

Fondements des systèmes de preuves – TP n° 7

Un peu de programmation en théorie des types de Martin-Löf

Dans ce TP, on utilisera des constructions pré-définies en Coq :

- La constante

```
sum : Type -> Type -> Type
```

permet de représenter des types somme. On utilisera en particulier la notation associée $A + B$ qui représente `sum A B`, lorsqu'elle ne rentre pas en conflit avec l'opération sur les entiers. On aura également besoin de ses constructeurs :

```
inl : forall A B : Type, A -> A + B   inr : forall A B : Type, B -> A + B
```

- La constante

```
sigT : forall A : Type, (A -> Type) -> Type
```

permet de représenter les Σ -types, qui sont définis grâce au constructeur

```
existT : forall (A : Type) (P : A -> Type) (x : A), P x -> sig P
```

Le plus souvent, on utilisera plutôt la notation $\{x : A \& P x\}$ qui représente `sigT (fun x : A => P x)`. On aura également besoin des projections associées à un Σ -type, en particulier de la première :

```
projT1 : forall (A : Type) (P : A -> Type), sigT P -> A
```

Exercice 1 (Division par 2)

1. En utilisant le récursif `nat_rect` déjà croisé la semaine dernière, montrer qu'on peut définir le Σ -type suivant :

```
Definition div2_spec : forall n, {m : nat & sum (n = m + m) (n = S m + m)}
```

On prendra soin de conclure la preuve par la commande `Defined` et non par la commande `Qed`¹.

2. Comment en déduire une définition de la fonction `div2 : nat -> nat` qui donne le quotient de la division par 2 de son argument ?
3. Montrer qu'on a bien :

```
Lemma twotimesdiv2 : forall n,
  n = (div2 n) + (div2 n) \ / n = S (div2 n) + (div2 n).
```

Exercice 2 (Soustraction et ordre) Charger la bibliothèque de lemmes sur l'arithmétique par la commande : `Require Import Arith.`

1. De manière analogue, spécifier puis définir la fonction `sub : nat -> nat -> nat` qui calcule la différence en valeur absolue de ses arguments. Indication : on pourra utiliser la tactique `elim` pour réaliser la récurrence.

¹La commande `Qed` "opacifie" la définition, c'est à dire la rend inerte au calcul.

2. Définir la relation d'ordre `le : nat -> nat -> Set` sur les entiers toujours à l'aide d'un Σ -type. Montrer qu'il s'agit bien d'une relation d'ordre.
3. Définir la relation `lt : nat -> nat -> Set` de manière analogue.
4. Définir `lt_le_dec : forall n m : nat, (lt n m) + (le n m)`.
5. Définir `lt_dec : forall n m, (lt n m) + (n = m) + (lt m n)`.
6. Définir `eq_dec : forall n m, {b : bool & b = true <-> n = m}`.

Exercice 3 (Partitions) Dans tout ce qui suit on travaillera plus facilement avec le mode : `Set Implicit Arguments`.

On généralise la définition des listes vue la semaine dernière pour construire des liste *polymorphes*, à l'aide de la définition suivante :

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

1. Définir à l'aide d'un `Fixpoint` la fonction `append` de concaténation. Montrer que :


```
forall (A : Type)(l : list A), append l nil = l.
```
2. Définir à l'aide d'un `Fixpoint` la fonction `count : list nat -> nat -> nat` qui compte les occurrences d'un entier donné dans une liste d'entiers. Montrer que :

```
forall l1 l2 n, count (append l1 l2) n = (count l1 n) + (count l2 n).
```

3. On définit comme suit l'égalité modulo permutation sur les listes d'entiers :

```
Definition perm_eq (l1 l2 : list nat) := forall n, count l1 n = count l2 n.
```

Montrer qu'il s'agit d'une relation d'équivalence ainsi que les deux propriétés suivantes :

```
- forall l1 l2 a, perm_eq l1 l2 -> perm_eq (cons a l1) (cons a l2).
```

```
- forall l1 l2, perm_eq (append l1 l2) (append l2 l1).
```

4. On se donne deux couleurs définies comme :

```
Inductive Color : Type := Red | Black.
```

ainsi qu'une fonction de coloration des entiers :

```
Parameter col : nat -> Color.
```

On veut construire une fonction qui trie une liste d'entiers en mettant les entiers rouges en tête, suivis des entiers noirs.

À l'aide des définitions suivantes :

```
Definition colored (c : Color) := {x : nat & col x = c}.
```

```
Definition col_list (c : Color) := list (colored c).
```

```
Definition Reds := col_list Red.
```

```
Definition Blacks := col_list Black.
```

et de deux fonctions :

```
R2list : Reds -> list nat          B2list : Blacks -> list nat
```

décrire sous forme d'un `sigma type` la partition (`Partition (l : list nat)`) qui réalise le tri souhaité. Définir enfin une fonction de type : (`forall l, Partition l`).