

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

[http://www.cs.uiuc.edu/class
/fa06/cs421/](http://www.cs.uiuc.edu/class/fa06/cs421/)

Based in part on slides by Mattox Beckman, as updated
by Vikram Adve and Gul Agha

Semantics

- Expresses the meaning of syntax
- Static semantics
 - Meaning based only on the form of the expression without executing it
 - Usually restricted to type checking / type inference

Dynamic semantics

- Method of describing meaning of executing a program
- Several different types:
 - Operational Semantics
 - Axiomatic Semantics
 - Denotational Semantics

Dynamic Semantics

- Different languages better suited to different types of semantics
- Different types of semantics serve different purposes

Operational Semantics

- Start with a simple notion of machine
- Describe how to execute (implement) programs of language on virtual machine, by describing how to execute each program statement (ie, following the *structure* of the program)
- Meaning of program is how its execution changes the state of the machine
- Useful as basis for implementations

Axiomatic Semantics

- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from *axioms* and *inference rules*
- Mainly suited to simple imperative programming languages

Axiomatic Semantics

- Used to formally prove a property (*post-condition*) of the *state* (the values of the program variables) after the execution of program, assuming another property (*pre-condition*) of the state before execution
- Written :
 {Precondition} Program {Postcondition}
- Source of idea of *loop invariant*

Denotational Semantics

- Construct a function \mathcal{M} assigning a mathematical meaning to each program construct
- Meaning function is compositional: meaning of construct built from meaning of parts
- Useful for proving properties of programs

Transition Semantics

- Form of operational semantics
- Describes how each program construct transforms machine state by *transitions*
- Rules look like
$$(C, m) \rightarrow (C', m')$$
- C, C' is code remaining to be executed
- m, m' represent the state/store/memory/environment
 - Partial mapping from identifiers to values
 - Sometimes m (or C) not needed
- Indicates exactly one step of computation

Expressions and Values

- Special class of expressions designated as *values*
 - Eg 2, 3 are values, but 2+3 is only an expression
- Memory only holds values
- Transitions stop when C is a value
- Value is the final *meaning* of original expression (in the given state)
- C, C' used for commands; E, E' for expressions; U, V for values

Simple Imperative Programming Language

- $I \in \text{Identifiers}$
- $N \in \text{Numerals}$
- $B ::= \text{true} \mid \text{false} \mid B \ \& \ B \mid B \ \text{or} \ B \mid \text{not } B$
 $\mid E < E \mid E = E$
- $E ::= N \mid I \mid E + E \mid E * E \mid E - E \mid -E$
- $C ::= \text{skip} \mid C; C \mid I ::= E$
 $\mid \text{if } B \text{ then } C \text{ else } C \text{ fi} \mid \text{while } B \text{ do } C \text{ od}$

Transitions for Expressions

- Identifiers: $(l, m) \rightarrow m(l)$
- Numerals are values: $(N, m) \rightarrow N$
- Notation - Function update:
- $m[l \leftarrow V] = \lambda y. \text{ if } y = l \text{ then } V \text{ else } m(y)$

Booleans:

- Values = {true, false}
- Operators: (short-circuit)

$$\begin{array}{l} (false \ \& \ B, m) \ \rightarrow \ false \\ (true \ \& \ B, m) \ \rightarrow \ B \end{array} \quad \frac{(B, m) \ \rightarrow \ (B'', m)}{(B \ \& \ B', m) \ \rightarrow \ (B'' \ \& \ B, m)}$$

$$\begin{array}{l} (true \ or \ B, m) \ \rightarrow \ true \\ (false \ or \ B, m) \ \rightarrow \ B \end{array} \quad \frac{(B, m) \ \rightarrow \ (B'', m)}{(B \ or \ B', m) \ \rightarrow \ (B'' \ or \ B, m)}$$

$$\begin{array}{l} (not \ true, m) \ \rightarrow \ false \\ (not \ false, m) \ \rightarrow \ true \end{array} \quad \frac{(B, m) \ \rightarrow \ (B', m)}{(not \ B, m) \ \rightarrow \ (not \ B', m)}$$

