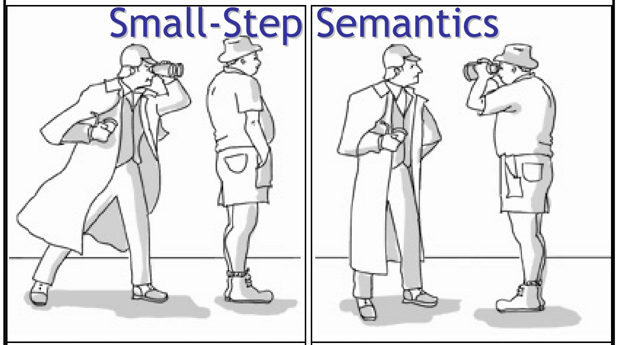
Operational Semantics
Small-Step Semantics



Sherlock saw the man using binoculars.

Sherlock saw the man using binoculars.

Today's Cunning Plan

- Review, Truth, and Provability
- Large-Step Opsem Commentary
- Small-Step Contextual Semantics
  - Reductions, Redexes, and Contexts
- Applications
- (Induction)

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

$\langle e, \sigma\rangle \Downarrow n$

  - In state σ, expression e evaluates to n

$\langle c, \sigma\rangle \Downarrow \sigma'$

  - After evaluating command c in state σ the new state will be σ'

- State σ maps variables to values ($\sigma : L \to Z$)
- Inferences equivalent up to variable renaming:

$\langle c, \sigma\rangle \Downarrow \sigma' \quad == \quad \langle c', \sigma_7\rangle \Downarrow \sigma_8$

Summary - Rules

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

$$\frac{}{\langle x, \sigma\rangle \Downarrow \sigma(x)} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 - e_2, \sigma\rangle \Downarrow n_1 \text{ minus } n_2}$$

- A derivation involves interlocking instances of rules of inference

$$\dfrac{\dfrac{\langle 4, \sigma_3\rangle \Downarrow 4 \quad \langle 2, \sigma_3\rangle \Downarrow 2}{\langle 4*2, \sigma_3\rangle \Downarrow 8} \qquad \langle 6, \sigma_3\rangle \Downarrow 6}{\langle (4*2) - 6, \sigma_3\rangle \Downarrow 2}$$

Provability

- Given an opsem system, $\langle e, \sigma\rangle \Downarrow n$ is provable if there exists a well-formed derivation with $\langle e, \sigma\rangle \Downarrow n$ as its conclusion
  - "well-formed" = "every step in the derivation is a valid instance of one of the rules of inference for this opsem system"
  - "$\vdash \langle e, \sigma\rangle \Downarrow n$" = "it is provable that $\langle e, \sigma\rangle \Downarrow n$"
- 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
  - $\langle 2+2, \sigma\rangle \Downarrow 4$ -- should be true
  - $\langle 2+2, \sigma\rangle \Downarrow 5$ -- 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:

$$\frac{\langle e_1, \sigma\rangle \Downarrow n \qquad \langle e_2, \sigma\rangle \Downarrow 0}{\langle e_1 - e_2, \sigma\rangle \Downarrow n}$$

- Our opsem would be incomplete:
  - $\langle 4\text{-}2, \sigma\rangle \Downarrow 2$ -- 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:

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \qquad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 - e_2, \sigma\rangle \Downarrow n_1 + 3}$$

- Our opsem would be inconsistent (or unsound):
  - $\langle 6\text{-}1, \sigma\rangle \Downarrow 9$ -- 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

$$\frac{\langle e, \sigma\rangle \Downarrow n}{\langle x := e, \sigma\rangle \Downarrow \sigma[x := n]}$$

Def: $\sigma[x{:=}\ n](x) = n$
$\sigma[x{:=}\ n](y) = \sigma(y)$

$$\frac{\langle b, \sigma\rangle \Downarrow \text{false}}{\langle \text{while } b \text{ do } c, \sigma\rangle \Downarrow \sigma}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \text{true} \qquad \langle c;\ \text{while } b \text{ do } c, \sigma\rangle \Downarrow \sigma'}{\langle \text{while } b \text{ do } c, \sigma\rangle \Downarrow \sigma'}$$

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., $b_1 \vee b_2$) have multiple evaluation rules
  - but only one can be applied at one time

Command Evaluation Trials

- The evaluation rules are not syntax-directed
  - See the rules for while, $\wedge$
  - 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 σ' 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

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 $<c, \sigma> \to <c', \sigma'>$
  - 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:

$$<x+(7-3),\sigma> \to <x+(4),\sigma> \to <5+4,\sigma> \to <9,\sigma>$$

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

Redexes

- A redex is a syntactic expression or command that can be reduced (transformed) in one atomic step
- Defined as a grammar:

$$
\begin{aligned}
r ::=\ & x && (x \in L)\\
\mid\ & n_1 + n_2 \\
\mid\ & x := n \\
\mid\ & \text{skip; } c \\
\mid\ & \text{if true then } c_1 \text{ else } c_2 \\
\mid\ & \text{if false then } c_1 \text{ else } c_2 \\
\mid\ & \text{while } b \text{ do } c
\end{aligned}
$$

- 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: $<r, \sigma> \to <e, \sigma'>$
  - means that in state σ, the redex r can be replaced in one step with the expression e

