

Where we are

- Done: Caml basics, IMP syntax, structural induction
- Today: IMP operational semantics
- Tonight: You could (almost?) finish homework 1

<u>Review</u>

IMP's abstract syntax is defined inductively:

$$s ::= skip | x := e | s; s | if e s s | while e s$$

$$e ::= c | x | e + e | e * e$$

$$(c \in \{\dots, -2, -1, 0, 1, 2, \dots\})$$

$$(x \in \{x_1, x_2, \dots, y_1, y_2, \dots, z_1, z_2, \dots\})$$

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

Encode our "social understanding" about variables and control flow

<u>Outline</u>

- 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 $oldsymbol{H}$ to encode 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 it is).

Heaps

 $H ::= \cdot \mid H, x \mapsto c$

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

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:

H; $e \Downarrow c$

to mean, "e evaluates to c under heap H".

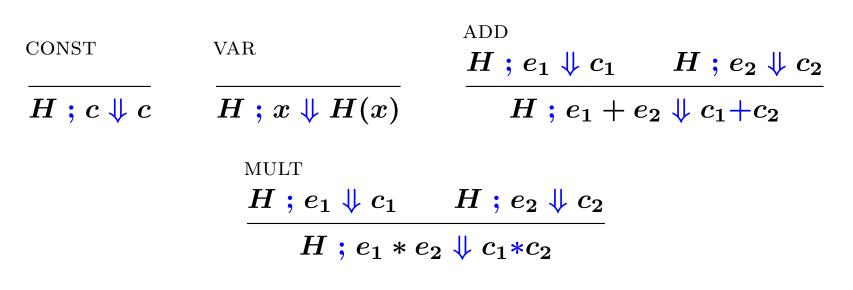
It is just a relation on triples of the form (H, e, c).

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



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:

$$\frac{\cdot, \mathsf{y} \mapsto 4 \text{ ; } 3 + \mathsf{y} \Downarrow 7 \quad \cdot, \mathsf{y} \mapsto 4 \text{ ; } 5 \Downarrow 5}{\cdot, \mathsf{y} \mapsto 4 \text{ ; } (3 + \mathsf{y}) + 5 \Downarrow 12}$$

Instantiates:

$$\begin{array}{ccc} H \ ; e_1 \Downarrow c_1 & H \ ; e_2 \Downarrow c_2 \\ \hline H \ ; e_1 + e_2 \Downarrow c_1 + c_2 \end{array}$$

with $H = \cdot, \mathrm{y} \mapsto 4, \, e_1 = (3 + \mathrm{y}), \, c_1 = 7, \, e_2 = 5, \, c_2 = 5 \end{array}$

Dan Grossman

Derivations

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

Example:

 $\cdot, \mathsf{y} {\mapsto} 4 ; \mathbf{3} \Downarrow \mathbf{3} \quad \cdot, \mathsf{y} {\mapsto} 4 ; \mathsf{y} \Downarrow 4$ $\cdot, \mathbf{y} \mapsto 4 \mathbf{; 3} + \mathbf{y} \Downarrow \mathbf{7} \qquad \cdot, \mathbf{y} \mapsto 4 \mathbf{; 5} \Downarrow \mathbf{5}$ \cdot , y \mapsto 4; (3 + y) + 5 \Downarrow 12 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 ↓ c such that we can instantiate some inference rule to have conlusion H; e ↓ 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_∞ is the relation we defined
 - All triples at the bottom of complete derivations.

For the math folks: R_{∞} 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* ↓ *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.

<u>On to statements</u>

A statement doesn't produce a constant.

It produces a new, possibly-different heap.

