Announcements

- Homework 1 is out, due Feb 2
  - Coding part using OCaml
- Additional text for more background reading
  - Check the front page and the schedule page

Survey

+++++ classroom interaction/discussion
++ lecture style
++ PPT presentations clear and “not boring”
++ pace
+ top-down and bottom-up presentation
+ new technologies
+ math

-- forum requirement (vs. paper summaries?)
- prior PL background or no background (discussions)
- readings not completely understandable
- more advanced PL topics
- more languages, less tools
- slides before class

Today’s Plan

- Review of Proof Systems
- Truth and Provability
- Reductions, Redexes, and Contexts
- “Real World”

60 Second Review: Semantics

- A formal semantics is a system for how programs work/meaning.

- In operational semantics the meaning of a program is “what it evaluates to”.

- Any operational semantics gives rules of inference that recursive structure of evaluation of programs.

- A formal semantics is a system for assigning meanings to programs.

- In operational semantics the meaning of a program is “what it evaluates to”.

- Any operational semantics gives rules of inference that tell you how to evaluate programs.
Review: Judgments

- **Judgments** ("something that is knowable") are derived using rules of inference like $<e, \sigma> \Downarrow n$
  - In state $\sigma$, expression $e$ evaluates to $n$
  - 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: $<e, \sigma> \Downarrow \sigma' \iff <c', \sigma_i> \Downarrow \sigma_i$

Review: Rules of Inference

- We express the evaluation rules as **rules of inference** 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:
  $<e_1, \sigma> \Downarrow n_1$ $<e_2, \sigma> \Downarrow n_2$
  $\Downarrow <e_1 + e_2, \sigma> \Downarrow n_1 + n_2$
  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. $e_1 + e_2 \Downarrow n$
  - 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

Summary: Proof Systems

- **Rules of inference** list the hypotheses necessary to arrive at a conclusion
  $<x, \sigma> \Downarrow \sigma(x)$ $<e_1, \sigma> \Downarrow n_1$ $<e_2, \sigma> \Downarrow n_2$
  $\Downarrow <e_1 - e_2, \sigma> \Downarrow n_1 - n_2$
- A **derivation** involves interlocking (well-formed) instances of rules of inference
  $<4, \sigma> \Downarrow 4$ $<2, \sigma> \Downarrow 2$
  $\Downarrow <4*2, \sigma> \Downarrow 8$ $<6, \sigma> \Downarrow 6$
  $\Downarrow <(4*2) - 6, \sigma> \Downarrow 2$

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

End of Review
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

- "follows the laws of nature"?
- baseline "obvious" true
  - system looks?
  - problem: provably = truth

Completeness

- A proof system (like our operational semantics) is complete if every true judgment is provable.
- If we replaced the subtract rule with:
  \[
  \begin{align*}
  \langle e_1, \sigma \rangle & \Downarrow n_1, \\
  \langle e_2, \sigma \rangle & \Downarrow 0, \\
  \langle e_1 - e_2, \sigma \rangle & \Downarrow n
  \end{align*}
  \]
- 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:
  \[
  \begin{align*}
  \langle e_1, \sigma \rangle & \Downarrow n_1, \\
  \langle e_2, \sigma \rangle & \Downarrow n_2, \\
  \langle e_1 - e_2, \sigma \rangle & \Downarrow n_1 + 3
  \end{align*}
  \]
- Our op. sem. would be inconsistent (or unsound):
  - \( <6-1, \sigma> \Downarrow 9 \) -- 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)

Returning to Our Op. Sem. for IMP

\[
\begin{align*}
\langle e, \sigma \rangle & \Downarrow n \\
\langle x := e, \sigma \rangle & \Downarrow \sigma[x := n] \\
\langle b, \sigma \rangle & \Downarrow false \\
\langle \text{while } b \text{ do } c, \sigma \rangle & \Downarrow \sigma \\
\langle b, \sigma \rangle & \Downarrow true <c; \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma'
\end{align*}
\]
Returning to Our Op. Sem. for IMP

\[
\begin{align*}
<\text{skip}, \sigma> \Downarrow \sigma' & \quad <c_1, \sigma> \Downarrow \sigma' \\
<\text{if } b \text{ then } c_1 \text{ else } c_2, \sigma> \Downarrow \sigma' & \quad <b, \sigma> \Downarrow \text{true} \\
\end{align*}
\]

Observations: Command Evaluation

• 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 have multiple evaluation rules
  - But only one can be applied at one time

