CS 477: Operational Program Semantics

Sasa Misailovic

Based on previous slides by Gul Agha, Elsa Gunter, Madhusudan Parthasarathy, Mahesh Viswanathan, and Vikram Adve

University of Illinois at Urbana-Champaign

Previously, on CS 477

Propositional Logic:

- Syntax
- Semantics
- Proof

(Homework/Quiz #1 is out: due next Thursday)

Simple Imperative Programming Language

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

Syntax -> Graphs

Reminder: Graph: (V, E)

- V is a set of vertices (nodes)
- $\mathrm{E} \subseteq V \times V$ is a relation denoting "connected" nodes. Elements $e \in E$ are edges: pairs of connected vertices $e = (v_1, v_2)$. Can be directed or undirected.

Common definitions:

- Post(v) – successor vertices of v, Pre(v) – direct predecessor vertices of v
- Path: a sequence of vertices s.t. $v_i \in Pre(v_{i+1})$. Cycle when the same vertex multiple times in the path, else simple. Length: number of vertices in a path.
- Acyclic graphs: no cycles.
- Tree: exists v_{root} (without predecessors) such that all other vertices reachable along unique paths
- Strongly connected component: all pairs of vertices mutually reachable
- Search: DFS, BFS; traversal: preorder, postorder, etc.

Syntax -> Graphs

- Parse Tree (from CS 374)
- Abstract Syntax Tree
- Control-flow Graph

Flow Graphs

- **Flow Graph:** A triple G=(N,A,s), where (N,A) is a (finite) directed graph, s ∈ N is a designated "initial" node, and there is a path from node s to every node n ∈ N.

- An *entry node* in a flow graph has no predecessors.
- An *exit node* in a flow graph has no successors.
- There is exactly one entry node, s. We can modify a general DAG to ensure this. *How?*
- We can also transform the graph to have only one exit node. *How?*

Control Flow Graph (CFG)

- **Flow Graph:** A triple G=(N,A,s), where (N,A) is a (finite) directed graph, s ∈ N is a designated "initial" node, and there is a path from node s to every node n ∈ N.

- **Control Flow Graph (CFG)** is a flow graph that represents all ***paths*** (sequences of statements) that might be traversed during program execution.
- Nodes in CFG are program statements, and edge (S_1,S_2) denotes that statement S_1 can be followed by S_2 in execution.
- In CFG, a node unreachable from s can be safely deleted. *Why?*
- Control flow graphs are usually *sparse*. I.e., $|A| = O(|N|)$. In fact, if only binary branching is allowed $|A| \leq 2|N|$.

Control Flow Graph (CFG)

- **Basic Block** is a sequence of statements S_1 ... S_n such that execution control must reach S_1 before S_2, and, if S_1 is executed, then S_2 . . . S_n are all executed in that order
  - Unless some statement S_i causes the program to halt

- **Leader** is the first statement of a basic block

- **Maximal Basic Block** is a basic block with a maximum number of statements (n)

Control Flow Graph (CFG)

Let us refine our previous definition

- **CFG** is a directed graph in which:
- Each node is a single basic block
- There is an edge b1 → b2 if block b2 *may* be executed after block b1 in *some* execution

- We typically define it for a single procedure

- A CFG is a conservative approximation of the control flow! Why?

Example

Source Code

```
unsigned fib(unsigned n) {
   int i;
   int f0 = 0, f1 = 1, f2;

   if (n <= 1) return n;

   for (i = 2; i <= n; i++) {
      f2 = f0 + f1;
      f0 = f1;
      f1 = f2;
   }
   return f2;
}
```

LLVM bitcode (ver 3.9.1)

```
define i32 @fib(i32 %0) {
  %2 = icmp ult i32 %0, 2
  br i1 %2, label %12, label %3

; <label>:3:
  br label %4

; <label>:4:
  %5 = phi i32 [ %8, %4 ], [ 1, %3 ]
  %6 = phi i32 [ %5, %4 ], [ 0, %3 ]
  %7 = phi i32 [ %9, %4 ], [ 2, %3 ]
  %8 = add i32 %5, %6
  %9 = add i32 %7, 1
  %10 = icmp ugt i32 %9, %0
  br i1 %10, label %11, label %4

; <label>:11:
  br label %12

; <label>:12:
  %13 = phi i32 [%0, %1], [%8, %11]
  ret i32 %13
}
```

