Operational Semantics

• Small-Step Semantics
• Large-Step Opsem Commentary
• Small-Step Contextual Semantics

Today’s Cunning Plan
• Review, Truth, and Provability
• Applications

Summary - Semantics
• A formal semantics is a system for assigning meanings to programs.
• For now, programs are IMP commands and expressions.
• In operational semantics the meaning of a program is “what it evaluates to.”
• Any opsem system gives rules of inference that tell you how to evaluate programs.

Summary - Judgments
• Rules of inference allow you to derive judgments (“something that is knowable”) like
  \(<e, \sigma> \downarrow n\)
  - In state \(\sigma\), expression \(e\) evaluates to \(n\)
  \(<c, \sigma> \downarrow \sigma'\)
  - After evaluating command \(c\) in state \(\sigma\) the new state will be \(\sigma'\)
• State \(\sigma\) maps variables to values (\(\sigma : L \rightarrow Z\))
• Inferences equivalent up to variable renaming:
  \(<c, \sigma \uplus \sigma' \equiv \equiv \equiv c', \sigma' \uplus \sigma_9>\)

Summary - Rules
• Rules of inference list the hypotheses necessary to arrive at a conclusion
  \(<x, \sigma> \uplus \sigma(x)\)
  \(<e_1, \sigma \downarrow n_1, e_2, \sigma \downarrow n_2\)
  \(<e_1 - e_2, \sigma \downarrow n_1 \text{ minus } n_2\).
• A derivation involves interlocking (well-formed) instances of rules of inference
  \(<4, \sigma_5 \downarrow 4\)
  \(<2, \sigma_5 \downarrow 2\)
  \(<4*2, \sigma_5 \downarrow 8\)
  \(<6, \sigma_5 \downarrow 6\)
  \(<(4*2) - 6, \sigma_5 \downarrow 2\)

Provability
• Given an opsem system, \(<e, \sigma> \uplus n\) is provable if there exists a well-formed derivation with \(<e, \sigma> \uparrow n\) as its conclusion.
  - “well-formed” = “every step in the derivation is a valid instance of one of the rules of inference for this opsem system.”
  - “\(\vdash \vdash \vdash\) \(<e, \sigma> \uplus n\)” = “it is provable that \(<e, \sigma> \uplus n\)”
• We would like truth and provability to be closely related.
Truth?
• “A Vorlon said understanding is a three-edged sword. Your side, their side and the truth.”
  - Sheridan, Into The Fire
• 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

Completeness
• A proof system (like our operational semantics) is complete if every true judgment is provable.
• If we replaced the subtract rule with:
  \[ \frac{\langle e_1, \sigma \rangle \downarrow n \quad \langle e_2, \sigma \rangle \downarrow 0}{\langle e_1 - e_2, \sigma \rangle \downarrow n} \]
• Our opsem would be incomplete:
  \( <4-2, \sigma> \downarrow 2 \) -- true but not provable

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

Desired Traits
• Typically a system (of 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
  - Recent research (e.g., Engler, ESP) into useful but unsound systems exists, however
• In this class your work should be complete and consistent (e.g., on homework problems)

With That In Mind
• We now return to opsem for IMP

\[ \frac{\langle e, \sigma \rangle \downarrow n}{\langle x := e, \sigma \rangle \downarrow \sigma[x := n]} \]
\[ \text{Def: } \sigma[x := n](x) = n \quad \sigma[x := n](y) = \sigma(y) \]
\[ \frac{\langle b, \sigma \rangle \downarrow \text{false}}{\langle \text{while } b \text{ do } c, \sigma \rangle \downarrow \sigma} \]
\[ \frac{\langle b, \sigma \rangle \downarrow \text{true} \quad \langle c; \text{while } b \text{ do } c, \sigma \rangle \downarrow \sigma'}{\langle \text{while } b \text{ do } c, \sigma \rangle \downarrow \sigma'} \]

Command Evaluation Notes
• The order of evaluation is important
  - \( c_1 \) is evaluated before \( c_2 \) in \( c_1; c_2 \)
  - \( c_2 \) is not evaluated in “if true then \( c_1 \) else \( c_2 \)”
  - \( c \) is not evaluated in “while false do \( c \)”
  - \( b \) is evaluated first in “if \( b \) then \( c_1 \) else \( c_2 \)”
  - this is explicit in the evaluation rules
• Conditional constructs (e.g., \( b_1 \lor b_2 \)) have multiple evaluation rules
  - but only one can be applied at one time

“A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines.”
  -- Ralph Waldo Emerson, Essays. First Series. Self-Reliance.
Command Evaluation Trials

- The evaluation rules are **not syntax-directed**
  - See the rules for `while, ∧`
  - The evaluation might not terminate
- Recall: the evaluation rules suggest an interpreter
- Natural-style semantics has two big disadvantages (continued ...)

