CSE 505: Concepts of Programming Languages

Dan Grossman
Fall 2009
Lecture 3—Operational Semantics for IMP
Where we are

- Done: Caml basics, IMP syntax, structural induction
- Today: IMP operational semantics
- Tonight: You could (almost?) finish homework 1
IMP’s abstract syntax is defined inductively:

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

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

Encode our “social understanding” about variables and control flow
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
  - ...
Informal idea

Given $e$, what $c$ does it evaluate to?

It depends on the values of variables (of course).

Use a heap $H$ to encode 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 it is).
Heaps

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

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

Last case avoids “errors” (makes function total)

“What heap to use” will arise in the statement semantics

- For expression evaluation, “we are given an \( H \)”
The judgment

We will write: \[ \text{H} \; ; \; e \downarrow c \]
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 \( \text{H} \; ; \; e \downarrow c \) to follow PL convention and
to distinguish it from other relations.

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

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

**CONST**

\[
H ; c \Downarrow c
\]

**VAR**

\[
H ; x \Downarrow H(x)
\]

**ADD**

\[
\begin{align*}
H ; e_1 \Downarrow c_1 & \quad & H ; e_2 \Downarrow c_2 \\
H ; e_1 + e_2 \Downarrow c_1 + c_2
\end{align*}
\]

**MULT**

\[
\begin{align*}
H ; e_1 \Downarrow c_1 & \quad & H ; e_2 \Downarrow c_2 \\
H ; e_1 \ast e_2 \Downarrow c_1 \ast c_2
\end{align*}
\]

---

**Bottom:** conclusion

**Top:** hypotheses

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.
Instantiating rules

Example instantiation:

\[
\begin{align*}
\cdot, y \mapsto 4 ; 3 + y \downarrow 7 & \quad \cdot, y \mapsto 4 ; 5 \downarrow 5 \\
\cdot, y \mapsto 4 ; (3 + y) + 5 \downarrow 12
\end{align*}
\]

Instantiates:

\[
\begin{align*}
H ; e_1 \downarrow c_1 & \quad H ; e_2 \downarrow c_2 \\
H ; e_1 + e_2 \downarrow c_1 + c_2
\end{align*}
\]

with \( H = \cdot, y \mapsto 4, e_1 = (3 + y), c_1 = 7, e_2 = 5, c_2 = 5 \)
Derivations

A \textit{(complete) derivation} is a tree of instantiations with \textit{axioms} at the leaves.

Example:

\begin{align*}
\cdot, y \mapsto 4 ; 3 \Downarrow 3 & \quad \cdot, y \mapsto 4 ; y \Downarrow 4 \\
\cdot, y \mapsto 4 ; 3 + y \Downarrow 7 & \quad \cdot, y \mapsto 4 ; 5 \Downarrow 5 \\
\cdot, y \mapsto 4 ; (3 + y) + 5 \Downarrow 12
\end{align*}

So \( H ; e \Downarrow c \) if there exists a derivation with \( H ; e \Downarrow c \) at the root.
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.
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.
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?

Note: Our semantics is syntax-directed.

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**

\[
H ; e \downarrow c \\
\rightarrow \\
H ; x := e \rightarrow H, x \mapsto c ; \text{skip}
\]

**SEQ1**

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

**SEQ2**

\[
H ; s_1 \rightarrow H' ; s_1' \\
\rightarrow \\
H ; s_1 ; s_2 \rightarrow H' ; s_1' ; s_2
\]

**IF1**

\[
H ; e \downarrow c \quad c > 0 \\
\rightarrow \\
H ; \text{if } e s_1 s_2 \rightarrow H ; s_1
\]

**IF2**

\[
H ; e \downarrow c \quad c \leq 0 \\
\rightarrow \\
H ; \text{if } e s_1 s_2 \rightarrow H ; s_2
\]
Statement semantics cont’d

What about \textbf{while} \( e \ s \) (do \( s \) and loop if \( e > 0 \))?

\begin{align*}
\text{WHILE} \\
H ; \text{while } e \ s \rightarrow H ; \text{if } e \ (s; \text{while } e \ s) \text{ skip}
\end{align*}

Many other equivalent definitions possible
Program semantics

We defined $H; s \rightarrow H'; s'$, but what does “s” mean/do?

Our machine iterates: $H_1; s_1 \rightarrow H_2; s_2 \rightarrow H_3; s_3 \ldots$,

with each step justified by a complete derivation using our single-step statement semantics

Let $H_1; s_1 \rightarrow^* H_2; s_2$ mean “becomes after 0 or more steps” and pick a special “answer” variable $ans$

The program $s$ produces $c$ if $\cdot; s \rightarrow^* H; \text{skip}$ and $H(ans) = c$

Does every $s$ produce a $c$?
Example program execution

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

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

\[
\begin{align*}
\cdot; x := 3; y := 1; \textbf{while} \ x \ s \\
\rightarrow \ & \cdot \cdot; x \leftrightarrow 3; \ \textbf{skip}; y := 1; \textbf{while} \ x \ s \\
\rightarrow \ & \cdot \cdot; x \leftrightarrow 3; y := 1; \textbf{while} \ x \ s \\
\rightarrow^2 \ & \cdot \cdot; x \leftrightarrow 3, y \leftrightarrow 1; \ \textbf{while} \ x \ s \\
\rightarrow \ & \cdot \cdot; x \leftrightarrow 3, y \leftrightarrow 1; \ \textbf{if} \ x \ (s; \textbf{while} \ x \ s) \ \textbf{skip} \\
\rightarrow \ & \cdot \cdot; x \leftrightarrow 3, y \leftrightarrow 1; y := y \times x; x := x - 1; \textbf{while} \ x \ s
\end{align*}
\]
Continued...

\[ \rightarrow^2 \cdot, x \leftarrow 3, y \leftarrow 1, y \leftarrow 3; \ x := x-1; \textbf{while} \ x \ s \]

\[ \rightarrow^2 \cdot, x \leftarrow 3, y \leftarrow 1, y \leftarrow 3, x \leftarrow 2; \textbf{while} \ x \ s \]

\[ \rightarrow \ldots, y \leftarrow 3, x \leftarrow 2; \textbf{if} \ x \ (s; \textbf{while} \ x \ s) \ \textbf{skip} \]

\[ \ldots \]

\[ \rightarrow \ldots, y \leftarrow 6, x \leftarrow 0; \textbf{skip} \]
Where we are

We have defined $H ; e \downarrow c$ and $H ; s \rightarrow H' ; s'$ and extended the latter to give $s$ a meaning.

The way we did expressions is “large-step”.

The way we did statements is “small-step”.

So now you have seen both.

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 \( n \),
\[
\cdot \; s \rightarrow^n H \; \text{skip} \text{ cannot be derived.}
\]

Example: \textbf{while 1 skip}

By induction on \( n \) with stronger induction hypothesis: If we can derive
\[
\cdot \; s \rightarrow^n H \; s' \text{ then } s' \text{ is while 1 skip or if 1 (skip; while 1 skip) skip or skip; while 1 skip.}
\]
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.