Feedback. Thanks!

- Like interactivity
- Good pace
- Readings and classes connection
  - My goal: readings provide initial exposure and details, while classes decode “denseness”
- Forum posts “pressure”
  - Goal: take a few minutes to reflect on reading or class discussion
  - Not supposed to be overly burdensome

Questions?

Key Ideas

- Proof systems: judgments, inference rules, derivations
- Reading and using rules
  - inversion
  - syntax-directed
- Evaluation rule for while
Review: Rules of Inference

- We express the evaluation rules as **inference rules** for our judgment.
  - called the **derivation rules** for the judgment.
  - also called the **evaluation rules** (for operational semantics).
- In general, we have one rule for each language construct:

  $\frac{<e_1, \sigma> \Downarrow n_1 \quad <e_2, \sigma> \Downarrow n_2}{<e_1 + e_2, \sigma> \Downarrow n_1 + n_2}$

  This is the only rule for $e_1 + e_2$.

Review: Inversion

- **Backward (bottom-up) reasoning**
  - Suppose we want to evaluate $e_1 + e_2$, i.e., find $n$ s.t. $e_1 + e_2 \Downarrow n$ is derivable using the previous rules.
  - By inspection of the rules we notice that the last step in the derivation of $e_1 + e_2 \Downarrow n$ must be the addition rule.
  - the other rules have conclusions that would not match $e_1 + e_2 \Downarrow n$.
  - this is called reasoning by **inversion** on the derivation rules.

Review: Syntax-Directed Evaluation

- Thus we must find $n_1$ and $n_2$ such that $e_1 \Downarrow n_1$ and $e_2 \Downarrow n_2$ are derivable.
  - This is done recursively.
- If there is exactly one rule for each kind of expression we say that the rules are **syntax-directed**.
  - At each step at most one rule applies.
  - This allows a simple evaluation procedure as above (recursive tree-walk).

Summary: Proof Systems

- **Rules of inference** list the hypotheses necessary to arrive at a conclusion.

  $\frac{<x, \sigma> \Downarrow \sigma(x) \quad <e_1, \sigma> \Downarrow n_1 \quad <e_2, \sigma> \Downarrow n_2}{<e_1 + e_2, \sigma> \Downarrow n_1 + n_2}$

- A **derivation** involves interlocking (well-formed) instances of rules of inference.

  $\frac{<4, \sigma> \Downarrow 4 \quad <2, \sigma> \Downarrow 2}{<4*2, \sigma> \Downarrow 8}$

  $\frac{<4*2, \sigma> \Downarrow 8 \quad <6, \sigma> \Downarrow 6}{<(4*2) - 6, \sigma> \Downarrow 2}$

Evaluation Rules (for Com, while)

$\frac{<b, \sigma> \Downarrow \text{true}}{<\text{while } b \text{ do } c, \sigma> \Downarrow \sigma}$

$\frac{<b, \sigma> \Downarrow \text{true}}{<\text{while } b \text{ do } c, \sigma> \Downarrow \sigma}$

$\frac{<b, \sigma> \Downarrow \text{false}}{<\text{while } b \text{ do } c, \sigma> \Downarrow \sigma}$
Returning to Our Op. Sem. for IMP

\[
\begin{align*}
\text{Observations?} & \quad \text{• What is constrained/captured or not constrained/not captured by our rules?}
\end{align*}
\]

\[
\begin{align*}
\text{Observations: Command Evaluation} & \quad \text{• The order of evaluation is important} \\
& \quad \text{– } c_1 \text{ is evaluated before } c_2 \text{ in } c_1 ; c_2 \\
& \quad \text{– } c_2 \text{ is not evaluated in } "\text{if true then } c_1 \text{ else } c_2" \\
& \quad \text{– } c \text{ is not evaluated in } "\text{while false do } c" \\
& \quad \text{– } b \text{ is evaluated first in } "\text{if false then } c_1 \text{ else } c_2" \\
& \quad \text{– this is explicit in the evaluation rules} \\
& \quad \text{• Conditional constructs have multiple evaluation rules} \\
& \quad \text{– but only one can be applied at one time}
\end{align*}
\]

\[
\begin{align*}
\text{Disadvantages of Natural-Style or Big-Step Operational Semantics} & \quad \text{• Thoughts?} \\
& \quad \text{– Non-trivial control is burdensome} \\
& \quad \text{– No easy way to specify "detail"} \\
& \quad \text{– Hard to talk about "non-terminating"} \\
& \quad \text{– Hard to talk about intermediate states}
\end{align*}
\]
Solution

