Chapter 1

theme02boolarithstudent

Dans ce second thème, nous commençons par enrichir notre catalogue de types de données manipulables dans Coq avec les booléens et les entiers naturels. Nous terminons avec les couples et le type produit cartésien de deux types.

Nos exemples seront souvent liés aux listes donc commençons par charger la bibliothèque idoine.

```
Require Import List.
```

1.1 booléens

Coq propose deux façons distinctes d'envisager la vérité et la fausseté :

- la vérité logique *True* et la fausseté logique *False* qui sont des types du `super-type` `Prop` (type des propositions logiques)

```
Check True. (*  => True : Prop *)
Check False. (*  => False : Prop *)
```

- la valeur de vérité *true* et la valeur de fausseté *false* qui sont des termes (valeurs) du type *bool* , ce dernier étant du type `Set` soit le type es structures de données.

```
Check true. (*  => true : bool *)
Check false. (*  => false : bool *)
Check bool. (*  => bool : Set *)
```

Cette dichotomie n'est pas évidente à comprendre, et dans certains cas il peut en résulter une certaine confusion. Mais intuitivement, la différence est assez claire :

- les propositions logiques sont vraies (ou fausses) au sens de la logique
- les valeurs de vérité permettent d'implémenter des prédicats calculables : des fonctions qui retournent des booléens.

On pourrait vouloir se contenter des valeurs de vérité mais alors toute proposition devrait être décidable, c'est-à-dire représentées par des fonctions calculables au sens des fonctions totales de Coq. On ne pourrait donc même pas énoncer une propriété dont on ne sait pas encore si on peut la démontrer ou non. Bref, on ne pourrait pas faire de conjecture. Et on sait aussi qu'il existe des propriétés indécidables que l'on aimerait énoncer. Pour toutes ces raisons les types *True*, *False* et `Prop` jouent un rôle fondamental alors que le type *bool* n'est qu'un type de donnée utile parmi d'autres.

Nous reviendrons aux types *True* et *False* lors du prochain thème, et retournons dès à présent au type *bool*.

```
Print bool.
```

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

On voit ici que *bool* est un type somme simple que l'on peut facilement traduire en pseudo-Ocaml :

```
type bool = true | false
```

On peut donc calculer avec ce type, en implémentant par exemple la conjonction.

```
Definition et (a:bool) (b:bool) : bool :=
  if a then b else false.
```

```
Example et_ex1:
  et true true = true.
Proof.
  compute. reflexivity.
Qed.
```

On peut bien sûr démontrer des lemmes un peu plus généraux.

```
Lemma et_false:
  ∀ b : bool, et false b = false.
Proof.
  intro b.
  unfold et.
  reflexivity.
Qed.
```

1.1.1 Exercice

Définir la disjonction *ou* et démontrer un lemme comparable à *et_false* ci-dessus.

1.1.2 Exercice

Les booléens nous permettent d'enrichir notre catalogue de combinateurs de listes avec le fameux *filter*.

Question 1 :

Définir la fonction *filter* telle que l'exemple suivant est prouvable :

```
Example filter_ex1:
  filter (fun a:bool => a) (false::true::false::false::true::nil) = true::true::nil.
```

Question 2

Montrer deux lemmes qui vous semblent des propriétés "limites" de *filter*.

1.2 Arithmétique

Si les booléens ne posent pas de difficulté particulière (à part cette confusion possible avec les types *True* et *False* de `Prop`), il n'en est rien de l'arithmétique, une branche particulièrement ardue des mathématiques.

Coq supporte différents types numériques : les entiers naturels avec *nat*, les entiers relatifs avec *Z*, et mêmes les réels et d'autres systèmes plus exotiques (entiers codés en binaire, etc.).

Le plus simple des types numériques est celui des entiers naturels (mais qui n'est pas simple pour autant).

Nous commençons par importer la bibliothèque d'arithmétique sur les naturels, qui nous offre un certain nombre de lemme déjà démontrés.

```
Require Import Arith.
```

Depuis *Peano* au XIXè siècle on sait que le type *nat* des entiers naturels peut être défini de façon inductive.

La définition proposée par Coq est la suivante :

```
Print nat.
```

```
Inductive nat : Set :=
  O : nat
| S : nat → nat
```

Coq accepte les notations standard des entiers, mais il ne vaut pas se tromper ce sont bien des constructions inductives.

```
Example Zero: 0 = O.
Proof. reflexivity. Qed.
```

```
Example Cinq: 5 = S (S (S (S (S O)))).
Proof. reflexivity. Qed.
```

Considérons une fonction *double* permettant de doubler un entier naturel passé en paramètre.

```
Definition double (n : nat) : nat := 2 × n.
```

```
Example square_2_is_2plus2:
```

```
  double 2 = 2 + 2.
Proof.
  compute.
  reflexivity.
Qed.
```

