$

est de type  $\forall A, B: \text{Prop}, \mathbf{et}(A, B) \rightarrow A$ .

– Montrons maintenant

$$\forall A: \text{Prop}, \mathbf{non}(\mathbf{et}(A, \mathbf{faux})).$$

On a  $\mathbf{et}(A, \mathbf{faux}) \rightarrow_{\delta}^* \forall X, (A \rightarrow (\forall Y, Y) \rightarrow X) \rightarrow X$ .<sup>10</sup>

et

$$\forall A, \mathbf{non}(\mathbf{et}(A, \mathbf{faux})) \rightarrow_{\delta}^* \forall A, \forall Z, (\forall X, (A \rightarrow (\forall Y, Y) \rightarrow X) \rightarrow X) \rightarrow Z$$

9. on note  $\mathbf{et}(A)(B)$  par  $\mathbf{et}(A, B)$ . On pourra aussi le noter avec la syntaxe du  $\lambda$ -calcul:  $(\mathbf{et} A B)$ .

10. pour clarifier, on omet les types Prop de X et Y.

Il s'agit donc de trouver un terme dont le type est  $\forall A, \forall Z, (\forall X, (A \rightarrow (\forall Y, Y) \rightarrow X) \rightarrow X) \rightarrow Z$ .

Soit  $f$  un terme de type  $\forall X, (A \rightarrow (\forall Y, Y) \rightarrow X) \rightarrow X$ .

Alors  $f(Z)$  est de type  $(A \rightarrow (\forall Y, Y) \rightarrow Z) \rightarrow Z$ .

Cherchons maintenant un terme  $g$  de type  $A \rightarrow (\forall Y, Y) \rightarrow Z$ .

Le terme  $g = \lambda xy. y(Z)$  convient.

Ainsi le terme  $f(Z)(g)$  est de type  $Z$ .

En conclusion, le terme  $\lambda A. \lambda Z. \lambda f. f(Z)(\lambda xy. y(Z))$  est bien de type  $\forall A, \text{non}(\text{et}(A, \text{faux}))$ .

**Exercice 1.3.** Montrer:

- i.  $\forall A, B: \text{Prop}, \text{et}(A, B) \rightarrow B$ .
- ii.  $\forall A, B: \text{Prop}, (A \rightarrow \text{faux}) \rightarrow \text{et}(A, B) \rightarrow \text{faux}$

**Exercice 1.4.** Montrer  $\forall A: \text{Prop}, A \rightarrow (\text{non}(\text{non } A))$ . Peut montrer la réciproque?

**Exercice 1.5.** Trouver une définition pour le connecteur logique "ou" et vérifier que

$$\forall A, B: \text{Prop}, A \rightarrow \text{ou}(A, B).$$

**Exercice 1.6.** Peut-on prouver  $\forall A: \text{Prop}, \text{ou}(A, \text{non}(A))$ ?

**Définition 1.9.** On définit les entiers (de Church) et leur type `nat` ainsi

$$n := \lambda X: \text{Type}. \lambda f: X \rightarrow X. \lambda x: X. \underbrace{f(\dots f(x)\dots)}_n$$

$$\text{nat} := \forall X: \text{Type}, (X \rightarrow X) \rightarrow (X \rightarrow X)$$

Ainsi  $0 := \lambda X. \lambda f. \lambda x. x$ ,  $1 := \lambda X. \lambda f. \lambda x. f(x)$ . L'entier  $n$  est la fonction qui itère une fonction  $n$  fois.

**Exercice 1.7.** Vérifier que  $n$  est de type `nat`.

**Exercice 1.8.** On définit `plus` :=  $\lambda n. \lambda m. \lambda X. \lambda f. \lambda x. n(f)(m(f))(x)$ .

Montrer que `plus` est de type `nat → nat → nat`.

Calculer `2 + 2`.

**Exercice 1.9.** Définir la multiplication, la puissance ( $n \mapsto m \mapsto n^m$ ).

On peut pousser plus loin la représentation de l'informatique dans le CCI, par exemple en définissant les listes, ce qui permet d'avoir une mémoire où stocker des données.

Mais cette représentation des entiers, des booléens, des listes, n'est pas très pratique... Et on n'a toujours pas de moyen de définir des fonctions récursives (les opérateurs de point fixe classiques du  $\lambda$ -calcul pur ne sont pas typables dans le CCI<sup>1</sup>).

## 1.4 Les types inductifs.

Les *types inductifs* permettent de définir par récurrence (*induction* en anglais) des ensembles d'arbres. Pour définir un type inductif, on donne les *constructeurs* de ses éléments: ce sont des constantes et/ou des fonctions.

Ces ensembles étant définis par récurrence on pourra alors

- définir des fonctions récursives sur leurs éléments
- raisonner par cas sur la forme de leurs éléments
- faire des démonstrations par récurrence de propriétés de leurs éléments

11. essayez par exemple de trouver un type  $T$  tel que le terme  $\omega := \lambda x: (X: \text{Type}) X \rightarrow X. x(T)(x)$  soit typable!

Commençons par des exemples.

### 1.4.1 Exemples.

#### 1.4.1.1 Les entiers naturels.

Pour définir le type `nat` des entiers de Peano:  $\{O, S(O), S(S(O)), \dots\}$ , c'est-à-dire les arbres

$$\begin{array}{ccc} O & S & S \\ | & | & | \\ O & S & \dots \\ & | & \\ & O & \end{array}$$

on se donne deux constructeurs  $O: \text{nat}$  et  $S: \text{nat} \rightarrow \text{nat}$ .

En Coq, on utilise la commande

```
Inductive nat:Set:=
| 0:nat
| S:nat -> nat.
```

Une telle définition produit alors la définition des constantes suivantes:

**nat:** de type `Set`, c'est le type inductif qu'on vient de définir (on aurait aussi pu le mettre dans la sorte `Type`).

