CS152: Programming Languages
Lecture 3 — Operational Semantics

Dan Grossman
Spring 2011

Where we are

- Done: Caml basics, "IMP" syntax, structural induction
- Now: Operational semantics for our little "IMP" language
  - Most of what you need for Homework 1
  - (But Problem 4 requires proofs over semantics)

Outline

- Semantics for expressions
  1. Informal idea; the need for heaps
  2. Definition of heaps
  3. The evaluation judgment (a relation form)
  4. The evaluation inference rules (the relation definition)
  5. Using inference rules
    - Derivation trees as interpreters
    - Or as proofs about expressions
  6. Metatheory: Proofs about the semantics
- Then semantics for statements
- ...

Review

IMP’s abstract syntax is defined inductively:

\[
\begin{align*}
  s & ::= \text{skip} \mid x := e \mid s; s \mid \text{if } e \text{ s s} \mid \text{while } e \text{ s} \\
  e & ::= c \mid x \mid e + e \mid e * e \\
  c & \in \{ \ldots, -2, -1, 0, 1, 2, \ldots \} \\
  x & \in \{ x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots \} 
\end{align*}
\]

We haven’t yet said what programs mean! (Syntax is boring)

Encode our “social understanding” about variables and control flow

Informal idea

Given \( e \), what \( c \) does it evaluate to?

\[
1 + 2 \quad x + 2
\]

It depends on the values of variables (of course)

Use a heap \( H \) for a total function from variables to constants
  - Could use partial functions, but then \( \exists H \) and \( e \) for which there is no \( c \)

We’ll define a relation over triples of \( H, e, \) and \( c \)
  - Will turn out to be function if we view \( H \) and \( e \) as inputs and \( c \) as output
  - With our metalanguage, easier to define a relation and then prove it is a function (if, in fact, it is)

Heaps

\[
H ::= \cdot \mid H, x \mapsto c
\]

A lookup-function for heaps:

\[
H(x) = \begin{cases} 
  c & \text{if } H = H', x \mapsto c \\
  H'(x) & \text{if } H = H', y \mapsto c' \text{ and } y \neq x \\
  0 & \text{if } H = \cdot 
\end{cases}
\]

- Last case avoids “errors” (makes function total)

“What heap to use” will arise in the semantics of statements
  - For expression evaluation, “we are given an \( H \)”
The judgment

We will write: \( H ; e \Downarrow c \)

to mean, "\( e \) evaluates to \( c \) under heap \( H \)"

It is just a relation on triples of the form \((H, e, c)\)

We just made up metasyntax \( H ; e \Downarrow c \) to follow PL convention
and to distinguish it from other relations

We can write: \(. , x \mapsto \rightarrow 3 ; x + y \Downarrow 3\), which will turn out to be true
(this triple will be in the relation we define)

Or: \(. , x \mapsto \rightarrow 3 ; x + y \Downarrow 6\), which will turn out to be false
(this triple will not be in the relation we define)

Inference rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>( H ; c \Downarrow c )</td>
</tr>
<tr>
<td>var</td>
<td>( H ; x \Downarrow H(x) )</td>
</tr>
<tr>
<td>add</td>
<td>( H ; e_1 \Downarrow c_1 ) ( H ; e_2 \Downarrow c_2 ) ( H ; e_1 + e_2 \Downarrow c_1 + c_2 )</td>
</tr>
<tr>
<td>mult</td>
<td>( H ; e_1 \Downarrow c_1 ) ( H ; e_2 \Downarrow c_2 ) ( H ; e_1 \ast e_2 \Downarrow c_1 \ast c_2 )</td>
</tr>
</tbody>
</table>

Top: hypotheses
Bottom: conclusion (read first)

By definition, if all hypotheses hold, then the conclusion holds

Each rule is a schema you “instantiate consistently”

- So rules “work” “for all” \( H, c, e_1 \), etc.
- But “each” \( e_1 \) has to be the “same” expression

Derivations

A (complete) derivation is a tree of instantiations with axioms at the leaves

Example:

\( , y \mapsto \rightarrow 4 \) \( 3 + y \Downarrow 7 \) \( , y \mapsto \rightarrow 4 \) \( 5 \Downarrow 5 \)
(\( y \mapsto \rightarrow 4 \) \( 3 + y \Downarrow 7 \) \( , y \mapsto \rightarrow 4 \) \( 5 \Downarrow 5 \))

By definition, \( H ; e \Downarrow c \) if there exists a derivation with \( H ; e \Downarrow c \) at the root

What are these things?

We can view the inference rules as defining an interpreter

- Complete derivation shows recursive calls to the “evaluate expression” function
  - Recursive calls from conclusion to hypotheses
  - Syntax-directed means the interpreter need not “search”

- See OCaml code in Homework 1

Or we can view the inference rules as defining a proof system

- Complete derivation proves facts from other facts starting with axioms
  - Facts established from hypotheses to conclusions

Back to relations

So what relation do our inference rules define?