Relations

$$\frac{(E, m) \dashrightarrow (E'', m)}{(E \sim E', m) \dashrightarrow (E'' \sim E', m)}$$

$$\frac{(E, m) \dashrightarrow (E', m)}{(V \sim E, m) \dashrightarrow (V \sim E', m)}$$

$(U \sim V, m) \dashrightarrow$ true or false, depending on whether $U \sim V$ holds or not

Arithmetic Expressions

$$\frac{(E, m) \dashrightarrow (E'', m)}{(E \text{ op } E', m) \dashrightarrow (E'' \text{ op } E', m)}$$

$$\frac{(E, m) \dashrightarrow (E', m)}{(V \text{ op } E, m) \dashrightarrow (V \text{ op } E', m)}$$

$(U \text{ op } V, m) \dashrightarrow N$ where N is the specified value for $U \text{ op } V$

Commands - in English

- skip is done evaluating
- When evaluating an assignment, evaluate the expression first
- If the expression being assigned is already a value, update the memory with the new value for the identifier
- When evaluating a sequence, work on the first command in the sequence first
- If the first command evaluates to a new memory (ie completes), evaluate remainder with new memory

Commands

$$(\text{skip}, m) \dashrightarrow m$$
$$(E, m) \dashrightarrow (E', m)$$
$$\frac{(E, m) \dashrightarrow (E', m)}{(l ::= E, m) \dashrightarrow (l ::= E', m)}$$
$$(l ::= V, m) \dashrightarrow m[l \leftarrow V]$$
$$(C, m) \dashrightarrow (C'', m')$$
$$\frac{(C, m) \dashrightarrow (C'', m')}{(C; C', m) \dashrightarrow (C''; C', m')}$$
$$(C, m) \dashrightarrow m'$$
$$\frac{(C, m) \dashrightarrow m'}{(C; C', m) \dashrightarrow (C', m')}$$

If Then Else Command - in English

- If the boolean guard in and if_then_else is true, then evaluate the first branch
- If it is false, evaluate the second branch
- If the boolean guard is not a value, then start by evaluating it first.

If Then Else Command

$(\text{if true then } C \text{ else } C' \text{ fi, } m) \dashrightarrow (C, m)$

$(\text{if false then } C \text{ else } C' \text{ fi, } m) \dashrightarrow (C', m)$

$$\frac{(B, m) \dashrightarrow (B', m)}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi, } m) \dashrightarrow (\text{if } B' \text{ then } C \text{ else } C' \text{ fi, } m)}$$

While Command

(while B do C od, m)

--> (if B then C;while B do C od else skip fi, m)

In English: Expand a While into a test of the boolean guard, with the true case being to do the body and the try the while loop again, and the false case being to stop.

Example Evaluation

- First step:

(if $x > 5$ then $y := 2 + 3$ else $y := 3 + 4$ fi,
 { $x \rightarrow 7$ })
 $\rightarrow ?$

Example Evaluation

- First step:

$$(x > 5, \{x \rightarrow 7\}) \rightarrow ?$$

$$\begin{aligned} &(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\ &\quad \{x \rightarrow 7\}) \\ &\quad \rightarrow ? \end{aligned}$$

Example Evaluation

- First step:

$$\frac{\quad}{(x, \{x \rightarrow 7\}) \rightarrow 7}$$

$$(x > 5, \{x \rightarrow 7\}) \rightarrow ?$$

(if $x > 5$ then $y := 2 + 3$ else $y := 3 + 4$ fi,
 $\{x \rightarrow 7\}$)
 $\rightarrow ?$

Example Evaluation

- First step:

$$\frac{\quad}{(x, \{x \rightarrow 7\}) \rightarrow 7}$$

$$(x > 5, \{x \rightarrow 7\}) \rightarrow (7 > 5, \{x \rightarrow 7\})$$

(if $x > 5$ then $y := 2 + 3$ else $y := 3 + 4$ fi,
 $\{x \rightarrow 7\}$)
 $\rightarrow ?$

