A tour of programming language semantics

Yves Bertot

August 2011

Describing a programming language

- One early applications of theorem proving tools
- Describe languages, guarantee properties of programs
- Need to know what the programming language means
- Define new tools for the programmers
- Explain what the tools really guarantee
- Application: defining a verified compiler

Several style of programming semantics

- Operational semantics
  - relation between inputs, programs, and outputs
  - Rely on inductive predicates
- Denotational semantics
  - Define the value of programs as functions
  - Rely on recursive programming
- Axiomatic semantics
  - Link properties satisfies by states before and after
  - Basis for verification condition generation
- Abstract interpretation
  - Automatic, static analysis of data properties

Writing an operational semantics

- Use a data-type to represent states
- Describe a relation between inputs, programs, and outputs
- Two different styles
  - Natural semantics, a.k.a. big step semantics, a three place relation,
    from the input, the program terminates with this output
  - Structural operational semantics, a.k.a. small step semantics,
    a four place relation,
    from this input and this program, an elementary step gives the
    new state and new program

Describing the syntax of a language

Require Import ZArith List String.
Open Scope Z_scope.
Open Scope string_scope.

Inductive aexpr : Type :=
  avar (s : string) | anum (n :Z) | aplus (a1 a2:aexpr).

Inductive bexpr : Type :=
  blt (a1 a2 : aexpr).

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

Evaluation of expressions

Definition env := list(string*Z).

Inductive aeval : env -> aexpr -> Z -> Prop :=
  a_int : forall r n, aeval (anum n) r n
  | a_var1 : forall r x v, aeval (avar (x,v)) r (avar x) v
  | a_var2 : forall r x y v v',
    x <> y -> aeval (avar x) r y v' ->
    aeval (y,v) r (avar x) y' v'
  | a_plus : forall r e1 e2 v1 v2,
    aeval r (aplus e1 e2) v1 v2 ->
    aeval r (aplus e1 e2) v1 + v2.
Evaluation and update

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

Remarks on natural semantics

▶ Programs may not terminate, or their execution may fail
▶ Their execution cannot be expressed with recursive function
▶ Description as an inductive relation is good work around
  but not executable
▶ Solution for execution limited in recursive calls

Partiality monad

- Use option types to describe partial functions
- Awkward to combine functions
- undefined propagates

Definition bind (A B:Type)(v:option A)(f:A->option B) :
  option B :=
match v with Some x => f x | None => None end.

Implicit Arguments bind.

Definition bind2 (A B C:Type)(v:option(A*B))
  (f: A->B->option C) :option C:=
match v with Some(a,b) => f a b | None => None end.

Implicit Arguments bind2.

Looking for the value of a variable

Fixpoint lookup (r:env)(name:string) : option Z :=
match r with
  nil => None
| (a,b)::tl =>
  if (string_dec a name) then Some b else lookup tl name end.

Evaluating expressions

Fixpoint af (r:env)(a:aexpr) : option Z :=
match a with
  avar index => lookup r index
| anum n => Some n
| aplus e1 e2 =>
  bind (af r e1)
  (fun v1 => bind (af r e2) (fun v2 => Some (v1+v2)))
end.
Structural operational semantics

Inductive sos_step : env->instr->instr->env->Prop :=
SOS1 : forall r r' x e v,
aeval r e v -> s_update r x v r' ->
sos_step r (assign x e) skip r'
| SOS2 : forall r i2, sos_step r (sequence skip i2) i2 r
| SOS3 : forall r r' i1 i1' i2, sos_step r i1 i1' r' ->
sos_step r (sequence i1 i2)(sequence i1' i2) r'
| SOS4 : forall r b i, beval r b true ->
sos_step r (while b i) (sequence i (while b i)) r
| SOS5 : forall r b i, beval r b false ->
sos_step r (while b i) skip r.

Consistency between the two semantics

- One can prove the following statement

\[ \forall r \ i \ r', \ sos\_star \ r \ i \ skip \ r' \iff \ exec \ r \ i \ r' \]

Executing sos

Fixpoint f_sos (r:env)(i:instr) : option (env*instr) :=
match i with
| skip => None
| assign x e =>
  bind (bind (af r e) (fun n => ufrxn ) )
  (fun r' => Some(r', skip))
| sequence i1 i2 =>
  if eq_skip i1 then Some(r, i2) else
  bind2 (f_sos r i1) (fun r' i' => Some(r', sequence i' i2))
| while b i =>
  bind (bf r b)
  (fun v => if v then Some(r, sequence i (while b i))
  else Some(r, skip))
end.