Dominance in Flow Graphs

- Let d, d1, d2, d3, n be nodes in G.
- d **dominates** n ("**d dom n**") *iff* every path from s to n contains d
- d **properly dominates** n if d dominates n and d ≠ n
- d **is the immediate dominator of** n ("**d idom n**")
  if d is the last proper dominator on any path from initial node to n,
- **DOM(**x**)** denotes the set of dominators of x,
- **Dominator tree:** the children of each node d are the nodes n such that "d idom n" (immediately dominates)

Dominator Properties

- **Lemma 1:** DOM(s) = { s }.
- **Lemma 2:** s dom d, for all nodes d in G.
- **Lemma 3:** The dominance relation on nodes in a flow graph is a ***partial ordering***
- ***Reflexive*** — *n dom n* is true for all n.
- ***Antisymmetric*** — If *d dom n*, then cannot be *n dom d*
- ***Transitive*** — $d1\ dom\ d2 \wedge d2\ dom\ d3 \Rightarrow d1\ dom\ d3$
- **Lemma 4:** The dominators of a node form a list.
- **Lemma 5:** Every node except s has a unique immediate dominator.

Postdominance

Def. Postdomination: node *p* postdominates a node *d* iff all paths to the exit node of the graph starting at *d* must go through *p*

Def. **Reverse Control Flow Graph** (RCFG) of a CFG has the same nodes as CFG and has edge Y → X if X → Y is an edge in CFG.

- p is a postdominator of d iff p dominates d in the RCFG.

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

- 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

Denotational Semantics

- Construct a function $\mathcal{M}$ assigning a mathematical meaning to each program construct
- Lambda calculus often used as the range of the meaning function
- Meaning function is compositional: meaning of construct built from meaning of parts
- Useful for proving properties of programs

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 :

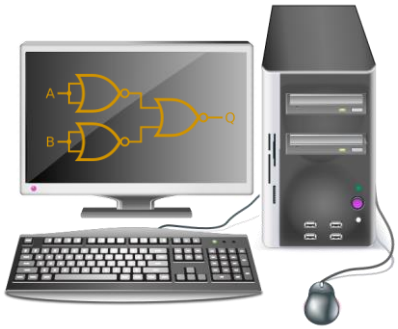
{**Precondition**} ***Program*** {**Postcondition**}

Much more about it later in the course!

Modeling Program Environment

Source Program:

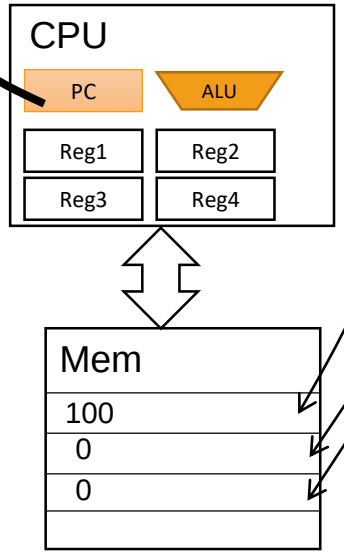
```
int binsearch(int x, int v[], int n)
{
    int low, high, mid;
  1 low = 0;
    high = n - 1;
    while (low <= high) | 2
    {
      3 mid = (low + high)/2;
        if (x < v[mid])
                high = mid - 1;  | 4
      5 else if (x > v[mid])
                low = mid + 1;   | 6
      7 else return mid;
    }
    return -1; | 8
} | 9
```