**O:** de type `nat`, c'est un constructeur constant, donc un élément de `nat`.

**S:** de type `nat → nat`, c'est un constructeur qui prend un élément de `nat` et en rend un autre (on peut le voir comme  $n \mapsto n + 1$ ).

**nat\_ind:** de type  $\forall P: \text{nat} \rightarrow \text{Prop}, P(O) \rightarrow (\forall n: \text{nat}, P(n) \rightarrow P(n + 1)) \rightarrow \forall n: \text{nat}, P(n)$ .

Ce n'est autre que le principe de démonstration par récurrence classique sur les entiers: pour prouver qu'une propriété est vraie pour tous les entiers naturels, il suffit de montrer qu'elle vraie en 0 et de montrer que si elle est vraie en  $n$ , alors elle vraie en  $n + 1$ .

**nat\_rec:** de type  $\forall P: \text{nat} \rightarrow \text{Set}, P(O) \rightarrow (\forall n: \text{nat}, P(n) \rightarrow P(n + 1)) \rightarrow \forall n: \text{nat}, P(n)$ .

**nat\_rect:** de type  $\forall P: \text{nat} \rightarrow \text{Type}, P(O) \rightarrow (\forall n: \text{nat}, P(n) \rightarrow P(n + 1)) \rightarrow \forall n: \text{nat}, P(n)$ .

Ces deux dernières constantes permettent de définir des fonctions récursives de type `nat → X`, comme on le verra par la suite.

La constante `nat_ind` exprime aussi le fait que dans le type `nat`, il n'y a pas d'autres éléments que ceux construits avec les constructeurs de `nat`, i.e.  $O$  et  $S$ . Autrement dit, `nat` est le plus ensemble de termes qui contient  $O$  et tel que s'il contient un terme  $t$ , alors il contient  $S(t)$ .

Une autre propriété importante des types inductifs (du moins ceux qui ne sont pas dans la sorte `Prop`) est que deux éléments construits avec des constructeurs différents sont différents:  $O$  est différent de  $S(O)$ .

#### 1.4.1.2 L'égalité de Leibniz.

L'égalité de Leibniz dit que deux éléments d'un type sont égaux s'ils sont indiscernables pour toute propriété. On la définit en Coq comme la plus petite relation d'équivalence:

```
Inductive eq[A:Set;x:A]:A->Prop:=
refl_equal:(eq A x x).
```

Ici, le type inductif `eq` a deux paramètres:  $A$  un ensemble, et  $x$  un élément de  $A$ . Son type est  $\forall A: \text{Set}, A \rightarrow A \rightarrow \text{Prop}$ . Il n'a qu'un seul constructeur `refl_equal`, de type  $(\text{eq } A x x)$ , ce qui signifie que les seules preuves d'égalité de deux éléments sont des preuves de  $(\text{eq } A x x)$ , autrement dit de  $x = x$ .

On a ici un exemple de définition d'un prédicat inductif: un prédicat inductif est défini comme le plus petit prédicat qui vérifie les axiomes de sa définition, qui sont les constructeurs. On peut aussi voir cela comme le fait que les seules manières de démontrer qu'un tel prédicat est vrai sont celles qui utilisent les axiomes de sa définition: ses preuves sont exactement les arbres de preuves qu'il contient en tant que type.

Notation: en Coq,  $(\text{eq } A \ x \ y)$  est noté aussi  $x = y$ .

La constante  $\text{eq\_ind}$ , définie avec le type inductif  $\text{eq}$ , a pour type

$$\forall A: \text{Set}, \forall x: A, \forall P: A \rightarrow \text{Prop}, P(x) \rightarrow \forall y: A, x = y \rightarrow P(y)$$

Elle exprime bien que deux éléments égaux ont les mêmes propriétés. C'est elle qui permet de faire des preuves d'égalités.

**Exercice 1.10.** Montrons que l'égalité de Leibniz est symétrique:

$$\forall A: \text{Set}, \forall x, y: A, x = y \rightarrow y = x.$$

Il suffit de trouver une fonction de type  $x = y \rightarrow y = x$ . Soit  $H$  un terme de type  $x = y$ , i.e. une preuve de  $x = y$ . On veut prouver  $y = x$ . Pour pouvoir utiliser le terme  $\text{eq\_ind}$ , prenons  $P := \lambda z. z = x$ .

Le terme  $(\text{eq\_ind } A \ x \ \lambda z. z = x)$  a le type  $x = x \rightarrow \forall y: A, x = y \rightarrow y = x$

Supposons qu'on a une preuve  $H_1$  de  $x = x$ . Appliquons le terme précédent à  $H_1$ .

On obtient  $(\text{eq\_ind } A \ x \ \lambda z. z = x \ H_1)$ , qui est de type  $\forall y: A, x = y \rightarrow y = x$ .

Pour  $H_1$ , on n'a pas le choix, vu la définition de  $\text{eq}$ : prenons  $H_1 := (\text{refl\_equal } A \ x)$ .

Finalement, le terme  $\lambda A: \text{Set}. \lambda x: A. (\text{eq\_ind } A \ x \ \lambda z. z = x \ (\text{refl\_equal } A \ x))$  a bien le type  $\forall A: \text{Set}, \forall x, y: A, x = y \rightarrow y = x$ , c'est donc bien une preuve de cet énoncé.

**Exercice 1.11.** Montrer la transitivité de  $\text{eq}$ .

## 1.4.2 Raisonnement et calcul par cas: la construction Cases.

Pour raisonner ou calculer par cas sur les éléments d'un type inductif, on se donne une nouvelle construction du CCI: la construction Cases. Sa forme la plus simple est la suivante:

```

Cases  $t$  of
  |  $c_1(x_1, \dots, x_{n_1})$   $\Rightarrow$   $r_1(x_1, \dots, x_{n_1})$ 
  ...
  |  $c_m(x_1, \dots, x_{n_m})$   $\Rightarrow$   $r_m(x_1, \dots, x_{n_m})$ 
end

```

où si  $T$  est un type inductif,  $t$  est de type  $T$ , les  $c_i$  sont les constructeurs de  $T$ , les  $x_1, \dots, x_{n_i}$  sont des variables correspondant à tous les arguments du constructeur  $c_i$ , et les termes  $r_i$  sont les résultats qu'on souhaite dans les différents cas possibles pour le terme  $t$ .

