

Operational Semantics

- Winskel, Ch. 2
- Slonneger and Kurtz Ch 8.4, 8.5, 8.6

1

Operational vs. Axiomatic

- **Axiomatic semantics**
 - Describes properties of program state, using first-order logic
 - Concerned with constructing proofs for such properties
- **Operational semantics**
 - Explicitly describes the effects of program constructs on program state
 - Shows not only **what** the program does, but also **how** it does it

2

Defining an Operational Semantics

- Define an interpreter for the language
- Define a compiler for the language, plus an interpreter for the assembly language used
- Specify how the state changes as various statements are executed

3

Specification Language

- Should be high-level
- Should be concise
- Efficiency does not matter
- Examples
 - Post system (next slides)
 - Attribute grammar (seen earlier)
 - "Nice" expressive languages: e.g. ML, Prolog

4

IMP

- IMP: simple imperative language
 - Already used in the discussion of axiomatic semantics
- Only integer variables
- No procedures or functions
- No explicit var declarations

5

IMP Syntax

```
<c>1 ::= skip | <id> := <ae> | <c>2 ; <c>3
      | if <be> then <c>2 else <c>3
      | while <be> do <c>2
<ae>1 ::= <id> | <int> | <ae>2 + <ae>3
      | <ae>2 - <ae>3 | <ae>2 * <ae>3
<be>1 ::= true | false
      | <ae>1 = <ae>2 | <ae>1 < <ae>2
      | ¬ <be>2 | <be>2 ∧ <be>3
      | <be>2 ∨ <be>3
```

6

State

- State: a function σ from variable names to values
- E.g., program with 2 variables x, y
 $\sigma(x) = 9$
 $\sigma(y) = 5$
- For simplicity, we will only consider integer variables
 - σ : Variables $\rightarrow \{0, -1, +1, -2, 2, \dots\}$

7

Operational Semantics for IMP

- Post system (proof system)
- If the state is σ and expression e is evaluated, what is the resulting value?
 - $\langle ae, \sigma \rangle \rightarrow n$ for arithmetic expression
 - $\langle be, \sigma \rangle \rightarrow bv$ for boolean expressions
 - ae, be : parse trees; n : integer; bv : boolean
- If the state is σ and statement c is executed to termination, what is the resulting state?
 - $\langle c, \sigma \rangle \rightarrow \sigma'$ (c is a parse tree)

8

Evaluation of Arithmetic Expressions

$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1$

$\langle n, \sigma \rangle \rightarrow n$

$\langle X, \sigma \rangle \rightarrow \sigma(X)$

$\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1$
 $\langle a_0 + a_1, \sigma \rangle \rightarrow n$ n is the sum of n_0 and n_1

similarly for $a_0 - a_1$ and $a_0 * a_1$

E.g. if $\sigma(P) = 4$ and $\sigma(Q) = 6$, $\langle P+Q, \sigma \rangle \rightarrow 10$

9

Inference Rules

- Here again we represent the semantics with inference rules
 - Zero or more premises
 - Conclusion
 - Optional condition (shown to the right): the rule applies only if the condition is true
 - e.g. "n is the sum of n_0 and n_1 "
- Instances of such rules are applied for a given code fragment, in order to derive (prove) values and states

10

Level of Detail

- "n is the sum of n_0 and n_1 "
 - This assumes that "sum" is a primitive notion that we will not define
- In some cases, we may decide to define it precisely
 - e.g. "sum" is not trivial for roman numerals
 - or maybe if we are describing a low-level language for some hardware device
- In this class: we will not specify how addition is done

11

Evaluation of Boolean Expressions

$b ::= \text{true} \mid \text{false} \mid a_0 = a_1 \mid a_0 < a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$

$\langle \text{true}, \sigma \rangle \rightarrow \text{true}$ $\langle \text{false}, \sigma \rangle \rightarrow \text{false}$

$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 = a_1, \sigma \rangle \rightarrow \text{true}}$ n_0 and n_1 are equal

