# UPMC/MASTER/INFO/STL/SVP
# Spécification et Vérification de Programmes
## Éléments d'une logique d'ordre supérieur avec le système Coq.

2016

## Table des matières

# Prolégomènes

## Programmes

**Il n'est de programmation que fonctionnelle** Un programme est ce qui permet à une machine de réaliser un calcul. Un calcul est réalisé par la composition d'opérations de manipulations dans le but d'obtenir la transformations de données numériques (entrées du calcul) vers une autre (sortie ou résultat du calcul). Les opérations de manipulations de données sont réalisées physiquement dans les divers composants des machine : micro-processeur, mémoire, etc. Les données sont implémentées par des suites binaires de taille fixe : octet, mot mémoire. Les données (valeurs binaires) résident dans des supports physiques des machines (registres, support magnétique, optique). Les opérations sont soit de pures transformations (incrément, négation, décalage), des compositions de plusieurs (souvent 2) données (addition, conjonction, comparaison) ou de simples opérations de déplacement d'une donnée vers un autre support (copier, dépiler, sauter).

Si l'on appelle ≪état mémoire≫ l'ensemble des valeurs numériques résidant à un instant dans l'ensemble des supports d'une machine, un calcul est une fonction d'un état mémoire vers un autre. Tout programme définit une fonction.

Au plus bas niveau, les programmes réalisent ces fonctions comme des séquences d'instructions machines et restent implicites. Au niveau supérieur des langages de programmation structurés, les fonctions peuvent être définies et utilisées explicitement par le truchement de leurs constructions syntaxiques.

## Spécification

Spécifier un programme c'est exprimer la relation à réaliser entre entre les entrées et les sorties du programme. Avec la vision fonctionnelle, c'est exprimer la relation à obtenir entre les arguments d'une fonction et son résultat.

Selon que le programme est donné comme une suite d'instructions (pas nécessairement de très bas niveau) ou comme une expression fonctionnelle, la spécification pourra prendre diverses formes :

— la relation entre entrées et sorties établie par un programme peut être donné comme un *triplet de Hoare*

$$P\{S\}Q$$

où $P$ et $Q$ décrivent l'état de la mémoire (la valeur des variables) avant (*pré-condition*) et après (*post-condition*) exécution du programme $S$ ;

— la relation entre les arguments $x$ et le résultat d'une fonction $f$ peut s'exprimer par une formule logique

$$\forall x.(P[x] \to Q[f(x)])$$

où $P$ donne la propriété que doivent satisfaire les arguments $x$ pour que la valeur de l'application $f(x)$ satisfasse $Q$.

## Vérification

Vérifier un programme, c'est s'assurer qu'il satisfait sa spécification. C'est-à-dire, dans la vision ≪suite d'instructions≫, que les couples formés par les entrées sorties du programme appartiennent à la relation définies par la spécification ; ou, dans la vision fonctionnelle, que la formule qui exprime la spécification est vraie.

La vision ≪suite d'instructions≫ des programmes réclame que soit définie une *logique* spécifique à l'activité de vérification, par exemple, la *logique de Hoare*. La vision fonctionnelle peut utiliser les moyens usuels de

la logique si toutefois celle-ci permet d'exprimer certaines constructions particulières que les langages de programmations utilisent pour définir des fonctions.

# 1 Programmes et structures de données

Par *programme* il faut entendre *fonction*. Le style fonctionnel est celui des langages de la famille ML.

## 1.1 Fonctions booléennes

Le type des booléens est un *type énuméré* défini par les deux constantes `true` et `false` appelées *constructeurs* du type `bool`.

On peut définir les fonctions booléennes de base en utilisant la structure de contrôle `if-then-else` :

```
Definition negb (b:bool) := if b then false else true.

Definition andb (b1 b2:bool) : bool := if b1 then b2 else false.

Definition orb (b1 b2:bool) : bool := if b1 then true else b2.
```

On spécifie le type des arguments et du résultat.

On peut aussi utiliser la construction `match-with`, dite construction de *filtrage de motif* (*pattern matching*, en anglais) :

```
Definition negb (b:bool) : bool :=
  match b with
    true => false
  | false => true
  end.
```

Dans l'expression `match b with true => false | false => true`, les constructeurs `true` et `false` sont appelés *motifs* (du filtrage).

En fait, l'expression `if` $e$ `then` $e_1$ `else` $e_2$ est une abréviation pour `match` $e$ `with true =>` $e_1$ `| false =>` $e_2$.

Le type `bool` est défini comme un *type inductif* :

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

Le type `bool` est lui-même de type `Set` qui est le type utilisé en général pour les structures de données.

C'est parce que le type `bool` est défini comme un type inductif, c'est-à-dire en termes de constructeurs, que l'on peut utiliser la structure de contrôle `match-with` pour décomposer les arguments booléens d'une fonction. On dit alors que la fonction est définie *par cas de constructeurs*.

**Évaluation symbolique** On munit notre *langage de programmation fonctionnelle* d'un système d'*évaluation symbolique*. Nous donnerons une définition précise de ces termes en **??**. En attendant, posons en les principes intuitifs avec les fonctions booléennes.

Le type énuméré `bool` est entièrement défini par les deux constantes `true` et `false`. Elles sont considérées comme les deux *valeurs symboliques* du type `bool`. Donc toute expression de type `bool` a pour *valeur*, soit la constante symbolique `true`, soit la constante symbolique `false`.

D'après la *définition* de la fonction `negb`, la valeur de l'application `(negb true)` est égale à la valeur de l'expression `match b with true => false | false => true` dans laquelle le *paramètre formel* `b` est remplacé par `true`. C'est-à-dire que la valeur de `(negb true)` est égale à la valeur de `match true with true => false | false => true`.

Le principe d'évaluation d'une construction de la forme `match` $e$ `with true =>` $e_1$ `| false =>` $e_2$ est exactement celui du `if-then-else` :

— si la valeur de $e$ est `true` alors la valeur de `match` $e$ `with true =>` $e_1$ `| false =>` $e_2$ est la valeur de $e_1$ ;

— sinon, la valeur de $e$ est nécessairement `false`, et la valeur de `match` $e$ `with true =>` $e_1$ `| false =>` $e_2$ est la valeur de $e_2$.

D'après ce principe, la valeur de `match true with true => false | false => true` est `false`.

Si l'on compose deux applications de `negb`, on pose que la valeur de l'expression `(negb (negb true))` est celle de `(negb false)`, puisque la valeur de `(negb true)` (paramètre d'appel du premier `negb`) est `false`. La valeur de `(negb false)` est celle de `match false with true => false | false => true` ; c'est-à-dire : `true`. Ce principe général d'évaluation des applications est appelé *appel par valeur* (en anglais : *call by value*).

## 1.2 Arithmétique et fonctions récursives

On travaillera avec des entiers idéaux et non des *entiers machines* (64 bits signés). On utilise l'ensemble des *entiers formels* de Peano qui est construit avec la constante 0 et la fonction *successeur* : *l'ensemble des entiers (naturels) est le plus petit ensemble qui contient 0 et qui est clos par la fonction successeur.*

On formalise ainsi cette définition :

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

La définition dit :

1. `O` est un élément du type `nat`
2. `S` est une fonction (abstraite) qui, à tout entier associe un entier (type `nat -> nat`).

`O` et `S` sont les constructeurs du type `nat`. Ce type mérite pleinement son qualificatif d'*inductif*, puisque le constructeur `S` a besoin d'un entier pour donner un nouvel entier.

Les expressions construites avec les constructeurs `O` et `S` ont la forme suivante : `O`, `(S O)`, `(S (S O))`, `(S (S (S O)))`, etc. Chacune d'elle représente un nombre entier (naturel) : le nombre de symboles `S` qu'elle contient. Toutes ces expressions sont les valeurs du type `nat`. Il y en a une infinité (*dénombrable*).

Le mécanisme de filtrage de la construction `match-with` est applicable aux expressions de type `nat`. Toutefois, il va différer un peu de ce que nous avions vu avec le type (fini) des booléens. Son utilisation la plus simple consiste à distinguer une valeur entière nulle (constructeur `O`) d'une valeur non nulle (constructeur `S`). Par exemple, on définit de cette manière la fonction *prédécesseur*, inverse de l'opération successeur, avec la particuliarité que, sur les entiers naturels, le prédécesseur de `O` est lui-même :

```
Definition pred (n:nat) : nat :=
  match n with
  | O => O
  | (S p) => p
  end.
```

Ce qui est nouveau ici, par rapport aux booléens, c'est l'utilisation du *symbole de variable* `p` dans le motif `(S p)`. Ce motif a deux propos :

1. *reconnaître* que la valeur de `n` est un entier non nul, puisque l'expression de cette valeur ≪commence≫ par l'application du constructeur `S` ;
2. *nommer* la valeur de l'entier qui suit l'application du ≪premier≫ constructeur `S` : si `n` a la valeur `(S 0)` alors, `p` prend la valeur `0` ; si `n` a la valeur `(S (S 0))`, alors `p` prend la valeur `(S 0)`, etc.

Du point de vue de l'évaluation symbolique la valeur de l'application `(pred (S (S 0)))` est la valeur l'expression `match (S (S 0)) with 0 => 0 | (S p) => p`. Selon le principe d'évaluation du filtrage, avec liaison de la variable de filtrage `p`, cette expression a pour valeur `(S 0)`.

Techniquement, on obtient `(S 0)` car l'expression `(S (S 0))` est égale à l'expression `(S p)` où l'on remplace `p` par `(S 0)`.

Sur la base des deux constructeurs `0` et `S`, jointe à la possibilité d'analyser une valeur entière par filtrage et à celle de définir des fonctions récursives, on reconstruit toute l'arithmétique.

Commençons par l'addition :

```
Fixpoint add (n m:nat) : nat :=
  match n with
  | O => m
  | (S p) => (S (add p m))
  end.
```

Intuitivement : pour faire la somme de $n$ et $m$, il suffit d'ajouter $n$ fois 1 à $m$ (≪ajouter 1≫ est représenté par le constructeur `S`). Ainsi, on réduit l'opération d'addition à l'*itération* de l'opération primitive `S`. L'itération est réalisée par la *définition récursive* (mot clé `Fixpoint`).

Suivons les étapes de l'évaluation symbolique de `(add (S (S 0)) (S 0))` : cette expression a pour valeur celle de l'expression `match (S (S 0)) with 0 => (S 0) | (S p) => (S (add p (S 0)))`. On a obtenu cette expression en remplaçant les paramètres formels `n` et `m` de la définition de `add` par, respectivement `(S (S 0))` et `(S 0)`. Selon le principe de l'évaluation du filtrage, `p` prend le valeur `(S 0)` et notre expression a pour valeur celle de `(S (add (S 0) (S 0)))`. Par définition de `add`, cette expression a pour valeur celle de `(S (match (S 0) with 0 => (S 0) | (S p) => (S (add 0 (S 0)))))` (notez le `S` en début d'expression). Par évaluation du filtrage, on obtient `(S (S (add 0 (S 0))))`, dont la valeur est celle de `(S (S (match 0 with 0 => (S 0) | (S p) => (S (add p (S 0))))))`. Ce qui donne, pour ce cas du filtrage : `(S (S (S 0)))`. Nous résumons ces étapes dans la tables suivante :

```
(add (S (S 0)) (S 0)
match (S (S 0)) with 0 => (S 0) | (S p) => (S (add p (S 0)))
(S (add (S 0) (S 0)))
(S (match (S 0) with 0 => (S 0) | (S p) => (S (add 0 (S 0)))))
(S (S (add 0 (S 0))))
(S (S (match 0 with 0 => (S 0) | (S p) => (S (add p (S 0))))))
(S (S (S 0)))
```

La multiplication est définie selon un principe analogue d'itération :

```
Fixpoint mult (n m:nat) : nat :=
  match n with
  | O => 0
  | (S p) => (add m (mult p m))
  end.
```

L'opération itérée ici est l'addition de $m$.