Disadvantages of Natural-Style Operational Semantics

- It is hard to talk about commands whose evaluation does not terminate
  - i.e., when there is no $\sigma'$ such that $\langle c, \sigma \rangle \Downarrow \sigma'$
  - But that is true also of ill-formed or erroneous commands (in a richer language!)
- It does not give us a way to talk about intermediate states
  - Thus we cannot say that on a parallel machine the execution of two commands is interleaved (= no modeling threads)

Semantics Solution

- **Small-step semantics** addresses these problems
  - Execution is modeled as a (possible infinite) sequence of states
- Not quite as easy as large-step natural semantics, though
- **Contextual semantics** is a small-step semantics where the atomic execution step is a **rewrite** of the program

Contextual Semantics

- We will define a relation $\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$
  - $c'$ is obtained from $c$ 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")

Contextual Derivations

- In small-step contextual semantics, derivations are not tree-structured
- A **contextual semantics derivation** is a sequence (or list) of atomic rewrites:

  $\langle x+(7-3), \sigma \rangle \rightarrow \langle x+(4), \sigma \rangle \rightarrow \langle 5+4, \sigma \rangle \rightarrow \langle 9, \sigma \rangle$

  What is an Atomic Reduction?

- What is an atomic reduction step?
  - Granularity is a choice of the semantics designer
- How to select the next reduction step, when several are possible?
  - This is the order of evaluation issue

What is an Atomic Reduction?

- Granularity is a choice of the semantics designer
- 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 (transformed) in one atomic step.
- Redexes are defined via a grammar:
  \[
  r ::= x \quad (x \in L) \\
  | n_1 + n_2 \\
  | x := n \\
  | \text{skip;} \; c \\
  | \text{if true then} \; c_1 \; \text{else} \; c_2 \\
  | \text{if false then} \; c_1 \; \text{else} \; c_2 \\
  | \text{while} \; b \; \text{do} \; c
  \]
- For brevity, we mix exp and command redexes.
- Note that \((1 + 3) + 2\) is not a redex, but \(1 + 3\) is.

Local Reduction Rules for IMP

- One for each redex: \( \langle r, \sigma \rangle \rightarrow \langle e, \sigma' \rangle \)
  - means that in state \( \sigma \), the redex \( r \) can be replaced in one step with the expression \( e \)
  - \( \langle x, \sigma \rangle \rightarrow \langle \sigma(x), \sigma \rangle \)
  - \( \langle n_1 + n_2, \sigma \rangle \rightarrow \langle n, \sigma \rangle \) where \( n = n_1 + n_2 \)
  - \( \langle n_1 = n_2, \sigma \rangle \rightarrow \langle \text{true}, \sigma \rangle \) if \( n_1 = n_2 \)
  - \( \langle x := n, \sigma \rangle \rightarrow \langle \text{skip}, \sigma[x := n] \rangle \)
  - \( \langle \text{skip;} \; c, \sigma \rangle \rightarrow \langle c, \sigma \rangle \)
  - \( \langle \text{if true then} \; c_1 \; \text{else} \; c_2, \sigma \rangle \rightarrow \langle c_1, \sigma \rangle \)
  - \( \langle \text{if false then} \; c_1 \; \text{else} \; c_2, \sigma \rangle \rightarrow \langle c_2, \sigma \rangle \)
  - \( \langle \text{while} \; b \; \text{do} \; c, \sigma \rangle \rightarrow \langle \text{if } \; b \; \text{then } \; c; \; \text{while} \; b \; \text{do} \; c \; \text{else skip}, \; \sigma \rangle \)

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 a context.
- Reduce the redex “\( r \)” to some other expression “\( e \)”.
- The resulting (reduced) expression consists of “\( e \)” with the original context.

As A Picture (1)

1. Find The Redex
2. Reduce The Redex

As A Picture (2)

1. Find The Redex
2. Reduce The Redex

As A Picture (3)

1. Find The Redex
2. Reduce The Redex

Not happy? I’ll explain with pictures soon!

As A Picture (1)

1. Find The Redex

As A Picture (2)

1. Find The Redex

As A Picture (3)

1. Find The Redex
As A Picture (4)

( Context )

... x := 4

...

