Examen partiel de sémantique des langages de programmation

Avril 2012

1 Programmation en Coq

1. *Quel est le type usuel d'une fonction à trois arguments, tous des nombres, et dont le résultat est un booléen?*
   Son type est `nat -> nat -> nat -> bool`.
   Si `f` *est une fonction ayant ce type, comment écrit-on "f appliquée aux nombres 1, 2, et 3"?*
   L'application de la fonction s'écrit `f 1 2 3`.

2. *Ecrire une fonction qui prend en entrée trois entiers naturels et qui retourne* `true` *si ces trois entiers naturels sont dans l'ordre croissant (au sens large) et* `false` *sinon. On pourra utiliser une fonction appelé* `leb` *qui compare deux entiers naturels, et retourne une valeur booléenne (*`true` *si le premier argument est inférieur ou égal au deuxième).*
   Une telle fonction peut s'écrire :

   ```
   Definition leb3 m n p := andb (leb m n) (leb n p).
   ```

3. *Ecrire une déclaration pour un type de donnée* `t23` *qui contient trois cas:*

   - *le premier cas comporte 2 composantes, qui sont dans le type lui-même,*
   - *le deuxième cas comporte 3 composantes, qui sont dans le type lui-même,*
   - *le troisième cas comporte 1 composante, qui est un nombre entier.*

   On peut définir ce type de la manière suivante :

   ```
   Inductive t23 :=

     | Cas2 (t1 t2 t3 : t23)
     | Cas3 (n : nat).
   ```

4. *Ecrire une fonction qui prend en entrée un terme de type* `t23` *et retourne la liste de toutes les valeurs portées par* `t23`*. On pourra utiliser une fonction* `app` *qui concatène deux listes.*
   Cette fonction peut s'écrire :

   ```
   Fixpoint to_list (t : t23) :=
     match t with
     | Cas1 t1 t2 => app (to_list t1) (to_list t2)
     | Cas2 t1 t2 t3 => app (app (to_list t1) (to_list t2)) (to_list t3)
     | Cas3 n => n :: nil
     end.
   ```

2 Tactiques simples

5. *Donnez une tactique (éventuellement composée) qui permet de résoudre le but suivant:*

```
A : Prop
B : Prop
H : B
H1 : (A -> B) -> A
=========================
  A
```

   Ce but est résolu par la tactique composée suivante :

```
apply H1; intro HA; assumption.
```

6. *Donnez une tactique ou une séquence de tactiques pour résoudre le but suivant:*

```
P : nat -> Prop
H : forall x : nat, P x -> P (S x)
H1 : P 3 \/ P 2
=========================
  P 4 /\ P (7 - 3)
```

   Il faut remarquer que `P 4` et `P (7 - 3)` sont deux termes convertibles. On a donc juste à faire une disjonction de cas selon H1 et appliquer H pour se ramener au bon indice :

```
destruct H1 as [H2 | H3].
  split; apply H; assumption.
split; apply H; apply H; assumption.
```

7. *Donnez une tactique ou une séquence de tactiques pour résoudre le but suivant:*

```
P : nat -> Prop
x : nat
y : nat
H : P x
H1 : S x = S y
========================
  P y
```

   Voici une solution pour ce problème.

```
injection H1.
(* nouveau but :
  ...
  ========================
  x = y -> P y
*)
intros H2; rewrite <- H2; exact H.
```

8. *Donnez un exemple de but pour lequel la tactique* `ring` *s'applique et pas la tactique* `omega`. Il suffit de choisir une égalité prouvable ou le produit entre plusieurs formules contenant des variables apparaissent. Par exemple la formule suivante a dans un membre le produit `x * x` et dans l'autre membre le produit `(x + 1) * x`.

```
x : Z
======================
 (x + 1) * x = x * x + x
```

3 Preuves de programmes

9. *Donnez un script de preuve qui permet de prouver l'énoncé suivant*

```
forall x, x <> 0 ->
   match x with 0 => false | S _ => true end = true
```

Voici la solution. Le programme `match x with ... end` est étudié cas par cas grâce à la tactique `destruct x`. Dans le cas où `x` vaut 0, cela fait apparaitre une hypothèse `0 <> 0`, qui est contradictoire. Dans le cas où `x` est un successeur, le programme se réduit à `true` et l'énoncé à prouver est alors très simple.

```
Lemma l1 : forall x, x <> 0 ->
  match x with 0 => false | S _ => true end = true.
Proof.
intros x xn0.
destruct x.
(* premier but
  xn0 : 0 <> 0
  ============================
   false = true *)
case xn0.
(* ...
  ============================
   0 = 0 *)
reflexivity.
(* deuxième but provenant du traitement par cas sur x.
  S n <> 0
  ============================
   true = true *)
reflexivity.
Qed.
```