On peut définir la soustraction, fonction inverse de l'addition, par itération de la fonction inverse du successeur : le prédécesseur.

```
Fixpoint sub (n m : nat) : nat :=
  match m with
    | 0 => n
    | (S p) => (pred (sub n p))
end.
```

Notez que c'est ici le second argument qui sert à contrôler le nombre de répétitions. Soustraire `m` de `n`, c'est appliquer `m` fois le prédécesseur sur `n`. On remarque qu'alors `(sub 0 m)` est égal à `0`, quelque soit `m`. Plus généralement, `(sub n m)` est égal à `0` si `n` est plus petit que `m`. Un entier est plus petit qu'un autre si sa valeur s'exprime avec moins de symbole `S`.

On peut d'ailleurs définir la fonction booléenne :

```
Fixpoint leb n m : bool :=
  match n, m with
    | 0, _ => true
    | _, 0 => false
    | (S n'), (S m') => (leb n' m')
    end.
```

telle que `(leb n m)` a la valeur `true` si et seulement si `n` est inférieur ou égal à `m`. Notez comment cette définition analyse à la fois les arguments `n` et `m` par filtrage [1] et comment elle faite appel au *motif anonyme* noté _.

Sur ce modèle, on peut donner une définition alternative de la soustraction :

```
Fixpoint sub (n m:nat) : nat :=
  match n, m with
    0, _ => 0
  | _, 0 => n
  | (S p), (S q) => (sub p q)
  end.
```

**Fonctions partielles** La division euclidienne (division entière) ne peut pas être définie car c'est une *fonction partielle* : la valeur de $n/0$ n'est pas définie (dans les entiers). Dans le *calcul des constructions inductives* (le formalisme logique sous-jacent au système Coq) on ne peut définir que des *fonctions totales* sur leur domaine. C'est-à-dire que, pour qu'une fonction `f` ait le type `nat -> nat`, par exemple, il faut que *pour tout* `x:nat`, *il existe* `y:nat` *tel que* `(f x) = y`.

Ainsi la *discipline de type* de Coq est beaucoup plus exigente que celle de nos langages de programmation usuels où le typage garanti seulement que si la valeur d'une application existe alors elle a le bon type.

## 1.3 Listes paramètrées

Les listes paramètrées sont les listes *polymorphes* de ML. C'est un type inductif défini par les constructeurs `nil` et `cons` :

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

1. en fait, l'expression analysée est celle du *couple* `(n,m)`.

La fonction de calcul de la longueur d'une liste est définie récursivement :

```
Fixpoint length (A:Set) (xs: list A) : nat :=
  match xs with
    nil => 0
  | (cons x xs) => (S (length A xs))
  end.
```

*Stricto sensu*, la fonction `length` a deux arguments : un type (`A`) et une liste (`xs`). Toutefois, dans la définition de `length`, on peut s'arranger pour que la « valeur » du premier argument (le type `A`) soit *implicite* (puisque cette information figure dans le type du second argument : `(list A)`). On écrit pour cela :

```
Fixpoint length {A:Set} (xs: (list A)) : nat :=
  match xs with
    nil => 0
  | (cons x xs) => (S (length xs))
  end.
```

L'argument implicite est indiqué entre accolades.

La concaténation :

```
Fixpoint app {A:Set} (xs ys : (list A)) : (list A) :=
  match xs with
    nil => ys
  | (cons x xs) => (cons x (app xs ys))
  end.
```

Schéma d'application : la *fonction d'ordre supérieur* `map`.

```
Fixpoint map A B:Set (f:A -> B) (xs:list A) : (list B) :=
  match xs with
    nil => nil
  | (cons x xs) => (cons (f x) (map f xs))
  end.
```

Schéma de filtrage : `filter`.

```
Fixpoint filter (A:Set) (p:A -> bool) (xs:list A) : (list A) :=
  match xs with
    nil => nil
  | (cons x xs) => if (p x) then (cons x (filter p xs))
                   else (filter p xs)
  end.
```

Schémas d'accumulation :

```
Fixpoint fold_left A B:Set (f:A -> B -> B) (xs:list A) (b:B) : B :=
  match xs with
    nil => b
  | (cons x xs) => (fold_left f xs (f x b))
  end.

Fixpoint fold_right A B:Set (f:B -> A -> A) (a:A) (xs:list B) : A :=
  match xs with
    nil => a
  | (cons x xs) => (f x (fold_right f a xs))
```

```
  end.
```

### 1.4 Fonctions partielles

Le type paramétré `option` est prédéfini en Coq

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

> Les valeurs du type `option` ne sont pas simplement des `Set`, mais des `Type`.

Ce type permet de « compléter » les définitions de fonctions partielles de manière à les rendre totales. Par exemple, la fonction qui donne l'élément d'une liste à une certaine position est partielle, on la rend totale de cette manière :

```
Fixpoint nth A:Set (i:nat) (xs:(list A)) : (option A) :=
  match i, xs with
    i, nil => None
  | 0, (cons x xs) => (Some x)
  | (S i), (cons x xs) => (nth i xs)
  end.
```

Notez la définition par cas sur les deux arguments `i` et `xs`.

## 2 Spécification

On entend par *spécification* toute *formule* ou ensemble de formules qui décrit les propriétés attendues d'une fonction.

**Spécification *vs* définition** On élabore une spécification dans le but de décrire l'ensemble des valeurs que doit calculer une fonction et ce, en préalable à la définition de la fonction.

Cette règle idéale se heurte bien entendu à quelques exceptions. Il n'existe parfois pas de meilleure spécification d'une fonction que sa définition. Par exemple, comment spécifier ce qu'est la longueur d'une liste mieux qu'en donnant la définition de la fonction qui la calcule ?

### 2.1 Formules

Une formule est une expression qui a une *valeur de vérité*. Ces expressions ont le type `Prop`.

> Les valeurs de vérités qui ont le type `Prop` ne peuvent pas être confondues avec les valeurs booléennes, qui ont le type `bool`.

Le langage des formules que nous utiliserons est constitué des éléments suivants :

1. Des symboles de *prédicat* et de *relation*. Un symbole de prédicat est une *fonction de vérité* avec un type de la forme `A -> Prop`, où `A` est de type `Set`. Plus généralement, une relation est une fonction de vérité avec un type de la forme $\texttt{A}_1$ `-> ...->` $\texttt{A}_n$ `-> Prop`.
2. L'implication entre deux formules (symbole `->`), la conjonction (symbole `/\`) et la disjonction (symbole `\/`) de deux formules, la négation (symbole `not`) d'une formule.

3. La quantification universelle (symbole `forall`) ou existentielle (symbole `exists`) typée d'une formule. On ne dira jamais simplement *pout tout $x$, blabla*, mais *pour tout $x$ de type $A$, blabla*. Idem pour la quantification existentielle.

De manière standard, la spécification d'une fonction `f` prend la forme d'une implication :

$$\texttt{forall (x}_1\texttt{:}\tau_1\texttt{)...(x}_n\texttt{:}\tau_n\texttt{), (P x}_1\texttt{...x}_n\texttt{) -> (Q (f x}_1\texttt{...x}_n\texttt{))}$$

où `P` pose les (éventuelles) conditions initiales des arguments de `f` (les *pré-conditions*) et `Q` donne les conditions que doit satisfaire le résultat de l'application de `f` (les *post-conditions*).

Nous reviendrons sur la notion de spécification ultérieurement.

# 3 Vérification

On entend par *vérification* toute *preuve* qu'une formule de spécification est un théorème. Nous nous intéresserons particulièrement aux *preuves formelles* que nous développerons dans le système d'aide à la preuve Coq.

Une preuve formelle est un enchaînement de *règles de déductions*. Pour prendre un exemple simple : pour prouver la formule `P -> Q`, on suppose que `P` est vrai et on prouve `Q`. C'est-à-dire, on se donne l'*hypothèse* `P` et on démontre `Q`. Cette étape de raisonnement est en général implicite, mais, pour être *vérifiable par machine*, une preuve formelle doit expliciter tous les détails d'utilisation des règles de déduction.

Pour nous familiariser avec le raisonnment formel, nous présentons le système de preuve de la *déduction naturelle* et montrons comment on réalise ses règles avec les commandes de preuves appelées *tactiques* du système Coq.

Puis nous étudierons quelques exemples de théorèmes équationnels en relation avec la manière dont sont définies les fonctions. Pour cela nous présenterons tout d'abord le mécanisme d'*évaluation symbolique*, nous verrons également comment utiliser des équations (*raisonnement équationnel*) et nous aborderons le *raisonnement par induction structurelle*.

Enfin, nous nous interesserons aux preuves de propriétés qui mettent en œuvre les définitions inductives de prédicats et de relations et nous verrons comment on y trouve également une forme de raisonnement par induction.

## 3.1 Règles de *déduction naturelle*

La déduction naturelle est un formalisme de la théorie de la démonstration qui énonce un ensemble de règles de déduction assez proche de la pratique usuelle du raisonnement. Ces règles sont classées en deux catégorie : les règles d'*introduction* et les règles d'*élimination*. *Grosso modo*, les règles d'introduction sont des règles de décomposition de la formule à monter et les règles d'élimination sont des règles d'utilisation d'hypothèses, d'axiomes ou de formules dont on possède un preuve (lemmes et théorèmes).

Les règles sont, pour la plupart *dirigées par la syntaxe*. Il existe des règles d'introduction et d'élimination pour chaque cas de construction de formule.

Nous présentons les règles de déduction comme une relation entre *séquents*. Un séquent est un couple formé d'un ensemble d'*hypothèses* et la formule à montrer. Une hypothèse sera soit une formule, soit une assertion de typage. Une règle de déduction sera donc une relation entre un ensemble de séquents que l'on appelle *prémisses* et un séquent que l'on appelle *conclusion* de la règle. Nous présentons ci-après l'adaptation des règles de déduction naturelle au langage de formules de Coq [2]. Si $\Gamma$ est un ensemble de formule ou d'assertions

2. La version originale de la déduction naturelle traite des formules du calcul des prédicats du premier ordre alors que le système Coq autorise des formules d'un calcul des prédicats d'ordre supérieur.

de typage, et si $F$ est une formule, on note $\Gamma \vdash F$ le séquent formé des hypothèses contenues dans $\Gamma$ et de la formule à monter $F$. On peut distinguer une formule particulière ($G$, par exemple) parmi les hypothèses en notant : $\Gamma, G \vdash F$.

### 3.1.1 La règle *axiome*

Cette règle échappe à la dichotomie introduction/élimination. C'est la règle qui permet de conclure «*par hypothèse*». On l'énonce :
— $\Gamma, F \vdash F$ est montré.

