Mechanized semantics, first lecture

Of expressions and commands: the semantics of an imperative language

Xavier Leroy
2019-11-28

Collège de France, chair of software sciences
Warming up: arithmetic expressions
A language of expressions comprising

- Integer constants 0, 1, -5, …, N
- Variables x, y, z, …
- Operations “plus” and “minus”: $e_1 + e_2$ et $e_1 - e_2$
  where $e_1$ and $e_2$ are sub-expressions.
Concrete syntax

The familiar algebraic notation, described by a BNF grammar:

\[
\begin{align*}
expr & ::= \text{term} \mid expr + \text{term} \mid expr - \text{term} \\
term & ::= \text{const} \mid \text{var} \mid ( expr ) \\
const & ::= -? \ [0 - 9]+ \\
var & ::= [a - z A - Z]+
\end{align*}
\]

Note: this grammar is not ambiguous: \(A+B-C\) is correctly read as \((A+B)-C\) and not as \(A+(B-C)\).
Abstract syntax trees

At leaves: constants and variables.

At nodes: operators $+$, $-$
Abstract syntax in research papers

A kind of grammar for abstract syntax trees:

Arithmetic expressions:

\[ a ::= x \quad \text{variables} \]
\[ N \quad \text{integer constants} \]
\[ a_1 + a_2 \quad \text{sum of two expressions} \]
\[ a_1 - a_2 \quad \text{difference of two expressions} \]

(No parentheses, no mention of precedence and associativity.)
The natural representation of abstract syntax trees in functional languages and proof assistants is an **inductive type**.

**In OCaml:**

```ocaml
type aexp =
    | CONST of int
    | VAR of string
    | PLUS of aexp * aexp
    | MINUS of aexp * aexp
```

**In Coq:**

```coq
Inductive aexp : Type :=
    | CONST (n: Z)
    | VAR (x: ident)
    | PLUS (a1: aexp) (a2: aexp)
    | MINUS (a1: aexp) (a2: aexp).
```
Abstract syntax trees as inductive types

Inductive aexp : Type :=
  | CONST (n: Z)
  | VAR (x: ident)
  | PLUS (a1: aexp) (a2: aexp)
  | MINUS (a1: aexp) (a2: aexp).

Defines 4 functions to construct values of type aexp:

CONST: Z -> aexp
VAR: ident -> aexp
PLUS: aexp -> aexp -> aexp
MINUS: aexp -> aexp -> aexp
Abstract syntax trees as inductive types

Inductive aexp : Type :=
 | CONST (n: Z)
 | VAR (x: ident)
 | PLUS (a1: aexp) (a2: aexp)
 | MINUS (a1: aexp) (a2: aexp).

Every value of type aexp is finitely generated by these 4 functions
⇒ case analysis + structural recursion

Fixpoint F (a: aexp) :=
 match a with
 | CONST n => ...
 | VAR x => ...
 | PLUS a1 a2 => ... F a1 ... F a2 ...
 | MINUS a1 a2 => ... F a1 ... F a2 ...
 end.
An arithmetic expression denotes a function values of variables $\rightarrow$ value of the expression.

The values of variables are given by a store (memory state) $s : \text{variable name} \rightarrow \text{variable value}$.

On paper, the denotational semantics is presented as a set of equations:

$$[[x]]s = s(x)$$
$$[[N]]s = N$$
$$[[a_1 + a_2]]s = [[a_1]]s + [[a_2]]s$$
$$[[a_1 - a_2]]s = [[a_1]]s - [[a_2]]s$$

(Note: $+$ and $-$ have different meanings on the left and on the right.)
Mechanizing this denotational semantics

On machine, this denotational semantics is presented as a recursive function defined by case analysis on the shape of the expression.

Definition store : Type := ident -> Z.

