Today's Challenge:
Develop a small-step semantics for IMP

Cormac Flanagan
University of California, Santa Cruz

Review: Big-Step vs. Small-Step Operational Semantics

- Big-Step Operational Semantics: \( e \downarrow n \)
  - Judgment \( e \downarrow n \) means that \( e \) evaluates to \( n \)
  - In one, big step, all the way to a result
  - Hard to talk about
    - commands that do not terminate.
    - intermediate states.

- Small-Step Operational Semantics: \( e \rightarrow e' \)
  - describe a single step in the evaluation
  - many steps may be needed to get a result

Review: Small-Step Evaluation Rules for ARITH

- Example: \( (3 + 4) + 5 \)
  - local rule: \( 3 + 4 \rightarrow 7 \)
  - global rule: \( H[3 + 4] \rightarrow H[7] \)
  - i.e. \( (3 + 4) + 5 \rightarrow 7 + 5 \)

Refactoring the Contextual semantics for ARITH

- Refactor semantics as local + global reduction rules
- Evaluation contexts for ARITH (same)
  - \( H ::= \ast \mid H \cdot e \mid n \cdot H \mid H \cdot e \mid n \cdot H \)
- Local reduction rules
  - \( n_1 \cdot n_2 \rightarrow n \) where \( n = n_1 + n_2 \)
  - \( n_1 \cdot n_2 \rightarrow n \) where \( n = n_1 \times n_2 \)
- Global reduction rule
  - \( H[e] \rightarrow H[e'] \) provided \( e \rightarrow e' \)
- Example: evaluate \( (3 + 4) + 5 \)
  - decompose: \( (3 + 4) + 5 \) is \( H[3+4] \) where \( H = \ast \cdot 5 \)
  - local rule: \( 3 + 4 \rightarrow 7 \)
  - global rule: \( H[3+4] \rightarrow H[7] \)
  - i.e. \( (3 + 4) + 5 \rightarrow 7 + 5 \)

Contextual, Small-Step Semantics for IMP
Contextual, Small-Step Semantics for IMP

- Each execution step is a rewrite of program plus store
- We will define a relation \( \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle \)
  - \( c' \) is obtained from \( c \) through an atomic rewrite step.
  - \( \langle x := 2 + 8, \sigma \rangle \rightarrow \langle x := 10, \sigma \rangle \rightarrow \langle \text{skip}, \sigma[x := 10] \rangle \)
- Terminal command: "skip"
  - evaluation stops here
  - evaluation continues until we get to "skip"
  - may never reach skip - eg: while true do skip

Review: IMP Syntax

- \( e ::= n \)
  - \( x \)
  - \( e_1 + e_2 \)
  - \( e_1 - e_2 \)

- \( b ::= \text{true} \)
  - \( \text{false} \)
  - \( e_1 = e_2 \)
  - \( \neg b \)
  - \( b_1 \land b_2 \)
  - \( b_1 \lor b_2 \)

- \( c ::= \text{skip} \)
  - \( x := e \)
  - \( c_1 ; c_2 \)
  - \( \text{if } b \text{ then } c_1 \text{ else } c_2 \)
  - \( \text{while } b \text{ do } c \)

Local Reduction Rules for IMP

- Local reduction rules specify what can be evaluated in one step: \( \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle \) (ditto for expressions)

\[ \langle x, \sigma \rangle \rightarrow \langle \sigma(x), \sigma \rangle \]
\[ \langle n_1 + n_2, \sigma \rangle \rightarrow \langle n, \sigma \rangle \quad \text{where } n = n_1 + n_2 \]
\[ \langle n_1 = n_2, \sigma \rangle \rightarrow \langle \text{true}, \sigma \rangle \quad \text{if } n_1 = n_2 \]
\[ \langle x := n, \sigma \rangle \rightarrow \langle \text{skip}, \sigma[x := n] \rangle \]
\[ \langle \text{skip}; c, \sigma \rangle \rightarrow \langle c, \sigma \rangle \]
\[ \langle \text{if } \neg b \text{ then } c_1 \text{ else } c_2, \sigma \rangle \rightarrow \langle c_2, \sigma \rangle \]
\[ \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \langle \text{if } b \text{ then } c; \text{while } b \text{ do } c \text{ else } \text{skip}, \sigma \rangle \]

Redexes

- A redex is a syntactic expression or command that can be reduced by a local reduction rule
- For brevity, we omit some redexes
- Redexes are defined by a grammar:

\[ r ::= x \]
\[ n + r \]
\[ n = r \]
\[ \neg r \]
\[ r \land b \]
\[ p \land r \]
\[ x := r \]
\[ r ; c \]
\[ \text{if } H \text{ then } c_1 \text{ else } c_2 \]

Review

- A redex is something that can be reduced in one step
  - E.g. 2+8
- Local reduction rules reduce these redexes
  - E.g. 2+8, \( \sigma \) \rightarrow 10, \( \sigma \)
- Next: global reduction rules
- Consider
  - \( \langle x := 1(2+8), \sigma \rangle \)
  - \( \langle \text{while false do } x := 1(2+8), \sigma \rangle \)
- Should we also reduce 2+8 in these cases?
Evaluation Contexts

- **Examples**
  - \( x := 1 + \)
  - Fill context \( H \) with \( 2 + 8 \) to yield \( H[2+8] = x := 1 + (2+8) \)
  - Or fill context with \( 10 \) to yield \( H[10] = x := 1 + 10 \)
  - \( \ast \)
  - NOT: while false do \( x := 1 + \)
  - NOT: while false do \( \ast \)

Contexts: Notes

- Evaluation contexts say how to find the next redex:
  - Consider \( e_1 + e_2 \) and its decomposition as \( H[r] \).
    - If \( e_1 \) is \( n_1 \) and \( e_2 \) is \( n_2 \)
      - then \( H = n_1 + n_2 \)
    - If \( e_1 \) is \( n_1 \) and \( e_2 \) is not \( n_2 \)
      - then \( H = n_1 + H_2 \) and \( e_2 = H_2[r] \)
    - If \( e_1 \) is not \( n_1 \)
      - then \( H = H_1 + e_2 \) and \( e_1 = H_1[r] \)

- In the last two cases decomposition is done recursively.