Example Evaluation

- First step:

$$\frac{}{(x, \{x \rightarrow 7\}) \rightarrow 7}$$

$$(x > 5, \{x \rightarrow 7\}) \rightarrow (7 > 5, \{x \rightarrow 7\})$$

(if $x > 5$ then $y := 2 + 3$ else $y := 3 + 4$ fi,
 $\{x \rightarrow 7\}$)

\rightarrow (if $7 > 5$ then $y := 2 + 3$ else $y := 3 + 4$ fi,
 $\{x \rightarrow 7\}$)

Example Evaluation

- Second Step:

$$(7 > 5, \{x \rightarrow 7\}) \rightarrow \text{true}$$

$$(\text{if } 7 > 5 \text{ then } y:=2 + 3 \text{ else } y:=3 + 4 \text{ fi, } \{x \rightarrow 7\})$$
$$\rightarrow (\text{if true then } y:=2 + 3 \text{ else } y:=3 + 4 \text{ fi, } \{x \rightarrow 7\})$$

- Third Step:

$$(\text{if true then } y:=2 + 3 \text{ else } y:=3 + 4 \text{ fi, } \{x \rightarrow 7\})$$
$$\rightarrow (y:=2+3, \{x \rightarrow 7\})$$

Example Evaluation

- Fourth Step:

$$\frac{(2+3, \{x \rightarrow 7\}) \rightarrow 5}{(y := 2+3, \{x \rightarrow 7\}) \rightarrow (y := 5, \{x \rightarrow 7\})}$$

- Fifth Step:

$$(y := 5, \{x \rightarrow 7\}) \rightarrow \{y \rightarrow 5, x \rightarrow 7\}$$

Example Evaluation

- Bottom Line:

(if $x > 5$ then $y := 2 + 3$ else $y := 3 + 4$ fi,
 $\{x \rightarrow 7\}$)

--> (if $7 > 5$ then $y := 2 + 3$ else $y := 3 + 4$ fi,
 $\{x \rightarrow 7\}$)

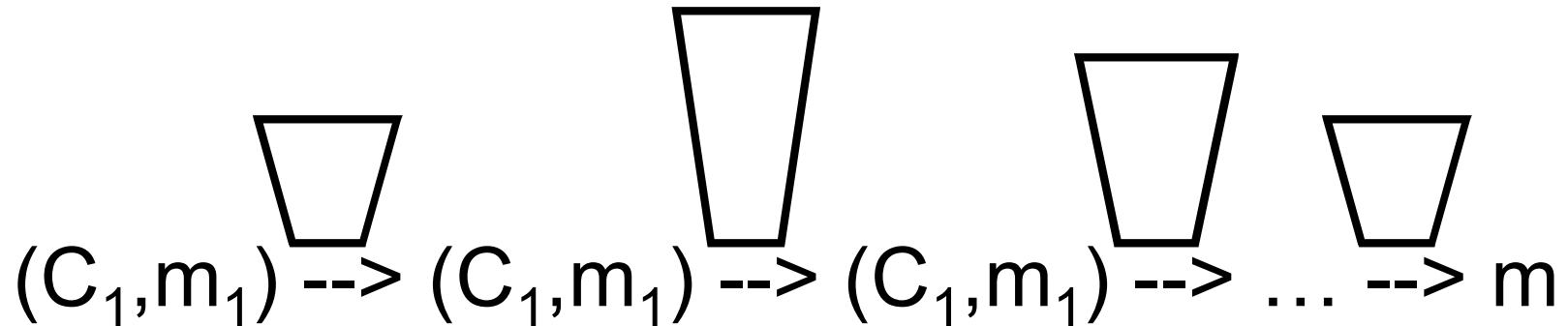
--> (if true then $y := 2 + 3$ else $y := 3 + 4$ fi,
 $\{x \rightarrow 7\}$)

--> ($y := 2 + 3$, $\{x \rightarrow 7\}$)