- Small-step operational semantics addresses these problems
  - Execution is modeled as a (possible infinite) sequence of states
- Contextual operational semantics is a small-step operational semantics where the atomic execution step is a rewrite of the program
  - Small-step op. sem. can be defined structurally (as we did with big-step)

Small-Step Operational Semantics

- We define a transition relation
  \[ (c, \sigma) \rightarrow (c', \sigma') \]
  - "c steps to c' via an atomic rewrite step"
  - Evaluation terminates when the program has been rewritten to a terminal program
    - one from which we cannot make further progress
- For IMP the terminal command is "skip"
  - As long as the command is not "skip" we can make further progress
    - some commands never reduce to skip (e.g., "while true do skip")

What is an Atomic Reduction?

- What is an atomic reduction step?
  - Granularity is a choice of the semantics designer
    - e.g., choice between an addition of arbitrary integers, or an addition of 32-bit integers
- How to select the next reduction step, when several are possible?
  - This is the order of evaluation issue

Redexes

- A redex is a syntactic expression or command that can be reduced in one atomic step
- Redexes are defined via a grammar:
  \[ r ::= x \quad (x \in L) \]
  \[ | n_1 + n_2 \]
  \[ | n_1 = n_2 \]
  \[ | \text{skip}; c \]
  \[ | \text{if true then } c_1 \text{ else } c_2 \]
  \[ | \text{if false then } c_1 \text{ else } c_2 \]
  \[ | \text{while } b \text{ do } c \]
- For brevity, we mix exp and com redexes

Redex or Not?

- 1 + 3 Yes
- (1 + 3) + 2 No
- x Yes
- 4 No where we mean reduces
- skip No
- if \(x \leq 0\) then \(x := -x\) else skip No
- while \(x \leq n\) do \(x := x + 1\) Yes

Redexes are never values!
Local Reduction Rules for IMP

- One for each redex: \(<r, \sigma> \rightarrow <e, \sigma'>\)
  - means that in state \(\sigma\), the redex \(r\) can be replaced in one step with the expression \(e\)

\(<x, \sigma> \rightarrow <\sigma(x), \sigma>\) where \(n = n_1 + n_2\)

\(<n_1 + n_2, \sigma> \rightarrow <n, \sigma'>\) if \(n_1 = n_2\)

\(<n_1 = n_2, \sigma> \rightarrow <true, \sigma'>\) if \(n_1 \neq n_2\)

\(<n_1 = n_2, \sigma> \rightarrow <false, \sigma'>\) if \(n_1 \neq n_2\)

\(<\bot, \sigma> \rightarrow <\bot, \sigma>\)

\(<3, \sigma> \rightarrow <\bot, \sigma>\)

Global Reduction Rules for IMP

\(<x := n, \sigma> \rightarrow <\text{skip, } \sigma[x := n]>\)

\(<\text{skip, } c, \sigma> \rightarrow <c, \sigma>\)

\(<\text{if true then } c_1 \text{ else } c_2, \sigma> \rightarrow <c_1, \sigma>\)

\(<\text{if false then } c_1 \text{ else } c_2, \sigma> \rightarrow <c_2, \sigma>\)

\(<\text{while } b \text{ do } c, \sigma> \rightarrow <\text{if } b \text{ then } c; \text{ while } b \text{ do } c \text{ else skip, } \sigma>\)

Global Reduction Pictorially

Step 1: Find The Redex

Step 2: Reduce The Redex

The Global Reduction Rule

- General idea of contextual semantics
  - Decompose the current expression into the redex-to-reduce-next and the remaining program
  - The remaining program is called the context
  - Reduce the redex “r” to some other expression “e”
  - The resulting (reduced) expression consists of “e” with the original context
Global Reduction Pictorially

Context
... x := 2+2 redex ; ...