$<x, \sigma> \to <\sigma(x), \sigma>$

$<n_1 + n_2, \sigma> \to <n, \sigma>$ where $n = n_1 + n_2$

$<n_1 = n_2, \sigma> \to <\text{true}, \sigma>$ if $n_1 = n_2$

$<x := n, \sigma> \to <\text{skip}, \sigma[x := n]>$

$<\text{skip}; c, \sigma> \to <c, \sigma>$

$<\text{if true then } c_1 \text{ else } c_2, \sigma> \to <c_1, \sigma>$

$<\text{if false then } c_1 \text{ else } c_2, \sigma> \to <c_2, \sigma>$

$<\text{while } b \text{ do } c, \sigma> \to$

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

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)

(Context)

…

x := 2+2

…

Step 1: Find The Redex

As A Picture (2)

(Context)

…

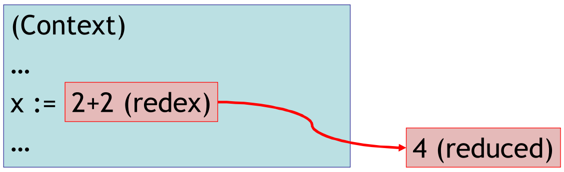
x := 2+2 (redex)

…

Step 1: Find The Redex

Step 2: Reduce The Redex

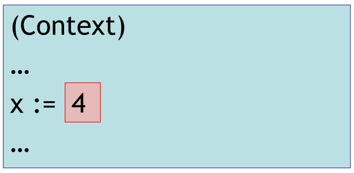
As A Picture (3)



Step 1: Find The Redex

Step 2: Reduce The Redex

As A Picture (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 $\langle r, \sigma\rangle \to \langle e, \sigma'\rangle$

then $\langle H[r], \sigma\rangle \to \langle H[e], \sigma'\rangle$

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 c_1 else c_2" we use the redex x and the context "if • > 2 then c_1 else c_2"

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

Contextual Semantics Example

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

<Comm, State>	Redex •	Context
<x := 1; x := x+1, [x := 0]>	x := 1	•; x := x+1
<skip; x := x+1, [x := 1]>	skip; x := x+1	•
<x := x+1, [x := 1]>	x	x := • + 1
What happens next?		

Contextual Semantics Example

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

<Comm, State>	Redex •	Context
<x := 1; x := x+1, [x := 0]>	x := 1	•; x := x+1
<skip; x := x+1, [x := 1]>	skip; x := x+1	•
<x := x+1, [x := 1]>	x	x := • + 1
<x := 1 + 1, [x := 1]>	1 + 1	x := •
<x := 2, [x := 1]>	x := 2	•
<skip, [x := 2]>		

More On 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
- 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 = • 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

- E.g. 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 = H'[r] with H' = 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

- Decomposition theorem:

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

Short-Circuit Evaluation

- What if we want to express short-circuit evaluation of $\wedge$?
  - Define the following contexts, redexes and local reduction rules

    H ::= ... | $H \wedge b_2$

    r ::= ... | true $\wedge$ b | false $\wedge$ b

    $\langle$true $\wedge$ b, $\sigma\rangle \to \langle$b, $\sigma\rangle$

    $\langle$false $\wedge$ b, $\sigma\rangle \to \langle$false, $\sigma\rangle$
  - the local reduction kicks in before b_2 is evaluated

Contextual Semantics Summary

- One can think of the • 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 is that 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

Real-World Example

- Cobbe and Felleisen, POPL 2005
- Small-step contextual opsem for Java
- Their rule for object field access:
- P ⊢ <E[obj.fd],S> → <E[F(fd)],S>
  - Where F=fields(S(obj)) and fd ∈ dom(F)

- They use “E” for context, we use “H”
- They use “S” for state, we use “σ”

Lost In Translation

- P ⊢ <H[obj.fd],σ> → <H[F(fd)],σ>
  - Where F=fields(σ(obj)) and fd ∈ dom(F)

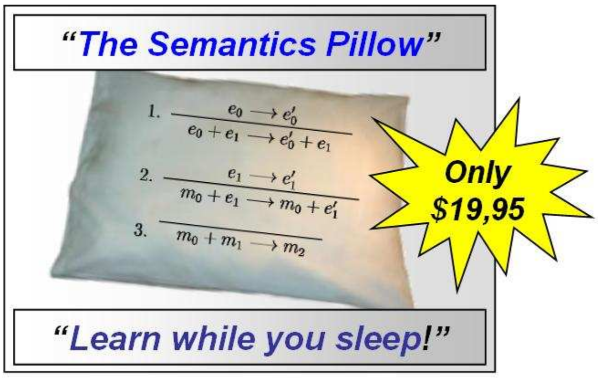
- They have “P ⊢”, but that just means “it can be proved in our system given P”

- <H[obj.fd],σ> → <H[F(fd)],σ>
  - Where F=fields(σ(obj)) and fd ∈ dom(F)

Lost In Translation 2

- <H[obj.fd],σ> → <H[F(fd)],σ>
  - Where F=fields(σ(obj)) and fd ∈ dom(F)
- They model objects (like obj), but we do not - let’s just make fd a variable:
- <H[fd],σ> → <H[F(fd)],σ>
  - Where F=σ and fd ∈ L
- Which is really just our rule:
- <H[fd],σ> → <H[σ(fd)],σ> (when fd ∈ L)

“Sleep On It”



Homework

- Straw Poll
- Homework 2 Out Today
  - Due Thursday, Feb 02
- Read Winskel Chapter 3
- Want an extra opsem review?
  - *Natural deduction* article
  - Plotkin Chapter 2
- Optional Philosophy of Science article