Démontrons un premier lemme arithmétique, que l'on peut penser simple mais qui va déjà nous donner un peu de fil à retordre du fait de la complexité intrinsèque des raisonnements arithmétiques (et aussi parce que pour l'instant on évite les procédures de décision automatique disponibles en Coq).

```
Lemma double_plus:
  ∀ n : nat, double n = n + n.
Proof.
  destruct n as [| n']. (*  raisonnement par cas *)
  - (*  cas de base: n=0 *)
    compute. (*  remarque : terme clos *)
    reflexivity.
  - (*  cas récursif: n=S n' *)
    unfold double.
    simpl.
    SearchPattern ( S _ = S _ ).
    (*  eq_S: forall x y : nat, x = y -> S x = S y *)
    apply eq_S.
    SearchPattern ( ?X + 0 = ?X).
    (*  plus_0_r: forall n : nat, n + 0 = n *)
    rewrite plus_0_r.
    reflexivity.
Qed.
```

Le seul frein ici est l'arithmétique. Et c'est normal car l'arithmétique (même restreinte aux entiers naturels) est loin d'être simple, elle occupe de nombreux mathématiciens et informaticiens depuis de nombreuses années (quelques siècles pour les mathématiciens).

Essayons tout de même de voir si Coq "en a sous le capot".

```
Lemma double_plus':
  ∀ n : nat, double n = n + n.
Proof.
  intro n.
  unfold double.
  ring.
Qed.
```

La procédure de décision arithmétique `ring` (sur l'anneau $(\mathbb{N}, +, \times)$) fonctionne à merveille ici.

Ceci illustre un élément fondamental de la preuve de programme.

Il existe de nombreuses techniques de démonstration, même pour un théorème donné. On verra par la suite que la plupart des théorème non-triviaux nécessitent un raisonnement de type inductif. Mais les étapes les plus fastidieuses de calcul peuvent souvent être traitées par `ring`, `auto with` *arith* ou autres procédures de décision automatique.

Pour effectuer des preuves sur les entiers naturels, on exploite le plus souvent le principe inductif *nat_ind*.

```
Check nat_ind.
```

```
nat_ind
     : ∀ P : nat → Prop,
       P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Montrons, en utilisant ce principe et en évitant `ring`, le lemme prédéfini *plus_0_r* que nous avons déjà utilisé.

```
Lemma plus_0_r':
  ∀ n : nat, n + 0 = n.
Proof.
  intro n.
  induction n as [| n'].
  - (*  cas n=0 *)
    trivial.
  - (*  cas n = S n' *)
    simpl.
    rewrite IHn'.
    reflexivity.
Qed.
```

On peut exploiter une autre tactique automatique pour démontrer notre lemme en une ligne : `auto with` *arith*. Cette procédure essaye d'appliquer un certain nombre de lemmes de façon automatique, en effectuant une recherche par unification et retour en arrière à la Prolog. Nous aurons l'occasion d'y revenir.

```
Lemma plus_0_r'':
  ∀ n : nat, n + 0 = n.
Proof.
  auto with arith.
Qed.
```

Remarque : ici on triche un peu puisque le Lemme original *plus_0_r* fait partie de la base des lemmes exploités par `auto with` *arith*. Donc la procédure n'a pas besoin de chercher longtemps pour conclure. Mais il est parfois utile d'essayer `auto with` *arith* (de même que `ring`) quand on se retrouve au milieu d'un preuve arithmétique.

1.2.1 Exercice

La relation naturelle entre les listes et les entiers naturels est bien sûr la notion de longueur de liste.

Question 1

Définir une fonction *longueur* retournant la longueur d'une liste.

Question 2

En fait *longueur* est prédéfinie en coq et se nomme *length*.

```
Print length.
```

```
length =
fun A : Type ⇒
fix length (l : list A) : nat :=
  match l with
  | nil ⇒ 0
  | _ :: l' ⇒ S (length l')
  end
     : ∀ A : Type, list A → nat
```

Montrer que les deux fonctions *longueur* et *length* effectuent le même calcul.

Question 3

Démontrer le lemme suivant :

```
Lemma length_app:
  ∀ A : Set, ∀ l1 l2 : list A,
    length (l1 ++ l2) = (length l1) + (length l2).
```

Remarque : ++ est la concaténation de listes, prédéfinie en Coq (notre *concat* de la semaine dernière).

Question 4

Montrer que *length* (*map f l*) = *length l*.

1.2.2 Exercice

Nous retournons maintenant à l'arithmétique "pure" avec le calcul de la somme des n premiers entiers (pour $n \geq 0$).

Question 1

Définir la fonction *sum* calculant : $\sum_{i=0}^{n} i$.

Question 2 (un peu plus corsée)

Démontrer le théorème classique de la somme : $2 \times \sum_{i=1}^{n} = n * (n+1)$.

Remarque : vous pourrez utiliser `ring` si vous vous retrouvez bloqués.

On remarque suite à ce dernier exerice que notre stratégie consiste globalement à pouvoir utiliser l'hypothèse d'induction avant de terminer avec `ring` ou autre procédure de décision. C'est généralement une stratégie gagnante.

1.2.3 Exercice : la factorielle

Question 1

Soit la fonction factorielle sur les entiers naturels.

```
Fixpoint fact (n:nat) : nat :=
  match n with
    | O ⇒ 1
    | S m ⇒ n × fact m
  end.