Fixpoint aeval (a: aexp) (s: store) : Z :=
match a with
| CONST n => n
| VAR x => s x
| PLUS a1 a2 => aeval a1 s + aeval a2 s
| MINUS a1 a2 => aeval a1 s - aeval a2 s
end.
Using this denotational semantics

As a pocket calculator (an interpreter for our language):

\[
\text{If } x \text{ is 10, then } 2 + x - 1 \text{ is 19.}
\]

To simplify expressions:

\[
\left[ x + (10 - 1) \right] s = s(x) + 9
\]

To prove algebraic properties of expressions:

\[
\left[ x + 1 \right] s > \left[ x \right] s \text{ for all } s
\]

To prove “meta” properties of the semantics:

\[
\text{If } s(x) = s'(x) \text{ for every } x \text{ free in } a, \text{ then } \left[ a \right] s = \left[ a \right] s'.
\]
Extensions and variants

Extending the language of expressions:

• with derived forms (e.g. \(-x \overset{\text{def}}{=} 0 - x\))
• with primitive forms (e.g. multiplication).

Modifying the semantics:

• Machine integers instead of mathematical integers \(\mathbb{Z}\).
• Reporting errors:
  overflows, division by 0, undefined variable, …
Modularizing denotational semantics using monads


\[
\begin{align*}
[N] &= \text{inj}(N) \\
[x] &= \text{get}(x) \\
[e_1 + e_2] &= \text{bind } [e_1] (\lambda v_1. \text{bind } [e_2] (\lambda v_2. v_1 \oplus v_2)) \\
[e_1 - e_2] &= \text{bind } [e_1] (\lambda v_1. \text{bind } [e_2] (\lambda v_2. v_1 \ominus v_2))
\end{align*}
\]

Parameterized by a reader monad \(M\) and an interpretation \(V\) of integer values:

\[
\begin{align*}
\text{ret} : \forall \alpha. \alpha &\rightarrow M \alpha & \text{inj} : \mathbb{Z} &\rightarrow M V \\
\text{bind} : \forall \alpha, \beta. M \alpha &\rightarrow (\alpha \rightarrow M \beta) \rightarrow M \beta & \cdot \oplus \cdot : V &\rightarrow V \rightarrow M V \\
\text{get} : \text{id} &\rightarrow M V & \cdot \ominus \cdot : V &\rightarrow V \rightarrow M V
\end{align*}
\]
Possible choices for $V$:

$V = \mathbb{Z}$ exact arithmetic
$V = [-2^{63}, 2^{63}]$ 64-bit signed machine arithmetic

Possible choices for $M$:

$M \alpha = (ident \to V) \to \alpha$ reader monad
$M \alpha = (ident \to \text{option } V) \to \text{option } \alpha$ reader and error monad

(See also the 2018-2019 lecture “Can we change the world? Imperative programming, monadic effects, algebraic effects”.)
The IMP language
and its reduction semantics
The language IMP

A minimalistic imperative language with structured control.

Arithmetic expressions:

\[ a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \]

Boolean expressions:

\[ b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \text{not } b \mid b_1 \text{ and } b_2 \]

Commands (statements):

\[ c ::= \text{skip} \quad \text{(do nothing)} \]
\[ \mid x ::= a \quad \text{(assignment)} \]
\[ \mid c_1; c_2 \quad \text{(sequence)} \]
\[ \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \quad \text{(conditional)} \]
\[ \mid \text{while } b \text{ do } c \quad \text{(loop)} \]
An IMP program

Euclidean division by repeated subtractions.

// entry: dividend in a, divisor in b

r := a;
q := 0;
while b <= r do
  r := r - b;
  q := q + 1
done

// exit: quotient in q, remainder in r
A routine denotational semantics, presented as a bool-valued function.

\[
\text{beval} : \text{bexp} \rightarrow \text{store} \rightarrow \text{bool}
\]

Many useful derived forms:

\[
a_1 \neq a_2 \quad a_1 < a_2 \quad a_1 \geq a_2 \quad a_1 > a_2 \quad a_1 \text{ or } a_2
\]
Denotational semantics of commands

Let’s attempt the naive denotational approach: the semantics of a command is a function \( \text{store “before” } \mapsto \text{store “after”} \).

