Preuves assistées par ordinateur – TP n° 4

Les listes en Coq

En Coq, le type des *listes polymorphes* est défini à l'aide de la définition inductive suivante

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

qui introduit dans l'environnement courant un constructeur de type `list` : `Type` → `Type` (qui à chaque type A : `Type` associe le type de listes correspondant `list` A : `Type`) ainsi que les deux constructeurs polymorphes

```
nil   :  forall A : Type, list A
cons  :  forall A : Type, A -> list A -> list A
```

Contrairement à Caml, le polymorphisme est *explicite* en Coq. Ceci se traduit à la fois :

- dans les types des constructeurs `nil` et `cons`, où l'on voit apparaître explicitement la quantification de type `forall A : Type,`... (qui demeure implicite en Caml)
- dans l'utilisation de ces constructeurs, qui attendent explicitement le paramètre de type A comme premier argument. Ainsi :
  - la liste vide d'éléments de A s'écrit : `nil` A (: `list` A) ;
  - l'ajout de l'élément $x : A$ à une liste xs : `list` A s'écrit : `cons` A x xs (: `list` A).

Remarque : En Coq, la quantification universelle `forall x : T, U(x)` introduit un *type fonctionnel* qui généralise le type flèche `T -> U`, et qu'on appelle *produit dépendant*.

Question préliminaire Le paramètre A : `Type` étant fixé, quel principe de récurrence est-il naturel d'introduire pour raisonner sur les listes ? On vérifiera la réponse avec `Check list_ind`.

Exercice 1 – Concaténation de listes
L'opération (polymorphe) de concaténation de listes est définie en Coq par :

```
Fixpoint concat (A : Type) (xs ys : list A) :=
  match xs with
  | nil       => ys
  | cons x xs => cons A x (concat A xs ys)
  end.
```

1. Quel est le type de `concat` ?
2. Montrer en Coq les propositions :

   ```
   forall (A : Type) (xs : list A), concat A (nil A) xs = xs
   forall (A : Type) (xs : list A), concat A xs (nil A) = xs
   ```

   Laquelle de ces deux propositions correspond à une égalité définitionnelle ?
3. Montrer que l'opération de concaténation est associative.

Exercice 2 – Longueur

1. Définir une fonction `length` à deux arguments A : `Type` et xs : `list` A telle que `length` A l retourne la longueur de la liste xs (exprimée comme un objet de type `nat`).
2. Quel est le type de la fonction `length` ?
3. Montrer que `length` A (`concat` A xs ys) = `length` A xs + `length` A ys.

Exercice 3 – Retournement

1. Définir une fonction `reverse : forall (A : Type), list A -> list A` retournant la liste donnée en second argument.
2. Montrer que `length` A (`reverse` A xs) = `length` A xs.
3. Montrer que `reverse` A (`concat` A xs ys) = `concat` A (`reverse` A ys) (`reverse` A xs)
4. Montrer que `reverse` A (`reverse` A xs) = xs