$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 = a_1, \sigma \rangle \rightarrow \text{false}}$ n_0 and n_1 are not equal

12

Evaluation of Boolean Expressions

$\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1$
 $\langle a_0 < a_1, \sigma \rangle \rightarrow \text{true}$ n_0 is less than n_1

$\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1$
 $\langle a_0 < a_1, \sigma \rangle \rightarrow \text{false}$ n_0 is greater than or equal to n_1

$\langle b, \sigma \rangle \rightarrow \text{true} \quad \langle b, \sigma \rangle \rightarrow \text{false}$
 $\langle \neg b, \sigma \rangle \rightarrow \text{false} \quad \langle \neg b, \sigma \rangle \rightarrow \text{true}$

13

Evaluation of Boolean Expressions

$\langle b_0, \sigma \rangle \rightarrow t_0 \quad \langle b_1, \sigma \rangle \rightarrow t_1$
 $\langle b_0 \wedge b_1, \sigma \rangle \rightarrow t$ t is true iff t_0 and t_1 are true

$\langle b_0, \sigma \rangle \rightarrow t_0 \quad \langle b_1, \sigma \rangle \rightarrow t_1$
 $\langle b_0 \vee b_1, \sigma \rangle \rightarrow t$ t is false iff t_0 and t_1 are false

How about short-circuit evaluation?

14

Short-circuit Evaluations

- $b_0 \wedge b_1$: if b_0 evaluates to false, no need to evaluate b_1
- $b_0 \vee b_1$: if b_0 evaluates to true, no need to evaluate b_1
- Most programming languages do this
- How do we represent this approach as inference rules?

15

Execution of Statements

- $\sigma[m/X]$ is the same as σ except for X
 - $\sigma[m/X](Y) = \sigma(Y)$ if Y is not X
 - $\sigma[m/X](X) = m$
 - Also written as $\sigma[X \leftarrow m]$

$\langle e, \sigma \rangle \rightarrow m$ $\langle \text{skip}, \sigma \rangle \rightarrow \sigma$

$\langle X := e, \sigma \rangle \rightarrow \sigma[m/X]$

$\langle c_0, \sigma \rangle \rightarrow \sigma'$ $\langle c_1, \sigma' \rangle \rightarrow \sigma''$

$\langle c_0; c_1, \sigma \rangle \rightarrow \sigma''$

16

Execution of Statements

$\langle b, \sigma \rangle \rightarrow \text{true}$ $\langle c_0, \sigma \rangle \rightarrow \sigma'$

$\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow \sigma'$

$\langle b, \sigma \rangle \rightarrow \text{false}$ $\langle c_1, \sigma \rangle \rightarrow \sigma'$

$\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow \sigma'$

$\langle b, \sigma \rangle \rightarrow \text{false}$

$\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma$

$\langle b, \sigma \rangle \rightarrow \text{true}$ $\langle c, \sigma \rangle \rightarrow \sigma'$ $\langle \text{while } b \text{ do } c, \sigma' \rangle \rightarrow \sigma''$

$\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma''$

17

Equivalence

- Expressions x and y are equivalent if for any σ and any z , $\langle x, \sigma \rangle \rightarrow z$ iff $\langle y, \sigma \rangle \rightarrow z$
 - e.g. $a+b$ is equivalent to $b-5+a+5$
- Statements x and y are equivalent if for any σ and σ' , $\langle x, \sigma \rangle \rightarrow \sigma'$ iff $\langle y, \sigma \rangle \rightarrow \sigma'$
 - e.g. statement " $c:=a+b; d:=c;$ " is equivalent to statement " $d:=b-5+a+5; c:=d;$ "
- Essential for ensuring the **correctness of compiler optimizations**
 - Optimized code vs. the original code

18

Example

- Loop peeling: transform "while b do c"
- **if b then** (c; **while b do c**) **else skip**
 - Take the first iteration out of the loop
 - Common compiler optimization
- Can we prove that this transformation is semantics-preserving?
 - i.e., are these statements equivalent?