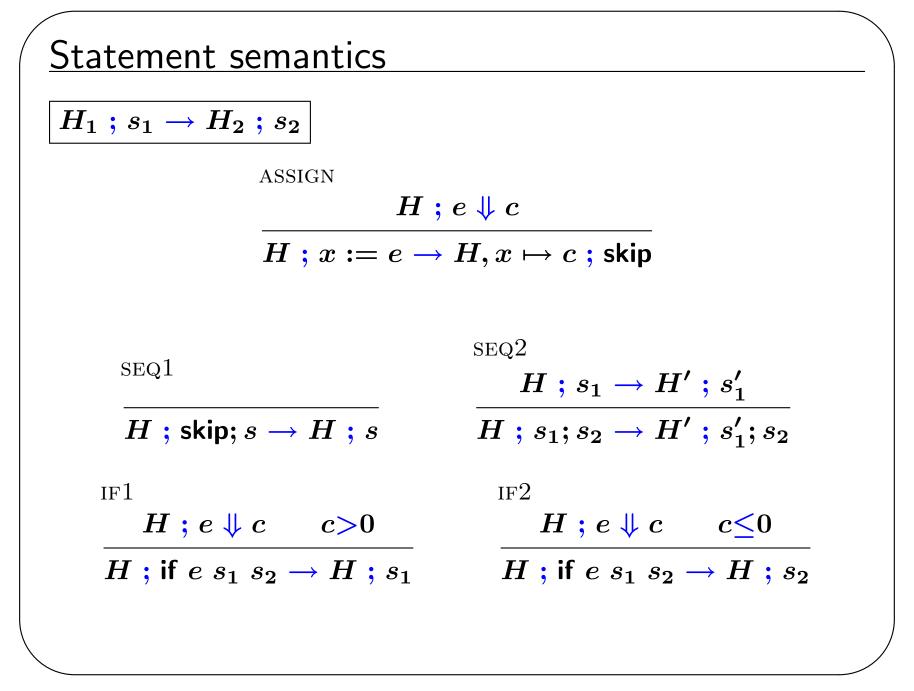
• If it terminates

We could define H_1 ; $s \Downarrow H_2$

- ullet 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 cont'd

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

WHILE

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

Many other equivalent definitions possible

Program semantics

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

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

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

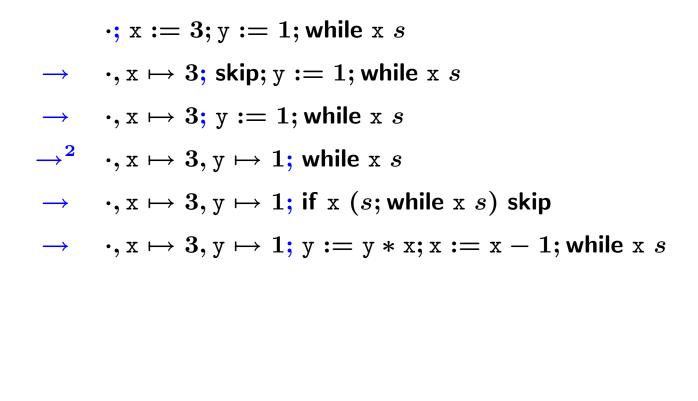
The program s produces c if \cdot ; $s \rightarrow^* H$; skip and H(ans) = c

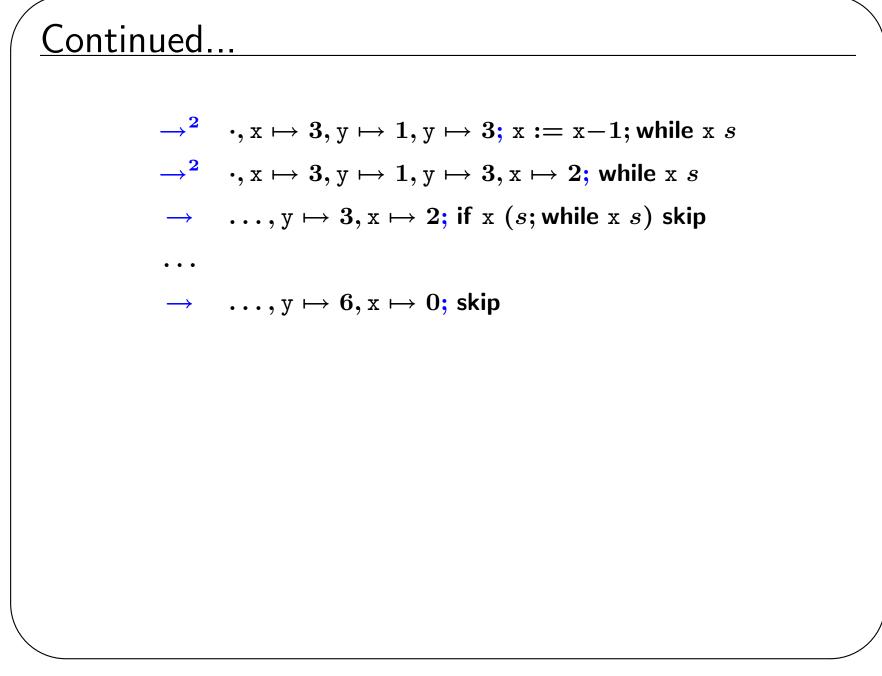
Does every s produce a c?

Example program execution

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).





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 $oldsymbol{H}$ and $oldsymbol{n}$,

 $\cdot ; s \rightarrow^{n} H ;$ skip cannot be derived.

Example: while 1 skip

By induction on n with stronger induction hypothesis: If we can derive $\cdot ; s \rightarrow^{n} H ; s'$ then s' 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.