### 3.1.2 Règles d'introduction

**L'implication** c'est la règle que nous avons utilisée en exemple.
— pour montrer $\Gamma \vdash F_1 \to F_2$, il suffit de montrer $\Gamma, F_1 \vdash F_2$.
Remarque : nous verrons que dans le système Coq, les hypothèses sont nommées.

**La conjonction**
— pour montrer $\Gamma \vdash F_1 \wedge F_2$, il suffit de montrer $\Gamma \vdash F_1$ et de montrer $\Gamma \vdash F_2$.
Cette règle a deux prémisses : les séquents $\Gamma \vdash F_1$ et $\Gamma \vdash F_2$. L'existence de règles à plusieurs prémisses confère une structure arborescente aux preuves formelles en déduction naturelle.

**La quantification universelle** cette règle pose des *hypothèses de typage*.
— pour monter $\Gamma \vdash \forall(x : A), F$, il suffit de monter $\Gamma, x : A \vdash F$, pour un $x$ *quelconque*.
Par «$x$ *quelconque*», on entend un $x$ sur lequel ne pèse aucune contrainte. Cocrètement, cela signifie que $x$ ne doit pas avoir d'occurrence libre dans aucune des formules de $\Gamma$.

**La disjonction**
— pour monter $\Gamma \vdash F_1 \vee F_2$, il sufit de montrer $\Gamma \vdash F_1$ ou de montrer que $\Gamma \vdash F_2$.
Il y a donc en fait deux règles d'introduction pour la disjonction.

**La quantification existentielle**
— pour montrer $\Gamma \vdash \exists x : A, F$, il suffit de disposer d'un terme $t : A$ est de monter que $\Gamma \vdash F[t/x]$.

**La négation**
— pour montrer $\Gamma \vdash \neg F$, il suffit que montrer que l'ensemble d'hypothèses $\Gamma, F$ mène à une contradiction ; ce que l'on note : $\Gamma, F \vdash \bot$.
C'est là une forme de *raisonnement par l'absurde*. Nous reviendrons sur ce point.

### 3.1.3 Règles d'élimination

**L'implication** c'est la règle connue sous le nom de *modus ponens*
— pour montrer $\Gamma \vdash F$, il suffit de montrer que $\Gamma \vdash G \to F$ et que $\Gamma \vdash G$.

**La conjonction**
— pour montrer $\Gamma \vdash F$, il suffit de montrer que $\Gamma \vdash F \wedge F'$ ou de montrer que $\Gamma \vdash F' \wedge F$.
Il y a donc ici également deux règles d'élimination de la conjonction.

**La quantification universelle**

— pour monter que $\Gamma \vdash F[t/x]$, pour un ceratin terme $t : A$, il suffit de montrer que $\Gamma \vdash \forall(x : A), F$.

C'est la règle d'instanciation ou de particularisation du quantificateur universel : ce qui est vrai de tout $x : A$ est vrai pour une valeur particulière $t : A$.

**La disjonction**

— pour montrer $\Gamma \vdash F$, il suffit de montrer que l'on a deux formules $F_1$ et $F_2$ telles que $\Gamma \vdash F_1 \vee F_2$ et que $\Gamma, F_1 \vdash F$, ainsi que $\Gamma, F_2 \vdash F$.

C'est là une forme du *raisonnement pas cas*.

**La quantification existentielle**

— pour montrer $\Gamma \vdash F$, il suffit de monter que $\Gamma \vdash \exists(x : A), F'$ et que $\Gamma, F'[y/x] \vdash F$, avec $y$ non libre dans $\Gamma$, $F'$ et $F$.

**La négation**

— pour montrer $\Gamma \vdash \bot$, c'est-à-dire que $\Gamma$ est contradictoire, il suffit d'avoir une formule $F$ telle que $\Gamma \vdash F$ et que $\Gamma \vdash \neg F$.

### 3.1.4 Absurde

La règle de raisonnement par l'absurde *classique* n'existe pas dans le système Coq. Pour montrer $\Gamma \vdash F$ on ne peut pas supposer que $\neg F$ et en déduire une contradiction.

On dispose toutefois d'une règle de *réduction à l'absurde* :

— pour montrer $\Gamma \vdash F$, il suffit de montrer que $\Gamma$ est contradictoire, c'est-à-dire que $\Gamma \vdash \bot$.

C'est la règle du *ex falso quodlibet* (du faux découle ce que l'on veut).

L'absence de la règle *classique* de raisonnement par l'absurde fait de la logique implémentée dans le système Coq une logique *intutionniste*.

**Tiers exclus** La logique intuitionniste n'admet pas le *principe du tiers exclus*. Ce principe pose que une formule $F$ est nécessairement vraie ou fausse ; c'est-à-dire que la formule $F \vee \neg F$ est vraie, quelque soit $F$.

**Double négation** Refuser le tiers exclus est équivalent à refuser l'équivalence entre une formule $F$ est sa double négation $\neg\neg F$.

## 3.2 Des règles aux tactiques

Nous allons voir comment le système Coq réalise les règles de raisonnement de la déduction naturelle. Nous procèderons en plusieurs étapes. Nous verrons tout d'abord comment sont réalisées les règles d'introduction et d'élimination pour la logiques des formules qui n'utilisesnt que la flèche et le quantificateur universel que l'on peut appeler *logique minimale*. Nous verrons ensuite les connecteurs de conjonction et de disjonction. Puis nous nous intéresserons à la négation.

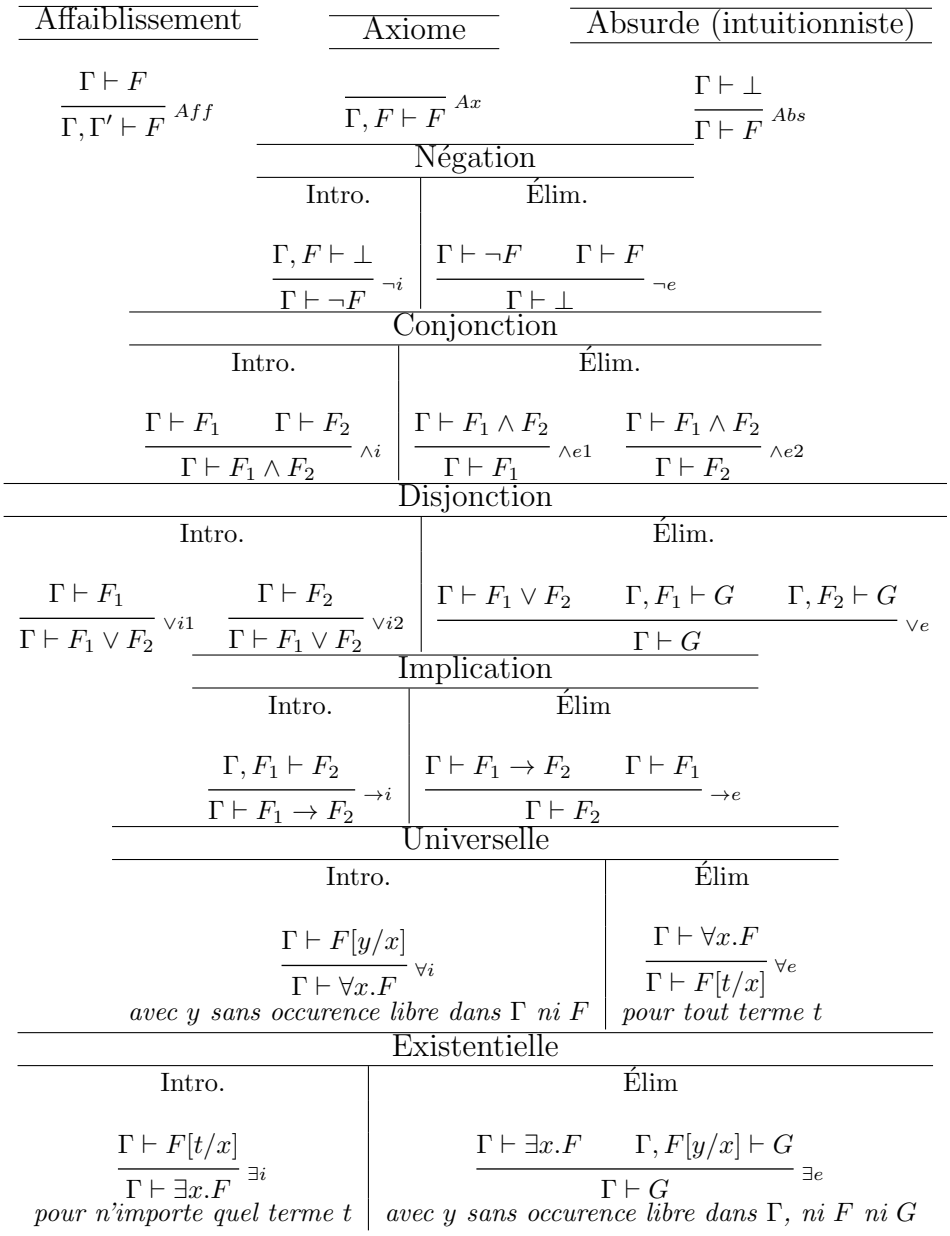


FIGURE 1 – Règles de déduction naturelle

### 3.2.1 Logique minimale

**Introductions et axiomes** Pour faire simple, commençons par prouver que `P -> P`, pour toute proposition `P`. Ce «théorème» est évidemment trivial, mais étudions cependant les étapes de sa preuve en Coq.

Nous voulons démontrer : `forall (P:Prop), P -> P`. Si l'on suit les règles de la déduction naturelle, on prouve ce théorème ainsi :

- pour montrer `forall (P:Prop), P -> P`, on se donne un `P` quelconque, de type `Prop` et on montre `P -> P` (régle d'introduction du quantificateur universel) ;
- pour monter `P -> P`, on suppose `P` et on montre `P` (règle d'introduction de la flèche) ;
- par hypothèse, on a démontré `P` (règle axiome).

Voici comment se déroule cette preuve avec le système interactif Coq :

- on pose le théorème à montrer :

```
Coq < Theorem tauto1: forall (P:Prop), P -> P.
1 subgoal

  ============================
   forall P : Set, P -> P

tauto1 <
```

Le système a entegistré qe nous souhaitions démonter un théorème (mot clé `Theorem`) auquel nous donnerons le nom de «`tauto1`». Par l'indication «`1 subgoal`», il nous informe qu'il y a une formule à démontrer, un *sous-but* de preuve. Cette formule est l'énoncé du théorème. La ligne de `=` qui surmonte cette formule est l'équivalent du symbole $\vdash$ des séquents qui sépare les hypothèses de la formule à montrer – ici, il n'y a pas encore d'hypothèse. Enfin, il attend que nous lui fournissions les tactiques pour démontrer le théorème `tauto1` (*prompt* `tauto1 <`).

Procédons à la preuve :

- on invoque la règle d'introduction du quantificateur universel avec la tactique `intro`

```
tauto1 < intro.
1 subgoal

  P : Prop
  ============================
   P -> P
```

Le système a fait passer l'assertion de typage `(P:Prop)` de la formule à montrer dans les hypothèses ;

- on invoque ensuite la règle d'introduction de l'implication, toujours avec la tactique `intro` :

```
tauto1 < intro.
1 subgoal

  P : Prop
  H : P
  ============================
   P
```

Le système a fait passer le `P` à gauche de la flèche dans notre ensemble d'hypothèses et a donné le nom `H` à cette nouvelle hypothèse. La formule à montrer est `P`.