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

Contextual Analysis

- 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

If <r, σ> → <e, σ'>
then <H[r], σ> → <H[e], σ'>

Contexts

- A context is like an expression (or command) with a marker • 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 c₁ else c₂" we use the redex x and the context "if • > 2 then c₁ else c₂"

Context Terminology

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

"Avoid context and specifics; generalize and keep repeating the generalization."
— Jack Schwartz

Contextual Semantics Example

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

What happens next?

Contextual Semantics Example

<table>
<thead>
<tr>
<th>&lt;Comm, State&gt;</th>
<th>Redex •</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;x := 1; x := x+1, [x := 0]&gt;</td>
<td>x := 1 •; x := x+1</td>
<td></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 x := • + 1</td>
<td></td>
</tr>
<tr>
<td>&lt;x := 1 + 1, [x := 1]&gt;</td>
<td>1 + 1 x := •</td>
<td></td>
</tr>
<tr>
<td>&lt;x := 2, [x := 1]&gt;</td>
<td>x := 2 •</td>
<td></td>
</tr>
<tr>
<td>&lt;skip, [x := 2]&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More On Contexts

- Contexts are defined by a grammar:
  \[ H ::= \bullet \mid n + H \mid H + e \mid x := H \mid \text{if } H \text{ then } c \text{ else } c \mid H; c \]
- A context has exactly one • marker
- A redex is never a value

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 \( H = \bullet \) 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 \( H = n_1 + e_2 \) 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
  - \( c_1 = \text{skip} \) and then \( c = H[\text{skip}; c_2] \) with \( H = \bullet \)
  - or \( c_1 \neq \text{skip} \) and then \( c_1 = H[r] \); so \( c = H'[r] \) with \( H' = H; c_2 \)
- e.g. \( c = "\text{if } b \text{ then } c_1 \text{ else } c_2" \)
  - either \( b = \text{true} \) or \( b = \text{false} \) and then \( c = H[r] \)
    with \( H' = H; c_2 \)
  - or \( b \) is not a value and \( b = H[r] \); so \( c = H'[r] \) with \( H' = \text{if } H \text{ then } c_1 \text{ else } c_2 \)

Context Decomposition

- Decomposition theorem:
  \[ \text{If } c \text{ is not } "\text{skip}" \text{ then exist unique } H \text{ and } r \text{ such that } c = H[r] \]
  - "Exist" means progress
  - "Unique" means determinism

Short-Circuit Evaluation

- What if we want to express short-circuit evaluation of \( \land \) ?
  - Define the following contexts, redexes and local reduction rules
    \[ H ::= \ldots \mid H \land b_2 \]
    \[ r ::= \ldots \mid \text{true} \land b \mid \text{false} \land b \]
    \[ <\text{true} \land b, \sigma> \rightarrow <b, \sigma> \]
    \[ <\text{false} \land b, \sigma> \rightarrow <\text{false}, \sigma> \]
  - the local reduction kicks in before \( b_2 \) is evaluated

Contextual Semantics Summary

- Can view • as representing the program counter
- The advancement rules for • 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: it 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
  - We’ll do that when we study memory allocation, etc.
**Reading Real-World Examples**
- Cobbe and Felleisen, POPL 2005
- Small-step contextual opsem for Java
- Their rule for object field access:
  \[ P \vdash \langle E[\text{obj.fd}],S \rangle \rightarrow \langle E[F(fd)],S \rangle \]
  - where \( F = \text{fields}(S(\text{obj})) \) 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**
- \( P \vdash <H[\text{obj.fd}],\sigma> \rightarrow <H[F(fd)],\sigma> \)
  - where \( F = \text{fields}(\sigma(\text{obj})) \) and \( fd \in \text{dom}(F) \)
- They have “\( P \vdash \)” but that just means “it can be proved in our system given \( P \)”

**Lost In Translation 2**
- \( <H[\text{obj.fd}],\sigma> \rightarrow <H[F(fd)],\sigma> \)
  - 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 \( fd \in L \)
- Which is just our variable-lookup rule:
  - \( <H[fd],\sigma> \rightarrow <H[\sigma(fd)],\sigma> \) (when \( fd \in L \))

**“Sleep On It”**

---

**Homework**
- Straw Poll
- Homework 2 Out Today
  - Due Next Week
- Read Winskel Chapter 3
- Want an extra opsem review?
  - Natural deduction article
  - Plotkin Chapter 2
- Optional Philosophy of Science article