CSE505: Graduate Programming Languages Lecture 3 — Operational Semantics

Dan Grossman Fall 2012

Where we are

- Done: OCaml tutorial, "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)

Review

IMP's abstract syntax is defined inductively:

We haven't yet said what programs *mean*! (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 e evaluate to?

1+2 x+2

Informal idea

Given e, what c does e evaluate to?

1+2 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 ∃ 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:

$$H(x) = \left\{egin{array}{ccc} c & ext{if} & H = H', x \mapsto c \ H'(x) & ext{if} & H = H', y \mapsto c' ext{ and } y
eq x \ 0 & ext{if} & H = \cdot \end{array}
ight.$$

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"

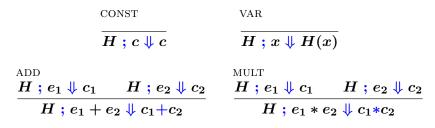
It is just a relation on triples of the form (H, e, c)

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

We can write: $., x \mapsto 3$; $x + y \Downarrow 3$, 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



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" e₁ has to be the "same" expression

Instantiating rules

Example instantiation:

$$\frac{\cdot, \mathtt{y} \mapsto 4 \ ; \ 3 + \mathtt{y} \Downarrow 7 \qquad \cdot, \mathtt{y} \mapsto 4 \ ; \ 5 \Downarrow 5}{\cdot, \mathtt{y} \mapsto 4 \ ; \ (3 + \mathtt{y}) + 5 \Downarrow 12}$$

Instantiates:

 $\frac{H}{H}; e_1 \Downarrow c_1 \qquad H; e_2 \Downarrow c_2 \\ H; e_1 + e_2 \Downarrow c_1 + c_2$

with

$$H = \cdot, y \mapsto 4$$

$$e_1 = (3 + y)$$

$$c_1 = 7$$

$$e_2 = 5$$

$$c_2 = 5$$

Derivations

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

Example:

$$\begin{array}{c} \hline \cdot, \texttt{y} \mapsto \texttt{4} \texttt{; } \texttt{3} \Downarrow \texttt{3} & \hline \cdot, \texttt{y} \mapsto \texttt{4} \texttt{; } \texttt{y} \Downarrow \texttt{4} \\ \hline \cdot, \texttt{y} \mapsto \texttt{4} \texttt{; } \texttt{3} + \texttt{y} \Downarrow \texttt{7} & \hline \cdot, \texttt{y} \mapsto \texttt{4} \texttt{; } \texttt{5} \Downarrow \texttt{5} \\ \hline \cdot, \texttt{y} \mapsto \texttt{4} \texttt{; } (\texttt{3} + \texttt{y}) + \texttt{5} \Downarrow \texttt{12} \end{array}$$

By definition, 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) R₀
- 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}
 - \blacktriangleright So R_i is all triples at the bottom of height- j complete derivations for $j \leq i$
- R_∞ is the relation we defined
 - All triples at the bottom of complete derivations

For the math folks: ${\boldsymbol R}_\infty$ 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* ↓ *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 \boldsymbol{e}

A statement doesn't produce a constant

A statement doesn't produce a constant

It produces a new, possibly-different heap

If it terminates

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

A statement doesn't produce a constant

It produces a new, possibly-different heap

If it terminates

We could define H_1 ; $s \Downarrow H_2$

- \blacktriangleright 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

$$egin{aligned} & H \ ; e \Downarrow c \ \hline H \ ; x := e o H, x \mapsto c \ ; ext{skip} \end{aligned}$$

| seq1 | ${ m SEQ2} \ H \ ; \ s_1 ightarrow H' \ ; \ s_1'$ |
|--|--|
| $\overline{H \ ; skip; s 	o H \ ; s}$ | $\overline{H ; s_1; s_2 ightarrow H' ; s_1'; s_2}$ |
| $ \overset{\text{IF1}}{H; e \Downarrow c} c > 0 $ | $H; e \Downarrow c c \leq 0$ |
| $\overline{H \ ; \ if \ e \ s_1 \ s_2 ightarrow H \ ; \ s_1}$ | $\overline{H \ ; \ { m if} \ e \ s_1 \ s_2 ightarrow H \ ; \ s_2}}$ |

Statement semantics cont'd

What about while $e \ s$ (do s and loop if e > 0)?

Statement semantics cont'd

What about while $e \ s$ (do s and loop if e > 0)?

WHILE

H ; while $e \ s \to H$; if $e \ (s;$ while $e \ s)$ skip

Many other equivalent definitions possible

Program semantics

Defined $H : s \rightarrow H' : s'$, but what does "s" mean/do?