19

First Direction

$\langle \text{while } b, \sigma \rangle \rightarrow \sigma'$ implies $\langle \text{if } b, \sigma \rangle \rightarrow \sigma'$

There must be some derivation, leading to

$\langle b, \sigma \rangle \rightarrow \text{false}$ (σ and σ' are the same state), or

$\langle b, \sigma \rangle \rightarrow \text{true}$ $\langle c, \sigma \rangle \rightarrow \sigma''$ $\langle \text{while } b \text{ do } c, \sigma'' \rangle \rightarrow \sigma'$

Case 1: $\langle b, \sigma \rangle \rightarrow \text{false}$, and $\langle \text{skip}, \sigma \rangle \rightarrow \sigma'$, so

$\langle \text{if } b \text{ then } \dots \text{ else skip}, \sigma \rangle \rightarrow \sigma'$

Case 2: $\langle b, \sigma \rangle \rightarrow \text{true}$ and $\langle c; \text{while } b, \sigma \rangle \rightarrow \sigma'$, so

$\langle \text{if } b \text{ then } c; \text{while } b \dots \text{ else } \dots, \sigma \rangle \rightarrow \sigma'$

20

Second Direction

$\langle \text{if } b, \sigma \rangle \rightarrow \sigma'$ implies $\langle \text{while } b, \sigma \rangle \rightarrow \sigma'$

There must be some derivation, leading to

$\langle b, \sigma \rangle \rightarrow \text{false}$ $\langle \text{skip}, \sigma \rangle \rightarrow \sigma'$ (so $\sigma = \sigma'$) or

$\langle b, \sigma \rangle \rightarrow \text{true}$ $\langle c; \text{while } b, \sigma \rangle \rightarrow \sigma'$

Case 1: $\langle b, \sigma \rangle \rightarrow \text{false}$, so $\langle \text{while } b \text{ do } \dots, \sigma \rangle \rightarrow \sigma'$

Case 2: $\langle b, \sigma \rangle \rightarrow \text{true}$ and $\langle c; \text{while } b, \sigma \rangle \rightarrow \sigma'$, so

must have had $\langle c, \sigma \rangle \rightarrow \sigma''$ and $\langle \text{while } b, \sigma'' \rangle \rightarrow \sigma'$,

and therefore $\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma'$

21

Another Example

- Partial redundancy elimination
 - In its general form, an advanced compiler optimization
- **if** b **then** x:=e1 **else** y:=e2 **fi**; x:=e1
- **if** b **then** x:=e1 **else** y:=e2; x:=e1; **fi**
- Under what conditions are these two code fragments semantically equivalent?
 - Try this at home ...

22

Yet Another Example

Claim: $\langle \text{while true do skip}, \sigma \rangle \rightarrow \sigma'$ cannot be derived.

Proof: suppose that a derivation $\langle \text{while...}, \sigma \rangle \rightarrow \sigma'$ exists. Consider a minimal length derivation.

The last step must be

$\langle \text{true}, \sigma \rangle \rightarrow \text{true} \quad \langle \text{skip}, \sigma \rangle \rightarrow \sigma' \quad \langle \text{while...}, \sigma' \rangle \rightarrow \sigma'$
 $\langle \text{while true do skip}, \sigma \rangle \rightarrow \sigma'$

But $\langle \text{skip}, \sigma \rangle \rightarrow \sigma'$ means σ and σ' are the same;
premise $\langle \text{while...}, \sigma' \rangle \rightarrow \sigma'$ means that the derivation is not minimal

23

Big-Step vs. Small-Step Semantics

- Until now: "coarse" semantics
 - Abstracts away some details about the individual steps taken during execution
 - "Big-step" semantics: based on the productions of the underlying grammar
- Alternative semantics: captures smaller steps in the execution
- Expressions: $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$
- Statements: $\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$