Example fact_5: fact 5 = 120.
Proof.
  compute.
  reflexivity.
Qed.
```

Définir une version *fact_it* récursive terminale de la factorielle.

Question 2

Démontrer le lemme suivant :

```
Lemma fact_it_lemma:
  ∀ n k:nat, fact_it n k = k × (fact n).
Proof.
```

Question 3

En déduire un théorème permettant de relier les deux versions de la factorielle.

1.2.4 Exercice : Fibonacci

Question 1

Définir une fonction *fib* de calcul de la suite de Fibonacci en traduisant la définition arithmétique suivante :

$$\mathrm{fib}(n) = \begin{cases} 0 \text{ si } n = 0 \\ 1 \text{ si } n = 1 \\ \mathrm{fib}(n-2) + \mathrm{fib}(n-1) \text{ sinon} \end{cases}$$

Question 2

Démontrer par cas le lemme suivant :

```
Lemma fib_plus_2:
  ∀ n:nat, fib (S (S n)) = fib n + fib (S n).
Proof.
```

Question 3 :

Donner une définition *fib_it* récursive terminale de Fibonacci.

Question 4

Démontrer le théorème

```
Theorem fib_it_fib:
  ∀ n:nat, fib_it n 1 1 = fib n.
```

Remarque : on sera sans doute amené à généraliser ce théorème, sous la forme d'un lemme *fib_it_fib_aux* (cf. exercice précédent avec *fact*).

1.3 Couples

Pour terminer ce thème, nous allons manipuler le type produit cartésien des couples.

Le type produit cartérisien de deux types A et B se note $A \times B$.

Par exemple :

```
Check (1, true). (* : nat × bool *)
```

L'écriture $A \times B$ est en fait un racourcis d'écriture pour le type *prod* A B.

Voici en Coq la définition du type produit *prod* :

```
Print prod.

Inductive prod (A B : Type) : Type :=
```

$pair : A \to B \to A \times B$

La notation classique est donc un raccourcis d'écriture pour le constructeur *pair*, comme le montre l'exemple suivant.

```
Check (pair 1 true). (*  : nat * bool *)
```

Les triplets, quadruplets, etc. sont simplement des extensions de la notation.

```
Check (1, true, (1::2::3::nil)). (*  : nat * bool * list nat *)
Check (pair 1 (pair true (1::2::3::nil))). (*  : nat * (bool * list nat) *)
Check (pair (pair 1 true) (1::2::3::nil)). (*  : nat * bool * list nat *)
```

Cette gestion par le parseur peut rendre la manipulation des n-uplets un peu fastidieuse pour le débutant.

Les deux accesseurs principaux pour les pairs son *fst* et *snd*.

```
Eval compute in fst (1, true). (*  = 1 : nat] *)
Eval compute in snd (1, true). (*  = true : bool] *)
```

Le théorème suivant montre que les paires ne jouent pas un rôle de premier plan dans la logique de Coq, même si l'intérêt de la notation ne fait aucun doute.

```
Theorem product_arrow:
  ∀ A B : Type,
  ∀ P : (A × B) → Prop,
  ∃ P' : A → B → Prop,
    ∀ a : A, ∀ b : B,
      P (a, b) = P' a b.
Proof.
  intros A B P.
  ∃ (fun (a:A) (b:B) ⇒ P (a,b)).
  intros a b.
  reflexivity.
Qed.
```

1.3.1 Exercice

Montrer le théorème réciproque *arrow_product*.

1.3.2 Exercice : le zip/unzip

Question 1

Définir la fonction *zip* qui à partir de deux listes *l1* et *l2* construit une liste des couples des éléments successifs de *l1* et *l2*. Si une liste est plus longue que l'autre alors les éléments restants sont omis.

Question 2

Montrer le lemme suivant :

```
Lemma zip_length_l:
  ∀ A B : Set, ∀ l1 : list A, ∀ l2 : list B,
    length l1 = length l2
    → length (zip l1 l2) = length l1.
```

En déduire le lemme complémentaire *zip_length_r* qui conclut : *length (zip l1 l2) = length l2.*

Question 3

Démontrer un lemme intéressant nommé *zip_map* et reliant comme son nom l'indique les fonctions *zip* et *map*.

Question 4 (plus difficile)

Définir la fonction complémentaire *unzip* qui à partir d'une liste de couples produit un couple de listes.

Montrer les propriétés suivantes :

```
Lemma unzip_cons_fst:
  ∀ A B : Set, ∀ l : list (A × B), ∀ e : A × B,
    fst (unzip (e::l)) = (fst e)::(fst (unzip l)).
```

```
Lemma unzip_cons_snd:
  ∀ A B : Set, ∀ l : list (A × B), ∀ e : A × B,
    snd (unzip (e::l)) = (snd e)::(snd (unzip l)).
```

```
Theorem zip_unzip:
  ∀ A B : Set, ∀ l : list (A × B),
    let (l1, l2) := unzip l
    in zip l1 l2 = l.
```

Démontrer finalement le lemme symétrique *unzip_zip*.