Our machine iterates: $H_1;s_1 \rightarrow H_2;s_2 \rightarrow H_3;s_3 \dots$, with each step justified by a complete derivation using our single-step statement semantics

Let H_1 ; $s_1 \rightarrow^n H_2$; s_2 mean "becomes after n steps"

Let H_1 ; $s_1 \rightarrow^* H_2$; s_2 mean "becomes after 0 or more steps"

Pick a special "answer" variable ans

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

Does every s produce a c?

$$x := 3; (y := 1; while x (y := y * x; x := x-1))$$

$$x := 3; (y := 1; while x (y := y * x; x := x-1))$$

$$\cdot$$
; x := 3; y := 1; while x s

$$x := 3; (y := 1; while x (y := y * x; x := x-1))$$

$$\cdot$$
; x := 3; y := 1; while x s

$$\rightarrow$$
 $\cdot, x \mapsto 3;$ skip; $y := 1;$ while $x s$

$$x := 3; (y := 1; while x (y := y * x; x := x-1))$$

$$\cdot$$
; x := 3; y := 1; while x s

$$\rightarrow$$
 $\cdot, x \mapsto 3;$ skip; $y := 1;$ while $x s$

$$\rightarrow$$
 $\cdot, x \mapsto 3; y := 1;$ while $x s$

$$x := 3; (y := 1; while x (y := y * x; x := x-1))$$

$$\cdot$$
; x := 3; y := 1; while x s

$$ightarrow \ \cdot, \mathrm{x} \mapsto \mathbf{3};$$
 skip; $\mathrm{y} := 1;$ while $\mathrm{x} \ s$

$$\rightarrow$$
 $\cdot, x \mapsto 3; y := 1;$ while $x s$

$$\rightarrow^2$$
 $\cdot, x \mapsto 3, y \mapsto 1;$ while x s

$$x := 3; (y := 1; while x (y := y * x; x := x-1))$$

$$\cdot$$
; x := 3; y := 1; while x s

$$ightarrow \ \cdot, \mathrm{x} \mapsto \mathbf{3};$$
 skip; $\mathrm{y} := 1;$ while $\mathrm{x} \ s$

$$\rightarrow$$
 $\cdot, x \mapsto 3; y := 1;$ while $x s$

$$ightarrow^2$$
 $\cdot, x \mapsto 3, y \mapsto 1;$ while x s

$$\rightarrow$$
 $\cdot, x \mapsto 3, y \mapsto 1;$ if $x (s;$ while $x s)$ skip

$$x := 3; (y := 1; while x (y := y * x; x := x-1))$$

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).

$$\cdot$$
; x := 3; y := 1; while x s

$$ightarrow \ \cdot, \mathrm{x} \mapsto \mathbf{3};$$
 skip; $\mathrm{y} := 1;$ while $\mathrm{x} \ s$

$$ightarrow \ \cdot, \mathtt{x} \mapsto \mathbf{3}; \, \mathtt{y} := 1;$$
 while $\mathtt{x} \ s$

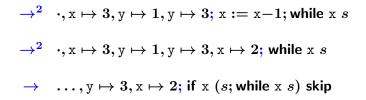
$$ightarrow^2$$
 $\cdot, x \mapsto 3, y \mapsto 1;$ while x s

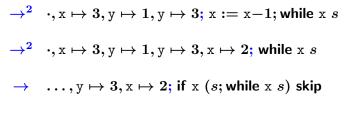
$$\rightarrow$$
 $\cdot, x \mapsto 3, y \mapsto 1$; if $x (s; while x s)$ skip

 \rightarrow $\cdot, x \mapsto 3, y \mapsto 1; y := y * x; x := x - 1;$ while x s

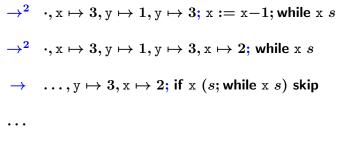
 \rightarrow^2 $\cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3; x := x-1;$ while x s

 $\begin{array}{l} \rightarrow^2 \quad \cdot, \mathbf{x} \mapsto \mathbf{3}, \mathbf{y} \mapsto \mathbf{1}, \mathbf{y} \mapsto \mathbf{3}; \, \mathbf{x} := \mathbf{x} - \mathbf{1}; \, \text{while } \mathbf{x} \, s \\ \\ \rightarrow^2 \quad \cdot, \mathbf{x} \mapsto \mathbf{3}, \mathbf{y} \mapsto \mathbf{1}, \mathbf{y} \mapsto \mathbf{3}, \mathbf{x} \mapsto \mathbf{2}; \, \text{while } \mathbf{x} \, s \end{array}$





. . .



$$ightarrow \ldots, \mathtt{y} \mapsto 6, \mathtt{x} \mapsto 0;$$
 skip

Where we are

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 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 \boldsymbol{x} holding $\boldsymbol{0}$

Establishing Properties

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

Example: Our last program terminates with \mathbf{x} holding $\mathbf{0}$

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

Example: while 1 skip

Establishing Properties

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

Example: Our last program terminates with \mathbf{x} holding $\mathbf{0}$

We can prove a program diverges, i.e., for all H and n, \cdot ; $s \rightarrow^{n} H$; 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.