Par exemple, on peut définir ainsi la fonction prédécesseur sur les entiers, qui à  $O$  associe  $O$ , et à  $n + 1$  associe  $n$ :

```

pred :=  $\lambda n: \text{nat}$ . Cases  $n$  of
  |  $O$   $\Rightarrow$   $O$ 
  |  $S(n_1)$   $\Rightarrow$   $n_1$ 
end

```

## 1.4.3 La $\iota$ -réduction.

Comme la  $\beta$ -réduction est la règle de calcul associée à l'abstraction, la construction Cases a une règle de calcul associée, la  $\iota$ -réduction. Elle est définie ainsi:

```

Cases  $c_k(a_1, \dots, a_{n_k})$  of
  |  $c_1(x_1, \dots, x_{n_1})$   $\Rightarrow$   $r_1(x_1, \dots, x_{n_1})$ 
  ...
  |  $c_m(x_1, \dots, x_{n_m})$   $\Rightarrow$   $r_m(x_1, \dots, x_{n_m})$ 
end
 $\rightarrow_{\iota} r_k(a_1, \dots, a_{n_k})$ 

```

Par exemple,

$$\text{pred}(S(S(O))) \rightarrow_{\delta} (\lambda n: \text{nat}. \mathbf{Cases} \ n \ \mathbf{of} \dots \mathbf{end})(S(S(O))) \rightarrow_{\beta} \mathbf{Cases} \ S(S(O)) \ \mathbf{of} \dots \mathbf{end} \rightarrow_i S(O)$$

#### 1.4.4 Fonctions récursives: la construction **Fix**.

##### 1.4.4.1 Un exemple.

Commençons par un exemple simple d'une fonction récursive sur les entiers: la fonction  $n \mapsto 2n$ .

On la définit en Coq ainsi:

```
Fixpoint double[n:nat]:nat:=
  Cases n of
    | 0      => 0
    | (S n1)=> (S (S (double n1)))
  end.
```

On définit donc la fonction double par récurrence sur son argument, de type nat, et elle rend un élément de type nat. Cette récurrence est bien fondée, car l'appel récursif de double se fait sur un terme visiblement plus petit que son argument: le terme  $n1$  est *structurellement plus petit* que le terme  $n$ , qui est dans ce contexte égal à  $S(n1)$ . Pour assurer la terminaison des calculs, cette condition doit être satisfaite pour tous les appels récursifs dans la définition d'une fonction récursive. Elle est clairement décidable, car basée sur l'écriture syntaxique des termes.

Par exemple, la fonction  $f:0 \mapsto 0, 1 \mapsto 0, 2n \mapsto f(n), 2n+1 \mapsto f(6n+4)$  ne peut être définie en Coq:  $6n+4$  n'est pas structurellement plus petit que  $2n+1$ . En fait c'est tant mieux car, à ma connaissance, personne ne sait à l'heure actuelle (16 janvier 2003, 23h10mn57s) si le calcul de  $f(n)$  termine toujours!<sup>12</sup>

##### 1.4.4.2 Un exemple de récursion mutuelle.

Continuons avec un exemple un peu plus compliqué, toujours sur les entiers: on va définir les prédicats pair et impair.

Supposons qu'on a deux propositions vrai et faux.

Informellement, on peut définir pair et impair ainsi:

$$\begin{aligned} \text{pair}(O) &= \text{vrai} \\ \text{impair}(O) &= \text{faux} \\ \text{pair}(n+1) &= \text{impair}(n) \\ \text{impair}(n+1) &= \text{pair}(n) \end{aligned}$$

On voit que ces deux prédicats sont définis par récurrence, et l'un avec l'autre: ils sont *mutuellement récursifs*.

Dans le CCI on les définira ainsi:

```
pair := Fix pair { pair[n: nat]: bool :=      Cases n of
                                                | O => vrai
                                                | S(n1) => impair(n1)
                                                end
with
  impair[n: nat]: bool :=      Cases n of
                                | O => faux
                                | S(n1) => pair(n1)
                                end
}
```

<sup>12</sup>. on le pense, et on soupçonne même  $f$  d'être identiquement nulle.

et

```

impair := Fix impair { pair[n: nat]: bool :=
  Cases n of
  | O ⇒ vrai
  | S(n1) ⇒ impair(n1)
  end
with
impair[n: nat]: bool := Cases n of
  | O ⇒ faux
  | S(n1) ⇒ pair(n1)
  end
}

```

Noter que:

- ce qu'il a dans les accolades est identique pour les deux fonctions: c'est le coeur de leur définition mutuellement récursive.
- les noms pair et impair dans la construction Fix sont des variables muettes. On a gardé les noms des fonctions qu'on voulait définir mais on aurait pu en prendre d'autres, comme le montre l'exemple qui suit.

Pour la fonction double on a:

```

double := Fix f { f[n: nat]: nat := Cases n of
  | O ⇒ O
  | S(n1) ⇒ S(S(f(n1)))
  end
}

```

#### 1.4.4.3 Le cas général.

En général, on définira des fonctions mutuellement récursives  $f_1, \dots, f_n$  ainsi: pour tout  $i$ ,  $f_i$  est définie par

```

Fix fi { f1[x1: T11; ...; xp1: T1p1]: U1 := d1
with
...
with
fn[x1: Tn1; ...; xpn: Tnpn]: Un := dn
}

```

Pour chacune des définitions  $d_i$ , la récursion doit se faire sur le dernier argument des crochets:  $x_{p_i}$ , et les appels récursifs doivent se faire sur des termes structurellement plus petits<sup>13</sup>.