Disadvantages of Natural-Style or Big-Step Operational Semantics

• It is hard to talk about commands whose evaluation does not terminate
  - i.e., when there is no \(\sigma’\) such that \(<c, \sigma> \Downarrow \sigma’\)
  - 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 of threads)

Solution

• Small-step operational 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 operational semantics is a small-step operational semantics where the atomic execution step is a rewrite of the program
Contextual Operational Semantics

- We define a transition relation
  \[ \langle c, \sigma \rangle \rightarrow \rightarrow \rightarrow \rightarrow \langle c', \sigma' \rangle \]
  - "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")

Small-Step Operational Semantics

- We can view an evaluation of an expression or a command as a sequence of atomic rewrites:
  \[ \langle x - (7 - 3), \sigma \rangle \rightarrow \langle x - (4), \sigma \rangle \rightarrow \langle 5 + 4, \sigma \rangle \rightarrow \langle 9, \sigma \rangle \]
  \( \sigma(x) = 5 \)

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:
  \[ r ::= x \quad (x \in \mathcal{L}) \]
  \[ | n_1 + n_2 | n_1 = n_2 | ... \]
  \[ | x := n \]
  \[ | \text{skip}; c \]
  \[ | \text{if true then } c_1 \text{ else } c_2 \]
  \[ | \text{if false then } c_1 \text{ else } c_2 \]
  \[ | \text{while } b \text{ do } c \]
- For brevity, we mix exp and com redexes

Redex or Not?

- 1 + 3 \[ \text{Yes} \]
- \((1 + 3) + 2 \) \[ \text{No} \]
- \( x \) \[ \text{Yes} \]
- 4 \[ \text{No} \]
  - skip \[ \text{No} \]
  - if \( x \leq 0 \) then \( x := -x \) else skip \[ \text{No} \]
  - while \( x \leq n \) do \( x := x + 1 \) \[ \text{Yes} \]

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 \( e \)
  (or command \( c \))

\[ \langle x, \sigma \rangle \rightarrow \langle \sigma(x), \sigma \rangle \]
\[ \langle n_1 + n_2, \sigma \rangle \rightarrow \langle n, \sigma \rangle \quad \text{where } n = n_1 + n_2 \]
\[ \langle n_1 = n_2, \sigma \rangle \rightarrow \langle \text{true}, \sigma \rangle \quad \text{if } n_1 = n_2 \]
\[ \langle n_1 = n_2, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle \quad \text{if } n_1 \neq n_2 \]
Local Reduction Rules for IMP

\[
\begin{align*}
<x := n, \sigma> & \rightarrow <\text{skip}, \sigma[x := n]> \\
<\text{skip}; c, \sigma> & \rightarrow <c, \sigma>
\end{align*}
\]

\[
\begin{align*}
<\text{if true then } c_1 \text{ else } c_2, \sigma> & \rightarrow <c_1, \sigma> \\
<\text{if false then } c_1 \text{ else } c_2, \sigma> & \rightarrow <c_2, \sigma>
\end{align*}
\]

\[
\begin{align*}
<\text{while } b \text{ do } c, \sigma> & \rightarrow <\begin{align*}
\text{if } b \text{ then } c_1 \text{ while } b \text{ do } c \text{ else } \text{skip}, \sigma
\end{align*}>
\end{align*}
\]

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

Global Reduction Pictorially

Context

\[
\begin{align*}
\text{x := 2+2 ;}
\end{align*}
\]

Step 1: Find The Redex

Step 2: Reduce The Redex

Global Reduction Pictorially

Context

\[
\begin{align*}
\text{x := 2+2 redex ;}
\end{align*}
\]

Step 1: Find The Redex

Step 2: Reduce The Redex
Global Reduction Pictorially

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

- 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 and Notation

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

Contextual Semantics Example

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

<table>
<thead>
<tr>
<th>Comm, State</th>
<th>Redex •</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1 ; x := x+1, [x := 0]</td>
<td>x := 1</td>
<td>• ; x := x+1</td>
</tr>
<tr>
<td>skip; x := x+1, [x := 1]</td>
<td>skip; x := x+1</td>
<td>•</td>
</tr>
<tr>
<td>x := x+1, [x := 1]</td>
<td>x</td>
<td>x := • + 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>Comm, State</th>
<th>Redex •</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1 ; x := x+1, [x := 0]</td>
<td>x := 1</td>
<td>• ; x := x+1</td>
</tr>
<tr>
<td>skip; x := x+1, [x := 1]</td>
<td>skip; x := x+1</td>
<td>•</td>
</tr>
<tr>
<td>x := x+1, [x := 1]</td>
<td>x</td>
<td>x := • + 1</td>
</tr>
<tr>
<td>x := 1 + 1 , x := 1</td>
<td>1+1</td>
<td>x := •</td>
</tr>
<tr>
<td>y := 2 , x := 1</td>
<td>y := 2</td>
<td>•</td>
</tr>
<tr>
<td>skip , x := 2</td>
<td>x := 2</td>
<td>•</td>
</tr>
</tbody>
</table>
Contextual Semantics Example

