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:

\[
\begin{align*}
  s &::= \text{skip} | x := e | s; s | \text{if } e s s | \text{while } e s \\
  e &::= c | x | e + e | e \cdot e
\end{align*}
\]

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 \( e \) evaluate to?

\[
\begin{align*}
  1 + 2 &\quad \text{or} \quad x + 2
\end{align*}
\]

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 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 \( H \)"

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
  - ...

Boyana Norris
2015
Derivations

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

The judgment

We will write: \( H \vdash e \downarrow c \)

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

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

Inference rules

\[
\begin{align*}
\text{CONST} & : & H ; c \downarrow c & \quad \text{VAR} & : & H ; x \downarrow H(x) \\
\text{ADD} & : & H ; e_1 \downarrow c_1 & \quad \text{MULT} & : & H ; e_1 \downarrow c_1 \\
& & H ; e_2 \downarrow c_2 & & H ; e_1 * e_2 \downarrow c_1 * c_2 \\
\end{align*}
\]

Example instantiation:

\[
\cdot, y \mapsto 4 ; 3 \downarrow 7 \\
\cdot, y \mapsto 4 ; 5 \downarrow 5 \\
\cdot, y \mapsto 4 ; (3 + y) \downarrow 12
\]

Instantiates:

\[
\begin{align*}
\text{ADD} & : & H ; e_1 \downarrow c_1 & \quad \text{MULT} & : & H ; e_1 \downarrow c_1 \\
& & H ; e_2 \downarrow c_2 & & H ; e_1 * e_2 \downarrow c_1 * c_2 \\
\end{align*}
\]

with

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

Derivations

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

Example:

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

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

Instantiating rules

Example instantiation:

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

Or:

\[
\begin{align*}
\cdot, x \mapsto 3 ; 3 \downarrow 3 & \\
\cdot, y \mapsto 4 ; y \downarrow 4 & \\
\cdot, y \mapsto 4 ; 5 \downarrow 5 &
\end{align*}
\]

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

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
**Some theorems**

- Progress: For all $H$ and $e$, there exists a $c$ such that $H; c \downarrow c$
- 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 does not 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**

$H_1; s_1 \rightarrow H_2; s_2$

**Statement semantics cont’d**

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

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

Many other equivalent definitions possible

**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 $s = (y := y * x; x := x−1)$.

- $x := 3; y := 1; \text{while } x s$
  - $x := 3; y := 1; \text{while } x s$
  - $x := 3; y := 1; \text{while } x s$
  - $x := 3; y := 1; \text{while } x s$
  - $x := 3; y := 1; \text{while } x s$
  - $x := 3; y := 1; \text{while } x s$
  - $x := 3; y := 1; \text{while } x s$

**Program semantics**

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

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

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 $\cdot; s \rightarrow^* H; s$ and $H(ans) = c$

Does every $s$ produce a $c$?
Continued...

\[ x \mapsto 3, y \mapsto 1; \text{while } x \]
\[ x \mapsto 3, y \mapsto 1, y \mapsto 3; x := x - 1; \text{while } x \]
\[ \ldots, y \mapsto 3, x \mapsto 2; \text{if } x (s; \text{while } x) \text{ skip} \]
\[ \ldots \]
\[ \ldots, y \mapsto 6, x \mapsto 0; \text{skip} \]

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 \( x \) 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: \textbf{while 1 skip}

By induction on \( n \), but requires a \textit{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.