Step 1: Find The Redex
Step 2: Reduce The Redex

4 reduced

Global Reduction Pictorially

Context
... x := 4 ; print x

Step 1: Find The Redex
Step 2: Reduce The Redex
Step 3: Replace It In The Context

The Global Reduction Rule

- We use $H$ to range over contexts
- We write $H[r]$ for the expression obtained by placing redex $r$ in context $H$

- Now we can define a small step

$$<r, \sigma> \rightarrow <e, \sigma'>$$

$$<H[r], \sigma> \rightarrow <H[e], \sigma'>$$

Context Terminology and Notation

- A context is also called an "expression with a hole"
- The marker $\bullet$ is sometimes called a hole
- $H[r]$ is the expression obtained from $H$ by replacing $\bullet$ with the redex $r$

Contexts

- A context is like an expression (or command) with a marker $\bullet$ in the place where the redex goes
- Examples:
  - To evaluate "$(1 + 3) + 2" we use the redex $1+3$ and the context "+$2"
  - To evaluate "if $x > 2$ then $c_1$ else $c_2$" we use the redex $x$ and the context "if $\bullet > 2$ then $c_1$ else $c_2$"

Contextual Semantics Example

- $x := 1 ; x := x + 1$ with initial state $[x:=0]$

<table>
<thead>
<tr>
<th>&lt;Comm, State&gt;</th>
<th>Redex $\bullet$</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := 1 ; x := x+1, [x := 0]$</td>
<td>$x := 1$</td>
<td>$\bullet : x := x+1$</td>
</tr>
<tr>
<td>$\langle \text{skip}, x := x+1, [x := 1]\rangle$</td>
<td>skip: $x := x+1$</td>
<td>$\bullet$</td>
</tr>
<tr>
<td>$x := x+1, [x := 1]$</td>
<td>$x$</td>
<td>$x := \bullet + 1$</td>
</tr>
</tbody>
</table>

What happens next?
Contextual Semantics Example

- x := 1; x := x + 1 with initial state [x:=0]

<table>
<thead>
<tr>
<th>&lt;Comm, State&gt;</th>
<th>Redex</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;skip; x := x+1, [x := 0]&gt;</td>
<td>x := 1</td>
<td>•: x := x+1</td>
</tr>
<tr>
<td>&lt;skip; x := x+1, [x := 1]&gt;</td>
<td>skip; x := x+1</td>
<td>•</td>
</tr>
<tr>
<td>&lt;x := x+1, [x := 1]&gt;</td>
<td>x</td>
<td>x := • + 1</td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Defining Contexts

- Contexts are defined by a grammar:
  \[ H ::= • | n + H \]
  \[ H + e | ... \]
  \[ | x := H \]
  \[ | if H then c_1 else c_2 \]
  \[ | H; c \]

- A context has exactly one • marker
- Does this say something about order of evaluation?

What's in a context?

- Contexts specify precisely 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 = • and r = 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 the decomposition is done recursively
  - Check that in each case the solution is unique

Unique Next Redex:

“Proof” By Handwaving Examples

- e.g., c = "c_1; c_2" - either
  - c_1 = skip and then c = H[skip; c_2] with H = •
  - or c_1 ≠ skip and then c_1 = H[r]; so c = H[r] with H = c_2
- e.g., 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] with H = if H then c_1 else c_2
Context Decomposition Theorem

- If \( c \) is not "skip" then there exist unique \( H \) and \( r \) such that \( c = H[r] \)
  - "Exist" means progress
  - "Unique" means determinism

What if we want short-circuit evaluation of \( \land \)?