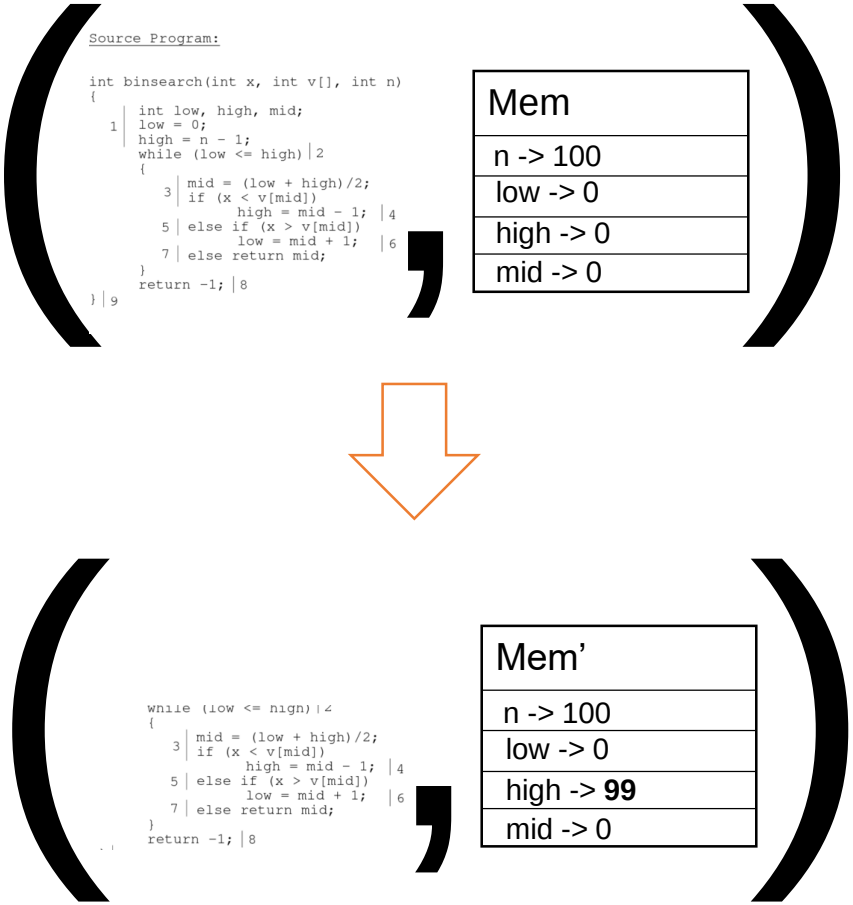
Real machine

Source Program:

```
int binsearch(int x, int v[], int n)
{
    int low, high, mid;
  1 low = 0;
    high = n - 1;
    while (low <= high) | 2
    {
      3 mid = (low + high)/2;
        if (x < v[mid])
                high = mid - 1;  | 4
      5 else if (x > v[mid])
                low = mid + 1;   | 6
      7 else return mid;
    }
    return -1; | 8
} | 9
```



Virtual machine



Mathematical Program Environment

Sources: https://www.researchgate.net/figure/Example-of-Control-Flow-Graph_fig5_4065402 and https://freesvg.org/computer-station-vector-graphics

Program Environment

Pair of code to execute + a valuation (aka state)

Code to execute: Next statement and program text that remains to be executed:

Statement_1; Other_Statements

A valuation of program variables:

- Mapping m: Identifiers-> Value

Program statements ("S_1; S_2; … S_n") transform the valuations. Execution is then:

- $m_2 = [[S_1]](m_1)$
- $m_3 = [[S_2]](m_2)$
- …
- $m_{n+1} = [[S_n]](m_n)$

- Also $(s_1, m_1) \rightarrow (s_2, m_2) \rightarrow (s_3, m_3) \rightarrow \ldots \longrightarrow (s_n, m_n) \longrightarrow (\cdot, m_{n+1})$.
  We can define the sequence $(s_1, m_1), (s_2, m_2), (s_3, m_3), \ldots, (s_n, m_n), (\cdot, m_{n+1})$ or its projection $(m_1, \ldots m_n)$ as the **trace** of execution

Natural Semantics ("Big-step Semantics")

- Aka Structural Operational Semantics, aka "Big Step Semantics"

- Provide value for a program by rules and derivations, similar to type derivations

- Rule conclusions look like

$$(C, m) \Downarrow m'$$

"Evaluating a command C in the state m results in the new state m' "

or

$$(E, m) \Downarrow v$$

"Evaluating an expression E in the state m results in the value v'"

Simple Imperative Programming Language

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

Natural Semantics of Atomic Expressions

- **Identifiers:** $(k,m) \Downarrow m(k)$

- **Numerals are values:** $(N,m) \Downarrow N$

- **Booleans:** $(true,m) \Downarrow true$

  $(false\ ,m) \Downarrow false$

Booleans:

$$\frac{(B, m) \Downarrow \text{false}}{(B \,\&\, B', m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{true} \qquad (B', m) \Downarrow b}{(B \,\&\, B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(B \text{ or } B', m) \Downarrow \text{true}} \qquad \frac{(B, m) \Downarrow \text{false} \quad (B', m) \Downarrow b}{(B \text{ or } B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(\text{not } B, m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{false}}{(\text{not } B, m) \Downarrow \text{true}}$$

Binary Relations

$$\frac{(E, m) \Downarrow U \qquad (E', m) \Downarrow V \qquad U \text{ rop } V = b}{(E \text{ rop } E', m) \Downarrow b}$$

- By U rop V = b, we mean does (the meaning of) the relation rop hold on the meaning of U and V

- May be specified by a mathematical expression/equation or rules matching U and V

Arithmetic Expressions

$$\frac{(E, m) \Downarrow U \qquad (E', m) \Downarrow V \qquad U \text{ op } V = N}{(E \text{ op } E', m) \Downarrow N}$$

where N is the specified value for (mathematical) U op V

Commands

Skip: $(skip, m) \Downarrow m$

Assignment:

$$\frac{(E,m) \Downarrow V}{(k := E, m) \Downarrow m\,[k \leftarrow V]}$$

Sequencing:

$$\frac{(C,m) \Downarrow m' \quad (C',m') \Downarrow m''}{(C; C', m) \Downarrow m''}$$

If Then Else Command

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

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

Example: If Then Else Rule

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

$\Downarrow$?