#### 1.4.5 La $\iota$ -réduction, bis.

La règle de réduction associée à la construction Fix est encore la  $\iota$ -réduction, qu'on étend de la manière suivante.

Soit  $F = \{ f_1[\dots; x_{p_1}: T_{p_1}]: U_1 := d_1 \text{ with } \dots \text{ with } f_n[\dots; x_{p_n}: T_{np_n}]: U_n := d_n \}$ .

Alors, si le terme  $a_{p_i}$  commence par un constructeur de son type (qui doit être un type inductif), on a

$$(\text{Fix } f_i F)(a_1, \dots, a_{p_i}) \rightarrow_{\iota} d_i[x_s \leftarrow a_s]_{s=1 \dots p_i} [f_k \leftarrow (\text{Fix } f_k F)]_{k=1 \dots n}$$

où  $t[x \leftarrow a]$  désigne le terme obtenu en substituant  $a$  à  $x$  dans  $t$ .

<sup>13</sup>. la définition de cette notion de "structurellement plus petit" est assez technique dans le cas de la récursion mutuelle, nous n'entrerons pas dans ses détails ici.

Par exemple, on a

$$\begin{aligned}
& \text{double}(S(O)) \\
& \rightarrow_{\delta} \mathbf{Fix} \ f \{ \ f[n: \text{nat}]: \text{nat} := \ \text{Cases } n \text{ of} \\
& \qquad \qquad \qquad |O \Rightarrow O \\
& \qquad \qquad \qquad |S(n_1) \Rightarrow S(S(f(n_1))) \\
& \qquad \qquad \qquad \text{end} \\
& \qquad \qquad \qquad \} (S(O)) \\
& \rightarrow_{\iota} \text{Cases } S(O) \text{ of} \\
& \qquad \qquad \qquad |O \Rightarrow O \\
& \qquad \qquad \qquad |S(n_1) \Rightarrow S(S(\mathbf{Fix} \ f \{ \ f[n: \text{nat}]: \text{nat} := \ \text{Cases } n \text{ of} \\
& \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad |O \Rightarrow O \\
& \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad |S(n_1) \Rightarrow S(S(f(n_1))) \\
& \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{end} \\
& \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \} (n_1))) \\
& \qquad \qquad \qquad \text{end} \\
& \rightarrow_{\iota} S(S(\mathbf{Fix} \ f \{ \ f[n: \text{nat}]: \text{nat} := \ \text{Cases } n \text{ of} \\
& \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad |O \Rightarrow O \\
& \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad |S(n_1) \Rightarrow S(S(f(n_1))) \\
& \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{end} \\
& \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \} (O))) \\
& \rightarrow_{\iota} S(S(\text{Cases } O \text{ of} \\
& \qquad \qquad \qquad |O \Rightarrow O \\
& \qquad \qquad \qquad |S(n_1) \Rightarrow S(S(\mathbf{Fix} \ f \{ \ f[n: \text{nat}]: \text{nat} := \ \text{Cases } n \text{ of} \\
& \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad |O \Rightarrow O \\
& \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad |S(n_1) \Rightarrow S(S(f(n_1))) \\
& \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{end} \\
& \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \} (n_1))) \\
& \qquad \qquad \qquad \text{end})) \\
& \rightarrow_{\iota} S(S(O))
\end{aligned}$$

On d'abord déplié la définition de double, puis réduit le terme  $\mathbf{Fix} \ f \ \{\dots\}(S(O))$ , puis réduit le terme  $\text{Cases } S(O)\dots$ , puis réduit le terme  $\mathbf{Fix} \ f \ \{\dots\}(O)$ , et enfin réduit le terme  $\text{Cases } O \text{ of } \dots$ , pour obtenir le résultat  $S(S(O))$ , en forme normale. On voit ici tout l'intérêt d'une machine pour effectuer ce type de calculs!

**Théorème 1.10.** *En ajoutant la  $\iota$ -réduction à la  $\beta$ -réduction et à la  $\delta$ -réduction, on obtient encore une relation de réduction fortement normalisante, et compatible avec le typage.*

Encore un théorème dont la preuve est ardue.

### 1.4.6 Typage des Cases et des Fix.

Les constructions  $\mathbf{Fix}$  et  $\text{Cases}$  ont chacune leur propre règle de typage, assez complexes, qu'on n'abordera pas ici.

# Chapitre 2

## Introduction au système Coq.

La documentation complète de Coq se trouve ici: <http://pauillac.inria.fr/coq/doc/main.html>  
Voir aussi le [livre de Yves Bertot et Pierre Casteran](http://www-sop.inria.fr/lemme/Yves.Bertot/coqart.html):  
<http://www-sop.inria.fr/lemme/Yves.Bertot/coqart.html>

### 2.1 Commandes élémentaires.

#### 2.1.1 Lancer coq depuis un shell

La commande est `coqtop`

```
Welcome to Coq 7.2 (January 2002)
```

```
Coq
Coq]
```

Noter que toutes les commandes envoyées à Coq doivent se terminer par un point. Le seul retour à la ligne significatif est celui qui suit le point final.