The Global Reduction Rule

- General idea of the contextual semantics:
  - Decompose the current expression into
    - the next redex \( r \)
    - and an evaluation context \( H \) (the remaining program).
  - Reduce the redex "\( r \)" to some other expression "\( e \)."
  - Put "\( e \)" back into the original context, yielding \( H[e] \).

- Formalized as a small step rule:
  - If \( \langle r, \sigma \rangle \rightarrow \langle e, \sigma' \rangle \) then \( \langle H[r], \sigma \rangle \rightarrow \langle H[e], \sigma' \rangle \)

Contextual Semantics: Example

- Consider the small-step evaluation of \( x := 1; x := x + 1 \) in the initial state \( \{ x := 0 \} \)

<table>
<thead>
<tr>
<th>State</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1; x := x + 1 )</td>
<td>( x := 1 )</td>
<td>( x := x + 1 )</td>
</tr>
<tr>
<td>( \text{skip}; x := x + 1 )</td>
<td>( \text{skip}; x := x + 1 )</td>
<td></td>
</tr>
<tr>
<td>( x := 1 + 1 )</td>
<td>( x := 1 + 1 )</td>
<td></td>
</tr>
<tr>
<td>( x := 2 )</td>
<td>( x := 2 )</td>
<td></td>
</tr>
</tbody>
</table>

Normal vs Short-Circuit Boolean Operators

- What if we want normal evaluation of \( \land \)?
  - Define the following contexts, redexes, and local rules:
    - \( H := \ldots \) | \( H \land b_2 \) | \( p_1 \land H \)
    - \( r := \ldots \) | \( p_1 \land p_2 \)
    - \( p_1 \land p_2 \) \( \rightarrow \) \( p_2 \) \( \Rightarrow \) where \( p = p_1 \lor p_2 \)

- What if we want short-circuit evaluation of \( \land \)?
  - Define the following contexts, redexes, and local rules:
    - \( H := \ldots \) | \( H \land b_2 \)
    - \( r := \ldots \) | \( \text{true} \land b_2 \) | \( \text{false} \land b_2 \)
    - \( \text{true} \land b_2 \) \( \rightarrow \) \( \text{true} \land b_2 \)
    - \( \text{false} \land b_2 \) \( \rightarrow \) \( \text{false} \land b_2 \)
    - The local reduction kicks in before \( b_2 \) is evaluated.
Some Further Topics

• Treatment of errors in operational semantics
  - with an explicit “error” result, as in (3/0) \rightarrow error,
  - with an “error” expression, as in (3 + error),
  - with “stuck” computations, so (3/0) \rightarrow r for no r.

Contextual Semantics: Notes

• For example: c = c_1 ; c_2
  - either c_1 = skip and then c = H[skip; c_2] with H = *
  - or c_1 \neq skip and then c_1 = H[r];
    so c = c_1 ; c_2 = H[r]; c_2 = H'[r] where H' = H : c_2

• For example: c = if b then c_1 else c_2
  - either b = true or b = false and then c = H[r] with H = *
  - or b is not a value and b = H[r];
    so c = H'[r] where H' = if H then c_1 else c_2

• Decomposition theorem: If c is not “skip” then there exist unique H and r such that c is H[r].
  \implies \text{Progress and determinism.}

Summary of Operational Semantics

• Precise specification of dynamic semantics:
  - order of evaluation (or that it doesn’t matter)
  - error conditions (sometimes implicitly, by rule applicability)
• Simple and abstract (cf. implementations)
  - no low-level details such as stack and memory management,
    data layout, etc.
• Often not compositional (as for while)
• Basis for some important proofs about languages
  - e.g. type soundness
• Basis for some reasoning about particular programs
• Point of reference for other semantics

Homework 4 (for October 18)

Consider a variant of ARITH where integers can be between 0 and $2^{64}-1$ only, and the arithmetic operations yield an “OVERFLOW” error rather than results larger than $2^{64}-1$ (and otherwise behave as usual).

1. Revise the abstract syntax of the language.
   Note that the “overflow” error is not an expression of the language.

2. Revise the big-step operational semantics, using judgments of the form $e \Downarrow v$ where $e$ is an expression and $v$ is either an integer $n$ or the new result OVERFLOW.

3. Write a contextual semantics for this variant of ARITH.
   In other words, define redexes, local reduction rules, contexts, and global reduction rules.
   Hint: think about the “type” of $\rightarrow$.
   What kinds of things should be on the left and right of this relation?

4. Turn in on paper at start of class Tuesday October 18th.
   Please attempt before Thursday, so we can discuss in class if need be.

Project Proposals

• Project proposals due Thursday October 20th
  - One (or two) page document describing
    - your proposed project
    - time budget (how much work is it?)
    - timeline (when will you do which parts?)
    - risks (always consider - and mitigate - risks!)
  - Office hours
    - next week Jessica has office hours at 11am instead of 10am.
  - Next class
    - axiomatic semantics
    - read Winskel Chapter 6