**Examen cours 2-7-2 Assistants de preuve**

Lundi 12 février 2007

L'énoncé est composé de ?? pages. L'examen dure 3 heures. Les notes de cours manuscrites ainsi que les supports de cours distribués cette année sont les seuls documents autorisés.
Les quelques éléments de la bibliothèque standard de Coq dont vous pourriez avoir besoin sont rappelés en annexe.

# 1 Problème de la somme du sous-ensemble

On considère dans cet exercice le problème dit de la « somme du sous-ensemble » qui consiste à déterminer si un ensemble fini d'entiers contient un sous-ensemble dont la somme est égale à un entier donné. On utilise ici les entiers naturels de Coq (type `nat`) et des listes d'entiers naturels pour représenter les ensembles finis (type `list nat`).

1. Définir une fonction `sum : list nat → nat` calculant la somme des éléments d'une liste d'entiers.
2. Définir un prédicat `sublist : list nat → list nat → Prop` tel que l'on a `sublist` $l_1$ $l_2$ si et seulement si la liste $l_1$ est obtenue en supprimant des éléments dans $l_2$ (éventuellement aucun).
3. Écrire une fonction booléenne `ssp : nat → list nat → bool` décidant le problème de la somme du sous-ensemble *i.e.* telle que `ssp` $s$ $l$ renvoie `true` si et seulement un sous-ensemble des éléments de $l$ a pour somme $s$. On ne cherchera pas à écrire un algorithme efficace (le problème est NP-complet).
4. Utiliser `sublist` et `sum` pour énoncer la correction de la fonction `ssp`. (On ne cherchera pas à faire la preuve.)
5. Donner le type d'une version fortement spécifiée de la fonction `ssp`.

# 2 L'opérateur $\varepsilon$ de Hilbert

L'opérateur de description indéfinie d'Hilbert, dit opérateur $\varepsilon$, permet de dissocier l'écriture d'un terme des conditions de l'existence de l'objet correspondant. Ainsi on peut vouloir écrire $\log_2(x)$ dans un discours mathématique comme une abbréviation de l'expression « un nombre $n$, s'il existe, tel que $2^n = x$ ». L'idée est que l'on pourra toujours écrire $\log_2(x)$ mais qu'on ne pourra rien dire de sa valeur lorsque $x$ n'est pas une puissance de 2.

Plus précisément, un $\varepsilon$-terme est de la forme $\varepsilon\ P$ où $P : A \to$ `Prop` est un prédicat. L'interprétation du terme $\varepsilon\ P$ est « un $x$ satisfaisant $P$, s'il existe, sinon un objet arbitraire ». On introduit donc ainsi l'opérateur $\varepsilon$ en Coq :

```
Parameter epsilon : forall (A:Set), A → (A → Prop) → A.
Axiom epsilon_spec :
  forall (A:Set) (a:A), forall P, (exists x, P x) → P (epsilon A a P).
```

1. Le deuxième argument de `epsilon` sert à garantir que le type `A` est habité. Justifier cette condition.
2. Définir une fonction `power2 : nat → nat` telle que `power2`$(n) = 2^n$ et en déduire, à l'aide de l'opérateur $\varepsilon$, une fonction `log2 : nat → nat` donnant le logarithme en base 2 exact, lorsqu'il existe.
3. Donner un schéma d'utilisation naturel de l'opérateur $\varepsilon$, de la forme

   ```
   forall (A:Set) (a:A) (P Q:A→Prop), ... → Q (epsilon A a P)
   ```

4. En utilisant l'opérateur $\varepsilon$, établir une preuve de

   ```
   forall (P Q : Prop), P∨Q → {P}+{Q}
   ```

   (On ne demande pas la preuve Coq dans tous ses détails, mais seulement l'idée générale et les différentes étapes.) Que peut-on en conclure quant à la distinction `Prop`/`Set` et au mécanisme d'extraction associé ? Plus simplement, que dire d'`epsilon` du point de vue de l'extraction ?
5. De même, utiliser l'opérateur $\varepsilon$ pour établir une preuve de

   ```
   forall (A B:Prop), (A→B)∨(B→A).
   ```

6. En déduire une preuve de

   ```
   forall (A:Prop), ~A ∨ ~~A
   ```

   Était-ce une proposition prouvable dans le Calcul des Constructions Inductives ? Pourquoi ?
7. L'opération de description *définie*, notée $\iota$, est un cas particulier de l'opérateur $\varepsilon$ lorsqu'il y a de plus *unicité* de l'objet correspondant à $\varepsilon\ P$. Définir l'opérateur $\iota$ à partir de l'opérateur $\varepsilon$ et donner pour $\iota$ un théorème `iota_spec` comparable à `epsilon_spec`.

