Travaux pratiques
TD7 - Sémantique naturelle

Sylvain Chevillard et Damien Pous

7 avril 2008

Dans ce TD, nous avons besoin de charger quelques bibliothèques :

```
Require Import List.
Require Import Arith.
Require Export ArithRing.
```

Nous allons définir et utiliser (sur un petit exemple jouet) une sémantique pour un langage de programmation minimal.

On définit la fonction de division par 2 dans les entiers de la façon suivante :

```
Fixpoint div2 (n:nat) : nat :=
match n with
| 0 => 0
| 1 => 0
| S (S p) => (div2 p)+1
end.
```

1 Le langage.

On représente un petit langage de programmation par le type inductif suivant :

```
Inductive expr : Set :=
  var : nat -> expr
| num : nat -> expr
| plus : expr -> expr -> expr
| moins : expr -> expr -> expr.

Inductive bexpr : Set :=
  greater : expr -> expr -> bexpr.

Inductive instr : Set :=
  assign : nat -> expr -> instr
| sequence : instr -> instr -> instr
| while : bexpr -> instr -> instr.
```

Ce langage de programmation est impératif, les variables sont appelées var_1 var_2 var_3 ... var_i, et sont représentées par le constructeur `var i` lorsqu'elles apparaissent dans une expression. L'affectation `var_i := e` est représentée directement par l'instruction `assign i e`. Les nombres entiers sont représentés par le constructeur `num`.

Les seuls tests que l'on peut faire sont des tests de comparaison stricte représentés par `greater`, les seules operations arithmétiques que l'on peut faire sont des additions et les soustractions (étant entendu que si $n > m$, $m - n = 0$; c'est le comportement de la fonction correspondante dans Coq).

Pour décrire le comportement des programmes, nous considérons que le comportement d'un programme consiste en la modification d'un état mémoire. Les états mémoires seront représentés par des éléments du type `state`, à savoir une liste de couple (i, n) indiquant que la valeur n est actuellement affectée à la variable i. On prendra les conventions suivantes :

– si un état ne contient aucune entrée de la forme $(i, \cdot)$, l'expression `var i` est dénuée de sens ;
– il n'y a pas de portée des variables. La liste des variables qui apparaît dans un état ne change pas au cours de l'algorithme (autrement dit, on supposera qu'il y a un processus d'initialisation des variables : ce processus introduit les variables dans l'état initial. Après quoi, on se contentera de mettre à jour les valeurs associées à chaque variable dans la liste).

```
Definition state := list (nat*nat).
```

Nous allons utiliser cinq prédicats inductifs :

– `(lookup s i v) : Prop` signifie que `var i` vaut v dans l'état s ;
– `(update s i v s') : Prop` signifie que, partant de l'état s, l'affectation de la valeur v dans `var i` conduit à l'état s' ;
– `(eval s e v) : Prop` signifie que l'expression e s'évalue en la valeur v dans l'état s ;
– `(beval s e b) : Prop` signifie que l'expression booléenne e s'évalue en b dans l'état s ;
– `(nat_sem s p s') : Prop` signifie que l'exécution de l'instruction p à partir de l'état s conduit à l'état s'.

Écrire ces prédicats.

2 Un petit programme.

Soit le programme C suivant :

```
while(i>0) {
  x = x+i;
  i = i-1;
}
```

Dans un environnement où i et x sont déclarées et valent initialement respectivement n et x_0, le programme calcule

$$x_0 + \sum_{k=1}^{n} k = x_0 + \frac{n(n+1)}{2}.$$

La traduction de ce programme dans notre langage est :

```
Definition mon_prog : instr :=
  while (greater (var 1) (num 0))
   (sequence (assign 2 (plus (var 2) (var 1)))
             (assign 1 (moins (var 1) (num 1)))
    ).
```

Démontrer que la sémantique du programme est bien celle attendue :

```
Theorem mon_prog_sum : forall n v,
   nat_sem ((1,n)::(2,v)::nil) mon_prog
           ((1,0)::(2,v + (div2(n*(n+1))))::nil).
```

On n'hésitera pas à créer des lemmes auxiliaires pour les calculs purement techniques sur les entiers. En outre, on pourra utiliser la tactique `ring` pour traiter les égalités arithmétiques.

3 Exécution.

Nous avons précédemment défini des prédicats inductifs. Il aurait été plus naturels peut-être d'utiliser des fonctions. Cependant, puisqu'un programme donné peut n'avoir aucun sens dans un état donné (par exemple `var 1` dans l'état `nil`), les fonctions ainsi définies ne peuvent être totales. Pour contourner cette difficulté, on utilise le type paramétré `option` et ses constructeurs `Some` et `None`. Définir les fonctions :

- `flookup` : `state -> nat -> option nat`
- `fupdate` : `state -> nat -> nat -> option state`
- `feval` : `state -> expr -> option nat`
- `fbeval` : `state -> bexpr -> option bool`

Remarque : pour définir ces fonctions, vous aurez besoin de pouvoir tester l'égalité entre deux entiers et d'effectuer des tests de comparaison. Vous utiliserez les prédicats `eq_nat_dec` et `le_gt_dec`.

Prouver alors les théorèmes suivants :

```
Theorem lookup_equiv :
  forall s i v, lookup s i v -> flookup s i = Some v.
Theorem update_equiv :
  forall s i v s', update s i v s' -> fupdate s i v = Some s'.
Theorem eval_equiv :
```

```
  forall s e v, eval s e v -> feval s e = Some v.
Theorem beval_equiv :
  forall s e b, beval s e b -> fbeval s e = Some b.
```