- on invoque la règle axiome avec la tactique `assumption` :

```
tauto1 < assumption.
No more subgoals.
```

Le système nous informe qu'il n'y a plus rien à montrer, la preuve est achevée.

- on clôt effectivement la preuve avec la commande `Qed` :

```
tauto1 < Qed.
intro.
intro.
assumption.

Qed.
tauto1 is defined
```

Le système résume l'ensemble de tactiques utilisées pour la preuve, ce que l'on appelle un *script de preuve* et nous informe que le théorème `tauto1` est défini. Il pourra être utilisé par la suite.

Nous pouvons effectuer les deux étapes d'introduction avec la tactique `intros` :

```
Coq < Theorem tauto1: forall (P:Set), P -> P.
1 subgoal

  ============================
   forall P : Set, P -> P

tauto1 < intros.
1 subgoal

  P : Set
  H : P
  ============================
   P

tauto1 < assumption.
No more subgoals.

tauto1 < Qed.
intros.
assumption.

Qed.
tauto1 is defined

Coq <
```

Ainsi, une tactique est plus qu'une règle de déduction. Une tactique peut enchaîner et combiner les règles de déduction.

Le «théorème» que nous venons de montrer est tellement trivial qu'il est démontrable automatiquement, par la tactique... `trivial` :

```
Coq < Theorem tauto1: forall (P:Set), P -> P.
1 subgoal
```

```
  ============================
   forall P : Set, P -> P

tauto1 < trivial.
No more subgoals.

tauto1 < Qed.
trivial.

Qed.
tauto1 is defined
```

Toujours à titre d'exemple trivial, considérons que `P` n'est pas une proposition, mais un prédicat portant sur des valeur d'un certain type `A`.

```
Theorem tauto2 : forall (A:Set) (P:A -> Prop) (x:A), (P x) -> (P x).
1 subgoal

  ============================
   forall (A : Set) (P : A -> Prop) (x : A), P x -> P x

tauto2 <
```

Notez le type de `P` et la quantification *du premier ordre* sur `x`.

Le schéma de preuve est identique au théorème `tauto1` : une suite d'introduction, puis la conclusion avec la règle axiome. Observons simplement les assertions de typage et l'hypothèse que nous donne la suite d'introductions.

```
tauto2 < intros.
1 subgoal

  A : Set
  P : A -> Prop
  x : A
  H : P x
  ============================
   P x

tauto2 < assumption.
No more subgoals.

tauto2 < Qed.
intros.
assumption.

Qed.
tauto2 is defined
```

**Éliminations** Nous allons regarder à présent comment utiliser des hypothèses avec la règle d'élimination de la flèche.

Nous voulons montrer que la flèche est transitive, c'est-à-dire que

```
  forall (P Q R : Prop), (P -> Q) -> (Q -> R) -> P -> R.
```

En langue naturelle, on écrirait la preuve ainsi : soit `P:Prop`, `Q:Prop` et `R:Prop`, supposons `H1 : P -> Q`, `H2 : Q -> R` et `H3 : P`, il faut montrer `R`.
De `H1` et `H3`, on déduit `Q`, par élimination de la flèche ; et de `H2` et `Q`, toujours par élémination de la flèche, on déduit `P`. CQFD.

Raisonner de cette manière, c'est procéder par *chaînage avant* : des ≪faits≫ exprimés par les hypothèses, on déduit de nouveaux ≪faits≫ (ici, `Q`), jusqu'à trouver celui que l'on cherche à montrer. Mais ce n'est pas la manière naturelle de Coq qui est plutôt conçu pour raisonner en *chaînage arrière*. Cette manière correspond à la façon dont nous avons exprimé les régles de déduction : ≪pour montrer $F$, if suffit de montrer $G \to F$ et $G$≫. Voici comment on rédigerait la preuve dans ce style : soit `P:Prop`, `Q:Prop` et `R:Prop`, supposons `H1 : P -> Q`, `H2 : Q -> R` et `H3 : P`, il faut montrer `R`.
Pour montrer `R`, *par application* de `H2`, il suffit de montrer `Q`.
Pour montrer `Q`, *par application* de `H1`, il suffit de montrer `P`. Ce que l'on a par hypothèse.

Nous avons souligné l'usage de l'expression ≪*par application*≫. C'est à dessein, car la tactique Coq qui réalise cela s'appelle précisément `apply`. Voici donc les étapes de la preuve de la transitivité de la flèche en Coq :

```
Coq <   Theorem arrow_trans : forall (P Q R : Prop),
    (P -> Q) -> (Q -> R) -> P -> R.
Coq < 1 subgoal

  ============================
   forall P Q R : Prop, (P -> Q) -> (Q -> R) -> P -> R

arrow_trans < intros P Q R H1 H2 H3.
1 subgoal

  P, Q, R : Prop
  H1 : P -> Q
  H2 : Q -> R
  H3 : P
  ============================
   R
```

Notez comment on peut nommer ses hypothèses lorsque l'on invoque la tactique `intros`.

```
arrow_trans < apply H2.
1 subgoal

  P, Q, R : Prop
  H1 : P -> Q
  H2 : Q -> R
  H3 : P
  ============================
   Q
```

Le système nous indique ce qu'il reste à montrer.

La fin de la preuve est facile :

```
arrow_trans < apply H1.
1 subgoal

  P, Q, R : Prop
```

```
  H1 : P -> Q
  H2 : Q -> R
  H3 : P
  ============================
   P

arrow_trans < assumption.
No more subgoals.

arrow_trans < Qed.
intros P Q R H1 H2 H3.
apply H2.
apply H1.
assumption.

Qed.
arrow_trans is defined

Coq <
```

Lorsque l'on dit «par application de l'hypothèse...» ou, plus fréquemment, «par application du théorème...», on dit en fait quelque chose que les logiciens ont appelé, en théorie de la démonstration, *l'interprétation algorithmique des preuves*. Dans cette interprétation, une preuve de `A -> B` est une *fonction* qui, à toute preuve de `A`, associe une preuve de `B`.

Dans notre exemple, quand nous supposons `H1 : P -> Q` et `H3 : P`, nous supposons que nous avons une preuve `H1` de `P -> Q` et une preuve `H3` de `P`. *Dans ce context*, l'application `(H1 H3)` est une preuve de `Q` (*i.e.* `(H1 H3) : Q`) et alors `(H2 (H3 H1))` est une preuve `R`.

Cette manière de noter est celle utilisée par le système Coq : les preuves sont des fonctions. On peut le voir en demandant au système d'afficher la preuve que nous avons réalisée avec les tactiques :

```
Coq < Print arrow_trans.
arrow_trans =
fun (P Q R : Prop) (H1 : P -> Q) (H2 : Q -> R) (H3 : P) => H2 (H1 H3)
     : forall P Q R : Prop, (P -> Q) -> (Q -> R) -> P -> R
```

La formule à démontrer est le *type* de la fonction qui la démontre.

Comme nous l'avons fait au paragraphe précédent, considérons que `P`, `Q` et `R` sont des prédicats. On veut montrer :

```
Theorem arrow_trans0 :
  forall (A:Set) (P Q R : A -> Prop),
    (forall (x:A), (P x) -> (Q x)) -> (forall (x:A), (Q x) -> (R x))
    -> forall (x:A), (P x) -> (R x).
```

Il est important de remarquer ici que cette formule n'est pas la même que

```
  forall (A:Set) (P Q R : A -> Prop) (x:A),
     ((P x) -> (Q x)) -> ((Q x) -> (R x)) -> (P x) -> (R x)
```

En effet, dans la formule ci-dessus, le `x` universellement quantifié est commun à `(P x) -> (Q x)`, `(Q x) -> (R x)`, et `(P x) -> (R x)`. Ce qui n'est pas le cas dans la formule que nous voulons montrer. Celle-ci

contient trois quantification sur `x`, ce qui rend indépendantes leurs occurences dans `(P x) -> (Q x)`, `(Q x) -> (R x)`, et `(P x) -> (R x)`. La formule à montrer est en fait équilavente à :

```
Theorem arrow_trans0 :
  forall (A:Set) (P Q R : A -> Prop),
    (forall (x:A), (P x) -> (Q x)) -> (forall (y:A), (Q y) -> (R y))
    -> forall (z:A), (P z) -> (R z).
```

Le script de preuve de ce théorème peut être identique à celui de sa version propositionnelle. Toutefois, l'effet de la tactique `apply` est assez différent. Pour le voir, suivons les étapes de cette preuve.

- on se donne les hypothèses nécessaires (introductions) :

```
arrow_trans0 < intros.
1 subgoal

  A : Set
  P, Q, R : A -> Prop
  H : forall x : A, P x -> Q x
  H0 : forall y : A, Q y -> R y
  z : A
  H1 : P z
  ============================
   R z
```

Pour obtenir `(R z)`, on peut utiliser l'hypothèse `H0` où `y` est instancié par `z` (éléminations du quantificateur universel et de la flèche :

```
arrow_trans0 < apply H0.
1 subgoal

  A : Set
  P, Q, R : A -> Prop
  H : forall x : A, P x -> Q x
  H0 : forall y : A, Q y -> R y
  z : A
  H1 : P z
  ============================
   Q z
```

La commande `apply` a donc combiné ici deux règles d'éliminations. On procède de même avec l'hypothèse `H` :

```
arrow_trans0 < apply H.
1 subgoal

  A : Set
  P, Q, R : A -> Prop
  H : forall x : A, P x -> Q x
  H0 : forall y : A, Q y -> R y
  z : A
  H1 : P z
  ============================
   P z
```

Et on conclut :

```
arrow_trans0 < assumption.
No more subgoals.

arrow_trans0 < Qed.
intros.
apply H0.
apply H.
assumption.

Qed.
arrow_trans0 is defined
```

Les preuves que nous avons réalisées jusqu'à présents sont tellement simple qu'un algorithme de recherche automatique de preuve peut les trouver. La tactique `trivial`, que nous avons déjà vue, n'est pas assez puissante pour montrer ce dernier théorème. On utilisera sa grande sœur, la tactique `auto`/

```
arrow_trans0 < auto.
No more subgoals.

arrow_trans0 < Qed.
auto.

Qed.
arrow_trans0 is defined
```

### 3.2.2 Conjonction

Nous illustrons l'utilisation de l'introduction et de l'élimination d'une conjonction par la preuve de la formule suivante :

```
Theorem tauto3 :
  forall (A:Set) (P Q : A -> Prop),
    (forall (x:A), (P x)) /\ (forall (y:A), (Q y))
    -> forall (z:A), (P z) /\ (Q z).
```

On peut rédiger la preuve en déduction naturelle de cette formule de la manière suivante :

- soit `A:Set`, `P Q : A -> Prop`, supposons `H : (forall (x:A), (P x)) /\(forall (y:A), (Q y))` et soit `z:A`, il faut montrer `(P z) /\(Q z)` ;

- pour montrer `(P z) /\(Q z)`, par introduction de la conjonction, il suffit de montrer que `(P z)` et que `(Q z)` ;

- pour montrer `(P z)`, par élimination du quantificatuer universel, il suffit de montrer que `forall (x:A), (P x)` ;

- ce que l'on déduit de l'hypothèse `H`, par élimination de la conjonction ;

- reste à montrer `(Q z)`. Pour montrer `(Q z)`, il suffit de montrer `forall (y:A), (Q y)` ;

- ce que l'on déduit de l'hypothèse `H`, par élimination de la conjonction et qui achève la preuve de notre théorème.