#### 2.1.2 Obtenir le type d'un terme

```
coq] Check nat.
```

```
nat
      : Set
```

```
Coq
Coq] Check Set.
```

```
Set
      : Type
```

```
Coq
Coq] Check Type.
```

```
Type
      : Type
```

```
Coq
Coq] Check Prop.
```

```
Prop
      : Type
```

```
Coq
Coq] Check (A:Prop)A.
      (A:Prop)A
          : Prop
```

```
Coq
Coq] Check [n:nat]n.
      [n:nat]n
          : nat->nat
```

```
Coq
Coq]
```

#### 2.1.3 Définir une constante

```
coq] Definition id_nat:= [n:nat]n.
      id_nat is defined
```

```
Coq
Coq] Print id_nat.
      id_nat = [n:nat]n
          : nat->nat
```

```
Coq
Coq]
```

Pour plus de clarté, on peut donner son type:

```
coq] Definition Id:(A:Set)A->A:= [A:Set] [x:A] x.
      Id is defined
```

```
Coq
Coq]
```

### 2.1.4 Obtenir la valeur d'une constante

coq] `Print Id.`

```
Id = [A:Set; x:A]x
      : (A:Set)A->A
```

Coq

Coq] `Print nat.`

```
Inductive nat : Set := 0 : nat |
S : nat->nat
```

Coq

Coq]

### 2.1.5 Calculer.

Avec la  $\beta$ -réduction:

coq] `Eval Cbv Beta in ([n:nat](S n) (S 0)).`

```
= (S (S 0))
   : nat
```

Coq

Coq]

Avec la  $\delta$ -réduction:

coq] `Eval Cbv Delta in (Id nat (S 0)).`

```
= ([A:Set; x:A]x nat (S 0))
   : nat
```

Coq

Coq]

Avec les deux:

coq] `Eval Cbv Beta Delta in (Id nat (S 0)).`

```
= (S 0)
   : nat
```

Coq

Coq]

Avec la  $\iota$ -réduction:

coq] `Eval Cbv Iota in (Cases (S 0) of 0 => true | (S n)=> false end).`

```
= ([_:nat]false 0)
   : bool
```

Coq

Coq] `Eval Cbv Iota Beta in (Cases (S 0) of 0 => true | (S n)=> false end).`

```
= false
   : bool
```

Coq

Coq]

Enfin avec tout:

coq] `Eval Compute in (Id nat (Cases (S 0) of 0 => (S 0) | (S n)=>0 end)).`

```
= 0
   : nat
```

Coq

Coq]

#### 2.1.5.1 Fonctions récursives.

Division par deux:

coq] `Fixpoint div2[n:nat]:nat:=`

```
  Cases n of
```

```
    | 0 => 0
```

```
    | (S n1)=>
```

```
      Cases n1 of
```

```
        | 0 => 0
```

```
        | (S
```

```
        n2) => (S (div2 n2))
```

```
      end
```

```
    end.
```

Coq] `Coqdiv2 is recursively defined`

Coq

Coq] `Print div2.`

```
div2 =
```

```
Fix div2
```

```
{div2 [n:nat] : nat :=
```

```
  Cases n of
```

```
    0 => 0
```

```
    | (S n1) => Cases n1 of
```

```
      0 => 0
```

```
      | (S n2) => (S
```

```
      (div2 n2))
```

```
    end
```

```
  end}
```

```
  : nat->nat
```

```

Coq
Coq] Eval Compute in (div2 (S (S (S (S
(S 0)))))).
      = (S (S 0))
      : nat

Coq
Coq]

```

## 2.1.6 Démontrer.

### 2.1.6.1 $\forall A, A \rightarrow A$

```

coq] Lemma 11:(A:Prop)A->A.
     1 subgoal
       =====
       (A:Prop)A->A

     11
Coq]

```

Coq entre en mode preuve (il était auparavant en mode commande). En dessous de la barre se trouve l'énoncé à démontrer, au dessus, les hypothèses disponibles (rien pour l'instant).

```

coq] Intros.
     1 subgoal

       A : Prop
       H : A
       =====
       A

     11
Coq]

```

`Intros` est une *tactique*, c'est-à-dire une commande qui effectue une étape de preuve élémentaire. Ici on fait passer en hypothèses les variables quantifiées universellement dans le but (noter que Coq invente un nom pour la variable du produit non dépendant  $A \rightarrow A$ ). On aurait aussi pu taper `Intros A H`.

```

coq] Exact H.
     Subtree proved!

     11
Coq]

```

La tactique `Exact` demande en argument un terme dont le type est le but à démontrer (modulo conversion). En cas de succès, le but est donc prouvé. Ici, on a fini la preuve du lemme.

```

coq] Qed.
     Intros.
     Exact H.

     11 is defined

Coq
Coq]

```

Pour enregistrer ce lemme dans l'environnement global de Coq, on utilise la commande `Qed`. Cela a pour effet de définir une nouvelle constante `11` dont la valeur est le terme construit par les tactiques de la preuve, et le type l'énoncé donné:

```

coq] Print 11.
     11 = [A:Prop; H:A]H
         : (A:Prop)A->A

```

```

Coq
Coq]

```

De plus, la commande `Qed` donne la liste des tactiques qui ont servi à construire la preuve, appelé le *script* de la preuve.

### 2.1.6.2 $\forall A, A \rightarrow \neg\neg A$

La proposition fautive est définie en Coq comme un type inductif vide, et la négation  $\neg A$ , comme  $A \rightarrow \text{False}$ . En syntaxe Coq on note  $\neg A$  par `~A`.

```

coq] Print False.
     Inductive False : Prop :=

```

```

Coq
Coq] Print not.
     not = [A:Prop]A->False
           : Prop->Prop

Coq
Coq] Lemma 12:(A:Prop)A->(not(not A)).
     1 subgoal
       =====
       (A:Prop)A->~~A

```

```

12
Coq] Intros.
1 subgoal

  A : Prop
  H : A
  =====
  ~~A

```

```

12
Coq] Unfold not.
1 subgoal

  A : Prop
  H : A
  =====
  (A->False)->False

```

12  
Coq]

La tactique `Unfold` remplace dans le but une constante par sa valeur. Si cela produit un radical, il est réduit.

```

coq] Intros.
1 subgoal

  A : Prop
  H : A
  H0 : A->False
  =====
  False

```

```

12
Coq] Apply H0.
1 subgoal

  A : Prop
  H : A
  H0 : A->False
  =====
  A

```

12  
Coq]

La tactique `Apply` permet d'utiliser une hypothèse (ou une constante de l'environnement global) dont le type est un produit, éventuellement itéré, qui se termine par un terme identique au but (ou s'y ramenant par instanciation de variables).

```

coq] Exact H.
      Subtree proved!