- Define the following contexts, redexes and local reduction rules:
  
  \[
  H ::= \ldots \mid H \land b \\
  r ::= \ldots \mid true \land b \mid false \land b \\
  <true \land b, \sigma> \rightarrow <b, \sigma> \\
  <false \land b, \sigma> \rightarrow <false, \sigma>
  \]
  
  - the local reduction kicks in before \( b_2 \) is evaluated

Summary: Contextual Operational Semantics

- Can view \( \bullet \) as representing the program counter
- The advancement rules for \( \bullet \) are non-trivial
  - At each step the entire command is decomposed
  - This makes contextual semantics inefficient to implement directly

- The major advantage of contextual semantics: allows a mix of local and global reduction rules
  - For IMP we have only local reduction rules: only the redex is reduced
  - Sometimes it is useful to work on the context too

Reading Real-World Examples

- Cobbe and Felleisen, POPL 2005
- Small-step contextual op. sem. for Java
- Their rule for object field access:
  
  \[
  P \vdash <E(obj.field), S> \rightarrow <E[F(fd)], S> \quad \text{where } F = \text{fields}(\sigma(obj)) \text{ and } fd \in \text{dom}(F)
  \]
  
  - They use "E" for context, we use "H"
  - They use "S" for state, we use "\( \sigma \)"

Lost In Translation

- \( P \vdash <H[obj.field], \sigma> \rightarrow <H[F(fd)], \sigma> \)
  
  \[
  \text{where } F = \text{fields}(\sigma(obj)) \text{ and } fd \in \text{dom}(F)
  \]
  
  - They have "\( P \vdash \)" but that just means "it can be proved in our system given \( P \)"
  
  - \( <H[obj.field], \sigma> \rightarrow <H[F(fd)], \sigma> \)
    
    \[
    \text{where } F = \text{fields}(\sigma(obj)) \text{ and } fd \in \text{dom}(F)
    \]
Lost In Translation

• \(<H[\text{obj fd}],\sigma> \rightarrow <H[F(fd)],\sigma>\)
  where \(F=\text{fields}(\sigma(\text{obj}))\) and \(fd \in \text{dom}(F)\)
• They model objects (like \(\text{obj}\)), but we do not (yet); let’s just make \(fd\) a variable:
  • \(<H[fd],\sigma> \rightarrow <H[F(fd)],\sigma>\)
    where \(F=\sigma\) and \(fd \in L\)
• That’s just our variable-lookup rule:
  • \(<H[fd],\sigma> \rightarrow <H[\sigma(fd)],\sigma>\) (when \(fd \in L\))

Commentary

Commentary: Provability

• Given a proof system, e.g., \(<e, \sigma> \Downarrow n\) is **provable** if there exists a well-formed derivation with \(<e, \sigma> \Downarrow n\) as its conclusion
  - “well-formed” = “every step in the derivation is a valid instance of one of the rules of inference for this system”
• We would **like** truth and provability to be closely related

Truth?

• We will not formally define “truth” yet
• Instead we appeal to your intuition
  - \(<2+2, \sigma> \Downarrow 4\) — should be true
  - \(<2+2, \sigma> \Downarrow 5\) — should be false
• Discussion Question: How might we define truth?

Discussion: Defining truth

Completeness

• A proof system (like our operational semantics) is **complete** if every true judgment is provable.
• If we **replaced** the subtract rule with:
  \[
  \frac{<e_1, \sigma> \Downarrow n \quad <e_2, \sigma> \Downarrow 0}{<e_1 - e_2, \sigma> \Downarrow n}
  \]
• Our op. sem. would be **incomplete**:
  \(<4-2, \sigma> \Downarrow 2\) — true but not provable
Consistency or Soundness

- A proof system is **consistent** (or **sound**) if every provable judgment is true.
- If we **replaced** the subtract rule with:
  
  \[
  \frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 - e_2, \sigma \rangle \Uparrow n_1 + 3}
  \]
- Our op. sem. would be **inconsistent** (or **unsound**):
  - \(\langle 6 - 1, \sigma \rangle \Downarrow 9\) -- false but provable

Desired Traits

- Typically an operational semantics is always **complete** (unless you forget a rule)
- If you are not careful, however, your system may be **unsound**
- Usually that is **very bad**
  - A paper with an unsound type system is usually rejected
  - Papers often prove (sketch) that a system is sound
- In this class your work should be complete and **consistent** (e.g., on homework problems)

For Next Time

- Homework 1, due Thu
- Read Winskel, Chapter 3
  - Optional: additional background
  - Optional: more details
  - see web page