Transcrivons cette preuve en script pour le système Coq.

On applique les règles d'introduction avec la tactique `intros` :

```
Coq < 1 subgoal

  ============================
   forall (A : Set) (P Q : A -> Prop),
   (forall x : A, P x) /\ (forall y : A, Q y) -> forall z : A, P z /\ Q z

tauto3 < intros.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) /\ (forall y : A, Q y)
  z : A
  ============================
   P z /\ Q z
```

Notez que le système n'a pas appliqué la règle d'introduction de la conjonction : la formule à montrer contient toujours la conjonction recherchée. Il faut utiliser une autre commande pour invoquer la règle d'introduction de la conjonction : la tactique `split` qui va engendrer 2 sous-buts de preuve `(P z)` et `(Q z)` :

```
tauto3 < split.
2 subgoals

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) /\ (forall y : A, Q y)
  z : A
  ============================
   P z

subgoal 2 is:
 Q z
```

On traite ces deux buts dans l'ordre où le système nous les présente.

On sait comment montrer `(P z)` si l'on a `forall (x:A), (P x)`. Et on a `forall (x:A), (P x)` par élimination de la conjonction, si l'on a la conjonction `(forall x : A, P x) /\(forall y : A, Q y)`. Or, cette conjonction est l'une de nos hypothèses. On peut réaliser en Coq un enchaînement analogue en commençant par invoquer la tactique `elim`.

```
tauto3 < elim H.
2 subgoals

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) /\ (forall y : A, Q y)
  z : A
  ============================
   (forall x : A, P x) -> (forall y : A, Q y) -> P z

subgoal 2 is:
 Q z
```

Notez que nous avons donné en argument à la tactique `elim` le nom d'une formule qui est une conjonction

(ici, l'hypothèse H). *Grosso modo*, l'invocation de `elim H` réalise le «raisonnement» général suivant ; pour montrer F, si on a une preuve que $F_1$ /\$F_2$, il suffit de montrer F sous hypothèses $F_1$ et $F_2$. Ici, «$F_1$ /\$F_2$» est notre hypothèse H ; sa preuve est donc immédiate.

Contrairement à ce qui est formulé dans la règle de déduction naturelle, le système Coq ne donne nous donne pas directement `forall (x : A), (P x)` et `forall (y:A), (Q y)` comme hypothèses. Nous devons les obtenir avec la commande `intros` :

```
tauto3 < intros.
2 subgoals

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) /\ (forall y : A, Q y)
  z : A
  H0 : forall x : A, P x
  H1 : forall y : A, Q y
  ============================
   P z

subgoal 2 is:
 Q z
```

On obtient `(P z)` par application de H0.

```
tauto3 < apply H0.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) /\ (forall y : A, Q y)
  z : A
  ============================
   Q z
```

Reste à montrer le second membre de la conjonction. Ce que l'on fait en trois étapes analogues et qui achève notre preuve :

```
tauto3 < elim H.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) /\ (forall y : A, Q y)
  z : A
  ============================
   (forall x : A, P x) -> (forall y : A, Q y) -> Q z

tauto3 < intros.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) /\ (forall y : A, Q y)
```

```
  z : A
  H0 : forall x : A, P x
  H1 : forall y : A, Q y
  ============================
   Q z

tauto3 < apply H1.
No more subgoals.

tauto3 < Qed.
intros.
split.

 elim H.
 intros.
 apply H0.

 elim H.
 intros.
 apply H1.

Qed.
tauto3 is defined
```

On peut raccourcir légèrement ce script en factorisant l'élimination de l'hypothèse `H` :

```
Coq < Theorem tauto3 :
  forall (A:Set) (P Q : A -> Prop),
    (forall (x:A), (P x)) /\ (forall (y:A), (Q y))
    -> forall (z:A), (P z) /\ (Q z).
Coq < Coq < Coq < 1 subgoal

  ============================
   forall (A : Set) (P Q : A -> Prop),
   (forall x : A, P x) /\ (forall y : A, Q y) -> forall z : A, P z /\ Q z

tauto3 < intros.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) /\ (forall y : A, Q y)
  z : A
  ============================
   P z /\ Q z

tauto3 < elim H.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) /\ (forall y : A, Q y)
```

```
  z : A
  ============================
   (forall x : A, P x) -> (forall y : A, Q y) -> P z /\ Q z

tauto3 < intros.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) /\ (forall y : A, Q y)
  z : A
  H0 : forall x : A, P x
  H1 : forall y : A, Q y
  ============================
   P z /\ Q z

tauto3 < split.
2 subgoals

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) /\ (forall y : A, Q y)
  z : A
  H0 : forall x : A, P x
  H1 : forall y : A, Q y
  ============================
   P z

subgoal 2 is:
 Q z

tauto3 < apply H0.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) /\ (forall y : A, Q y)
  z : A
  H0 : forall x : A, P x
  H1 : forall y : A, Q y
  ============================
   Q z

tauto3 < apply H1.
No more subgoals.

tauto3 < Qed.
intros.
elim H.
intros.
split.
```

```
 apply H0.

 apply H1.

Qed.
tauto3 is defined
```

### 3.2.3 Disjonction

Illustrons l'utilisation de l'élimination et de l'introduction d'une disjonction par le preuve de la foumule suivante :

```
Theorem tauto4 :
  forall (A:Set) (P Q : A -> Prop),
    (forall (x:A), (P x)) \/ (forall (x:A), (Q x))
    -> forall (y:A), (P y) \/ (Q y).
```

Le schèma de la preuve à réaliser est la suivant :

- supposons `A:Set`, `P Q : A -> Set`, `H: (forall (x:A), (P x)) \/(forall (x:A), (Q x))` et `y:A` et montrons `(P y) \/(Q y)` ;

- pour montrer `(P y) \/(Q y)`, par élimination de la disjonction sur l'hypothèse `H`, il suffit de montrer `(P y) \/(Q y)` en supposant `forall (x:A), (P x)` et de montrer `(P y) \/(Q y)` en supposant `forall (x:A), (Q x)` ;

- pour montrer `(P y) \/(Q y)` en supposant `forall (x:A), (P x)`, il suffit de montrer `(P y)` (introduction de la disjonction) ;

- on obtient `(P y)` par application de l'hypothèse `forall (x:A), (P x)` ;

- reste à montrer `(P y) \/(Q y)` en supposant `forall (x:A), (Q x)`. Pour cela, il suffit de montrer `(Q y)` (introduction de la disjonction) ;

- on obtient `(Q y)` par application de l'hypothèse `forall (x:A), (Q x)` et on achève ainsi la démonstration.

Transcrivons ce schèma en script pour le système Coq.

```
Coq < Theorem tauto4 :
  forall (A:Set) (P Q : A -> Prop),
    (forall (x:A), (P x)) \/ (forall (x:A), (Q x))
    -> forall (y:A), (P y) \/ (Q y).

1 subgoal

  ============================
   forall (A : Set) (P Q : A -> Prop),
   (forall x : A, P x) \/ (forall x : A, Q x) -> forall y : A, P y \/ Q y

tauto4 < intros.
1 subgoal

  A : Set
  P, Q : A -> Prop
```

```
  H : (forall x : A, P x) \/ (forall x : A, Q x)
  y : A
  ============================
   P y \/ Q y
```

À ce point, on ne sait encore s'il faut montrer le membre gauche de la disjonction ou son membre droit ; c'est-à-dire que l'on ne sait pas quelle règle d'introduction utiliser pour la disjonction. Mais l'hypothèse `H` nous donne deux cas (`forall (x : A), (P x)` et `forall (x : A), (Q x)`) dont chacun permet d'en décider. La tactique `elim` appliquée à une disjonction réalise le raisonnement par cas qu'est l'élimination de la disjonction.

```
tauto4 < elim H.
2 subgoals

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) \/ (forall x : A, Q x)
  y : A
  ============================
   (forall x : A, P x) -> P y \/ Q y

subgoal 2 is:
 (forall x : A, Q x) -> P y \/ Q y
```

Notez qu'ici encore, le système ne nous donne pas directement les hypothèses issues de l'éminitation de la disjonction et il faut utiliser une invocation de `intro`.

Dans le premier cas, on montre le membre gauche (`P y`) :

```
tauto4 < intro.
2 subgoals

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) \/ (forall x : A, Q x)
  y : A
  H0 : forall x : A, P x
  ============================
   P y \/ Q y

subgoal 2 is:
 (forall x : A, Q x) -> P y \/ Q y

tauto4 < left.
2 subgoals

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) \/ (forall x : A, Q x)
  y : A
  H0 : forall x : A, P x
  ============================
   P y
```

```
subgoal 2 is:
 (forall x : A, Q x) -> P y \/ Q y
```

La tactique `left` implente l'une des règle d'introduction de la disjonction.

On conclut ce cas par application de l'hypothèse obtenue :

```
tauto4 < apply H0.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) \/ (forall x : A, Q x)
  y : A
  ============================
   (forall x : A, Q x) -> P y \/ Q y
```

Le second cas est similaire, à cette différence que l'autre règle d'introduction de la disjonction s'appelle `right` :

```
tauto4 < intro.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) \/ (forall x : A, Q x)
  y : A
  H0 : forall x : A, Q x
  ============================
   P y \/ Q y

tauto4 < right.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : (forall x : A, P x) \/ (forall x : A, Q x)
  y : A
  H0 : forall x : A, Q x
  ============================
   Q y

tauto4 < apply H0.
No more subgoals.

tauto4 < Qed.
intros.
elim H.

 intro.
 left.
 apply H0.

 intro.
```

```
 right.
 apply H0.

Qed.
tauto4 is defined
```

### 3.2.4 Existentielle

On illustre l'élimination et l'introduction du quantificateur existentiel avec la preuve de :

```
Coq < Theorem tauto5 :
  forall (A:Set) (P : Prop) (Q : A -> Prop),
    (exists (x:A), P -> (Q x)) -> P -> exists (x:A), (Q x).

1 subgoal

  ============================
   forall (A : Set) (P : Prop) (Q : A -> Prop),
   (exists x : A, P -> Q x) -> P -> exists x : A, Q x
```

Comme c'est presque toujours le cas, on se donne toutes les hypothèses possibles.

```
tauto5 < intros.
1 subgoal

  A : Set
  P : Prop
  Q : A -> Prop
  H : exists x : A, P -> Q x
  H0 : P
  ============================
   exists x : A, Q x
```

Pour obtenir la preuve d'une formule existentielle, il faut trouver un terme, appelé *témoin*, pour la variable quantifiée – ce terme peut être lui-même une simple variable. C'est la règle d'introduction. À ce point de notre preuve, nous n'en disposons pas. Mais, comme pour l'exemple que nous avons pris pour la disjonction, nous pouvons obtenir ce témoin de l'hypothèse `H`, par règle d'élimination.

```
tauto5 < elim H.
1 subgoal

  A : Set
  P : Prop
  Q : A -> Prop
  H : exists x : A, P -> Q x
  H0 : P
  ============================
   forall x : A, (P -> Q x) -> exists x0 : A, Q x0
```

Notez comment le système a renommé la variable existentiellement quantifiée dans la formule à montrer.