```

```

12
Coq] Qed.
      Intros.
      Unfold not.
      Intros.
      Apply H0.
      Exact H.

```

```

12 is defined

Coq
Coq] Print 12.
      12 = [A:Prop; H:A; H0:(A->False)](H0 H)
           : (A:Prop)A->~~A

```

Coq  
Coq]

**Exercice 2.1.** Démontrer  $(A,B,C:\text{Prop})A \rightarrow (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow C$ .

### 2.1.6.3 Calcul des propositions.

Les connecteurs `and` et `or` sont définis ainsi en Coq:

```

Coq] Print and.
      Inductive and [A : Prop; B : Prop]
      : Prop := conj : A->B->A/\B

```

```

Coq
Coq] Print or.
      Inductive or [A : Prop; B : Prop]
      : Prop :=
      or_introl : A->A\/B |
      or_intror : B->A\/B

```

Coq  
Coq]

(`and`  $A B$ ) est noté  $A \setminus B$ , (`or`  $A B$ ) est noté  $A \setminus / B$ .

```

coq] Lemma 13:(A,B:Prop)A/\B -> B/\A.
1 subgoal

=====
(A,B:Prop)A/\B->B/\A

```

```

13
Coq] Intros.
1 subgoal

  A : Prop
  B : Prop
  H : A/\B
=====
  B/\A

13
Coq] Split.
2 subgoals

  A : Prop
  B : Prop
  H : A/\B
=====
  B

subgoal 2 is:
A

13
Coq]

La tactique Split décompose un but dont
le type est un type inductif avec un seul constructeur. Ici elle produit deux sous-buts. C'est sur le premier sous-but que vont maintenant agir les tactiques, et lorsqu'il sera prouvé, le deuxième sous-but le remplacera.

Coq] Elim H.
2 subgoals

  A : Prop
  B : Prop
  H : A/\B
=====
  A->B->B

subgoal 2 is:
A

13
Coq]

La tactique Elim introduit une discussion sur la forme d'un objet d'un type inductif. Ici la discussion est simple: il n'y a qu'un cas car le type and n'a qu'un constructeur. Les arguments de ce constructeur sont introduits dans le but.

```

```

coq] Intros.
2 subgoals

  A : Prop
  B : Prop
  H : A/\B
  H0 : A
  H1 : B
=====
  B

subgoal 2 is:
A

13
Coq] Exact H1.
1 subgoal

  A : Prop
  B : Prop
  H : A/\B
=====
  A

13
Coq] Elim H.
1 subgoal

  A : Prop
  B : Prop
  H : A/\B
=====
  A->B->A

13
Coq] Intros.
1 subgoal

  A : Prop
  B : Prop
  H : A/\B
  H0 : A
  H1 : B
=====
  A

13
Coq] Exact H0.
Subtree proved!

13
Coq] Qed.

```

```

Intros.
Split.
Elim H.
Intros.
Exact H1.

Elim H.
Intros.
Exact H0.

13 is defined

Coq
Coq]Print 13.
13 =
[A,B:Prop; H:(A\B)]
<B,A>{(and_ind A B B [_:A; H1:B]H1
H),(and_ind A B A [H0:A; _:B]H0 H)}
: (A,B:Prop)A\B->B\A

Coq
Un deuxième exemple, où une discussion
plus intéressante est nécessaire:
coq] Lemma 14:(A,B:Prop)A\B->B\A.
1 subgoal

=====
(A,B:Prop)A\B->B\A

14
Coq]Intros.
1 subgoal

A : Prop
B : Prop
H : A\B
=====
B\A

14
Coq]Elim H.
2 subgoals

A : Prop
B : Prop
H : A\B
=====
A->B\A

subgoal 2 is:
B->B\A

```

```

14
Coq]
Ici, le type inductif de H est or, qui a deux
constructeurs.
coq] Intros.
2 subgoals

A : Prop
B : Prop
H : A\B
H0 : A
=====
B\A

subgoal 2 is:
B->B\A

14
Coq]Right.
2 subgoals

A : Prop
B : Prop
H : A\B
H0 : A
=====
A

subgoal 2 is:
B->B\A

14
Coq]
Une preuve de B\A est soit une preuve de
A, soit une preuve de B. On choisit donc ici
une preuve de A (car on a A en hypothese),
avec la tactique Right .
coq] Exact H0.
1 subgoal

A : Prop
B : Prop
H : A\B
=====
B->B\A

14
Coq]Intros.
1 subgoal

A : Prop

```

```

B : Prop
H : A\B
H0 : B
=====
B\A

14
Coq]Left.
1 subgoal

A : Prop
B : Prop
H : A\B
H0 : B
=====
B

14
Coq]Exact H0.
Subtree proved!

14
Coq]Qed.

Intros.
Elim H.
Intros.
Right.
Exact H0.

Intros.
Left.
Exact H0.

14 is defined

Coq

Coq]Print 14.
14 =
[A,B:Prop; H:(A\B)]
(or_ind A B B\A [H0:A](or_intror
B A H0) [H0:B](or_introl B A H0) H)
: (A,B:Prop)A\B->B\A

Coq

Coq]

