Examen cours 2-7-2 Assistants de preuve

Mardi 12 février 2008

L'énoncé est composé de 3 pages. L'examen dure 2 heures 30. 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 Logique combinatoire

On modélise dans cet exercice quelques opérations sur la logique combinatoire qui est un formalisme équivalent au lambda-calcul mais qui n'utilise pas de lieurs.

Les termes de la logique combinatoire sont formés à partir de variables de deux constantes particulières appelées combinateurs notées S et K et d'un symbole binaire d'application. On notera $t\ u$ l'application du terme t au terme u. L'application associe à gauche.

Les règles pour une étape de réduction de la logique combinatoire sont les suivantes :

$$\mathsf{K}\ t\ u \longrightarrow t \qquad \mathsf{S}\ t\ u\ v \longrightarrow (t\ v)\ (u\ v) \qquad \frac{t \longrightarrow t'}{t\ u \longrightarrow t'\ u} \qquad \frac{u \longrightarrow u'}{t\ u \longrightarrow t\ u'}$$

1. Définir en Coq le type des termes de la logique combinatoire.
2. Définir une relation correspondant à une étape de réduction.
3. Définir un prédicat inductif qui exprime qu'un terme de la logique combinatoire est fortement normalisant (toutes les suites de réduction terminent).
4. Définir un prédicat inductif qui exprime qu'un terme de la logique combinatoire est faiblement normalisant (il existe une suite de réductions qui termine).
5. On souhaite écrire une fonction qui calcule une étape de réduction lorsque c'est possible (par exemple la plus externe) et qui échoue sinon. Ecrire un terme fonctionnel Coq qui simule ce comportement à l'aide d'un type option. Ecrire la propriété exprimant la correction de cette fonction (on ne cherchera pas à la démontrer).
6. Ecrire une fonction Coq qui étant donné un terme de la logique combinatoire et un entier n renvoie un terme dans lequel on a effectué le plus de réductions possibles sans dépasser n.
7. Ecrire une fonction Coq qui calcule la forme normale d'un terme fortement normalisable.

1.1 Cas typé

Partie optionelle On associe aux termes de la logique combinatoire un système de type simple. Un type est soit un type de base c soit de la forme $\tau_1 \to \tau_2$ avec τ_i des types. On convient que $\to$ associe à droite.

Le combinateur K a pour type $\tau_1 \to \tau_2 \to \tau_1$ et le combinateur S a pour type $(\tau_1 \to \tau_2 \to \tau_3) \to (\tau_1 \to \tau_2) \to \tau_1 \to \tau_3$ (pour τ_i quelconques). La règle de typage de l'application est la même que pour le lambda-calcul.

- Définir le type Coq typ correspondant aux types de la logique combinatoire.
- Définir le prédicat qui dit qu'un terme est bien typé de type τ dans un environnement donné (qui pourra être représenté comme une fonction des variables dans les types).
- Redéfinir les termes de la logique combinatoire typée comme une famille indexée par l'environnement et le type du combinateur.

2 Tableaux

Un tableau est une structure non persistente, ce qui est illustré dans le programme suivant :

```
let t = Array.create 4 'a'
let x = t
let y = t.(0) <- 'b'; t
let _ = x.(0)=y.(0)
```

Ce programme renvoie la valeur `true` car l'affectation dans le tableau `t` à la troisième ligne affecte le tableau associé à la variable `x`. Pour préserver l'ancienne valeur du tableau, on peut le recopier ou utiliser une structure fonctionnelle de liste mais cela est moins efficace en temps d'accès ou en espace. On peut également utiliser des structures de données persistantes qui allient l'efficacité de la modification en place et la préservation des valeurs passées.

On peut construire des tableaux persistants de la manière suivante. Un tableau persistant sera une référence sur un objet qui est soit un tableau ordinaire soit une différence donnée par un indice, une valeur et un tableau persistant.

Le type CAML des tableaux persistants est donc le type `'a t` défini par :

```
type 'a t = 'a data ref
and 'a data = | Array of 'a array | Diff of int * 'a * 'a t
```

Tout tableau persistant est donc un pointeur qui donne accès à un ensemble de différences et finalement à un tableau classique. L'accès est d'autant plus efficace qu'il y a moins d'étapes de dífférence. Il existe plusieurs représentations du même tableau. En particulier on peut toujours s'arranger pour qu'un tableau persistant t soit un pointeur sur une donnée de la forme $(\texttt{Array}\, a)$.

En effet si $!t$ est de la forme $(\texttt{Diff}\, i\, v\, t')$ et si t' est un pointeur vers $(\texttt{Array}\, a)$, soit v' la valeur $a.(i)$, on affecte $a.(i) < -v$, alors t est représenté par $(\texttt{Array}\, a)$ et pour assurer la persistance, on affecte à t' la valeur $(\texttt{Diff}\, i\, v'\, t)$.

Dans le cas général, on peut définir une fonction `reroot` qui étant donné un tableau persistant t, élimine si nécessaire les opérations `Diff` de manière à ce que t soit directement un pointeur sur une donnée de la forme $(\texttt{Array}\, a)$.

```
let rec reroot t = match !t with
  | Array _ → ()
  | Diff (i, v, t') →
      (reroot t';
        match !t' with
          | Array a as n →
             let v' = a.(i) in a.(i) <- v; t := n; t' := Diff (i, v', t)
          | Diff _ → assert false)
```

Les opérations `get` et `set` commencent par cette opération de reconfiguration et sont définies ainsi :

```
let rec get t i = match !t with
  | Array a → a.(i)
  | Diff _ →
      (reroot t; match !t with Array a → a.(i) | Diff _ → assert false)

let set t i v =
  reroot t;
```

```
match !t with
| Array a as n →
    let old = a.(i) in
    if old == v then t
    else (a.(i) <- v; let res = ref n in t := Diff (i, old, res); res)
| Diff _ → assert false
```

Questions

1. Proposer un ensemble de types Coq pour modéliser le comportement des tableaux persistants. On utilisera en particulier un type `pointer` pour représenter les références et un mémoire auxiliaire pour stocker les valeurs associées aux références.
2. En utilisant ce type de données, proposer une version fonctionelle des fonctions `reroot` et `get`.
3. Proposer une spécification pour les fonctions `reroot`, `get` et `set` .

3 Logique classique et proof-irrelevance

Exercice possible si cela n'a pas ete fait en cours. A detailler

This exercise show that the excluded middle implies proof-irrelevance. We assume as an axiom that $\forall A, A \vee \neg A$.

- Show $\forall A, \neg\neg A \to A$.
- Introduce `BOOL` of type `Prop` with two elements T and F.
- Assume $T \neq F$, build f : `Prop` $\to$ `BOOL` such that $\forall A, A \longrightarrow f\ A = T$.
- Load the module `Hurkens` in the standard library, look at the type of theorem `paradox`. Deduce a proof of $T = F$.
- Prove the proof-irrelevance property : $\forall (A : \texttt{Prop})(p\,q : A), p = q$.

A Extrait de la bibliothèque standard de Coq

A adapter en fonction du sujet

```
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 *)
```