24

Small-Step Evaluation of Expressions

$\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle \quad \langle e', \sigma' \rangle \rightarrow \langle e'', \sigma'' \rangle$

$\langle e, \sigma \rangle \rightarrow \langle e'', \sigma'' \rangle$

$\langle X, \sigma \rangle \rightarrow \langle \sigma(X), \sigma \rangle$ (axiom)

Example: addition is done left to right

- The left argument is evaluated first, the right arguments is evaluated next
- Big-step semantics does not capture this

25

Small-Step Addition

$\langle a_0, \sigma \rangle \rightarrow \langle a_0', \sigma \rangle$

$\langle a_1, \sigma \rangle \rightarrow \langle a_1', \sigma \rangle$

$\langle a_0 + a_1, \sigma \rangle \rightarrow \langle a_0' + a_1, \sigma \rangle$

$\langle n + a_1, \sigma \rangle \rightarrow \langle n + a_1', \sigma \rangle$

$\langle n + m, \sigma \rangle \rightarrow \langle p, \sigma \rangle$ if the sum of m and n is p

- If one step in the evaluation of a_0 leads to a_0' , then one step in the evaluation of $a_0 + a_1$ leads to $a_0' + a_1$ (evaluate a_0 first)
- After a_0 is evaluated, evaluate a_1
- Using rules 2 and 3, evaluate the sum

26

Example

Evaluate $a+b+c$, with $\sigma(a)=1, \sigma(b)=2, \sigma(c)=4$

$\langle a, \sigma \rangle \rightarrow \langle 1, \sigma \rangle$ rule 1: $\langle a+b, \sigma \rangle \rightarrow \langle 1+b, \sigma \rangle$

$\langle b, \sigma \rangle \rightarrow \langle 2, \sigma \rangle$ rule 2: $\langle 1+b, \sigma \rangle \rightarrow \langle 1+2, \sigma \rangle$

rule 3: $\langle 1+2, \sigma \rangle \rightarrow \langle 3, \sigma \rangle$

transitivity: $\langle a+b, \sigma \rangle \rightarrow \langle 3, \sigma \rangle$

rule 1: $\langle a+b+c, \sigma \rangle \rightarrow \langle 3+c, \sigma \rangle$

$\langle c, \sigma \rangle \rightarrow \langle 4, \sigma \rangle$ rule 2: $\langle 3+c, \sigma \rangle \rightarrow \langle 3+4, \sigma \rangle$

rule 3: $\langle 3+4, \sigma \rangle \rightarrow \langle 7, \sigma \rangle$

transitivity: $\langle a+b+c, \sigma \rangle \rightarrow \langle 7, \sigma \rangle$

27

Small-Step Execution of Statements

$\langle c_0, \sigma \rangle \rightarrow \sigma' \quad \langle c_1, \sigma' \rangle \rightarrow \sigma''$ (big step)

$\langle c_0; c_1, \sigma \rangle \rightarrow \sigma''$

$\langle c_0, \sigma \rangle \rightarrow \langle c_0', \sigma' \rangle$ (small step)

$\langle c_0; c_1, \sigma \rangle \rightarrow \langle c_0'; c_1, \sigma' \rangle$

$\langle c_0, \sigma \rangle \rightarrow \sigma'$ (think of σ' as $\langle \text{empty}, \sigma' \rangle$)

$\langle c_0; c_1, \sigma \rangle \rightarrow \langle c_1, \sigma' \rangle$

28

Example

$\langle X:=5; Y:=1, \sigma \rangle \rightarrow ?$

$\langle X:=5, \sigma \rangle \rightarrow \sigma[5/X]$

therefore

$\langle X:=5; Y:=1, \sigma \rangle \rightarrow \langle Y:=1, \sigma[5/X] \rangle$

also $\langle Y:=1, \sigma[5/X] \rangle \rightarrow \sigma[5/X][1/Y]$

through transitivity

$\langle X:=5; Y:=1, \sigma \rangle \rightarrow \sigma[5/X][1/Y]$

29
