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

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.

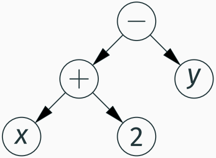
Concrete syntax

The familiar algebraic notation, described by a BNF grammar:

$$\begin{aligned} expr &::= term \mid expr \; \texttt{+} \; term \mid expr \; \texttt{-} \; term \\ term &::= const \mid var \mid \texttt{(} \; expr \; \texttt{)} \\ const &::= \texttt{-}? \; [\texttt{0} - \texttt{9}]+ \\ var &::= [\texttt{a} - \texttt{z} \; \texttt{A} - \texttt{Z}]+ \end{aligned}$$

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



$$x + 2 - y$$
$$(x + 2) - y$$
$$x + 2 - (y)$$

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	variables
	$\mid N$	integer constants
	$\mid a_1 + a_2$	sum of two expressions
	$\mid a_1 - a_2$	difference of two expressions

(No parentheses, no mention of precedence and associativity.)

Abstract syntax trees as inductive types

The natural representation of abstract syntax trees in functional languages and proof assistants is an inductive type.

In OCaml:

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

In 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
$\Rightarrow$ 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.
```

Denotational semantics of expressions

An arithmetic expression denotes a function values of variables $\to$ value of the expression.

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

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

$$[\![x]\!]\ s = s(x)$$
$$[\![N]\!]\ s = N$$
$$[\![a_1 \texttt{ + } a_2]\!]\ s = [\![a_1]\!]\ s + [\![a_2]\!]\ s$$
$$[\![a_1 \texttt{ - } a_2]\!]\ s = [\![a_1]\!]\ s - [\![a_2]\!]\ s$$

(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):

If `x` *is 10, then* `2 + x - 1` *is 19.*

To simplify expressions:

$[\![\mathtt{x} + (\mathtt{10} - \mathtt{1})]\!]\, s = s(x) + 9$

To prove algebraic properties of expressions:

$[\![\mathtt{x} + \mathtt{1}]\!]\, s > [\![\mathtt{x}]\!]\, s$ *for all* s

To prove "meta" properties of the semantics:

If $s(x) = s'(x)$ *for every* x *free in* a*, then* $[\![a]\!]\, s = [\![a]\!]\, s'$.

Extensions and variants

Extending the language of expressions:

- with derived forms (e.g. $-x \stackrel{def}{=} 0 - x$)
- 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

(Eugenio Moggi, *Notions of computations and monads*, 1989, 1991)

$$
\begin{aligned}
[\![N]\!] &= \texttt{inj}(N) \\
[\![x]\!] &= \texttt{get}(x) \\
[\![e_1 + e_2]\!] &= \texttt{bind}\ [\![e_1]\!]\ (\lambda v_1.\ \texttt{bind}\ [\![e_2]\!]\ (\lambda v_2.\ v_1 \oplus v_2)) \\
[\![e_1 - e_2]\!] &= \texttt{bind}\ [\![e_1]\!]\ (\lambda v_1.\ \texttt{bind}\ [\![e_2]\!]\ (\lambda v_2.\ v_1 \ominus v_2))
\end{aligned}
$$

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

$$
\begin{aligned}
\texttt{ret} &: \forall \alpha.\ \alpha \to M\ \alpha & \texttt{inj} &: \mathbb{Z} \to M\ V \\
\texttt{bind} &: \forall \alpha, \beta.\ M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta & \cdot \oplus \cdot &: V \to V \to M\ V \\
\texttt{get} &: \mathit{ident} \to M\ V & \cdot \ominus \cdot &: V \to V \to M\ V
\end{aligned}
$$

Modularizing denotational semantics using monads

Possible choices for V:

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

Possible choices for M:

$M\ \alpha = (\mathit{ident} \to V) \to \alpha$	reader monad
$M\ \alpha = (\mathit{ident} \to \texttt{option}\ V) \to \texttt{option}\ \alpha$	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:

$$b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \texttt{not}\ b \mid b_1\ \texttt{and}\ b_2$$

Commands (*statements*):

$$\begin{array}{lll} c & ::= \texttt{skip} & \text{(do nothing)} \\ & \mid x := a & \text{(assignment)} \\ & \mid c_1; c_2 & \text{(sequence)} \\ & \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 & \text{(conditional)} \\ & \mid \texttt{while}\ b\ \texttt{do}\ c & \text{(loop)} \end{array}$$

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
```