Example: If Then Else Rule

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

Example: Arith Relation

$$
\dfrac{\dfrac{? > ? = ?}{(x,\{x\text{->}7\}) \Downarrow ? \quad (5,\{x\text{->}7\}) \Downarrow ?}}{(x > 5, \{x \text{ -> } 7\}) \Downarrow ?}
$$

$$
\dfrac{(x > 5, \{x \text{ -> } 7\}) \Downarrow ? \qquad \Downarrow ?}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \text{ -> } 7\})}
$$

Example: Identifier(s)

$$\frac{\dfrac{7 > 5 = \text{true} \qquad (x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}{(x > 5, \{x \text{ -> } 7\})\Downarrow ?}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \Downarrow ?}$$

Example: Arith Relation

$$\frac{\dfrac{7 > 5 = \text{true} \qquad (x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}{(x > 5, \{x \text{ -> } 7\})\Downarrow \text{true}}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \ \{x \text{ -> } 7\}) \Downarrow\ ?}$$

Example: If Then Else Rule

$$\frac{\dfrac{7 > 5 = \text{true} \qquad (x,\{x\text{->}7\}) \Downarrow 7 \quad (5,\{x\text{->}7\}) \Downarrow 5}{(x > 5, \{x \text{ -> } 7\}) \Downarrow \text{true}} \qquad \dfrac{}{\begin{array}{c}(y := 2 + 3, \{x\text{-> } 7\} \\ \Downarrow ?\end{array}}}{\begin{array}{c}(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \\ \Downarrow ?\end{array}}$$

Example: Assignment

$$\frac{\dfrac{7 > 5 = \text{true} \qquad (x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}{(x > 5, \{x \text{ -> } 7\})\Downarrow \text{true}} \qquad \dfrac{(2+3, \{x\text{->}7\})\Downarrow ? \qquad (y{:=}\ 2 + 3, \{x\text{-> } 7\}}{\Downarrow\ ?}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \Downarrow\ ?}$$

Example: Arith Op

$$\frac{\dfrac{7 > 5 = \text{true} \qquad \dfrac{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}{(x > 5, \{x \text{ -> } 7\})\Downarrow \text{true}} \qquad \dfrac{\dfrac{? + ? = ? \qquad (2,\{x\text{->}7\})\Downarrow ? \quad (3,\{x\text{->}7\}) \Downarrow ?}{(2+3, \{x\text{->}7\})\Downarrow ?}}{\dfrac{(y:= 2 + 3, \{x\text{-> } 7\}}{\Downarrow ? \qquad .}}}{(\text{if } x > 5 \text{ then } y:= 2 + 3 \text{ else } y:=3 + 4 \text{ fi}, \{x \text{ -> } 7\})}$$

$$\Downarrow ?$$

Example: Numerals

$$\frac{\dfrac{\dfrac{7 > 5 = \text{true} \qquad (x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}{(x > 5, \{x \text{ -> } 7\})\Downarrow \text{true}} \qquad \dfrac{\dfrac{2 + 3 = 5 \qquad (2,\{x\text{->}7\})\Downarrow 2 \quad (3,\{x\text{->}7\}) \Downarrow 3}{(2+3, \{x\text{->}7\})\Downarrow ?}}{(y := 2 + 3, \{x \text{-> } 7\} \Downarrow ?}}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \Downarrow ?}$$