Executing SOS (continued)

Fixpoint f_star (n:nat) r i : option (env*instr) :=
match n with
| O => Some(r,i)
| Sn =>
  if eq_skip i then
  Some(r,i)
  else
  bind2 (f_sos r i) (fun r' i' => f_star n r' i')
end.

(* After many lines of proof. *)
Lemma f_star_exec : forall n r i r',
f_star n r i = Some(r',skip) -> exec r i r'.

Denotational semantics

- Use the Tarsky fixed point theorem
  \[ In every complete partial order, every continuous function has a least fixed point \]
- The Tarski fixed point theorem is provable in the logic of Coq
- But proving that partial functions have a cpo structure requires axioms from classical logic
- Used as a function to produce recursive function
  (with no termination guarantee)
- The function cannot be executed in Coq
  \[ \text{External executable code can be automatically produced} \]
  \[ \text{Internal simulation is possible (limiting number of recursive calls)} \]
While behavior: repeat test and body

Definition F:\_\_phi\
\( (A:\text{Type}) (t:A\rightarrow \text{option bool}) (f \ g :A\rightarrow \text{option } A) \) :=
\( \text{fun } r \Rightarrow \text{IF } (t \ r) \text{ THEN } (\text{bind } (f \ r) \ g) \text{ ELSE } (\text{Some } r). \)

Theorem F:\_\_phi\_continuous :
\( \forall (A : \text{Type}) \ t \ f, \) continuous \( (f\_\text{order } A \ A) (f\_\text{order } A \ A) \) \( (F:\_\_phi \ A \ t \ f). \)

Definition phi := fun A t f => Tarski\_fix \( (F:\_\_phi \ A \ t \ f). \)

The partial function for the full language

Fixpoint ds(i:instr) : env \rightarrow \text{option } env :=
\text{match } i \text{ with }
\text{assign } x e =>
\text{fun } l \Rightarrow \text{bind } (af l e) \text{ (fun } v \Rightarrow uf l x v) \\
\text{| sequence } i1 \ i2 => \text{fun } r \Rightarrow \text{bind } (ds i1 r) \ (ds i2) \\
\text{| while } e i => \text{fun } l \Rightarrow \phi (\text{fun } l' \Rightarrow \text{bf } l' e)(ds i) l \\
\text{| skip => fun } r \Rightarrow \text{Some } r \\
\text{end.} \\
Theorem ds\_eq\_sn :
\( \forall i \ l \ l', \ ds \ i \ l = \text{Some } l' \leftrightarrow \text{exec } l \ i \ l'. \)

Axiomatic semantics

▶ Relate properties instead of relating states
▶ Static analysis approach: Only one rule for each construct
▶ Relate logical reasoning with program constructs
▶ Requires a syntax for logical properties

Syntax for assertions and conditions

Inductive assert : Type :=
| a\_b(e: bexpr) | a\_not(a: assert) | a\_conj(a a': assert) |
| pred(s: string)(l: list aexpr).

Inductive condition : Type := c\_imp : (a1 a2 : assert).

Substitution

Fixpoint subst (e:aexpr)(s:string)(v:aexpr) : aexpr :=
\text{match } e \text{ with }
| avar s' => \text{if } \text{string\_dec } s \ s' \text{ then } v \text{ else } e \\
| anum n => anum n \\
| aplus e1 e2 => aplus (subst e1 s v) (subst e2 s v) \\
\text{end.}

Definition l\_subst (l:list aexpr) s v: list aexpr :=
\text{map } (\text{fun } x \Rightarrow \text{subst } x \ s \ v) \ l.