Denotational semantics of Boolean expressions

A routine denotational semantics, presented as a `bool`-valued function.

$$\texttt{beval} : \texttt{bexp} \rightarrow \texttt{store} \rightarrow \texttt{bool}$$

Many useful derived forms:

$$a_1 \neq a_2 \qquad a_1 < a_2 \qquad a_1 \geq a_2 \qquad a_1 > a_2 \qquad a_1 \ \texttt{or} \ a_2$$

Denotational semantics of commands

Let's attempt the naive denotational approach: the semantics of a command is a function store "before" $\mapsto$ store "after".

$$
\begin{aligned}
[\![\texttt{skip}]\!]\, s &= s \\
[\![x := a]\!]\, s &= s\{x \leftarrow [\![a]\!]\, s\} \\
[\![c_1; c_2]\!]\, s &= [\![c_2]\!]\, ([\![c_1]\!]\, s) \\
[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\, s &= \begin{cases} [\![c_1]\!]\, s & \text{if } [\![b]\!]\, s = \texttt{true} \\ [\![c_2]\!]\, s & \text{if } [\![b]\!]\, s = \texttt{false} \end{cases} \\
[\![\texttt{while}\ b\ \texttt{do}\ c]\!]\, s &= \begin{cases} s & \text{if } [\![b]\!]\, s = \texttt{false} \\ [\![\texttt{while}\ b\ \texttt{do}\ c]\!]\, ([\![c]\!]\, s) & \text{if } [\![b]\!]\, s = \texttt{true} \end{cases}
\end{aligned}
$$

Denotational semantics of commands

$$[\![\texttt{while}\ b\ \texttt{do}\ c]\!]\ s = [\![\texttt{while}\ b\ \texttt{do}\ c]\!]\ ([\![c]\!]\ s) \qquad \text{if } [\![b]\!]\ s = \texttt{true}$$

This equation is circular and fails to define the store "after" the execution of a `while` loop.

Besides, this store "after" is undefined if the loop doesn't terminate! (as in `while true do skip`)

The corresponding Coq function is rejected as not structurally recursive.

Denotational semantics of commands

Could we change the type of the denotation function to
$\texttt{com} \to \texttt{store} \to \texttt{option store}$, so that

$$\begin{array}{ll} [\![c]\!]\ s = \texttt{Some}\ s' & \text{means } c \text{ terminates with store } s' \\ [\![c]\!]\ s = \texttt{None} & \text{means } c \text{ diverges?} \end{array}$$

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-complet.

Reduction semantics of commands

Plan B: an operational semantics using sequences of reductions, in the style of lambda-calcul and its beta-reduction.

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

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

c: command s: initial store	one step of computation	c': residual command s': updated store

Reduction rules

Assignments:

$$(x := a)/s \to \texttt{skip}/s\{x \leftarrow [\![a]\!]\ s\}$$

Sequences:

$$(c_1; c_2)/s \to (c_1'; c_2)/s' \quad \text{si } c_1/s \to c_1'/s'$$
$$(\texttt{skip}; c_2)/s \to c_2/s$$

Example:

$$(x := 1; y := 2)/s \to (\texttt{skip}; y := 2)/s' \to (y := 2)/s' \to \texttt{skip}/s''$$

where $s' = s\{x \leftarrow 1\}$ and $s'' = s'\{y \leftarrow 2\}$.

Reduction rules

Conditional:

$$(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)/s \to c_1/s \qquad \text{if } [\![b]\!]\ s = \texttt{true}$$

$$(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)/s \to c_2/s \qquad \text{if } [\![b]\!]\ s = \texttt{false}$$

Loops:

$$(\texttt{while}\ b\ \texttt{do}\ c)/s \to \texttt{skip}/s \qquad \text{if } [\![b]\!]\ s = \texttt{false}$$

$$(\texttt{while}\ b\ \texttt{do}\ c)/s \to (c; \texttt{while}\ b\ \texttt{do}\ c)/s \quad \text{if } [\![s]\!]\ b = \texttt{true}$$