Example: Arith Op

$$\frac{\dfrac{7 > 5 = \text{true} \qquad \dfrac{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}{(x > 5, \{x \text{ -> } 7\})\Downarrow \text{true}} \qquad \dfrac{\dfrac{2 + 3 = 5 \qquad (2,\{x\text{->}7\})\Downarrow 2 \quad (3,\{x\text{->}7\}) \Downarrow 3}{(2+3, \{x\text{->}7\})\Downarrow 5}}{(y{:=}\ 2 + 3, \{x\text{-> } 7\}}}{\Downarrow ?}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \ \{x \text{ -> } 7\})}$$

$$\Downarrow \ ?$$

Example: Assignment

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

$$\Downarrow ?$$

Example: If Then Else Rule

$$\frac{\dfrac{\dfrac{(x,\{x\text{->}7\})\Downarrow 7 \quad (5,\{x\text{->}7\})\Downarrow 5}{(x > 5, \{x \text{ -> } 7\})\Downarrow \text{true}} \quad 7 > 5 = \text{true} \qquad \dfrac{\dfrac{(2,\{x\text{->}7\})\Downarrow 2 \quad (3,\{x\text{->}7\}) \Downarrow 3}{(2+3, \{x\text{->}7\})\Downarrow 5} \quad 2 + 3 = 5}{(y := 2 + 3, \{x\text{->} 7\} \Downarrow \{x\text{->}7, y\text{->}5\}}}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \Downarrow \mathbf{\{x\text{->}7, y\text{->}5\}}}$$

While Command

$$\frac{(B, m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

$$\frac{}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m''}$$

While Command

$$\frac{(B, m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

$$\frac{\overset{1}{(B,m) \Downarrow \text{true}} \quad \overset{2}{(C,m) \Downarrow m'} \quad \overset{3}{(\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m''}$$

Example: While Rule

$$\frac{(B, m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

$$\frac{\overset{1}{(B,m) \Downarrow \text{true}} \quad \overset{2}{(C,m) \Downarrow m'} \quad \overset{3}{(\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m''}$$

(while x > 5 do x := x-5 od, {x -> 7}) $\Downarrow$ {x->2}

Example: While Rule

1

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

2 $(x := x\text{-}5, \{x\text{->}7\}) \Downarrow \{x\text{->}2\}$

3 $(x > 5, \{x\text{->}2\}) \Downarrow \text{false}$

---

while x > 5 do x := x-5 od;

$\{x \text{ -> } 2\}) \Downarrow \{x\text{->}2\}$

$(\text{while } x > 5 \text{ do } x := x\text{-}5 \text{ od}, \{x \text{ -> } 7\}) \Downarrow \{x\text{->}2\}$

While Command and Termination?

$$\frac{(B, m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

$$\frac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m''}$$

The rule assumes the loop terminates!

While Command and Termination?

$$\frac{(B, m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

$$\frac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m''}$$

The rule assumes the loop terminates!

$$\frac{\textbf{? ? ?}}{\text{while } (x>0) \text{ do } x:=x+1 \text{ od}, \{x\text{->}1\} \Downarrow \textbf{? ? ?}}$$

Interpretation Versus Compilation

- A **compiler** from language L1 to language L2 is a program that takes an L1 program and for each piece of code in L1 generates a piece of code in L2 of same meaning
- An **interpreter** of L1 in L2 is an L2 program that executes the meaning of a given L1 program
- Compiler would examine the body of a loop once; an interpreter would examine it every time the loop was executed

Interpreter

- An *Interpreter* represents the operational semantics of a language L1 (source language) in the language of implementation L2 (target language)
- Built incrementally
  - Start with literals
  - Variables
  - Primitive operations
  - Evaluation of expressions
  - Evaluation of commands/declarations

Interpreter

- Takes abstract syntax trees as input
  - In simple cases could be just strings
- One procedure for each syntactic category (nonterminal)
  - eg one for expressions, another for commands
- If Natural semantics used, tells how to compute final value from code
- If Transition semantics used, tells how to compute next "state"
  - To get final value, put in a loop

Natural Semantics Interpreter Implementation

- Identifiers: ***(k,m)*** $\Downarrow$ ***m(k)***
- Numerals are values: ***(N,m)*** $\Downarrow$ ***N***

- Conditionals:

$$\frac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'} \qquad \frac{(B,m) \Downarrow \text{false} \quad (C',m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

```
compute_exp (Var(v), m) = look_up v m
compute_exp (Int(n), _) = Num (n)
…
compute_com (IfExp(b,c1,c2), m) =
       if compute_exp (b,m) = Bool(true)
       then compute_com (c1,m)
       else compute_com (c2,m)
```

Natural Semantics Interpreter Implementation

- Loop: $\dfrac{(B, m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$ $\quad \dfrac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''}{(\text{while } B \text{ do C od}, m) \Downarrow m''}$

```
compute_com (While(b,c), m) =
      if compute_exp (b,m) = Bool(false)
      then m
      else compute_com
              (While(b,c), compute_com(c,m))
```

- May fail to terminate - exceed stack limits
  - Returns no useful information then