\[\begin{array}{|c|c|}
\hline
\text{Context} & \text{Redex} \bullet \text{Context} \\
\hline
\langle x := 1; x := x + 1, [x := 0] \rangle & x := 1 \bullet x := x + 1 \\
\langle \text{skip}; x := x + 1, [x := 1] \rangle & \text{skip}; x := x + 1 \bullet \\
\langle x := x + 1, [x := 1] \rangle & x \bullet x := x + 1 \\
\langle x := 1 + 1, [x := 1] \rangle & 1 + 1 \bullet x := \\
\langle x := 2, [x := 1] \rangle & x := 2 \bullet \\
\langle \text{skip}, [x := 2] \rangle & \\
\hline
\end{array}\]

Defining Contexts

- Contexts are defined by a grammar:
  \[H ::= \bullet | n + H \]
  \[| H + e | \ldots \]
  \[| x := H \]
  \[| \text{if } H \text{ then } c_1 \text{ else } c_2 \]
  \[| H; c\]
- A context has exactly one \(\bullet\) marker
- Does this say something about order of evaluation?

What’s in a context?

- Contexts specify precisely how to find the next redex
  - Consider \(e_1 \cdot e_2\) and its decomposition as \(H[r]\)
  - If \(e_1\) is \(n_1\) and \(e_2\) is \(n_2\) then \(H = \bullet\) and \(r = n_1 \cdot n_2\)
  - If \(e_1\) is \(n_1\) and \(e_2\) is not \(n_1\) then \(H = n_1 \cdot H_r\) and \(e_2 = H_2[r]\)
  - If \(e_1\) is not \(n_1\) then \(H = H_1 \cdot e_2\) 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 \(c_1 = \text{skip and then } c_1 = H[r]; \text{ so } c = H[r]\)
  - 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 \(H = \bullet\)
  - or \(b\) is not a value and \(b = H[r]; \text{ so } c = H[r]\)
  - with \(H = \text{if } H \text{ then } c_1 \text{ else } c_2\)

Context Decomposition Theorem

- If \(c\) is not “\text{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:
  \[H ::= \ldots | H \land b\]
  \[r ::= \ldots | \text{true \land } b | \text{false \land } b\]
  \[< \text{true \land } b, a> \rightarrow < b, a>\]
  \[< \text{false \land } b, a> \rightarrow < \text{false}, a>\]
- the local reduction kicks in before \(b_2\) is evaluated
What if we want short-circuit evaluation of \( \land \)?

- Define the following contexts, redexes and local reduction rules:

\[
H ::= \ldots | H \land b \\
r ::= \ldots | \text{true} \land b | \text{false} \land b \\
\langle \text{true} \land b, \sigma \rangle \rightarrow \langle b, \sigma \rangle \\
\langle \text{false} \land b, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle
\]

- The local reduction kicks in before \( b \) 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:

\[
P \vdash \langle H[\text{obj.fd}], \sigma \rangle \rightarrow \langle H[F(fd)], \sigma \rangle
\]

where \( F = \text{fields}(\sigma(\text{obj})) \) 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

- \( P \vdash \langle H[\text{obj.fd}], \sigma \rangle \rightarrow \langle H[F(fd)], \sigma \rangle \)
  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 \( P \)"

- \( \langle H[\text{obj.fd}], \sigma \rangle \rightarrow \langle H[F(fd)], \sigma \rangle \)
  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:

\[
\langle H[fd], \sigma \rangle \rightarrow \langle H[F(fd)], \sigma \rangle
\]

where \( F = \sigma \) and \( fd \in L \)
- That's just our variable-lookup rule:

\[
\langle H[fd], \sigma \rangle \rightarrow \langle H[\sigma(fd)], \sigma \rangle \quad (\text{when } fd \in L)
\]

For Next Time

- Homework 1, due Mon Feb 2
- Read Winskel, Chapter 3
  - Optional: additional background
  - Optional: more details
  - see web page