- Start with empty relation (no triples) \( R_0 \)
- Let \( R_i \) be \( R_{i-1} \) union all \( H ; e \Downarrow c \) such that we can instantiate some inference rule to have conclusion \( H ; e \Downarrow c \) and all hypotheses in \( R_{i-1} \)
  - So \( R_i \) is all triples at the bottom of height-\( j \) complete derivations for \( j \leq i \)
- \( R_\infty \) is the relation we defined
  - All triples at the bottom of complete derivations

For the math folks: \( R_\infty \) is the smallest relation closed under the inference rules

Instantiating rules

Example instantiation:

\( , y \mapsto \rightarrow 4 \) \( 3 + y \Downarrow 7 \) \( , y \mapsto \rightarrow 4 \) \( 5 \Downarrow 5 \)
(\( y \mapsto \rightarrow 4 \) \( 3 + y \Downarrow 7 \) \( , y \mapsto \rightarrow 4 \) \( 5 \Downarrow 5 \))

Instantiates:

\( \text{add} \)

\( H ; e_1 \Downarrow c_1 \) \( H ; e_2 \Downarrow c_2 \)

\( H ; e_1 + e_2 \Downarrow c_1 + c_2 \)

with

\( H = , y \mapsto \rightarrow 4 \)
\( e_1 = (3 + y) \)
\( c_1 = 7 \)
\( e_2 = 5 \)
\( c_2 = 5 \)
Some theorems

- Progress: For all $H$ and $e$, there exists a $c$ such that $H ; e \Downarrow c$.
- Determinacy: For all $H$ and $e$, there is at most one $c$ such that $H ; e \Downarrow c$.

We rigged it that way... what would division, undefined-variables, or `gettime()` do?

Proofs are by induction on the the structure (i.e., height) of the expression $e$.

On to statements

A statement doesn’t produce a constant.

It produces a new, possibly-different heap.
- If it terminates

We could define $H_1 ; s \Downarrow H_2$
- Would be a partial function from $H_1$ and $s$ to $H_2$
- Works fine; could be a homework problem

Instead we’ll define a “small-step” semantics and then “iterate” to “run the program”

\[
H_1 ; s_1 \rightarrow H_2 ; s_2
\]

Statement semantics

\[
H_1 ; s_1 \rightarrow H_2 ; s_2
\]

ASSIGN

\[
H ; e \Downarrow c \quad \xrightarrow{\text{assign}} \quad H ; x := e \rightarrow H ; x \rightarrow c ; \text{skip}
\]

SEQ1

\[
H ; \text{skip}; s \rightarrow H ; s
\]

SEQ2

\[
H ; s_1 \rightarrow H' ; s_1' \quad \xrightarrow{\text{seq1}} \quad H ; s_1 ; s_2 \rightarrow H' ; s_1' ; s_2
\]

IF1

\[
H ; e \Downarrow c \quad c > 0 \quad \xrightarrow{\text{if1}} \quad H ; \text{if } e \rightarrow s_1 \rightarrow H ; s_1
\]

IF2

\[
H ; e \Downarrow c \quad c \leq 0 \quad \xrightarrow{\text{if2}} \quad H ; e \rightarrow s_1 \rightarrow s_2 \rightarrow H ; s_2
\]

Example program execution

\[
x := 3 ; (y := 1 ; \text{while } x (y := y * x ; x := x - 1))
\]

Let’s write some of the state sequence. You can justify each step with a full derivation. Let $s = (y := y * x ; x := x - 1)$.

\[
\begin{align*}
\vdots \quad x &:= 3 ; y := 1 ; \text{while } x (y := y * x ; x := x - 1) \\
\rightarrow \quad x &\mapsto 3 ; \text{skip}; y := 1 ; \text{while } x s \\
\rightarrow \quad x &\mapsto 3 ; y := 1 ; \text{while } x s \\
\rightarrow^2 \quad x &\mapsto 3 , y \mapsto 1 ; \text{while } x s \\
\rightarrow \quad x &\mapsto 3 , y \mapsto 1 ; \text{if } x (s ; \text{while } x s) \text{ skip} \\
\rightarrow \quad x &\mapsto 3 , y \mapsto 1 ; y := y * x ; x := x - 1 ; \text{while } x s
\end{align*}
\]

Program semantics

\[
\begin{align*}
\quad \text{Defined } \quad H ; s \rightarrow H' ; s', \text{ but what does } s \text{ mean/do?} \\
\quad \text{Our machine iterates: } H_1 ; s_1 \rightarrow H_2 ; s_2 \rightarrow H_3 ; s_3 \ldots , \text{ with each step justified by a complete derivation using our single-step statement semantics} \\
\quad \text{Let } H_1 ; s_1 \rightarrow^n H_2 ; s_2 \text{ mean “becomes after } n \text{ steps”} \\
\quad \text{Let } H_1 ; s_1 \rightarrow^* H_2 ; s_2 \text{ mean “becomes after 0 or more steps”} \\
\quad \text{Pick a special “answer” variable } \text{ans} \\
\quad \text{The program } s \text{ produces } c \text{ if } s \rightarrow^* H ; \text{skip and } H(\text{ans}) = c \\
\quad \text{Does every } s \text{ produce a } c?
\end{align*}
\]
Where we are

- Defined $H; e$ and $H; s \rightarrow H'; s'$ and extended the latter to give $s$ a meaning
  - The way we did expressions is "large-step operational semantics"
  - The way we did statements is "small-step operational semantics"
  - So now you have seen both

Definition by interpretation: program means what an interpreter (written in a metalanguage) says it means
  - Interpreter represents a (very) abstract machine that runs code

Large-step does not distinguish errors and divergence
  - But we defined IMP to have no errors
  - And expressions never diverge

Establishing Properties

We can prove a property of a terminating program by "running" it.

Example: Our last program terminates with $x$ holding 0.

We can prove a program diverges, i.e., for all $H$ and $n$, $s \rightarrow^n H; \text{skip}$ cannot be derived.

Example: while 1 skip

By induction on $n$, but requires a stronger induction hypothesis.

More General Proofs

We can prove properties of executing all programs (satisfying another property)

Example: If $H$ and $s$ have no negative constants and $H; s \rightarrow^* H'; s'$, then $H'$ and $s'$ have no negative constants.

Example: If for all $H$, we know $s_1$ and $s_2$ terminate, then for all $H$, we know $H;(s_1; s_2)$ terminates.