Programming Language Semantics
Exam solution

Yves Bertot

October 20, 2006

1 Programming language semantics

I expect students to be able to write semantic rules for this kind of language. These semantic rules could be given either as inference rules or as Coq definitions. Here are the Coq definition. Because they are data that can be processed with a computer, it was easier to test them and check that they correspond to the informal meaning. We will also provide theory-like inference rules, but the Coq definition should be used as main reference in case of doubt.

By comparison with the `little` language that was used in the course notes, we re-use the `lookup` and `update` predicates. There is now only one important syntactic category. Also, there is only one judgment, which combines the evaluation of expressions with the side-effects, which were previously described separately in the execution of instructions. We choose to use the wording "executing an expression". We define a predicate `exec` sucht that `exec` ρ e ρ' v means *executing* e *in environment* ρ *terminates with the new environment* ρ' *and the value* v.

```
Require Import ZArith List Omega.
Open Scope Z_scope.

Inductive exp : Set :=
  num : Z -> exp
| var : Z -> exp
| plus : exp -> exp -> exp
| mult : exp -> exp -> exp
| assign : Z -> exp -> exp
| while : exp -> exp -> exp
| seq : exp -> exp -> exp.

Definition env := list (Z * Z).

Inductive lookup : env -> Z -> Z -> Prop :=
  lk1 : forall i n s, lookup ((i,n)::s) i n
```

```
| lk2 : forall i1 i2 n1 n2 s, i1 <> i2 -> lookup s i1 n1 ->
        lookup ((i2,n2)::s) i1 n1.

Inductive update : env -> Z -> Z -> env -> Prop :=
  up1 : forall i n1 n2 s, update ((i,n1)::s) i n2 ((i,n2)::s)
| up2 : forall i1 i2 n1 n2 s1 s2,
        update s1 i1 n1 s2 ->
        update ((i2,n2)::s1) i1 n1 ((i2,n2)::s2).

Inductive exec : env -> exp -> env -> Z -> Prop :=
  e_n : forall n s, exec s (num n) s n
| e_v : forall i n s, lookup s i n -> exec s (var i) s n
| e_p : forall e1 e2 s s' s'' v1 v2,
        exec s e1 s' v1 -> exec s' e2 s'' v2 ->
        exec s (plus e1 e2) s'' (v1+v2)
| e_m : forall e1 e2 s s' s'' v1 v2,
        exec s e1 s' v1 -> exec s' e2 s'' v2 ->
        exec s (mult e1 e2) s'' (v1*v2)
| e_a : forall i e v s s' s'',
        exec s e s' v -> update s' i v s'' ->
        exec s (assign i e) s'' v
| e_wf : forall s s' e1 e2 n,
         exec s e1 s' n -> n <= 0 ->
         exec s (while e1 e2) s' n
| e_wt : forall s1 s2 s3 s4 e1 e2 n n' n'',
         exec s1 e1 s2 n -> 0 < n -> exec s2 e2 s3 n' ->
         exec s3 (while e1 e2) s4 n'' ->
         exec s1 (while e1 e2) s4 n''
| e_s : forall s1 s2 s3 e1 e2 n n',
        exec s1 e1 s2 n -> exec s2 e2 s3 n' ->
        exec s1 (seq e1 e2) s3 n'.
```

Here is the presentation with inference rules, we use the notation

$$\rho \vdash e \rightsquigarrow (\rho', v)$$

for the sentence *executing e in environment ρ terminates with the new environment ρ' and the value v* (and the coq proposition `exec` ρ e $\rho' v$). We rely on the same predicates `lookup` and `update` as in the course notes, here noted with judgments of the same form: $\rho \vdash x \rightarrow n$ for `lookup`, and $\rho \vdash x, n \mapsto \rho'$ for `update`.

$$\frac{}{\rho \vdash n \rightsquigarrow (\rho, n)}$$

$$\frac{\rho \vdash x \rightarrow n}{\rho\rho \vdash x \rightsquigarrow (\rho, n)}$$

$$\frac{\rho \vdash e_1 \rightsquigarrow \rho_1, v_1 \qquad \rho_1 \vdash e_2 \rightsquigarrow \rho_2, v_2}{\rho \vdash e_1{+}e_2 \rightsquigarrow (\rho_2, v_1 + v_2)}$$

$$\frac{\rho \vdash e_1 \rightsquigarrow \rho_1, v_1 \qquad \rho_1 \vdash e_2 \rightsquigarrow \rho_2, v_2}{\rho \vdash e_1 * e_2 \rightsquigarrow (\rho_2, v_1 \times v_2)}$$

$$\frac{\rho \vdash e \rightsquigarrow \rho_1, v \qquad \rho_1 \vdash x, v \mapsto \rho_2}{\rho \vdash x := e \rightsquigarrow (\rho_2, v)}$$

$$\frac{\rho \vdash e_1 \rightsquigarrow \rho_1, v \qquad v \leq 0}{\rho \vdash \texttt{while}\ e_1\ e_2 \rightsquigarrow (\rho_1, v)}$$

$$\frac{\rho \vdash e_1 \rightsquigarrow \rho_1, v_1 \quad 0 < v_1 \quad \rho_1 \vdash e_2 \rightsquigarrow (\rho_2, v_2) \qquad \rho_2 \vdash \texttt{while}\ e_1\ e_2 \rightsquigarrow (\rho_3, v_3)}{\rho \vdash \texttt{while}\ e_1\ e_2 \rightsquigarrow (\rho_3, v_3)}$$

$$\frac{\rho \vdash e_1 \rightsquigarrow \rho_1, v_1 \qquad \rho_1 \vdash e_2 \rightsquigarrow \rho_2, v_2}{\rho \vdash e_1 ; e_2 \rightsquigarrow (\rho_2, v_2)}$$

2 A proof that some programs don't terminate

Let's prove that the program `while 1 0` is non terminating.