Transition Semantics ("Small-step Semantics")

- Form of operational semantics
- **Describes how each program construct transforms machine state by *transitions***
- Rules look like

$$(C, m) \text{ --> } (C', m') \quad \text{or} \quad (C, m) \text{ --> } m'$$

- *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

- *C, C'* used for commands; *E, E' for* expressions; *U,V* for values
- Special class of expressions designated as *values*
  - Eg 2, 3 are values, but 2+3 is only an expression
- Memory only holds values
  - Other possibilities exist

Evaluation Semantics

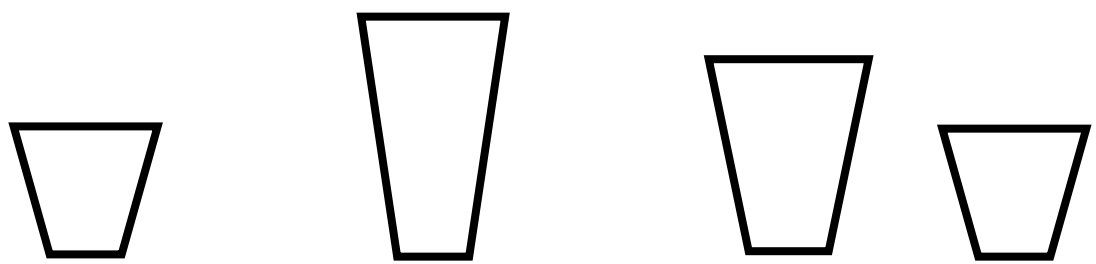
- Transitions successfully stops when E/*C* is a value/memory
- Evaluation fails if no transition possible, but not at value/memory
- Value/memory is the final *meaning* of original expression/command (in the given state)
- Coarse semantics: final value / memory
- More fine grained: whole transition sequence

Simple Imperative Programming Language

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

Transition Semantics Evaluation

- **A sequence of transitions**: trees of justification for each step



(C_1,m_1) --> (C_2,m_2) --> (C_3,m_3) --> … --> (skip, m) --> m

Transitions for Expressions

- Numerals are values
- Boolean values = {true, false}
- Identifiers: (k,m) --> (m(k), m)

Arithmetic Expressions

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

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

(U op V, m) --> (N,m)

where N is the specified value for (mathematical) “U op V”

Boolean Operations:

- Operators: (short-circuit)

(false & B, m) --> (false,m)

(true & B, m) --> (B,m)

$$\frac{(B, m) \text{ --> } (B'', m)}{(B \,\&\, B', m) \text{ --> } (B'' \,\&\, B', m)}$$

(true or B, m) --> (true,m)

(false or B, m) --> (B,m)

$$\frac{(B, m) \text{ --> } (B'', m)}{(B \text{ or } B', m) \text{ --> } (B'' \text{ or } B', m)}$$

(not true, m) --> (false,m)

(not false, m) --> (true,m)

$$\frac{(B, m) \text{ --> } (B', m)}{(\text{not } B, m) \text{ --> } (\text{not } B', m)}$$

Relations

$$\frac{(E, m) \text{ --> } (E'', m)}{(E \text{ rop } E', m) \text{ --> } (E'' \text{ rop } E', m)}$$

$$\frac{(E, m) \text{ --> } (E', m)}{(V \text{ rop } E, m) \text{ --> } (V \text{ rop } E', m)}$$

(U rop V, m) --> (true,m) or (false,m)

depending on whether U rop V holds or not

Commands - in English

- ***skip*** means we're 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 (i.e. it completes), evaluate remainder with the new memory

Commands

$$(\text{skip}, m) \dashrightarrow m$$

$$\frac{(E,m) \dashrightarrow (E',m)}{(k:=E,m) \dashrightarrow (k:=E',m)}$$

$$(k:=V,m) \dashrightarrow m[k \leftarrow V]$$

$$\frac{(C,m) \dashrightarrow (C'',m')}{(C;C', m) \dashrightarrow (C'';C',m')} \qquad \frac{(C,m) \dashrightarrow m'}{(C;C', m) \dashrightarrow (C',m')}$$

If Then Else Command - in English

- If the boolean guard in an 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

- Base Cases:

(if true then C else C' fi, m) --> (C, m)

(if false then C else C' fi, m) --> (C', m)

- Recursive Case:

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

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 check of the boolean guard, with the true case being to execute the body and then 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 -> 7}) --> ?