Fixpoint a\_subst (a:assert)(s:string)(v:aexpr) : assert :=
\text{match } a \text{ with }
| a\_b e => a\_b (b\_subst e s v) \\
| a\_not a => a\_not (a\_subst a s v) \\
| pred p l => pred p (l\_subst l s v) \\
| a\_conj a1 a2 => a\_conj(a\_subst a1 s v)(a\_subst a2 s v) \\
\text{end.}

Axiomatic semantics as an inductive definition

Inductive ax\_sem (pm : p\_env) :
| assert -> instr -> assert -> Prop :=
\text{ax1 : forall } P, \text{ ax\_sem pm } P \text{ skip } P \\
\text{| ax2 : forall } P x e, \text{ ax\_sem pm } (a\_subst P x e) \text{ assign } x e P \\
\text{| ax3 : forall } P Q R i i2, \text{ ax\_sem pm } P i1 Q \rightarrow \text{ax\_sem pm } Q i2 R \rightarrow \text{ax\_sem pm } P \text{ (sequence } i1 i2) R \\
\text{| ax4 : forall } P b i, \text{ ax\_sem pm } (a\_conj (a\_b b) P) i P \rightarrow \text{ax\_sem pm } P \text{ while } b i (a\_conj (a\_not (a\_b b)) P) \\
\text{| ax5 : forall } P P' Q' Q, \text{ valid pm } (c\_imp P P') \rightarrow \text{ax\_sem pm } P' \ i Q' \rightarrow \text{valid pm } (c\_imp Q' Q) \rightarrow \text{ax\_sem pm } P \ i Q.
Automatic axiomatic semantics

- Proofs of ax\_sem are very easy
- Except the consequence rule
  - Need to choose the new assertions
  - Need to prove the validity of implications
- A program can build the proof automatically
  - Let user annotate programs
  - Require invariants for loops
  - Leave the proofs of validity statements for later (verification conditions)

Verification Condition generation

Inductive a\_instr : Type :=
  prec(a:assert)(i:a\_instr) | a\_skip
  | a\_assign(x:string)(e:aexpr) | a\_sequence(i1 i2:a\_instr)
  | a\_while(b:bexpr)(a:assert)(i:a\_instr).

Fixpoint pc (i:a\_instr)(a:assert) : assert :=
  match i with
  prec a' i => a'
  | a\_skip => a
  | a\_assign x e => a\_subst a x e
  | a\_sequence i1 i2 => pc i1 (pc i2 a)
  | a\_while b a' i => a'
end.

Verification condition generation (continued)

Fixpoint vcg (i:a\_instr)(post : assert) : list condition :=
  match i with
  prec a i => c\_imp a (pc i post)::vcg i post
  | a\_skip => nil
  | a\_assign _ _ => nil
  | a\_sequence i1 i2 =>
    vcg i2 post ++ vcg i1 (pc i2 post)
  | a\_while e a i =>
    c\_imp (a\_conj (a\_not (a\_b e)) a) post ::
    c\_imp (a\_conj (a\_b e) a) (pc i a) :: vcg i a
end.

Correctness of the axiomatic semantics

- r\_g is a function that maps variables to their value in r or g

Theorem ax\_sem\_sound :
  forall mrir gPQ , exec r i r' -> ax\_sem m P i Q ->
  i\_a m (r\_g) P -> i\_a m (r'\_g) Q.

Theorem vcg\_sound :
  forall ps i A g r1 r2,
  valid_l ps (vcg i A) -> exec r1 (un\_annot i) r2 ->
  i\_a ps (r1\_g) (pc i A) -> i\_a ps (r2\_g) A.

Even more automation: static analysis

- Ingenuity is required to find correct annotations
- Static analysis infers properties of variable values in locations
- Abstract interpretation uses a systematic way to handle properties
- Best possible properties undecidable
  - Do over-approximation to ensure termination

Example with intervals

{x = 0}
While x < 10 do
  x := x + 1
end

- Problem: find an invariant for this loop
- (0,0), (0,1), (0,2) ... does not seem to converge
- (0, +\infty) is an over approximation
- Using knowledge from the test we know (0,9) before the assignment
- so (0,10) is an invariant!
Proved abstract interpretation

- Abstract interpreter written for this programming language
- Possible to change the kind of properties considered
- Behavior expressed by producing annotated programs
- Correctness statement expressed by using \texttt{vcg}
- Level of difficulty higher than other aspects of semantics seen so far

Availability

- For the first part
  \url{http://coq.inria.fr/pylons/contribs/view/Semantics/v8.3}
- For Abstract interpretation
  \url{http://hal.inria.fr/inria-00329572/}