



En Coq, l'introduction d'un nouveau type de données s'effectue à l'aide d'un mécanisme de *définition inductive* qui ressemble beaucoup à la définition d'un type concret en Caml. Ainsi, le type `nat` des entiers naturels est introduit<sup>1</sup> à l'aide de la définition inductive suivante :

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

Cette définition ajoute à l'environnement courant trois nouvelles constantes :

- le type `nat : Set` (Set est le type des petits types);
- le constructeur<sup>2</sup> `0 : nat` (constructeur constant);
- le constructeur `S : nat -> nat` (constructeur à un argument de type `nat`).

Le système utilise ensuite le sucre syntaxique `0`, `1`, `2`, etc. pour désigner les entiers `0`, `S 0`, `S (S 0)`, `S (S (S 0))`, etc.

La définition inductive ci-dessus engendre automatiquement un certain nombre de principes d'induction, dont le plus utilisé en pratique est le schéma de récurrence

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

utilisé en interne par les tactiques `elim` et `induction`.

**Exercice 1. – L'addition** En Coq, l'addition<sup>3</sup> est définie au moyen de la construction `Fixpoint`, qui est l'équivalent du « `let rec` » de Caml :

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

Il est important de noter que les appels récursifs se font ici sur un premier argument `n` de plus en plus petit. Il s'agit en fait de *décroissance structurelle*. Coq refuse les définitions pour lesquels il n'est pas en mesure de vérifier cette propriété, et qui risquent donc de ne pas forcément terminer<sup>4</sup>.

Le système utilise la notation `n + m` pour désigner le terme `plus n m`.

1. Vérifier à l'aide des tactiques `simpl` et `reflexivity` qu'on a les égalités définitionnelles

$$0 + m = m \quad \text{et} \quad S n + m = S (n + m).$$

A-t-on les égalités définitionnelles `n + 0 = n` et `n + S m = S (n + m)` ?

2. Montrer les deux lemmes suivants :

```
Lemma plus_n_0 : forall n, n + 0 = n.
Lemma plus_n_Sm : forall n m, n + S m = S (n + m).
```

On prouvera ces deux lemmes par récurrence sur `n`, à l'aide de la tactique `induction`.

3. Montrer que l'addition est commutative : `forall n m, n + m = m + n`.
4. Montrer que l'addition est associative : `forall n m p, (n + m) + p = n + (m + p)`.

1. cf fichier `theories/Init/Datatypes.v`

2. Attention! le constructeur s'appelle `0` (lettre « O ») et non `0` (« zéro »).

3. cf fichier `theories/Init/Peano.v`

4. Dans les versions de Coq antérieures à 8.2, il fallait indiquer l'argument de décroissance à l'aide d'une annotation `{struct n}`.

**Exercice 2. – La multiplication** En Coq, la multiplication est définie par

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

(Le système utilise le sucre syntaxique  $n * m$  pour désigner le terme `mult n m`.)

1. Vérifier à l'aide des tactiques `simpl` et `reflexivity` qu'on a les égalités définitionnelles

$$0 * m = 0 \quad \text{et} \quad S n * m = m + n * m.$$

2. Montrer la propriété de distributivité : `forall n m p, (n + m) * p = n * p + m * p`.
3. Montrer que la multiplication est commutative et associative.

**Exercice 3. – La relation d'ordre** On peut définir<sup>5</sup> la relation d'ordre usuelle sur les entiers `le : nat -> nat -> Prop` en posant :

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

Montrer que `le` est une relation d'ordre :

```
Lemma le_refl : forall n, le n n.
Lemma le_trans : forall n m p, le n m -> le m p -> le n p.
Lemma le_antisym : forall n m, le n m -> le m n -> n = m.
```

**Tactiques utiles :** `simpl`, `elim`, `induction`, `rewrite`, `generalize`.

---

5. Cette définition n'est pas celle de la librairie standard de Coq (cf fichier `theories/Init/Peano.v`), mais est bien entendu équivalente.