

Fondements des systèmes de preuves – TP n° 6

## Système T et types inductifs

### Exercice 1 (Système T de Gödel)

1. Définir en Coq (à l'aide de la commande `Fixpoint`) une constante

$$\text{rec} \quad : \quad \forall T : \text{Set}, T \rightarrow (\text{nat} \rightarrow T \rightarrow T) \rightarrow \text{nat} \rightarrow T$$

qui implémente le récursur du système T de Gödel.

Quelles sont les égalités définitionnelles attendues ? Vérifiez-les.

Dans la suite de l'exercice, on se place dans le système T. On implémentera chacune des fonctions demandées à l'aide du récursur `rec` (et sans utiliser la commande `Fixpoint`).

2. Implémentez la fonction prédécesseur `pred : nat → nat` définie par

$$\text{pred } 0 = 0 \quad \text{et} \quad \text{pred } (S x) = x.$$

Comment vérifier en Coq que les égalités ci-dessus sont vraies, tout en s'assurant qu'il s'agit d'égalités définitionnelles ?

3. Implémentez l'addition et la multiplication, définies par

$$\begin{array}{ll} \text{plus } 0 y = y & \text{mult } 0 y = 0 \\ \text{plus } (S x) y = S (\text{plus } x y) & \text{mult } (S x) y = \text{plus } (\text{mult } x y) y \end{array}$$

4. Implémentez la fonction d'Ackermann `ack : nat → nat → nat` définie par :

$$\begin{array}{l} \text{ack } 0 y = S y \\ \text{ack } (S x) 0 = \text{ack } x 1 \\ \text{ack } (S x) (S y) = \text{ack } x (\text{ack } (S x) y) \end{array}$$

**Exercice 2 (Anatomie d'un récursur)** En Coq le récursur de la théorie des types de Martin-Löf est implémenté par une constante polymorphe

$$\text{nat\_rec} \quad : \quad \forall P : \text{nat} \rightarrow \text{Set}, P 0 \rightarrow (\forall n : \text{nat}, P n \rightarrow P (S n)) \rightarrow \forall n : \text{nat}, P n$$

1. Vérifiez qu'on a les égalités définitionnelles `nat_rec P x f 0 = x` et `nat_rec P x f (S n) = f n (nat_rec P x f n)`
2. Définir la constante `rec` de l'exercice 1 à l'aide de la constante `nat_rec`.

Le système Coq dispose également de deux autres récursurs :

- `nat_ind` :  $\forall P : \text{nat} \rightarrow \text{Prop}, \dots$  pour les prédicats à valeur dans `Prop`. (C'est cette constante est utilisée par la tactique `induction`.)
- `nat_rect` :  $\forall P : \text{nat} \rightarrow \text{Type}, \dots$  pour les prédicats à valeur dans `Type`.

Calculatoirement, ces deux constantes se comportent comme la constante `nat_rec`.

3. Comment sont implémentées les trois constantes `nat_ind`, `nat_rec` et `nat_rect`?  
On comparera la définition du récursur `nat_rect` avec celle de la constante `rec` de l'exercice 1.

**Exercice 3 (Les listes)** Le type des listes d'entiers est défini en Coq par :

```
Inductive list : Set :=
| nil : list
| cons : nat -> list -> list.
```

Cette définition engendre la création de trois récursurs `list_ind`, `list_rec` et `list_rect`.

1. Quels sont les types de ces récursurs? les règles de réduction associées?
2. Définissez, en utilisant le récursur `list_rec`, la fonction de concaténation `append : list → list → list`.
3. Quelles sont les égalités définitionnelles de `append`? Vérifiez-les.
4. Montrez que `append` est une opération associative, pour laquelle la liste vide est élément neutre à gauche et à droite. (*Indication* : Il s'agit d'une preuve par induction.)

**Exercice 4 (Les listes dépendantes)** Le type des listes dépendantes est défini en Coq par :

```
Inductive vect : nat -> Set :=
| vnil : vect 0
| vcons : forall n : nat, nat -> vect n -> vect (S n).
```

1. Comment écrit-on la liste [5; 6; 7] de type `vect 3`?

Cette définition engendre la création de trois récursurs `list_ind`, `list_rec` et `list_rect`.

1. Vérifiez les types des récursurs `vect_ind`, `vect_rec` et `vect_rect` engendrés par cette définition. Quelles sont les règles de réduction associées?
2. Définissez, en utilisant le récursur `vect_rec`, la fonction de concaténation `vappend : ∀n p : nat, vect n → vect p → vect (n + p)`.
3. (★★) Comment exprimer que cette fonction est associative? Prouvez-le.

**Exercice 5 (Fonctions variadiques)**

1. À l'aide de `nat_rect`, définissez une constante `arity : nat → Set` telle que

$$\text{arity } n = \underbrace{\text{nat} \rightarrow \dots \rightarrow \text{nat}}_n \rightarrow \text{nat}.$$

2. À l'aide de `nat_rec`, définissez une constante `varsum : ∀n : nat, arity n` telle que

$$\text{varsum } n \ x_1 \ \dots \ x_n = x_1 + \dots + x_n.$$

(*Indication* : On pourra commencer par définir une constante `varsum_aux` de type  $\forall n : \text{nat}, \text{nat} \rightarrow \text{arity } n$  telle que  $\text{varsum\_aux } n \ a \ x_1 \ \dots \ x_n = a + x_1 + \dots + x_n$ .)

3. (★★) Écrire une fonction `varbuild` telle que

$$\text{varbuild } n \ x_1 \ \dots \ x_n = [x_1; \dots; x_n] : \text{vect } n.$$