CSE 230: Winter 2010  
Principles of Programming Languages

Lecture 2:  
Big-step Operational Semantics

Ranjit Jhala  
UC San Diego

Administrative Factoids

• Make-Up Lectures
  - Fri Jan 8, 12:30 - 3:00pm (Tomorrow!)
  - Mon Jan 25, 12:30 - 3:00pm

• Location: CSE 1202

ARITH: Arithmetic Expressions

Concrete syntax
• Rules to express programs as strings of characters
  - relevant to readability, familiarity, etc.

• Issues and choices
  - Should semicolon separate or terminate statements?
  - How should comments be indicated?

• “Solved Problem”  
  - Finite automata and context-free grammars.
  - Automatic parser generators

Irrelevant to 230

ARITH: A Simple Expression Language

syntax and operational semantics
Programs = Abstract Syntax Trees

AST is a parse tree
- Amenable to formal, algorithmic manipulation
- Fairly independent of concrete syntax

Abstract syntax for ARITH
\[ e ::= n \]
\[ | e_1 + e_2 \]
\[ | e_1 \times e_2 \]

An Operational Semantics
- Specifies how to evaluate expressions
- Defined by cases on the form of expressions:
  \[ n \] evaluates to \( n \)
  - \( n \) is a normal form, no need to evaluate further
  - \( e_1 + e_2 \) evaluates to \( n \) if
    - \( e_1 \) evaluates to \( n_1 \)
    - \( e_2 \) evaluates to \( n_2 \)
    - and \( n \) is the sum of \( n_1 \) and \( n_2 \)
  - \( e_1 \times e_2 \) evaluates to \( n \) if
    - \( e_1 \) evaluates to \( n_1 \)
    - \( e_2 \) evaluates to \( n_2 \)
    - and \( n \) is the product of \( n_1 \) and \( n_2 \)

Another Formulation
- Notation: \( e \Downarrow n \) means that \( e \) evaluates to \( n \)
  - Judgment: stmt about relation between \( e \) and \( n \)
  - Allows us to write evaluation rules concisely
    - \( n \Downarrow n \)
    - \( e_1 + e_2 \Downarrow n \) if
      - \( e_1 \Downarrow n_1 \)
      - \( e_2 \Downarrow n_2 \)
      - and \( n \) is the sum of \( n_1 \) and \( n_2 \)
    - \( e_1 \times e_2 \Downarrow n \) if
      - \( e_1 \Downarrow n_1 \)
      - \( e_2 \Downarrow n_2 \)
      - and \( n \) is the product of \( n_1 \) and \( n_2 \)
Operational Semantics as Inference Rules

\[ e_1 \downarrow n_1, \quad e_2 \downarrow n_2 \rightarrow n \text{ is the sum of } n_1 \text{ and } n_2 \]

\[ e_1 + e_2 \downarrow n \]

\[ e_1 \downarrow n_1, \quad e_2 \downarrow n_2 \rightarrow n \text{ is the product of } n_1 \text{ and } n_2 \]

\[ e_1 * e_2 \downarrow n \]

- Rule: “above the line” implies “below the line”
- Evaluation rules for big-step op. semantics
- Derivation rules for the judgement \( e \downarrow n \)

How to Read the Rules?

Ans 1: Forward as inference rules:
If hypothesis judgments (“numerator”) hold then conclusion judgment (“denominator”) holds \( e_1 \downarrow n_1, \quad e_2 \downarrow n_2 \rightarrow n \text{ is the sum of } n_1 \text{ and } n_2 \)

\[ e_1 + e_2 \downarrow n \]
e.g., If we know that \( e_1 \downarrow 5 \) and \( e_2 \downarrow 7 \), then we can infer that \( e_1 + e_2 \downarrow 12 \)

No “numerator”? Denominator is a fact

How to Read the Rules?

Ans 2: Backward, as evaluation rules:
Evaluating \( e \): find \( n \) s.t. \( e \downarrow n \) is derivable from rules

\[ \text{e.g., how to evaluate } e_1 * e_2? \]

Find \( n \) s.t. \( e_1 * e_2 \downarrow n \) is derivable

- Last step in derivation must be multiplication rule:
  (Conclusions of other rules do not match)
- Recursively find \( n_1 \) and \( n_2 \) s.t. \( e_1 \downarrow n_1 \) and \( e_2 \downarrow n_2 \) derivable
  - “Stitch” derivations together with product rule

\[
\begin{array}{c}
D_1 \quad D_2 \\
\hline
\quad e_1 \downarrow n_1 \\
\quad e_2 \downarrow n_2 \\
\hline
\quad e_1 * e_2 \downarrow n
\end{array}
\]