Example Evaluation

- First step:

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

Example Evaluation

- First step:

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

Example Evaluation

- First step:

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

Example Evaluation

- First step:

$$\frac{\dfrac{(x,\{x \to 7\}) \to (7, \{x \to 7\})}{(x > 5, \{x \to 7\}) \to (7 > 5, \{x \to 7\})}}{\begin{array}{c}(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \to 7\}) \\ \to (\text{if } 7 > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \to 7\})\end{array}}$$

Example Evaluation

- Second Step:

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

- Third Step:

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

Example Evaluation

- Fourth Step:

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

- Fifth Step:

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

Example Evaluation

- Bottom Line:

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

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

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

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

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

--> **{y -> 5, x -> 7}**

Adding Local Declarations

- Add to expressions:
- *E* ::= … | let *x* = *E* in *E'* | fun *x* -> *E* | *E* *E'*
  - Recall: fun *x* -> *E* is a value

- Could handle local binding using state, but have assumption that evaluating expressions does not alter the environment

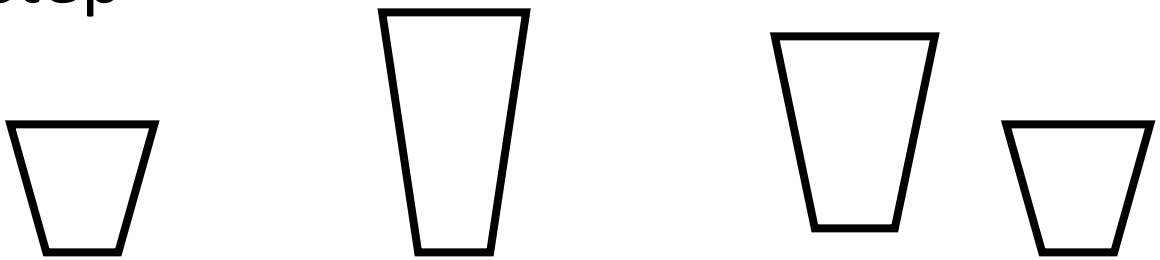
- We will use **substitution** here instead
- **Notation**: *E* [*E'*/ *x*] means replace all free occurrence of *x* by *E'* in *E*

Calling Conventions (Common Strategies)

- Call by value (eager evaluation): First evaluate the argument, then use its value
- Call by name: Refer to the computation by its name; evaluate every time it is called
- Call by need (lazy evaluation): Refer to the computation by its name, but once evaluated, store (“memoize”) the result for future reuse

Transition Semantics Evaluation

- **A sequence of transitions**: trees of justification for each step

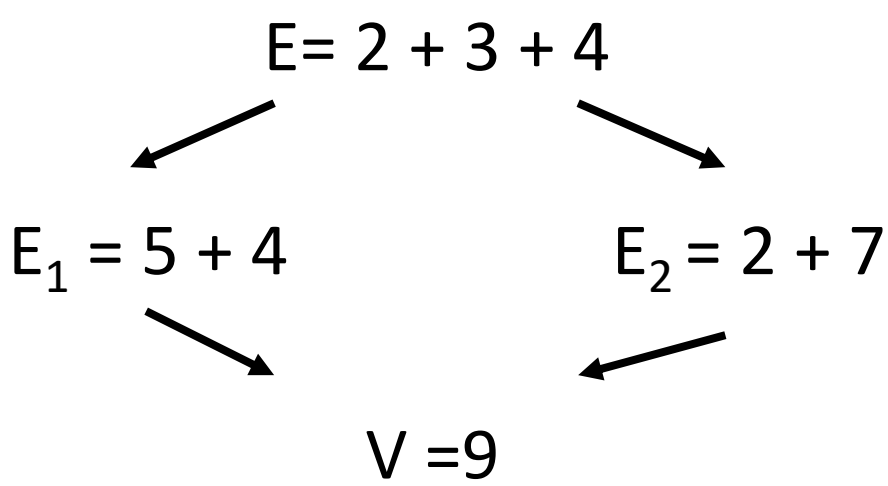


(C_1,m_1) --> (C_2,m_2) --> (C_3,m_3) --> … --> (skip, m) --> m

- **Definition:** let **-->*** be the transitive closure of **-->** i.e., the smallest transitive relation containing **-->**

Church-Rosser Property

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



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)

Extension: Abort

- Regular execution terminates when program in configuration (skip, m)

- Add another command “abort”.