Let D be a derivation for$\rho \vdash$ `while 1 0` $\rightsquigarrow (\rho', v)$ that is minimal in size for any ρ and ρ'. This derivation necessarily has exactly the following shape:

$$\frac{\dfrac{}{\rho \vdash 1 \rightsquigarrow (\rho, 1)} \quad 0 < 1 \quad \dfrac{}{\rho \vdash 0 \rightsquigarrow (\rho, 0)} \quad \dfrac{D'}{\rho \vdash \texttt{while 1 0} \rightsquigarrow (\rho', v')}}{\rho \vdash \texttt{while 1 0} \rightsquigarrow (\rho', v')}$$

Thus, D' is a derivation of the shape $\rho_0 \vdash$ `while 1 0` $\rightsquigarrow (\rho_1, v_1)$ and is smaller than D, this is contradictory with the definition of D. Here is the proof as it is written in Coq:

```
Definition loop_forever := while (num 1) (num 0).

Theorem loop_forever_prop :
  forall r e r' n, exec r e r' n -> e = loop_forever -> False.
unfold loop_forever.
intros r e r' n H; elim H; try (intros; discriminate).
(* case while-false *)
intros s s' e1 e2 n0 Hexec_e1 Hrec_e1 Hcomp Heq.
(* we have : Hexec_e1 : exec s e1 s' n0; Hcomp : n0 <= 0;
             Heq : while e1 e2 = while (num 1) (num 0) *)
injection Heq; intros; subst e1 e2.
(* we have : Hexec_e1 : exec s (num 1) s' n0 *)
inversion Hexec_e1; subst n0.
(* we have : Hcomp : 1 <= 0 *)
omega.
(* case while-true *)
intros s1 s2 s3 s4 e1 e2 n0 n' n'' Hexec_e1 Hrec_e1 Hexec_e2
  Hrec_e2 Hexec_w Hrec_w Heq.
(* we have : Hrec_w : while e1 e2 =
```

```
                    while (num 1) (num 0) -> False;
             Heq : while e1 e2 = while (num 1) (num 0) *)
auto.
Qed.
```

3 Extending with a conditional expression

We write the conditional expression as "e_1 ? e_2 : e_3", with the informal meaning: if e_1 is positive, compute e_2, otherwise compute e_3. In both cases accumulate the side effects of e_1 and the chosen e_i for the relevant $i \in \{2, 3\}$. To describe this construct we only need to add the following two rules:

$$\frac{\rho \vdash e_1 \rightsquigarrow \rho_1, v_1 \quad 0 < v_1 \quad \rho_1 \vdash e_2 \rightsquigarrow (\rho_2, v_2)}{\rho \vdash e_1 ? e_2 : e_3 \rightsquigarrow (\rho_2, v_2)}$$

$$\frac{\rho \vdash e_1 \rightsquigarrow \rho_1, v_1 \quad v_1 \leq 0 \quad \rho_1 \vdash e_3 \rightsquigarrow (\rho_2, v_2)}{\rho \vdash e_1 ? e_2 : e_3 \rightsquigarrow (\rho_2, v_2)}$$

In Coq, we prefer to define a new language and a new description of the `exec` predicate. The new definitions are suffixed with a prime character.

```
Inductive exp' : Set :=
  num' : Z -> exp'
| var' : Z -> exp'
| plus' : exp' -> exp' -> exp'
| mult' : exp' -> exp' -> exp'
| assign' : Z -> exp' -> exp'
| while' : exp' -> exp' -> exp'
| seq' : exp' -> exp' -> exp'
| if' : exp' -> exp' -> exp' -> exp'.

Inductive exec' : env -> exp' -> env -> Z -> Prop :=
  e_n' : forall n s, exec' s (num' n) s n
| e_v' : forall i n s, lookup s i n -> exec' s (var' i) s n
| e_p' : forall e1 e2 s s' s'' v1 v2,
        exec' s e1 s' v1 -> exec' s' e2 s'' v2 ->
        exec' s (plus' e1 e2) s'' (v1+v2)
| e_m' : forall e1 e2 s s' s'' v1 v2,
        exec' s e1 s' v1 -> exec' s' e2 s'' v2 ->
        exec' s (mult' e1 e2) s'' (v1*v2)
| e_a' : forall i e v s s' s'',
        exec' s e s' v -> update s' i v s'' ->
        exec' s (assign' i e) s'' v
| e_wf' : forall s s' e1 e2 n,
         exec' s e1 s' n -> n <= 0 ->
         exec' s (while' e1 e2) s' n
| e_wt' : forall s1 s2 s3 s4 e1 e2 n n' n'',
```

```
         exec' s1 e1 s2 n -> 0 < n -> exec' s2 e2 s3 n' ->
         exec' s3 (while' e1 e2) s4 n'' ->
         exec' s1 (while' e1 e2) s4 n''
| e_iff' : forall s1 s2 s3 e1 e2 e3 n n',
           exec' s1 e1 s2 n -> n <= 0 -> exec' s2 e3 s3 n' ->
           exec' s1 (if' e1 e2 e3) s3 n'
| e_ift' : forall s1 s2 s3 e1 e2 e3 n n',
           exec' s1 e1 s2 n -> 0 < n -> exec' s2 e2 s3 n' ->
           exec' s1 (if' e1 e2 e3) s3 n'
| e_s' : forall s1 s2 s3 e1 e2 n n',
        exec' s1 e1 s2 n -> exec' s2 e2 s3 n' ->
        exec' s1 (seq' e1 e2) s3 n'.
```