# 3 Le lièvre et la tortue

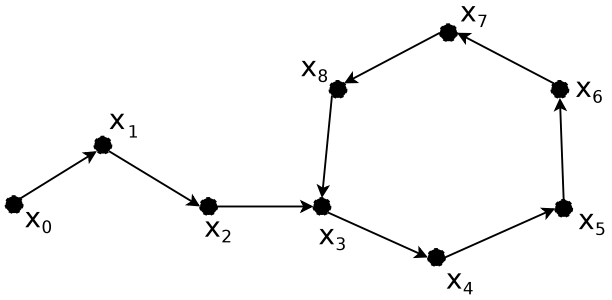
Cet exercice est une application de l'opérateur $\varepsilon$ introduit à la question précédente. Les questions de cet exercice sont indépendantes de celles du précédent mais on pourra utiliser `epsilon` et `epsilon_spec` introduits plus haut.

Considérons une fonction $f : A \to A$ sur un ensemble $A$ fini et la suite $(x_n)_{n\in\mathbb{N}}$ définie par

$$\begin{cases} x_0 \in A \\ x_{n+1} = f(x_n) \end{cases}$$

Il est clair que cette suite est cyclique à partir d'un certain rang, $A$ étant fini. Plus précisément, il existe deux entiers $\lambda \geq 0$ et $\mu \geq 1$ minimaux tels que

$$\forall i,\ \forall j,\ x_{\lambda+i} = x_{\lambda+i+j\mu}$$




Dit autrement, $\lambda$ est le nombre d'étapes pour atteindre le cycle et $\mu$ sa longueur. La figure ci-dessus illustre une situation où $\lambda = 3$ et $\mu = 6$. L'algorithme de Floyd, dit « du lièvre et de la tortue », calcule un entier $m > 0$ tel que $x_m = x_{2m}$, et donc un multiple de $\mu$, en temps $O(\lambda + \mu)$ et en espace $O(1)$. Il consiste en deux parcours simultanés de la séquence $(x_n)$, à deux vitesses différentes :

```
let rec course m x y =  (* x = x_m ∧ y = x_2m *)
    if x = y then m else course (m + 1) (f x) (f (f y))
let r = course 1 (f x0) (f (f x0))
```

Le but de cet exercice est d'écrire cet algorithme en Coq et donc en particulier de prouver sa terminaison.

1. On se donne les paramètres Coq suivants :

   ```
   Parameter A : Set.
   Parameter eq_A_dec : forall (x y:A), {x=y}+{~x=y}.
   Parameter f : A → A.
   Parameter x0 : A.
   ```

   Définir la fonction `x : nat → A` correspondant à la suite $(x_n)$.

2. On introduit ensuite $\lambda$ et $\mu$ de la manière suivante (leur minimalité n'est pas importante ici) :

   ```
   Parameter lambda : nat.
   Parameter mu : nat.
   Axiom mu_positive : mu > 0.
   Axiom lambda_mu : forall i j, x (lambda+i) = x (lambda+i+j*mu).
   ```

   La justification de la terminaison de la fonction `course` est la suivante : soit $x_m$ n'est pas encore entré dans le cycle et dans ce cas la distance de $x_m$ au cycle décroît, soit $x_m$ est déjà dans le cycle et dans ce cas la distance entre $x_{2m}$ et $x_m$ décroît. On va donc utiliser un ordre lexicographique.

   (a) Définir l'ordre lexicographique usuel sur les paires d'entiers naturels sous la forme d'un prédicat `lex : nat*nat → nat*nat → Prop`.

   (b) À l'aide de l'opérateur $\varepsilon$, définir la distance de $x_{2m}$ à $x_m$, c'est-à-dire le plus petit entier naturel $i$ tel que $x_{2m+i} = x_m$ lorsqu'il existe.

   (c) Justifier l'existence de cette distance lorsque $m \geq \lambda$. (On ne demande pas une preuve Coq.)

(d) En déduire une fonction `variant : nat → nat*nat` telle que `variant` $m$ est la quantité qui décroît lors de l'appel récursif pour la relation `lex`, *i.e.* telle que l'énoncé suivant soit prouvable :

$$\forall m,\ x_m \neq x_{2m} \Rightarrow \texttt{lex}\ (\texttt{variant}\ (S\ m))\ (\texttt{variant}\ m)$$

3. Donner une spécification à la fonction récursive `course`.
4. Indiquer comment utiliser l'ordre lexicographique `lex` pour définir cette fonction récursive, en supposant donnée une preuve `lex_wf : well_founded lex`.
5. En déduire un terme Coq de type `{ m :nat | m > 0 ∧ x m = x (2*m) }`.

# A Extrait de la bibliothèque standard de Coq

```
Inductive nat : Set := :=  O : nat | S : nat -> nat.
plus  : nat -> nat -> nat (* addition *)
minus : nat -> nat -> nat (* soustraction; minus n m = O si n<=m *)
eq_nat_dec : forall (n m : nat), {n = m} + {n <> m}
le_lt_dec : forall (n m : nat), {n <= m} + {m < n}

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

Inductive bool : Set :=  true : bool | false : bool
andb : bool -> bool -> bool (* ET booléen *)
orb  : bool -> bool -> bool (* OU booléen *)
```