Rules are syntax-directed:
- exactly one rule for each kind of expression

At each step at most one rule applies
- Simple evaluation procedure

Called reasoning by inversion on the derivation rules
Ex: evaluate \((3+5) \times (4+2)\)

<table>
<thead>
<tr>
<th>3</th>
<th>5</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3 \downarrow 3)</td>
<td>(5 \downarrow 5)</td>
<td>(4 \downarrow 4)</td>
<td>(2 \downarrow 2)</td>
</tr>
<tr>
<td>((3+5) \downarrow 8)</td>
<td>((4+2) \downarrow 6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((3+5) \times (4+2) \downarrow 48)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Semantic Properties: Uniqueness**

*ARITH* is a deterministic language:
all exprs evaluate to at most one value

\[
\forall e. \forall n. \forall n'. e \downarrow n \land e \downarrow n' \Rightarrow n = n'
\]

Q: How would we prove such a theorem?

**IMP: An Imperative Language**

syntax and operational semantics
**IMP Syntactic Entities**

- **Int** integer literals $n$
- **Bool** booleans ${\text{true}, \text{false}}$
- **Loc** locations $x, y, z, \ldots$ (assignable variables)
- **Aexp** arithmetic expressions $e$
- **Bexp** boolean expressions $b$
- **Comm** commands $c$

**Abstract Syntax: Arith Expressions (Aexp)**

$$e ::= \ n \quad \text{for } n \in \text{Int}$$
$$\mid x \quad \text{for } x \in \text{Loc}$$
$$\mid e_1 + e_2 \quad \text{for } e_1, e_2 \in \text{Aexp}$$
$$\mid e_1 - e_2 \quad \text{for } e_1, e_2 \in \text{Aexp}$$
$$\mid e_1 \times e_2 \quad \text{for } e_1, e_2 \in \text{Aexp}$$

**Abstract Syntax: Commands (Comm)**

$$c ::= \text{skip}$$
$$\mid x := e \quad \text{for } x \in \text{L} \ & \ e \in \text{Aexp}$$
$$\mid c_1 ; c_2 \quad \text{for } c_1, c_2 \in \text{Comm}$$
$$\mid \text{if } b \text{ then } c_1 \text{ else } c_2 \quad \text{for } b \in \text{Bexp} \ & \ c_1, c_2 \in \text{Comm}$$
$$\mid \text{while } b \text{ do } c \quad \text{for } c \in \text{Comm} \ & \ b \in \text{Bexp}$$

**Notes:**
- Variables are **not** declared
- All variables have integer type
- There are **no** side-effects

- Typing rules embedded in syntax definition
  - Other checks may **not** be context-free
  - Need to be specified separately (e.g., vars are declared)
- Commands contain all the side-effects in the language
- (Missing: pointers, function calls, exceptions ....)
Semantics of IMP: States

- Meaning of IMP expressions depends on the values of variables

- A state $\sigma$ is a function from Loc to Int
  - Value of variables at a given moment
  - Set of all states is $\Sigma = \text{Loc} \rightarrow \text{Int}$

\[
\begin{align*}
(c, \sigma) & \Downarrow \sigma^{-1} \\
(e, \sigma) & \Downarrow n \\
(b, \sigma) & \Downarrow t/f
\end{align*}
\]

Operational Semantics of IMP

Evaluation judgment for expressions:
- Ternary relation on expression, a state, and a value:
- We write: $\langle e, \sigma \rangle \Downarrow n$
  “Expression $e$ in state $\sigma$ evaluates to $n$”

Q: Why no state on the right?
  - Evaluation of expressions has no side-effects:
    - i.e., state unchanged by evaluating an expression

Q: Can we view judgment as a function of 2 args $e, \sigma$?
  - Only if there is a unique derivation ...

Evaluation Rules (for Aexp)