Pour obtenir en hypothèse un `x` tel que `P -> (Q x)`, on utilise la commande `intros` :

```
tauto5 < intros.
1 subgoal
```

```
  A : Set
  P : Prop
  Q : A -> Prop
  H : exists x : A, P -> Q x
  H0 : P
  x : A
  H1 : P -> Q x
  ============================
   exists x0 : A, Q x0
```

On dispose maintenant du témoin qui va nous permettre de conclure, c'est `x`. La tactique `exists` permet d'introduire ce témoin :

```
tauto5 < exists x.
1 subgoal

  A : Set
  P : Prop
  Q : A -> Prop
  H : exists x : A, P -> Q x
  H0 : P
  x : A
  H1 : P -> Q x
  ============================
   Q x
```

Le système peut se débrouiller pour achever la preuve avec la tactique `auto` :

```
tauto5 < auto.
No more subgoals.

tauto5 < Qed.
intros.
elim H.
intros.
exists x.
auto.

Qed.
tauto5 is defined
```

### 3.2.5 Absurde et négation

Montrons

```
Theorem tauto6 : forall (P:Prop), P -> not (not P).
1 subgoal

  ============================
   forall P : Prop, P -> ~ ~ P

tauto6 < intros.
```

```
1 subgoal

  P : Prop
  H : P
  ============================
   ~ ~ P
```

La tactique `intros` n'applique pas, par défaut la règle d'introduction de la négation. Il faut le demander explicitement :

```
tauto6 < intro.
1 subgoal

  P : Prop
  H : P
  H0 : ~ P
  ============================
   False
```

La proposition prédéfinie `False` correspond au $\bot$ utilisé pour poser les règles de la négation et de l'absurde. Pour l'obtenir, on applique l'élimination de la négation, avec l'hypothèse négative `H0`

```
tauto6 < elim H0.
1 subgoal

  P : Prop
  H : P
  H0 : ~ P
  ============================
   P
```

Le système nous demande alors de prouver l'autre prémisses de la régle d'élimination. Ce qui est ici immédiat, par hypothèse :

```
tauto6 < assumption.
No more subgoals.

tauto6 < Qed.
intros.
intro.
elim H0.
assumption.

Qed.
tauto6 is defined
```

La tactique correpondant à la règle d'absurdité intuitionniste est la tactique `exfalso`. Elle se contente de remplacer la formule à montrer par `False`.

Nous avons fait le tour du noyau logique pur. Nous allons à préent nous tourner vers deux éléments importants de la spécification et de la vérification formelle : les propriétés d'égalité entre valeurs et le raisonnement équationnel ainsi que le raisonnement avec les structures de données et les prédicats inductifs.

## 3.3 Raisonnement équationnel

### 3.3.1 Le prédicat d'égalité

Du point de vue logique, l'égalité est *la plus petite relation binaire réflexive*. On utilise le symbole usuel `=` pour noter l'égalité. La relation d'égalité est paramétrée par les types des valeurs comparées. La propriété de réflexivité pour les valeurs appartenant à un type `A` qui est un `Set` s'énonce

```
forall (A:Set) (x:A), x=x.
```

La réflexivité est l'outil de base des preuves d'égalités : deux expressions identiques désignent des valeurs égales. Par exemple, $1 + x = 1 + x$ est immédiatement prouvé par réflexivité.

Nous ne considèrerons pas d'autre égalité qu'entre valeurs d'un type est un un `Set`.

Le fait que l'égalité soit la *plus petite* relation réflexive lui confère une propriété primordiale : si $x$ vérifie une propriété $P$ est si $x = y$ alors $y$ vérifie également la propriété $P$ [3]. Formellement :

```
forall (A:Set) (x:A) (P : A -> Prop),
  (P x) -> forall (y:A), (x=y) -> (P y).
```

Cette propriété autorise, permet d'énoncer la règle raisonnement suivante :
— Pour montrer $F[t]$, il suffit de montrer que $F[u]$ et $t = u$.
La réflexivité et la propriété de substituabilité se transcrivent ainsi sous forme de règle de déduction :

$$\frac{}{\Gamma \vdash t \texttt{ = } t}\ refl \qquad \left|\ \frac{\Gamma \vdash F[u] \qquad \Gamma \vdash t \texttt{ = } u}{\Gamma \vdash F[t]}\ eq\right.$$

L'égalité est une *relation d'équivalence*. C'est-à-dire qu'elle est réflexive (par définition), symétrique et transitive. Montrons le :

```
Theorem eq_sym : forall (A:Set) (x y:A), (x=y) -> (y=x).
1 subgoal

  ============================
   forall (A : Set) (x y : A), x = y -> y = x

eq_sym < intros.
1 subgoal

  A : Set
  x, y : A
  H : x = y
  ============================
   y = x
```

On pourra conclure en remplaçant dans la formule à montrer `x` par `y` – ce à quoi nous autorise l'hypothèse `H` – et en invoquant la réflexivité de l'égalité. Les tactique permettant ces étapes de preuves sont `rewrite` et `reflexivity` :

```
eq_sym < rewrite H.
1 subgoal

  A : Set
```

3. C'est de cette manière que Liebniz définissait *l'identité* : *aedem sunt, quae mutuo substitui possunt, salva veritate*

```
  x, y : A
  H : x = y
  ============================
   y = y

eq_sym < reflexivity.
No more subgoals.

eq_sym < Qed.
intros.
rewrite H.
reflexivity.

Qed.
eq_sym is defined
```

Nous laissons la preuve de

```
Theorem eq_trans :
  forall (A:Set) (x y z:A), (x=y) -> (y=x) -> (x=z).
```

en exercice.

### 3.3.2 Évaluation symbolique

Nous avons vu que $1+2 = 1+2$ est trivialement vrai par réflexivité. Nous savons que $1+2 = 2+1$. mais cette égalité n'est pas une conséquence de la réflexivité. Pour montrer celle-ci nous pouvons *calculer* les valeurs des expressions $1+2$ et $2+1$ qui donnent toutes deux 3 et alors $3 = 3$ est vérifiée par réflexivité.

Nous présentons dans ce qui suit le mécanisme de *calcul* du système Coq.

**Le modèle du langage fonctionnel** que nous utilisons est le *λ-calcul*. C'est un langage de programmation très rudimentaire basé sur les symboles réservés `fun`, `=>`, les parenthèses `(` et `)` et un ensemble de symboles atomiques (symboles de variables ou de constructeurs) disjoint des symboles déjà réservés. La grammaire du langage s'exprime ainsi :

$$t ::= a \mid \texttt{fun}\ x\ \texttt{=>}\ t \mid \texttt{(}t\ t\texttt{)}$$

Il définit l'ensemble $t$ des *termes* du λ-calcul :

- $a$ tout symbole atomique est un terme.
- `fun` $x$ `=>` $t$ l'*abstraction* du symbole de variable $x$ vis-à-vis du terme $t$ est un terme : la fonction de paramètre $x$ et de corps $t$.
- `(`$t$ $t$`)` l'*application* d'un terme à un autre est un terme : le premier $t$ est la fonction, le second, son argument.

On considère un sous ensemble des termes correspondants à ceux qui sont *correctement typés* vis-à-vis d'un ensemble de *règles de typages*. Nous ne développons pas ce point, pour l'instant.

**Le modèle de calcul** associé à ce langage est la *β-réduction*, elle-même basée sur la *substitution* .

`(fun` $x$ `=>` $t$ $u$`)` se réduit en $t[u/x]$

L'écriture `(fun` $x$ `=>` $t$ $u$`)` se lit comme l'application de l'abstraction `fun` $x$ `=>` $t$ au terme $u$. L'écriture $t[u/x]$ se lit : $t$ dans lequel $u$ remplace (les occurrences libres de) $x$. C'est une *évaluation symbolique* en ce sens que la valeur d'un terme est elle-même un terme. Si, pour un terme $t$, il n'est pas possible d'appliquer la règle de réduction, sur $t$ lui-même ou aucun de ses sous-termes, alors $t$ est sa propre valeur. On dit que $t$ est en *forme normale*. La valeur d'un terme est donc sa forme normale.

Il existe des termes qui n'ont pas de forme normale. Une *théorie des types* comme le *calcul des constructions indicutives* garanti que tout terme correctement typé a une forme normale.

Avec les langages *à la ML*, ce noyau est enrichi de la construction `match-with` dont la forme générale est

`match` $t$ `with` $p_1$ `=>` $t_1$ `|` $\ldots$ `|` $p_n$ `=>` $t_n$ `end`.

Les termes $p_1 \ldots p_n$ appartiennent à un sous-ensemble de termes appelés *motifs* (*pattern*, en anglais). Ils ne doivent contenir que des symboles de constructeurs ou des symboles de variables. De plus, ils sont *linéaires*, c'est-à-dire qu'un symbole de variable ne peut y avoir au plus qu'une seule occurrence.

La construction `if-then-else` est un cas particulier de `match-with` : le terme `if` $e$ `then` $e_1$ `else` $e_2$ est égal à `match` $e$ `with true =>` $e_1$ `| false =>` $e_2$ `end.`

La règle de réduction de la construction `match-with` met en jeu la notion de *filtrage* (*pattern matching*, en anglais). Soit $p$ un motif dont les variables sont $x_1, \ldots, x_n$. On dit que le motif $p$ filtre un terme $t$ si et seulement si il existe des termes $t_1, \ldots, t_n$ tels que $p[t_1/x_1, \ldots, t_n/x_n] = t$.

`match` $t$ `with` $p_1$ `=>` $t_1$ `|` $\ldots$ `|` $p_n$ `=>` $t_n$ `end` se réduit en $t_i[u_1/x_1, \ldots, u_k/x_k]$
où $i$ est le plus petit parmi $[1 \ldots n]$ tel que $t$ se réduit en $p_i[u_1/x_1, \ldots, u_k/x_k]$.

*Stricto sensu* il existe une règle de réduction correspondant au fait de remplacer un symbole par sa définition, lorsqu'il en a une. Nous n'insistons pas ici sur ce point.

**Évaluation et valeurs** Pour se rapprocher des usages du domaine de langages de programmation, on emploiera le terme *évaluation* plutôt que réduction.

Le processus d'évaluation est la répétition de l'application de règles d'évaluation (*clôture transitive* de la relation définie par les règles d'évaluations).

C'est un *calcul formel* qui associe un terme à un autre terme. Ce calcul peut donner une ≪valeur≫ au sens usuel. Par exemple, `1+2` s'évalue en `3`. Sans utiliser le *sucre syntaxique* de l'arithmétique en Coq, on a que l'expression `(plus (S 0) (S (S 0)))` s'évalue en l'expression `(S (S (S 0)))`. On considère que l'expression `3` (c'est-à-dire `(S (S (S 0)))` est la *valeur* de l'expression `1+2` (c'est-à-dire `(plus (S 0) (S (S 0)))`). Le processus d'évaluation peut parfois donner une expression que l'on ne considère usuellement pas comme une ≪valeur≫. C'est la cas lorsque l'expression considérée contient des symboles de variables. Par exemple `0+y` s'évalue en `y`. On parle alors *d'évaluation symbolique*.

