Examen cours 2-7-2 Assistants de preuve

Mardi 10 février 2009

L'énoncé est composé de 3 pages. L'examen dure 2 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 Définitions inductives : le type W (8 points)

Le type W des arbres bien fondés est paramétré par un type A et une famille de types $B : A \to \texttt{Type}$. C'est un type avec un seul constructeur défini par : Inductive W (A :Type)(B :A->Type) : Type := node : forall (a :A), (B a -> W A B) -> W A B. Le type A sert à paramétrer les noeuds et le type $B\,a$ donne l'arité du nœud paramétré par a.

1. Donner le type de l'élimination dépendante pour le type W sur la sorte `Type`.
2. Pour coder le type `nat` des entiers avec `O` et `S`, il faut deux types de nœuds. On prendra donc $A = \texttt{bool}$. Le constructeur `O` va correspondre à $a = \texttt{false}$, ce constructeur ne prend aucun argument donc on aura $B\,\texttt{false} = \texttt{empty}$. Le constructeur `O` correspondra à $a = \texttt{true}$, ce constructeur prend un argument donc on aura $B\,\texttt{true} = \texttt{unit}$.
   Donner les termes correspondants à `nat`, `O` et `S` en utilisant ce codage.
3. Donner le codage en utilisant le type W du type `tree` des arbres binaires paramétré par un type des valeurs V, c'est-à-dire pour lequel on a un constructeur `leaf` de type $(\texttt{tree}\,V)$ et un constructeur `bin` de type $\texttt{tree}\,V \to V \to \texttt{tree}\,V \to \texttt{tree}\,V$. On donnera le codage du type et de ses constructeurs.
4. Si on se donne n de type `nat`, construire deux fonctions f_1 et f_2 de type $\texttt{unit} \to \texttt{nat}$ telles que $\forall x : \texttt{unit}, f_i\,x = n$ soit prouvable mais telles que f_1 et f_2 ne soient pas convertibles.
5. Quelle conséquence cela a sur le codage de `nat` à l'aide de W ? Proposer une égalité sur le type W qui permette de résoudre ce problème.

2 Logique classique et égalité des preuves (8 points)

Le but de cet exercice est de montrer que le tiers-exclu implique l'égalité des preuves. On se place dans un environnement qui comporte l'axiome `em` de type $\forall A : \texttt{Prop}, A \vee \neg A$.

1. Donner le type le plus général pour l'élimination dépendante : fun A P f g => match em A as x return P x with $(\text{or}_i ntrola) \Rightarrow fa | (or_i ntrorb) \Rightarrow gbend$.

2. Construire un terme `dneg` de type $\forall A, \neg\neg A \to A$
3. Construire un type `BOOL` of type `Prop` contenant deux éléments T et F (on peut choisir une définition inductive ou bien un codage imprédicatif).
4. Construire un terme $p2b$ de type `Prop` $\to$ `BOOL` tel que $(p2b\, A)$ est egal à T si A est vrai et F sinon.
5. Construire une preuve de $T \neq F \to (\forall A, A \leftrightarrow f\, A = T)$.
6. Dans le calcul des constructions, il est possible de construire un terme de type : paradox : forall (B : Prop) (p2b : Prop -> B) (b2p : B -> Prop), (forall A : Prop, b2p (p2b A) -> A) -> (forall A : Prop, A -> b2p (p2b A)) -> forall C : Prop, C En utilisant le terme `paradox` et le terme $p2b$ précédent, construire une preuve de $\neg T \neq F$ puis une preuve de $T = F$.
7. En déduire la propriété d'égalité des preuves : $\forall (A : \texttt{Prop})(p\, q : A), p = q$

3 Programmation en Coq (10 points)

On se donne une type des arbres binaires avec des valeurs entières : Inductive tree : Type := L : nat -> tree | N : nat -> tree -> tree -> tree

1. Ecrire une fonction récursive `mult` de type `tree` $\to$ `nat` en Coq qui calcule le produit des éléments de l'arbre.
2. Définir inductivement un prédicat `inO` sur les arbres qui exprime que l'entier 0 fait partie des éléments de l'arbre.
3. Afin d'effectuer moins de multiplications, on souhaite développer une fonction `multO` qui renvoie une exception lorsqu'elle trouve 0 dans l'arbre et sinon calcule le produit des éléments (c'est-à-dire la même chose que `mult`).
   (a) Proposer une spécification pour cette fonction en utilisant des types dépendants.
   (b) Donner un terme de preuve pour cette fonction. On ne détaillera pas les preuves logiques associées aux spécifications mais on se contentera d'énoncer les lemmes nécessaires.
   (c) Donner le terme extrait du programme précédent.
4. On remarque que lorsqu'un 0 est trouvé dans l'arbre, le calcul ne s'arrête pas mais l'exception est propagée récursivement jusqu'à la racine. Pour éviter ce comportement, on propose une autre transformation monadique basée sur les continuations :

   $$M\, A \equiv \forall C, (A \to C) \to C \to C$$

   Chaque calcul de type A est interprété comme un terme de type $M\, A$. Ce terme attend comme argument un programme f de type $A \to C$ (la continuation normale) et un programme x de type C (la continuation d'exception). Si le calcul se termine normalement sur une valeur a alors le résultat sera $f\, a$ par contre si le calcul se termine de manière exceptionnelle alors c'est la valeur x qui est renvoyée.
   (a) Quel est le lien entre le type inductif `option` A et le type $M\, A$?

(b) Donner les constructions monadiques `unit` de type $A \to M\,A$ et `bind` de type $M\,A \to (A \to M\,B) \to M\,B$.

(c) Donner le codage de l'opération de levée d'exception `raise` de type $M\,A$ et `catch` de type $M\,A \to A \to A$ telle que `catch` $e_1\,e_2$ renvoie la valeur de e_1 si e_1 ne lève pas d'exception et sinon se comporte comme e_2.

(d) Reconstruire en utilisant *de manière abstraite* les opérations `bind`, `unit`, `raise` et `catch` précédentes une fonction `mult1` qui calcule le produit d'un arbre en distingant le cas où l'arbre contient 0 et en construisant de manière récursive une valeur de type M `nat`. Déduire de `mult1` une fonction de type `tree` $\to$ `nat` qui calcule le produit des éléments de l'arbre.

Définitions Coq

Inductive or (A B : Prop) : Prop := or$_i$$ntrol : A- > AB|or_intror : B- > ABInductiveempty : Type := .Inductiveunit : Type := tt : unitInductivesumor(A : Type)(B : Prop) : Type := inleft : A- > A + B|inright : B- > A + BInductivesig(A : Type)(P : A- > Prop) : Type := exist : forallx : A, Px- > sigP