```

**Exercice 2.2.** Démontrer  $(A,B,C:\text{Prop})A\wedge(B\wedge C) \rightarrow (A\wedge B)\wedge(A\wedge C)$ .

#### 2.1.6.4 Calcul des prédicats.

Le quantificateur existentiel est défini constructivement ainsi:

```

coq] Print ex.
Inductive ex [A : Set; P : A->Prop]
: Prop :=
ex_intro : (x:A)(P x)->(Ex P)

Coq

Coq]
Coq abrège (ex A P) en (Ex P). Une
preuve de  $\exists x:A, P(x)$  est donc donnée par un
couple  $(x, p)$  où  $x$  est dans  $A$  et  $p$  est une
preuve de  $P(x)$ .

coq] Lemma 15:(n:nat)n=0\/(Ex
[m:nat]n=(S m)).
1 subgoal

=====
(n:nat)n=0\/(EX m:nat | n=(S m))

15
Coq]Intros.
1 subgoal

n : nat
=====
n=0\/(EX m:nat | n=(S m))

15
Coq]Elim n.
2 subgoals

n : nat
=====
0=0\/(EX m:nat | 0=(S m))

subgoal 2 is:
(n0:nat)
n0=0\/(EX m:nat | n0=(S m))->(S
n0)=0\/(EX m:nat | (S n0)=(S m))

15
Coq]

```

Ici la tactique `Elim` introduit une discussion sur  $n$ , qui est la plus générale possible: c'est une démonstration par récurrence qui commence. Le premier but correspond au cas de base, le second au cas de récurrence, où on note l'apparition de l'hypothèse de récurrence (dont on n'a pas besoin dans ce cas particulier).

```

coq] Left.

```

```

2 subgoals

n : nat
=====
0=0

subgoal 2 is:
(n0:nat)
n0=0\/(EX m:nat | n0=(S m))->(S
n0)=0\/(EX m:nat | (S n0)=(S m))

15
Coq]Apply refl_equal.
1 subgoal

n : nat
=====
(n0:nat)
n0=0\/(EX m:nat | n0=(S m))->(S
n0)=0\/(EX m:nat | (S n0)=(S m))

15
Coq]Print eq.
Inductive eq [A : Set; x : A] : A-
>Prop := refl_equal : x=x

15
Coq]
Ici on utilise la définition de l'égalité de
Leibnitz.
Maintenant commence le cas de récur-
rence.
coq] Intros.
1 subgoal

n : nat
n0 : nat
H : n0=0\/(EX m:nat | n0=(S m))
=====
(S n0)=0\/(EX m:nat | (S n0)=(S
m))

15
Coq]Right.
1 subgoal

n : nat
n0 : nat
H : n0=0\/(EX m:nat | n0=(S m))
=====
(EX m:nat | (S n0)=(S m))

```

```

15
Coq]Exists n0.
1 subgoal

n : nat
n0 : nat
H : n0=0\/(EX m:nat | n0=(S m))
=====
(S n0)=(S n0)

15
Coq]
Pour prouver un but existentiel, on doit
donner le témoin explicitement, avec la tac-
tique Exists. Ici c'est n0.
coq] Apply refl_equal.
Subtree proved!

15
Coq]Qed.
Intros.
Elim n.
Left.
Apply refl_equal.

Intros.
Right.
Split with n0.
Apply refl_equal.

15 is defined

Coq
Coq]Print 15.
15 =
[n:nat]
(nat_ind [n0:nat]n0=0\/(EX m:nat |
n0=(S m))
(or_introl 0=0 (EX m:nat | 0=(S
m)) (refl_equal nat 0))
[n0:nat; _:(n0=0\/(EX m:nat |
n0=(S m)))]
(or_intror (S n0)=0 (EX m:nat |
(S n0)=(S m))
(ex_intro nat [m:nat](S
n0)=(S m) n0 (refl_equal nat (S
n0)))) n)
: (n:nat)n=0\/(EX m:nat | n=(S
m))

Coq

```

Coq]

# Chapitre 3

## Exemples

A travers quelques exemples de théorie des ensembles classique, on montre dans ce chapitre comment le CCI et le système Coq permet de "faire" des maths, c'est-à-dire définir, énoncer, démontrer.

### 3.1 L'ensemble des parties est plus gros que l'ensemble.

**Théorème 3.1.** *Il n'y a pas de surjection d'un ensemble dans l'ensemble de ses parties.*

**Démonstration.** Soient  $E$  un ensemble,  $\mathcal{P}(E)$  l'ensemble de ses parties. On fait une démonstration par l'absurde: supposons qu'il existe une application surjective  $f: E \rightarrow \mathcal{P}(E)$ .

Soit  $A := \{x \in E \mid x \notin f(x)\}$ .  $A$  est une partie de  $E$ , comme  $f$  est surjective, il existe un  $a \in E$  tel que  $f(a) = A$ . Par définition de  $A$ ,  $a \in A$  est équivalent à  $a \notin f(a)$ , donc  $a \notin A$ : voilà une belle contradiction.  $\square$

Montrons maintenant ce théorème avec Coq.

```

coq] Definition P:Type->Type:=[E:Type]E->Prop.
      P is defined

      Coq
Coq]Definition appartient:(E:Type)E->(P E)-
>Prop:=[E:Type][x:E][y:(P E)][y x].
      appartient is defined

      Coq
Coq]Theorem t1:(E:Type)(f:E->(P E))((y:(P E))(exT E
[x:E](f x==y))>False.
      1 subgoal
      =====
      (E:Type; f:(E->(P E)))(y:(P E))(EXT x:E | (f
x)==y)>False

      t1
Coq]Intros.
      1 subgoal

      E : Type
      f : E->(P E)
      H : (y:(P E))(EXT x:E | (f x)==y)
      =====
      False

      t1
Coq]LetTac A:=[x:E](not (appartient E x (f x))).
      1 subgoal

      E : Type
      f : E->(P E)
      H : (y:(P E))(EXT x:E | (f x)==y)
      A := [x:E]~(appartient E x (f x)) : E->Prop
      =====
      False

      t1
Coq]Elim (H A).
      1 subgoal

      E : Type
      f : E->(P E)
      H : (y:(P E))(EXT x:E | (f x)==y)
      A := [x:E]~(appartient E x (f x)) : E->Prop
      =====
      (x:E)(f x)==A->False

      t1
Coq]Intros a Ha.
      1 subgoal

      E : Type
      f : E->(P E)
      H : (y:(P E))(EXT x:E | (f x)==y)
      A := [x:E]~(appartient E x (f x)) : E->Prop
      a : E
      Ha : (f a)==A
      =====
      False

      t1
Coq]Cut (not (appartient E a A))>(appartient E a A).
      2 subgoals

      E : Type
      f : E->(P E)
      H : (y:(P E))(EXT x:E | (f x)==y)
      A := [x:E]~(appartient E x (f x)) : E->Prop
      a : E
      Ha : (f a)==A
      =====
      (~(appartient E a A)>(appartient E a A))>False