**Évaluation et formules** Comme avec l'égalité, dans une formule $F$, on peut remplacer un terme $t$ par un terme $u$ lorsque $t$ s'évalue et $u$. Cela donne la règle de déduction :

— pour prouver $A[t]$, si $t$ s'évalue en $u$, il suffit de prouver $A[u]$ ;

Notons $t \hookrightarrow u$ pour ≪$t$ s'évalue en $u$≫. On note cette règle

$$\frac{\Gamma \vdash A[u] \qquad t \hookrightarrow u}{\Gamma \vdash A[t]}\ eval$$

Avec cette règle, on peut montrer que `1+2=2+1`. En effet, puisque `1+2`$\hookrightarrow$`3` et `2+1`$\hookrightarrow$`3`, il sufit de montrer que `3=3`. Ce que l'on a par réflexivité.

Le remplacement d'une expression par sa valeur est implémenté dans le système Coq par la tactique `simpl`.

```
Coq < Goal 1+2=2+1.
1 subgoal

  ============================
   1 + 2 = 2 + 1

Unnamed_thm < simpl.
1 subgoal

  ============================
   3 = 3

Unnamed_thm < reflexivity.
No more subgoals.

Unnamed_thm < Qed.
simpl.
reflexivity.

Qed.
Unnamed_thm is defined
```

La tactique `simpl` évalue tout ce qu'elle peut dans la formule à démontrer : ici, elle a réduit les deux termes de l'égalité.

### 3.3.3 Propriétés équationnelles des fonctions

Les équations intéressantes pour la spécification et la vérification formelle sont celles qui font intrevenir des variables. On distingue deux espèces de propriétés équationnelles : celles qui s'obtiennent simplement par évaluation et les autres.

**Équations de la première espèce** L'équation `forall (b:bool), (andb true b) = b` s'obtient par évaluation. En effet, comme `(andb b true)` s'évalue en `b`.

```
Coq < Lemma and_true_left : forall (b:bool), (andb true b)=b.
1 subgoal

  ============================
   forall b : bool, (true && b)%bool = b

and_true_left < simpl.
1 subgoal

  ============================
   forall b : bool, b = b

and_true_left < reflexivity.
No more subgoals.
and_true_left < Qed.
simpl.
reflexivity.
```

```
and_true_left is defined
```

L'existence des équations de première espèce nous dit que la relation d'évaluation *est contenue* dans la relation d'égalité puisque si $t \hookrightarrow u$ alors $t = u$.

**Équations de la seconde espèce** L'inverse n'est pas vrai. Par exemple, l'équation `(andb b true) = b` ne s'obtient pas simplement avec l'évaluation.

Cela tient à la manière dont la conjonction `andb` est définie :

```
Definition andb (b1 b2:bool) : bool := if b1 then b2 else false.
```

Dès lors, `(andb b true)` est égal à `if b then true else false`. Le processus d'évaluation est ≪bloqué≫ par la variable `b` dont il n'a pas la valeur.

```
Coq < Lemma andb_true_right : forall (b:bool), (andb b true) = b.
1 subgoal

  ============================
   forall b : bool, (b && true)%bool = b

andb_true_right < simpl.
1 subgoal

  ============================
   forall b : bool, (b && true)%bool = b

andb_true_right <
```

On se retrouve ≪bloqué≫.

### 3.3.4 Preuve par cas de constructeurs

Pour ≪débloquer≫ la situation, il faut envisager les deux valeurs possibles de `b`, c'est-à-dire ; *raisonner par cas* sur `b` :

- si `b`=`true` alors l'équation à montrer est `(andb true true) = true`, ce qui s'obtient par évaluation et réflexivité de l'égalité ;
- si `b`=`false` alors l'équation à montrer est `(andb false true) = false`, ce qui s'obtient de la même manière.

On parle de *cas de constructeurs* car `true` et `false` sont les deux constructeurs du type `bool` auquel appartient `b`.

On peut formuler ce *principe* de raisonnement avec les booléens par la règle

$$\frac{\Gamma \vdash F[\texttt{true}] \qquad \Gamma \vdash F[\texttt{false}]}{\Gamma \vdash \texttt{forall}(b : \texttt{bool}), F[b]}\ bool\_case$$

La tactique `destruct` permet un tel raisonnement par cas.

```
Coq < Lemma andb_true_right : forall (b:bool), (andb b true) = b.
1 subgoal

  ============================
   forall b : bool, (b && true)%bool = b
```

```
andb_true_right < destruct b.
2 subgoals

  ============================
   (true && true)%bool = true

subgoal 2 is:
 (false && true)%bool = false
andb_true_right < simpl; reflexivity.
1 subgoal

  ============================
   (false && true)%bool = false

andb_true_right < simpl; reflexivity.
No more subgoals.
andb_true_right < Qed.
destruct b.
 simpl; reflexivity.

 simpl; reflexivity.

andb_true_right is defined
```

Remarque, on peut remplacer la séquence de tactiques `simpl; reflexivity` par `trivial`. Ceci permet de simplifier notablement le script de cette preuve :

```
Lemma andb_true_right : forall (b:bool), (andb b true) = b.
1 subgoal

  ============================
   forall b : bool, (b && true)%bool = b

andb_true_right < destruct b; trivial.
No more subgoals.

andb_true_right < Qed.
destruct b; trivial.

Qed.
andb_true_right is define
```

**Pour tout type inductif** il existe une règle de raisonnement par cas de constructeur. Pour le type `nat`, on a

$$\frac{\Gamma \vdash F[\texttt{0}] \qquad \Gamma, x : \texttt{nat} \vdash F[(\texttt{S}\ x)]}{\Gamma \vdash \texttt{forall}(x : \texttt{nat}), A[x]}\ \texttt{nat}\_case$$

Pour le type `list`, pour tout `A:Set`, on a

$$\frac{\Gamma \vdash F[\texttt{nil}] \qquad \Gamma, (x : \texttt{A}), (xs : (\texttt{list A})) \vdash F[(\texttt{cons}\ x\ xs)]}{\Gamma \vdash \texttt{forall}(xs : (\texttt{list A}), F[xs]}\ \texttt{list}\_case$$

### 3.3.5 Preuve par induction structurelle

Considérons les deux propriétés équationneles de l'addition
- `forall (x:nat), (plus 0 x) = x`
- `forall (x:nat), (plus x 0) = x`

La première est obtenue par simple évaluation. En effet, si l'on se souvient de la définition de l'addition :

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

on observe que la fonction est définie par analyse de cas du premier argument. Dès lors, `(plus 0 x)` s'évalue en `(match 0 with 0 => x | (S x) => S (plus x 0) end.)`, qui lui-même s'évalue en `x`.

La seconde équation ne peut être simplement établie de cette manière. On retrouve la situation de ≪blocage≫ que l'on avait rencontré avec `(andb b true)`. En effet, `(plus x 0)` s'évalue en `match x with 0 => 0 | (S p) => S (plus p 0) end.`

Mais ici, un raisonnement par cas ne sera pas suffisant :

```
Lemma plus_0_right : forall (x:nat), (plus x 0) = x.
1 subgoal

  ============================
   forall x : nat, x + 0 = x

plus_0_right < destruct x.
2 subgoals

  ============================
   0 + 0 = 0

subgoal 2 is:
 S x + 0 = S x

plus_0_right < trivial.
1 subgoal

  x : nat
  ============================
   S x + 0 = S x

plus_0_right < simpl.
1 subgoal

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

Dans le second cas de constructeur, on n'a guère progressé : l'évaluation de `(S x)+0` a fait réapparaître l'expression ≪bloquante≫ `x+0`.

Pour pouvoir progresser, il faut utiliser un moyen de preuve plus puissant que le simple raisonnement par cas : le *raisonnement par induction structurelle*.

Le raisonnement par induction structurelle ajoute au raisonnement par cas une *hypothèse d'induction*. La règle de raisonnement par induction sur un entier peut s'écrire ainsi :

$$\frac{\Gamma \vdash F[0] \qquad \Gamma, x : \texttt{nat}, H : F[x] \vdash F[\texttt{S}x]}{\Gamma \vdash \texttt{forall}(x : \texttt{nat}), F[x]}\ nat\_ind$$

Appliquons ce principe avec la formule `(plus x 0) = x` :

- à montrer : `(plus 0 0) = 0`. C'est trivial.
- à montrer : `(plus (S x) 0) = (S x)`, sous les hypothèses `x:nat` et `H:(plus x 0) = x`.
  - En appliquant l'évaluation à `(plus (S x) 0) = (S x)`, reste à monter `S (plus x 0) = (S x)`.
  - En utilisant l'équation en hypothèse `H`, on remplace `(plus x 0)` dans `S (plus x 0) = (S x)` par `x` ; reste alors à monter `S x = S x`. Ce qui est trivial.

La tactique `induction` implémente le raisonnement par induction structurelle.

```
Coq < Lemma plus_0_right : forall (x:nat), (plus x 0) = x.
1 subgoal

  ============================
   forall x : nat, x + 0 = x

plus_0_right < induction x.
2 subgoals

  ============================
   0 + 0 = 0

subgoal 2 is:
 S x + 0 = S x
plus_0_right < trivial.
1 subgoal

  x : nat
  IHx : x + 0 = x
  ============================
   S x + 0 = S x

plus_0_right < simpl.
1 subgoal

  x : nat
  IHx : x + 0 = x
  ============================
   S (x + 0) = S x

plus_0_right < rewrite IHx.
1 subgoal

  x : nat
  IHx : x + 0 = x
  ============================
```

```
   S x = S x

plus_0_right < trivial.
No more subgoals.
plus_0_right < Qed.
induction x.
 trivial.

 simpl.
 rewrite IHx.
 trivial.

plus_0_right is defined
```

**Pour tout type inductif** il existe un raisonnement par induction structurelle. Pour les listes, pour tout `A:Set`, on a

$$\frac{\Gamma \vdash F[\texttt{nil}] \qquad \Gamma, (x : \texttt{A}), (xs : (\texttt{list A}), (IHxs : F[xs]) \vdash F[(\texttt{cons } x \ xs)]}{\Gamma \vdash \texttt{forall}(xs : (\texttt{list A})), F[xs]}$$

Il nous permet de montrer que `nil` est élément neutre à droite pour la concaténation :

```
Theorem app_nil_right :
  forall (A:Set) (xs:(list A)), (app xs nil) = xs.
1 subgoal

  ============================
   forall (A : Set) (xs : list A), (xs ++ nil)%list = xs

app_nil_right < induction xs.
2 subgoals

  A : Set
  ============================
   (nil ++ nil)%list = nil

subgoal 2 is:
 ((a :: xs) ++ nil)%list = (a :: xs)%list

app_nil_right < trivial.
1 subgoal

  A : Set
  a : A
  xs : list A
  IHxs : (xs ++ nil)%list = xs
  ============================
   ((a :: xs) ++ nil)%list = (a :: xs)%list

app_nil_right < simpl.
1 subgoal
```

```
  A : Set
  a : A
  xs : list A
  IHxs : (xs ++ nil)%list = xs
  ============================
   (a :: xs ++ nil)%list = (a :: xs)%list

app_nil_right < rewrite IHxs.
1 subgoal

  A : Set
  a : A
  xs : list A
  IHxs : (xs ++ nil)%list = xs
  ============================
   (a :: xs)%list = (a :: xs)%list

app_nil_right < trivial.
No more subgoals.

app_nil_right < Qed.
induction xs.
 trivial.

 simpl.
 rewrite IHxs.
 trivial.