\[
\begin{align*}
\langle n, \sigma \rangle & \Downarrow n \\
\langle x, \sigma \rangle & \Downarrow \sigma(x) \\
\langle e_1 + e_2, \sigma \rangle & \Downarrow n_1 + n_2 \\
\langle e_1 - e_2, \sigma \rangle & \Downarrow n_1 - n_2 \\
\langle e_1 * e_2, \sigma \rangle & \Downarrow n_1 * n_2
\end{align*}
\]
## Evaluation Rules (for Bexp)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;true, σ&gt;</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>&lt;false, σ&gt;</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>&lt;e_1, σ&gt;</code></td>
<td><code>n_1</code></td>
</tr>
<tr>
<td><code>&lt;e_2, σ&gt;</code></td>
<td><code>n_2</code></td>
</tr>
<tr>
<td><code>p</code> is <code>n_1 = n_2</code></td>
<td><code>n_1</code></td>
</tr>
<tr>
<td><code>&lt;e_1, σ&gt;</code></td>
<td><code>n_1</code></td>
</tr>
<tr>
<td><code>&lt;e_2, σ&gt;</code></td>
<td><code>n_2</code></td>
</tr>
<tr>
<td><code>p</code> is <code>n_1 &lt; n_2</code></td>
<td><code>n_1</code></td>
</tr>
<tr>
<td><code>&lt;e_1, σ&gt;</code></td>
<td><code>n_1</code></td>
</tr>
<tr>
<td><code>&lt;e_2, σ&gt;</code></td>
<td><code>n_2</code></td>
</tr>
<tr>
<td><code>p</code> is <code>n_1 &lt; n_2</code></td>
<td><code>n_1</code></td>
</tr>
<tr>
<td><code>&lt;e_1 = e_2, σ&gt;</code></td>
<td><code>p</code></td>
</tr>
<tr>
<td><code>&lt;e_1 &lt; e_2, σ&gt;</code></td>
<td><code>p</code></td>
</tr>
<tr>
<td><code>&lt;e_1 = e_2, σ&gt;</code></td>
<td><code>p</code></td>
</tr>
<tr>
<td><code>&lt;e_1 &lt; e_2, σ&gt;</code></td>
<td><code>p</code></td>
</tr>
<tr>
<td><code>&lt;b_1, σ&gt;</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>&lt;c_1, σ&gt;</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>&lt;if b then c_1 else c_2, σ&gt;</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>&lt;if b then c_1 else c_2, σ&gt;</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>&lt;if b then c_1 else c_2, σ&gt;</code></td>
<td><code>true</code></td>
</tr>
</tbody>
</table>

## Evaluation Rules (for Comm)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;skip, σ&gt;</code></td>
<td><code>σ</code></td>
</tr>
<tr>
<td><code>&lt;c_1, σ&gt;</code></td>
<td><code>σ'</code></td>
</tr>
<tr>
<td><code>&lt;c_2, σ&gt;</code></td>
<td><code>σ''</code></td>
</tr>
<tr>
<td><code>&lt;c_1 ; c_2, σ&gt;</code></td>
<td><code>σ''</code></td>
</tr>
</tbody>
</table>

Define `σ[x ↦ n]` as:

- `σ[x ↦ n](x) = n`
- `σ[x ↦ n](y) = σ(y)`
Evaluation Rules (for Comm)

\[
\begin{align*}
<b, \sigma> & \Downarrow false \\
<\text{while } b \text{ do } c, \sigma> & \Downarrow \sigma \\
\end{align*}
\]

Evaluation of Commands

Order of evaluation is important and stipulated:

- \( c_1 \) is evaluated before \( c_2 \) in \( c_1; c_2 \)
- \( c_2 \) not evaluated in \( \text{if true then } c_1 \text{ else } c_2 \)
- \( c \) is not evaluated in \( \text{while false do } c \)

Evaluation of Commands

Evaluation rules are not syntax-directed
- Multi rules for conditional constructs \( \text{if , while} \)
- Only one can be applied at one time
  - Depends on value of condition (semantics-directed)

Evaluate from the zero-initialized store:

while \( x < 5 \) do \( x := x + 1 \)
- i.e. Given \( \sigma \) maps all vars to 0, find \( \sigma' \) s.t.
  - \(<\text{while } x < 5 \text{ do } x := x + 1 , \sigma> \Downarrow \sigma' \)

Evaluation rules are not syntax-directed
- Multi rules for conditional constructs \( \text{if , while} \)
- Only one can be applied at one time
  - Depends on value of condition (semantics-directed)
Homework

- Implement op semantics in Ocaml
  - IMP + exceptions

- Some guidelines
  - types for locations, AExp, BExp, Comm
  - stores are ML map from vars to integers

- Documentation on Web page

Strongly recommend you do it asap

---

Homework Skeleton

type loc = string;;
type state = n LocMap.t;;

let lookup sigma x = 
  try LocMap.find x sigma with Not_found -> 0

let update (sigma:state) (x:loc) (v:n) : state = 
  LocMap.add x n sigma;;

type aexp =
  | Int of int | Loc of variable| ...;;
type bexp = ...;;
type comm = ...;;

let rec eval_aexp (e:aexp) (sigma:state) = ...

let rec eval_bexp (b:bexp) (sigma:state) = ...

let rec eval_comm (c:comm) (sigma:state) = ...

let c = While(Leq(Loc "x",Int 5), Asgn("x",Sum(Loc "x",Int 1)));;

let sigma0 = initial state () ;;

let sigma = eval_comm c sigma0 ;;

lookup sigma "z";;
lookup sigma "x";;