--> ($y := 5$, $\{x \rightarrow 7\}$) --> $\{y \rightarrow 5, x \rightarrow 7\}$

Transition Semantics Evaluation

- A sequence of steps with trees of justification for each step



- Let \dashrightarrow^* be the transitive closure of \dashrightarrow
- I.e., the smallest transitive relation containing \dashrightarrow

Adding Local Declarations

- Add to expressions:
- $E ::= \dots \mid \text{let } l = E \text{ in } E' \mid \text{fun } l \rightarrow E \mid E E'$
- $\text{fun } l \rightarrow E$ is a value
- Could handle local binding using state, but have assumption that evaluating expressions doesn't alter the environment
- We will use substitution here instead
- Recall: $E [E' / l]$ means replace all free occurrence of l by E' in E

Call-by-value (Eager Evaluation)

$$(\text{let } l = V \text{ in } E, m) \rightarrow (E[V/l], m)$$
$$(E, m) \rightarrow (E'', m)$$

$$(\text{let } l = E \text{ in } E', m) \rightarrow (\text{let } l = E'' \text{ in } E')$$
$$((\text{fun } l \rightarrow E) V, m) \rightarrow (E[V/l], m)$$
$$(E', m) \rightarrow (E'', m)$$

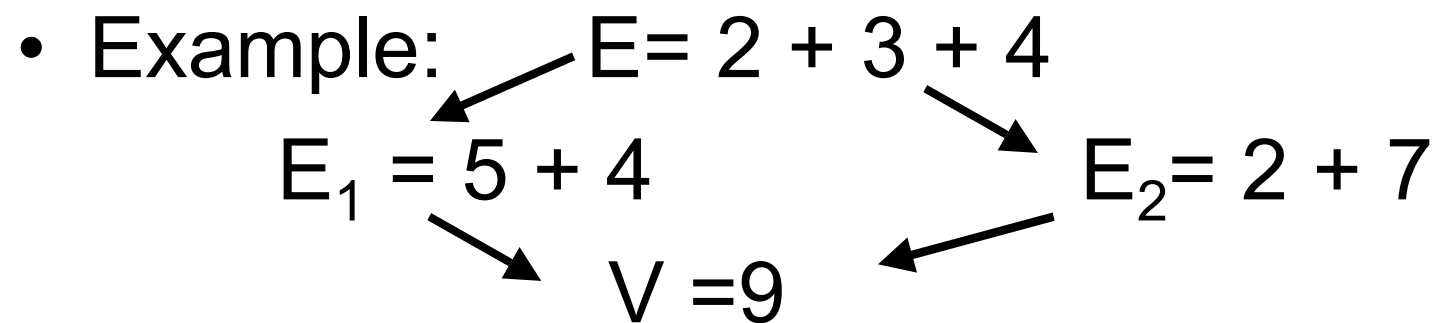
$$((\text{fun } l \rightarrow E) E', m) \rightarrow ((\text{fun } l \rightarrow E) E'', m)$$

Call-by-name (Lazy Evaluation)

- $(\text{let } I = E \text{ in } E', m) \rightarrow (E'[E / I], m)$
- $((\text{fun } I \rightarrow E') E, m) \rightarrow (E'[E / I], m)$
- Question: Does it make a difference?
- It can depending on the language

Church-Rosser Property

- Church-Rosser Property: If $E \rightarrow^* E_1$ and $E \rightarrow^* E_2$, if there exists a value V such that $E_1 \rightarrow V$, then $E_2 \rightarrow V$
- Also called **confluence** or **diamond property**



Does It always Hold?

- No. Languages with side-effects tend not be Church-Rosser with the combination of call-by-name and call-by-value
- Alonzo Church and Barkley Rosser proved in 1936 the λ -calculus does have it
- Benefit of Church-Rosser: can check equality of terms by evaluating them (Given evaluation strategy might not terminate, though)

Transition Semantics for λ -Calculus

- Application (version 1)

$$(\lambda x . E) E' \rightarrow E[E'/x]$$

- Application (version 2)

$$(\lambda x . E) V \rightarrow E[V/x]$$

$$\frac{E' \rightarrow E''}{(\lambda x . E) E' \rightarrow (\lambda x . E) E''}$$