Qed.
app_nil_right is defined
```

# 4 Spécification et prédicats inductifs

## 4.1 Une fonction et une spécification

Considérons la fonction suivante :

```
Fixpoint get_prefix A:Set (n:nat) (xs:list A) :=
  match n, xs with
    0, _ => nil
  | (S n), nil => nil
  | (S n), (cons x xs) => (cons x (get_prefix n xs))
  end.
```

Comme son nom le suggère, cette fonction calcule un préfixe de `xs`. On attend de surcroît que le résultat soit une liste de longueur `n`. Ce que l'on pourrait exprimer par la formule :

```
forall (A:Set) (n:nat) (xs:list A), (length (get_prefixe n xs)) = n.
```

Toutefois, cette formule n'est pas vraie en général. Par exemple, il est faux que `(length (get_prefixe 1 nil)) = 1` ; puisque `(get_prefixe 1 nil)` est égal à `nil` et `(length nil)` est égal à `0`.

L'égalité de la longueur du résultat et de l'argument `n` n'est vraie que si `n` n'est pas plus grand que la longueur de la liste dont on veut extraire le préfixe. Il y a donc une (pré)condition à satisfaire pour obtenir l'égalité attendue :

```
forall (A:Set) (n:nat) (xs:list A),
  (n <= (length xs)) -> (length (prefix n xs)) = n.
```

La propriété de longueur n'est bien entendue pas suffisante pour garantir le correction de la fonction `get_prefix`. Il faut également garantir que la valeur de l'application `(get_prefix n xs)` calcule effectivement un préfixe de la liste `xs`. Se pose alors la question de *formaliser* l'énoncé ≪*`(get_prefix n xs)` est un préfixe de `xs`*≫.

Informellement, une liste `zs` est un préfixe de la liste `xs` si `xs` ≪commence≫ par `zs`. Nous allons envisager dans un premier temps deux visions des listes qui nous permettront de définir la relation ≪est préfixe de≫. Puis, nous montrerons une troisième possibilité offerte par le système Coq.

**Les listes comme suites indicées** Une liste est une *séquence* de valeurs dans laquelle chacune a une position, comme dans les structures de tableau. Supposons que la fonction `nth`, nous donne, lorsqu'elle existe, la valeur présente à une certaine position dans une liste : `(nth i xs)` désigne, lorsqu'elle existe, la valeur en position `i` dans `xs`.

Avec cette vision, dire que `xs` commence par `zs`, c'est dire que pour tout indice `i` dans la liste `zs`, `(nth i zs) = (nth i xs)`. On garantit que `i` est ≪un indice dans la liste `zs`≫ en posant la condition que `i < (length zs)`. Ces éléments nous donnent une première définition de la relation cherchée :

```
Definition Is_prefix A:Set (zs xs:list A) : Prop :=
  forall (i:nat), (i < length zs) -> (nth i zs) = (nth i xs).
```

Cette définition est cependant incomplète : si elle garantit (par la prémisse `(i < (length zs))`) que `(nth i zs)` est bien défini, elle ignore le cas où `(nth i xs)` ne l'est pas. Pour obtenir une définition correcte, il faut poser :

```
Definition Is_prefix {A:Set} (zs xs:list A) : Prop :=
  (length zs <= length xs) /\
  forall (i:nat), (i < length zs) -> (length zs <= length xs)
     -> (nth i zs) = (nth i xs).
```

Toutefois, outre sa redondante lourdeur, cette manière de définir a pour principal inconvénient l'utilisation de la fonction partielle `nth`. Nous allons donc tenter une autre approche.

**Le *monoïde* des listes**[4] Il existe une vision des suite de valeurs, et donc des listes, dans laquelle c'est l'opération de concaténation qui est primitive.

Dans cette perspective, dire que `xs` ≪commence≫ par `zs`, c'est dire que `xs` est construite en concaténant `zs` avec une liste `ys`. En d'autre termes, c'est dire qu'il existe une liste `ys` telle que `xs` est égale à la concaténation de `zs` et `ys`. On peut écrire (formellement) cette égalité : `(app zs ys) = xs`. La *relation* ≪`zs` est un préfixe de `xs`≫, s'écrit alors à l'aide d'un quantificateur existentiel : `exists (ys:list A), (app zs ys) = xs`. Et nous pouvons poser comme définition de la relation ≪être préfixe de≫ :

```
Definition Is_prefix {A:Set} (zs xs : list A) : Prop :=
  exists (ys:list A), (app zs ys) = xs.
```

Cette formulation est assez limpide et plus aisée d'utilisation que la précédente bien que le traitement de l'existentiel reste un peu délicat. Nous allons donc nous tourner vers une autre possibilité de définition.

4. Structure algébrique munie d'une loi de composition interne associative et qui possède un élément neutre.

## 4.2 Relation inductive

Cette autre manière de définir la relation `Is_prefix` repose sur la structure inductive même des listes.

Pour que `zs` soit un préfixe de `xs`, il suffit de vérifier l'une ou l'autre des deux conditions suivante :

1. soit `zs` est la liste vide `nil`.
2. soit `zs` et `xs` commencent par une même valeur et `zs` se prolonge en un préfixe de `xs`.

C'est le second terme de la conjonction de la seconde condition fait de cette définition une *définition inductive*.

On peut préciser plus formellement cette seconde condition : l'énoncé ≪`zs` et `xs` commencent par une même valeur≫ signifie que l'on peut écrire `zs` sous la forme `(cons a zs')` et `xs` sous la forme `(cons a xs')`, pour un `a` quelconque ; l'énoncé ≪`zs` se prolonge en un préfixe de `xs`≫ devint alors simplement : `zs'` est un préfixe de `xs'`. Plus formellement : pour toute valeur `a` et toutes listes `zs'` et `xs'`, si `zs'` est un préfixe de `xs'` alors `(cons a zs')` est un préfixe de `(cons a xs')`.

La définition inductive d'une relation adopte la forme des définitions des types de données inductifs :

```
Inductive Is_prefix {A:Set} : (list A) -> (list A) -> Prop :=
  is_refix_nil : forall (xs:list A), (Is_prefix nil xs)
| is_refix_cons : forall (a:A) (zs xs:list A),
    (Is_prefix zs xs) -> (Is_prefix (cons a zs) (cons a xs)).
```

Les deux clauses de la définition sont nommées `is_prefix_nil` et `is_prefix_cons`. On peut les considérer comme les *constructeurs* de la relation. Le type de chacun de ces constructeurs est exprimé par des formules (expressions de type `Prop`). Pour montrer que deux listes `xs` et `ys` sont dans la relation `Is_prefix`, il faudra montrer que l'on peut établir la validité de la formule `(Is_prefix xs ys)`, soit par application de `is_prefix_nil`, soit par application de `is_prefix_cons` ; plus exactement, par application des formules données pour chaque cas de définition.

Illustrons cela en montrant quelques propriétés de la relation `Is_prefix`.

Montrons que la relation `Is_prefix` est réflexive :

```
Theorem is_prefix_refl : forall (A:Set) (xs:list A), (Is_prefix xs xs).
```

La preuve de cet énoncé est assez simple, par induction structurelle sur `xs` :

— si `xs` est la liste vide `nil`, il suffit de montrer que `(Is_prefix nil nil)`, ce que l'on a par `is_prefix_nil`.

— si `xs` a la forme `(cons x xs')`, il suffit de montrer que, `(Is_prefix (cons x xs') (cons x xs'))`, en supposant `IHxs: (Is_prefix xs' xs')`.
Pour montrer `(Is_prefix (cons x xs') (cons x xs'))`, par application de `is_prefix_cons`, il suffit de montrer que `(Is_prefix xs' xs')`. Ce qui est notre hypothèse d'induction.

Le script correspondant à cette preuve par induction est :

```
induction xs.
- apply is_prefix_nil.
- apply is_prefix_cons. assumption.
```

Montrons maintenant que notre définition inductive est cohérente avec la vision *monoïde* des listes, à savoir :

```
Theorem is_prefix_app : forall (A:Set) (xs ys : list A), (Is_prefix xs (app xs ys)).
```

On prouve cet énoncé par induction sur `xs` :

— si `xs` est `nil`, il suffit de montrer `(Is_prefix nil (app nil ys))`. Ce que l'on a par `is_prefix_nil`.

— si `xs` est de la forme `(cons x xs')`, il suffit de montrer que `(Is_prefix (cons x xs') (app (cons x xs') ys))`, en supposant que `IHxs: (Is_prefix xs' (app xs' ys))`.

Pour montrer `(Is_prefix (cons x xs') (app (cons x xs') ys))`, par évaluation, il suffit de montrer que `(Is_prefix (cons x xs') (cons x (app xs' ys)))`. En appliquant `is_prefix_cons`, il suffit de montrer que `(Is_prefix xs' (app xs' ys))`. Ce que l'on a par `IHxs`.

Le script correspondant est :

```
induction xs.
- intro. simpl. apply is_prefix_nil.
- intros. simpl. apply is_prefix_cons. apply IHxs.
```

**Lemmes d'inversion** La seconde clause de la définition de `Is_prefix` s'énonce comme une implication : `(Is_prefix xs ys) -> (Is_prefix (cons z xs) (cons z ys))` mais il s'agit en fait d'une équivalence. En effet, on peut montrer sa réciproque :

```
Theorem is_prefix_cons_inv : forall (A:Set) (z:A) (xs ys:list A),
  (Is_prefix (cons z xs) (cons z ys)) -> (Is_prefix xs ys).
```

Après les introductions d'usage, le but à prouver est :

```
A : Set
z : A
xs, ys : list A
H : Is_prefix (z :: xs) (z :: ys)
============================
 Is_prefix xs ys
```

La formule à montrer, sous l'hypothèse `H`, est nécessairement vraie *par définition* de `Is_prefix`. En effet, il n'y a qu'une seule manière d'avoir `(Is_prefix (z :: xs) (z :: ys))` : c'est par la clause `is_prefix_cons` de la définition de `Is_prefix` qui stipule que `(Is_prefix xs ys) -> (Is_prefix (cons z xs) (cons z ys))`, pour tout `z`, `xs` et `ys`. Dès lors, si on suppose que `(Is_prefix (z :: xs) (z :: ys))` satisfaite alors la prémisse `Is_prefix xs ys` est nécessairement satisfaite.

*Grosso modo*, ce raisonnement est réalisé par la tactique `inversion`. Si, ici, on l'applique à `H`, on obtient :

```
A : Set
z : A
xs, ys : list A
H : Is_prefix (z :: xs) (z :: ys)
z0 : A
xs0, ys0 : list A
H1 : Is_prefix xs ys
H0 : z0 = z
H2 : xs0 = xs
H3 : ys0 = ys
============================
 Is_prefix xs ys
```

Qui est trivialement établi avec la règle axiome (tactique `assumption`).

Le script de preuve de l'énoncé `is_prefix_cons_inv` est

```
intros. inversion H. assumption.
```