10. *On considère le type de données et la fonction suivants:*

```
Inductive bin := N (t1 t2 : bin) | L (x : Z).

Fixpoint flip t :=
  match t with
    N t1 t2 => N (flip t2) (flip t1)
  | L x => L x
  end.
```

Donnez un script de preuve qui permet de prouver

```
forall t, flip (flip t) = t
```

Vous pourrez indiquer les étapes principales de la preuve en donnant le but engendré à certains endroits. Voici la solution. Cette preuve se fait simplement par récurrence sur la structure de l'argument `t`. Lorsque `t` vaut `N t1 t2`, l'expression `flip (N t1 t2)` se calcule en `N (flip t2) (flip t1)` et `flip (flip (N t1 t2))` se calcule en
`N (flip (flip t1)) (flip (flip t2))`.

```
Lemma flip_inv : forall t, flip (flip t) = t.
induction t as [t1 IHt1 t2 IHt2 | x].
(*premier but provenant de la preuve par récurrence
  t1 : bin
  IHt1 : flip (flip t1) = t1
  t2 : bin
  IHt2 : flip (flip t2) = t2
  =============
    flip (flip (N t1 t2)) = N t1 t2 *)
simpl.
(* ...
   ============
    N (flip (flip t1)) (flip (flip t2))
    = N t1 t2 *)
rewrite IHt1.
(* ...
   ============
    N t1 (flip (flip t2)) = N t1 t2 *)
rewrite IHt2.
(* ...
   ============
    N t1 t2 = N t1 t2 *)
reflexivity.
(*deuxième but provenant de la preuve par récurrence
  x : Z
  =============
   flip (L x) = L x *)
reflexivity.
Qed.
```

4 Preuves en sémantique

```
Inductive aexpr : Type :=
  avar (s:string) | anum (x:Z) | aplus (e1 e2 : aexpr).

Inductive bexpr : Set :=
  blt (e1 e2 : aexpr).

Inductive instr : Type:=
  skip | assign (s:string) (e:aexpr)
| sequence (i1 i2 : instr) | while (b : bexpr) (i:instr).

Definition env := list (string * Z).

Inductive aeval : env -> aexpr -> Z -> Prop :=
  ae_num : forall r n, aeval r (anum n) n
| ae_var1 : forall r x v, aeval ((x,v)::r) (avar x) v
| ae_var2 : forall r x y v v' , x <> y -> aeval r (avar x) v' ->
              aeval ((y,v)::r) (avar x) v'
| ae_plus : forall r e1 e2 v1 v2,
              aeval r e1 v1 -> aeval r e2 v2 ->
              aeval r (aplus e1 e2) (v1 + v2).
Inductive beval : env -> bexpr -> bool -> Prop :=
| be_lt1 : forall r e1 e2 v1 v2,
            aeval r e1 v1 -> aeval r e2 v2 ->
```

```
            v1 < v2 -> beval r (blt e1 e2) true
| be_lt2 : forall r e1 e2 v1 v2,
            aeval r e1 v1 -> aeval r e2 v2 ->
            v2 <= v1 -> beval r (blt e1 e2) false.

Inductive update : env->string->Z->env->Prop :=
| s_up1 : forall r x v v', update ((x,v)::r) x v' ((x,v')::r)
| s_up2 : forall r r' x y v v', update r x v' r' ->
            x <> y -> update ((y,v)::r) x v' ((y,v)::r').

Inductive exec : env->instr->env->Prop :=
| SN1 : forall r, exec r skip r
| SN2 : forall r r' x e v,
   aeval r e v -> update r x v r' -> exec r (assign x e) r'
| SN3 : forall r r' r'' i1 i2,
    exec r i1 r' -> exec r' i2 r'' ->
    exec r (sequence i1 i2) r''
| SN4 : forall r r' r'' b i,
    beval r b true -> exec r i r' ->
    exec r' (while b i) r'' ->
    exec r (while b i) r''
| SN5 : forall r b i,
    beval r b false -> exec r (while b i) r.
```

11. *Ecrivez une fonction qui prend en entrée un environnement quelconque (de type* `env`*) et retourne une* **instruction** *composée, qui met à 0 toutes les variables de cet environment. Quelle formule pourrait-on écrire pour exprimer la correction de cette fonction?*

    La réponse est donnée par la fonction suivante:

    ```
    Fixpoint e2i (e : env) :=
     match e with
       nil => skip
     | (a,v)::tl => sequence (e2i tl) (assign a (anum 0))
     end.
    ```

    Une formule qui exprime que cette fonction est correcte est la suivante:

    ```
    forall e, exists e', exec e (e2i e) e' /\
      forall x n, aeval e (avar x) n ->
                  aeval e' (avar x) 0
    ```

    Faire la preuve de cette formule n'était pas demandé.