```

```

subgoal 2 is:
  ~(appartient E a A)->(appartient E a A)

t1
Coq]Intro.
2 subgoals

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
HO : ~(appartient E a A)->(appartient E a A)
=====
False

subgoal 2 is:
  ~(appartient E a A)->(appartient E a A)

t1
Coq]Cut (appartient E a A)->(not (appartient E a A)).
3 subgoals

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
HO : ~(appartient E a A)->(appartient E a A)
=====
((appartient E a A)->~(appartient E a A))->False

subgoal 2 is:
  (appartient E a A)->~(appartient E a A)
subgoal 3 is:
  ~(appartient E a A)->(appartient E a A)

t1
Coq]Intro.
3 subgoals

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
HO : ~(appartient E a A)->(appartient E a A)
H1 : (appartient E a A)->~(appartient E a A)
=====
False

subgoal 2 is:
  (appartient E a A)->~(appartient E a A)
subgoal 3 is:
  ~(appartient E a A)->(appartient E a A)

t1
Coq]Intro.
3 subgoals

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
HO : ~(appartient E a A)->(appartient E a A)
H1 : (appartient E a A)->~(appartient E a A)
=====
False

subgoal 2 is:
  (appartient E a A)->~(appartient E a A)
subgoal 3 is:
  ~(appartient E a A)->(appartient E a A)

t1
Coq]Unfold not in HO;Unfold not in H1.
3 subgoals

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
HO : ((appartient E a A)->False)->(appartient E a A)
H1 : (appartient E a A)->(appartient E a A)->False
=====
False

```

```

subgoal 2 is:
  (appartient E a A)->~(appartient E a A)
subgoal 3 is:
  ~(appartient E a A)->(appartient E a A)

t1
Coq]Auto.
2 subgoals

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
HO : ~(appartient E a A)->(appartient E a A)
=====
(appartient E a A)->~(appartient E a A)

subgoal 2 is:
  ~(appartient E a A)->(appartient E a A)

t1
Coq]Intro.
2 subgoals

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
HO : ~(appartient E a A)->(appartient E a A)
H1 : (appartient E a A)
=====
~(appartient E a A)

subgoal 2 is:
  ~(appartient E a A)->(appartient E a A)

t1
Coq]Unfold appartient A in H1.
2 subgoals

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
HO : ~(appartient E a A)->(appartient E a A)
H1 : ~((appartient E a (f a)))
=====
~(appartient E a A)

subgoal 2 is:
  ~(appartient E a A)->(appartient E a A)

t1
Coq]Rewrite Ha in H1.
2 subgoals

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
HO : ~(appartient E a A)->(appartient E a A)
H1 : ~((appartient E a A))
=====
~(appartient E a A)

subgoal 2 is:
  ~(appartient E a A)->(appartient E a A)

```

```

t1
Coq]Exact H1.
1 subgoal

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
=====
~(appartient E a A)

t1
Coq]Intro.
1 subgoal

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
HO : ~(appartient E a A)
=====
(appartient E a A)

t1
Coq]Unfold appartient A.
1 subgoal

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
HO : ~(appartient E a A)
=====
~(appartient E a (f a))

t1
Coq]Rewrite Ha.
1 subgoal

E : Type
f : E->(P E)
H : (y:(P E))(EXT x:E | (f x)==y)
A := [x:E]~(appartient E x (f x)) : E->Prop
a : E
Ha : (f a)==A
HO : ~(appartient E a A)
=====
~(appartient E a A)

t1
Coq]Exact HO.

```

```

Subtree proved!

t1
Coq]Defined.
Intros.
LetTac A:=[x:E](not (appartient E x (f x))).
Elim (H A).
Intros a Ha.
Cut (not (appartient E a A))->(appartient E a A).
Intro.
Cut (appartient E a A)->(not (appartient E a A)).
Intro.
Unfold not in HO; Unfold not in H1.
Auto.

Intro.
Unfold appartient A in H1.
Rewrite Ha in H1.
Exact H1.

Intro.
Unfold appartient A.
Rewrite Ha.
Exact HO.

t1 is defined

Coq
Coq]Print t1.
t1 =
[E:Type; f:(E->(P E)); H:((y:(P E))(EXT x:E | (f
x)==y))]
[A:=[x:E]~(appartient E x (f x))]
(exT_ind E [x:E](f x)==A False
[a:E; Ha:((f a)==A)]
[HO:=HO:(~(appartient E a A))]
(eqT_ind_r (P E) A [p:(P E)]~(appartient
E a p) HO (f a) Ha)]
[H1:=H1:(appartient E a A)]
(eqT_ind_r (P E) A
[p:(P E)]~(appartient E a p)-
>~(appartient E a A)
[H2:(~(appartient E a A))]H2 (f a) Ha
H1)]
(H1 (HO [H2:(appartient E a A)](H1 H2 H2))
(HO [H2:(appartient E a A)](H1 H2 H2))) (H
A))
: (E:Type; f:(E->(P E)))(y:(P E))(EXT x:E |
(f x)==y))->False

Coq]Check nat.
nat
: Set

Coq
Coq]

```