Reduction semantics as inference rules

$$(x := a)/s \to \texttt{skip}/s[x \leftarrow [\![a]\!]\ s])$$

$$\frac{c_1/s \to c_1'/s'}{(c_1; c_2)/s \to (c_1'; c_2)/s'} \qquad (\texttt{skip}; c)/s \to c/s$$

$$(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)/s \to \begin{cases} c_1/s & \text{if } [\![b]\!]\ s = \texttt{true} \\ c_2/s & \text{if } [\![b]\!]\ s = \texttt{false} \end{cases}$$

$$\frac{[\![b]\!]\ s = \texttt{true}}{(\texttt{while}\ b\ \texttt{do}\ c)/s \to (c; \texttt{while}\ b\ \texttt{do}\ c)/s}$$

$$\frac{[\![b]\!]\ s = \texttt{false}}{(\texttt{while}\ b\ \texttt{do}\ c)/s \to \texttt{skip}/s}$$

Writing inference rules in Coq

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

$$x := a/s \rightarrow \texttt{skip}/s[x \leftarrow [\![a]\!]\ s])$$

$$\frac{c_1/s \rightarrow c_1'/s'}{(c_1; c_2)/s \rightarrow (c_1'; c_2)/s'}$$

```
forall x a s,
    red (ASSIGN x a, s) (SKIP, update x (aeval s a) s)

forall c1 c2 s c1’ s’,
    red (c1, s) (c1’, s’) ->
    red (SEQ c1 c2, s) (SEQ c1’ c2, s’)
```

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 $\texttt{red}\ (c, s)\ (c', s')$ for some choices of c, s, c', s'.

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

$\Rightarrow$ 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 vers $\texttt{skip}/s'$.

$$c/s \to c_1/s_1 \to \cdots \to \texttt{skip}/s'$$

- Divergence: infinite sequence of reductions

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

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

$$c/s \to c_1/s_1 \to \cdots \to c'/s' \not\to \qquad c' \neq \texttt{skip}$$

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.

Intuitions of natural semantics

If the command $c; c'$ terminates, its reduction sequence has a very specific shape:

$$
\begin{aligned}
(c; c')/s \to (c_1; c')/s_1 \to \cdots \to (\texttt{skip}; c')/s_2 \\
\to c'/s_2 \to \cdots \to \texttt{skip}/s_3
\end{aligned}
$$

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

$$
\begin{aligned}
c/s \to c_1/s_1 \to \cdots \to \texttt{skip}/s_2 \\
c'/s_2 \to \cdots \to \texttt{skip}/s_3
\end{aligned}
$$

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

$$\frac{c_1/s \Downarrow s_2 \qquad c_2/s_2 \Downarrow s'}{c_1; c_2/s \Downarrow s'}$$

Rules for the natural semantics of IMP

$$\texttt{skip}/s \Downarrow s \qquad\qquad x := a/s \Downarrow s[x \leftarrow [\![a]\!]\, s]$$

$$\frac{c_1/s \Downarrow s' \qquad c_2/s' \Downarrow s''}{c_1; c_2/s \Downarrow s''} \qquad\qquad \frac{\begin{array}{c} c_1/s \Downarrow s' \text{ if } [\![b]\!]\, s = \texttt{true} \\ c_2/s \Downarrow s' \text{ if } [\![b]\!]\, s = \texttt{false} \end{array}}{\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2/s \Downarrow s'}$$

$$\frac{[\![b]\!]\, s = \texttt{false}}{\texttt{while } b \texttt{ do } c/s \Downarrow s}$$

$$\frac{[\![b]\!]\, s = \texttt{true} \qquad c/s \Downarrow s' \qquad \texttt{while } b \texttt{ do } c/s' \Downarrow s''}{\texttt{while } b \texttt{ do } c/s \Downarrow s''}$$

Equivalence with reduction semantics

A nice result:

$$c/s \Downarrow s' \quad \text{if and only if} \quad c/s \xrightarrow{*} \texttt{skip}/s'$$

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 $c/s \Downarrow s'$) that is very convenient for compiler verification proofs (3rd lecture) and soundness proofs for program logics (5th lecture).

A definitional interpreter

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`.

```
Fixpoint cexec_f (fuel: nat) (s: store) (c: com)
                                              : option store :=
  match fuel with
  | O => None
  | S fuel' => ... cexec_f fuel' s' c' ...
  end.
```

```
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

Summary so far

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.