\[
\begin{align*}
\sem{\text{skip}} s &= s \\
\sem{x := a} s &= s\{x \leftarrow \sem{a} s\} \\
\sem{c_1; c_2} s &= \sem{c_2}(\sem{c_1} s)
\end{align*}
\]

\[
\begin{align*}
\sem{\text{if } b \text{ then } c_1 \text{ else } c_2} s &= \begin{cases} 
\sem{c_1} s & \text{if } \sem{b} s = \text{true} \\
\sem{c_2} s & \text{if } \sem{b} s = \text{false}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\sem{\text{while } b \text{ do } c} s &= \begin{cases} 
\text{s} & \text{if } \sem{b} s = \text{false} \\
\sem{\text{while } b \text{ do } c}(\sem{c} s) & \text{if } \sem{b} s = \text{true}
\end{cases}
\end{align*}
\]
Denotational semantics of commands

\[
\text{[while } b \text{ do } c\text{]} \ s = \text{[while } b \text{ do } c\text{]} \ (\text{[c]} \ s) \quad \text{if } [b] \ s = \text{true}
\]

This equation is circular and fails to define the store “after” the execution of a \text{while} loop.

Besides, this store “after” is undefined if the loop doesn’t terminate! (as in \text{while true do } \text{skip})

The corresponding Coq function is rejected as not structurally recursive.
Could we change the type of the denotation function to $\text{com} \rightarrow \text{store} \rightarrow \text{option store}$, so that

$$\llbracket c \rrbracket s = \text{Some } s' \quad \text{means } c \text{ terminates with store } s'$$
$$\llbracket c \rrbracket s = \text{None} \quad \text{means } c \text{ diverges?}$$

In classical logic: yes.

In type theory (Coq, Agda, etc): no, because

- all definable functions are computable;
- the denotation function would decide the halting problem for IMP;
- IMP is Turing-complete.
Plan B: an operational semantics using sequences of reductions, in the style of lambda-calculus and its beta-reduction.

We reduce configurations $c/s$ comprising a command $c$ and the current store $s$:

$$c/s \rightarrow c'/s'$$

$c$: command \hspace{1cm} one step of \hspace{1cm} $c'$: residual command

$s$: initial store \hspace{1cm} computation \hspace{1cm} $s'$: updated store
Reduction rules

Assignments:

\[(x := a)/s \rightarrow \text{skip}/s\{x \leftarrow [a] s}\]

Sequences:

\[(c_1; c_2)/s \rightarrow (c'_1; c_2)/s' \quad \text{if} \quad c_1/s \rightarrow c'_1/s'\]
\[(\text{skip}; c_2)/s \rightarrow c_2/s\]

Example:

\[(x := 1; y := 2)/s \rightarrow (\text{skip}; y := 2)/s' \rightarrow (y := 2)/s' \rightarrow \text{skip}/s''\]

where \(s' = s\{x \leftarrow 1\}\) and \(s'' = s'\{y \leftarrow 2\}\).
Reduction rules

Conditional:

\[(\text{if } b \text{ then } c_1 \text{ else } c_2)/s \rightarrow c_1/s \quad \text{if } \llbracket b \rrbracket s = \text{true} \]
\[(\text{if } b \text{ then } c_1 \text{ else } c_2)/s \rightarrow c_2/s \quad \text{if } \llbracket b \rrbracket s = \text{false} \]

Loops:

\[(\text{while } b \text{ do } c)/s \rightarrow \text{skip}/s \quad \text{if } \llbracket b \rrbracket s = \text{false} \]
\[(\text{while } b \text{ do } c)/s \rightarrow (c; \text{while } b \text{ do } c)/s \quad \text{if } \llbracket s \rrbracket b = \text{true} \]
Reduction semantics as inference rules

\[(x := a)/s \rightarrow \text{skip}/s[x \leftarrow \llbracket a \rrbracket s]\]

\[c_1/s \rightarrow c'_1/s'\]

\[\frac{}{(c_1; c_2)/s \rightarrow (c'_1; c_2)/s'}\]

\[(\text{skip}; c)/s \rightarrow c/s\]

\[(c_1; c_2)/s \rightarrow (c'_1; c_2)/s'\]