- If the computation ends in (abort, m), then there is no transition from it => we reached the error state

Extensions: Parallel

- Statement C1 par C2: execute C1 and C2 in parallel
- We can apply multiple rules at the same time!
- (reflects nondeterminism; also hard to express using $\Downarrow$)

$$\frac{(C,m) \text{ --> } (C'', m')}{(C \text{ par } C', m) \text{ --> } (C'' \text{ par } C', m')}$$

$$\frac{(C,m) \text{ --> } (C'', m')}{(C \text{ par skip}, m) \text{ --> } (C'', m')}$$

$$\frac{(C',m) \text{ --> } (C'', m')}{(C \text{ par } C', m) \text{ --> } (C \text{ par } C'', m')}$$

$$\frac{(C',m) \text{ --> } (C'', m')}{(\text{skip par } C', m) \text{ --> } (C'', m')}$$

Extension: Nondeterministic

- E.g., nondeterministic assignment x = E1 [] E2
  - Nondeterministically assigns one of the two evaluated values to x
- How do we extend the semantics? (e.g., small step)
- What are our configurations?

Symbolic Execution

Symbolic formulas syntax (with symbolic variables α):

P ::= true | false

| not P | P1 bop P2 | Aexp1 rop Aexpr2

Aexp ::= α | n | Aexp1 + Aexp2 | Aexp1 * Aexp2

| Aexp1 – Aexp2 | Aexp1 / Aexp2

Memory store: Σ: $Var \rightarrow Aexp$

Analysis state (P, Σ):

- P is called ***path condition***, and Σ a ***symbolic state***.

Arithmetic And Relational Expressions

$$\frac{(E1, \Sigma) \Downarrow Aexp1' \quad (E2, \Sigma) \Downarrow Aexp2'}{(E1 \text{ op } E2, \Sigma) \Downarrow Aexp1' \text{ op } Aexp2'}$$

$$\frac{(E1, \Sigma) \Downarrow Aexp1' \quad (E2, \Sigma) \Downarrow Aexp2' \quad P = Aexp1' \text{ rop } Aexp2'}{(E \text{ rop } E', \Sigma) \Downarrow P}$$

Statements

Skip: $(P, \text{skip}, \Sigma) \Downarrow (P, \Sigma)$

Assignment:

$$\frac{(E, \Sigma) \Downarrow \text{Aexp}}{(P, k := E, \Sigma) \Downarrow (P, \Sigma\,[k \text{ <-- } \text{Aexp}\,])}$$

Sequencing:

$$\frac{(P, C, \Sigma) \Downarrow (P', \Sigma') \quad (P', C', \Sigma') \Downarrow \Sigma''}{(P, C; C', \Sigma) \Downarrow \Sigma''}$$

If Then Else Statement

$$\frac{(B, \Sigma) \Downarrow Pb \qquad SAT(P \wedge Pb) \quad (P \wedge Pb, C, \Sigma) \Downarrow (P', \Sigma')}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, \Sigma) \Downarrow (P', \Sigma')}$$

$$\frac{(B, \Sigma) \Downarrow Pb \qquad SAT(P \wedge \neg Pb) \qquad (P \wedge \neg Pb, C', \Sigma) \Downarrow (P', \Sigma')}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, \Sigma) \Downarrow (P', \Sigma')}$$

Both are possibly satisfiable (due to symbolic abstraction)!

Execution is then not a sequence but a tree of instructions!

Static Symbolic execution: We "merge" the formulas of both branches and simplify them. This will be clearer after we cover abstract interpretation next!

Example

```
int x = input()
int y = 0

if x > 0
      y = x + 1
else
      y = -x
```

// Question: Is $y \geq 0$
// after the execution?

(See the lecture video)

Another Example

```
int x = input()
int y = 1/x
```

// Question: can the code experience an error?

```
int x = input()
if x != 0
  y = 1 / x
else
  abort
```

Symbolic Execution of Loops?

- Most practical tools just “unroll” the loop k times
- Enough for finding various bugs:
  search under “Small scope hypothesis”

- A more general approach will require ***loop invariants*** (predicates that hold at any point of loop execution)
- Often requires manual intervention by developer!
- We will discuss invariants later when we cover deductive methods for reasoning about programs.

Symbolic Evaluaton for Loops: Rule

Together: Let us derive the rule for the finite loop
while_k (condition) -- for a constant $k > 0$

(See the lecture video)

Symbolic Execution and Testing

- Generalizes testing by using symbolic values and having means to explore all paths: exhaustive exploration
- Scalability is an issue (although the modern tools have made it more practical)
- Concolic execution: combines testing with symbolic execution
  - Use concrete execution to reach a certain point in the execution (e.g., an important subcomputation)
  - Use then symbolic execution to exhaustively explore the executions within that smaller scope