\[(\text{if } b \text{ then } c_1 \text{ else } c_2)/s \rightarrow \begin{cases} c_1/s & \text{if } \llbracket b \rrbracket s = \text{true} \\ c_2/s & \text{if } \llbracket b \rrbracket s = \text{false} \end{cases}\]

\[\llbracket b \rrbracket s = \text{true}\]

\[(\text{while } b \text{ do } c)/s \rightarrow (c; \text{while } b \text{ do } c)/s\]

\[\llbracket b \rrbracket s = \text{false}\]

\[(\text{while } b \text{ do } c)/s \rightarrow \text{skip}/s\]
Writing inference rules in Coq

Step 1: write every rule as a standard logical formula.

\[ x := a/s \rightarrow \text{skip}/s[x \leftarrow \{a\} s] \]

\[ c_1/s \rightarrow c'_1/s' \]
\[ (c_1; c_2)/s \rightarrow (c'_1; c_2)/s' \]

\[
\begin{align*}
\text{forall } x, a, s, \\
\text{red} \ (\text{ASSIGN} \ x \ a, s) \ (\text{SKIP}, \ \text{update} \ x \ (\text{aeval} \ s \ a) \ s)
\end{align*}
\]

\[
\begin{align*}
\text{forall } c_1, c_2, s, c_1', s', \\
\text{red} \ (c_1, s) \ (c_1', s') \rightarrow \\
\text{red} \ (\text{SEQ} \ c_1 \ c_2, s) \ (\text{SEQ} \ c_1' \ c_2, s')
\end{align*}
\]

Step 2: give a name to each rule and turn it into a case of an inductive predicate.
Inductive red: com * store -> com * store -> Prop :=
| red_assign: forall x a s,
  red (ASSIGN x a, s) (SKIP, update x (aeval s a) s)
| red_seq_done: forall c s,
  red (SEQ SKIP c, s) (c, s)
| red_seq_step: forall c1 c s1 c2 s2,
  red (c1, s1) (c2, s2) ->
  red (SEQ c1 c, s1) (SEQ c2 c, s2)
| red_ifthenelse: forall b c1 c2 s,
  red (IFTHENELSE b c1 c2, s)
  ((if beval s b then c1 else c2), s)
| red_while_done: forall b c s,
  beval s b = false ->
  red (WHILE b c, s) (SKIP, s)
| red_while_loop: forall b c s,
  beval s b = true ->
  red (WHILE b c, s) (SEQ c (WHILE b c), s).
Using an inductive predicate

Each case of the definition is a theorem allowing us to conclude \( \text{red} (c, s) (c', s') \) for some choices of \( c, s, c', s' \).

Moreover, the proposition \( \text{red} (c, s) (c', s') \) holds only if it was proved by applying these theorems a finite number of times.

\[ \Rightarrow \text{reasoning principles: by induction on the derivation and case analysis on the last rule used.} \]

(To better understand the foundations of this approach, see the 2018-2019 lecture “Weapons of mass construction: inductive types, inductive predicates”.)
Reduction sequences

The behavior of a command $c$ is obtained by forming sequences of reductions starting with $c/s$.

- **Termination with final state $s'$**: finite sequence of reductions versa $skip/s'$.

  $$c/s \rightarrow c_1/s_1 \rightarrow \cdots \rightarrow skip/s'$$

- **Divergence**: infinite sequence of reductions

  $$c/s \rightarrow c_1/s_1 \rightarrow \cdots \rightarrow c_n/s_n \rightarrow \cdots$$

- **Run-time error**: finite sequence of reduction to an irreducible state other than $skip$ (never happens in IMP)

  $$c/s \rightarrow c_1/s_1 \rightarrow \cdots \rightarrow c'/s' \not\rightarrow c' \neq skip$$
Other kinds of operational semantics: natural semantics, definitional interpreters
Natural semantics

Another style of operational semantics, intermediate between reduction semantics and evaluation function.

Often called *big-step semantics*, as opposed to *small-step semantics*, which is another name for reduction semantics.
If the command $c; c'$ terminates, its reduction sequence has a very specific shape:

$$(c; c')/s \rightarrow (c_1; c')/s_1 \rightarrow \cdots \rightarrow (\text{skip}; c')/s_2 \rightarrow c'/s_2 \rightarrow \cdots \rightarrow \text{skip}/s_3$$

This sequence shows that $c$ terminates from $s$ on an intermediate store $s_2$, and that $c'$ terminates from $s_2$ on $s_3$

$$c/s \rightarrow c_1/s_1 \rightarrow \cdots \rightarrow \text{skip}/s_2$$
$$c'/s_2 \rightarrow \cdots \rightarrow \text{skip}/s_3$$
Intuitions of natural semantics

Idea: define a predicate $c/s \downarrow s'$ meaning “from initial store $s$, command $c$ terminates on final store $s'$”, using inference rules that capture this structure of terminating executions.

Example: we saw that $(c; c')$ started in $s$ terminates in $s'$ iff $c$ started in $s$ terminates in $s_2$ and $c'$ started in $s_2$ terminates in $s'$, for an intermediate store $s_2$. Hence the rule

\[
\begin{align*}
  c_1/s \downarrow s_2 \quad & c_2/s_2 \downarrow s' \\
  \hline
  c_1; c_2/s \downarrow s'
\end{align*}
\]
Rules for the natural semantics of IMP

\[
\begin{align*}
\text{skip/s} \downarrow s \\
c_1/s \downarrow s' \quad c_2/s' \downarrow s'' \\
\hline 
\text{c_1; c_2/s} \downarrow s''
\end{align*}
\]

\[
\begin{align*}
\text{x := a/s} & \downarrow s[x \leftarrow [a] s] \\
c_1/s \downarrow s' & \text{ if } [b] s = \text{true} \\
\text{c_2/s} & \downarrow s' \text{ if } [b] s = \text{false} \\
\text{if b then c_1 else c_2/s} & \downarrow s'
\end{align*}
\]

\[
\begin{align*}
[b] s = \text{false} \\
\text{while b do c/s} \downarrow s
\end{align*}
\]

\[
\begin{align*}
[b] s = \text{true} \quad c/s \downarrow s' \quad \text{while b do c/s'} \downarrow s'' \\
\text{while b do c/s} \downarrow s''
\end{align*}
\]
Equivalence with reduction semantics

A nice result:

\[ \frac{c}{s} \downarrow s' \quad \text{if and only if} \quad \frac{c}{s} \rightarrow^{*} \text{skip/s}' \]

We can therefore use one semantics or the other to reason over terminating execution, whichever is most convenient.

Natural semantics provides an induction principle (on derivations of \( \frac{c}{s} \downarrow s' \)) that is very convenient for compiler verification proofs (3rd lecture) and soundness proofs for program logics (5th lecture).
We were unable to define the semantics of a command as a function store “before” $\mapsto$ store “after” because this function would be partial (non-termination).

We can, however, define an approximation of this function by bounding a priori the recursion depth using a fuel parameter of type nat.

\[
\text{Fixpoint } \text{cexec}_f \ (\text{fuel: nat}) \ (s: \text{store}) \ (c: \text{com}) \\
\quad \ : \ \text{option } \text{store} \ := \\
\quad \quad \text{match } \text{fuel} \ \text{with} \\
\quad \quad \mid 0 \ => \ \text{None} \\
\quad \quad \mid \text{S fuel'} \ => \ \ldots \ \text{cexec}_f \ \text{fuel'} \ s' \ c' \ \ldots \\
\quad \ \text{end}.
\]
A definitional interpreter

Fixpoint cexec_f (fuel: nat) (s: store) (c: com) : option store := ...

A result Some s’ means c terminates on s’ definitely.
A result None is not conclusive: either c diverges, either we need more fuel to finish the execution of c.
Very useful to test the semantics on sample programs.
Summary
The IMP language = expressions + imperative commands.

Semantics: naive denotational, operational (by reductions, or natural, or by bounded interpreter).

Coq formalization: inductive types, recursive functions, inductive predicates.

First proofs: equivalences between various semantics.