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

• $I \in \text{Identifiers}$
• $N \in \text{Numerals}$
• $B ::= \text{true} \mid \text{false} \mid B \& B \mid B \lor B \mid \neg B \mid E < E \mid E = E$
• $E ::= N \mid I \mid E + E \mid E \ast E \mid E - E \mid - E$
• $S ::= \text{skip} \mid S; S \mid I ::= E \mid \text{if } B \text{ then } S \text{ else } S \text{ fi} \mid \text{while } B \text{ do } S \text{ od}$
Syntax -> Graphs

Reminder: Graph: \((V, E)\)

- \(V\) is a set of vertices (nodes)
- \(E \subseteq V \times V\) 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, $s \in N$ is a designated “initial” node, and there is a path from node $s$ to every node $n \in N$.

- 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, $s \in N$ is a designated “initial” node, and there is a path from node $s$ to every node $n \in N$.

- **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 $S_1 \ldots S_n$ such that execution control must reach $S_1$ before $S_2$, and, if $S_1$ is executed, then $S_2 \ldots S_n$ 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 $b_1 \rightarrow b_2$ if block $b_2$ may be executed after block $b_1$ 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 $d, d_1, d_2, d_3, n$ be nodes in $G$.

• $d$ dominates $n$ ("$d \text{ dom } n$") iff every path from $s$ to $n$ contains $d$

• $d$ properly dominates $n$ if $d$ dominates $n$ and $d \neq n$

• $d$ is the immediate dominator of $n$ ("$d \text{ idom } n$") if $d$ is the last proper dominator on any path from initial node to $n$,

• $\text{DOM}(x)$ denotes the set of dominators of $x$,

• Dominator tree: the children of each node $d$ are the nodes $n$ such that "$d \text{ idom } n$" (immediately dominates)
Dominator Properties

• **Lemma 1**: $\text{DOM}(s) = \{ s \}$.
• **Lemma 2**: $s \text{ dom } d$, for all nodes $d$ in $G$.
• **Lemma 3**: The dominance relation on nodes in a flow graph is a **partial ordering**
  • **Reflexive** — $n \text{ dom } n$ is true for all $n$.
  • **Antisymmetric** — If $d \text{ dom } n$, then cannot be $n \text{ dom } d$
  • **Transitive** — $d1 \text{ dom } d2 \land d2 \text{ dom } d3 \Rightarrow d1 \text{ dom } d3$
• **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 $Y \rightarrow X$ if $X \rightarrow Y$ 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 $M$ 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:

\{\text{Precondition}\} \text{ Program} \{\text{Postcondition}\}

Much more about it later in the course!
Modeling 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 (“S₁; S₂; ... Sₙ”) transform the valuations. Execution is then:
  • m₂ = [[S₁]](m₁)
  • m₃ = [[S₂]](m₂)
  • ...
  • mᵣ₊₁ = [[Sₙ]](mₙ)

• Also (s₁, m₁) → (s₂, m₂) → (s₃, m₃) → ... → (sₙ, mₙ) → (... mᵣ₊₁).

We can define the sequence (s₁, m₁), (s₂, m₂), (s₃, m₃), ... , (sₙ, mₙ), (... mᵣ₊₁) or its projection (m₁, ... mᵣ) 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

• $B ::= \text{true} \mid \text{false}$
  $\mid B \& B \mid B \text{ or } B \mid \text{not } B \mid E < E \mid E = E$

• $E ::= N \mid I \mid E + E \mid E \ast E \mid E - E \mid - E$

• $C ::= \text{skip} \mid C;C \mid I ::= E$
  $\mid \text{if } B \text{ then } C \text{ else } C \text{ fi} \mid \text{while } B \text{ do } C \text{ od}$
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:

\[(B, m) \downarrow \text{false}\]
\[(B & B', m) \downarrow \text{false}\]
\[(B, m) \downarrow \text{true}\]
\[(B \lor B', m) \downarrow \text{true}\]
\[(\text{not } B, m) \downarrow \text{false}\]
\[(\text{not } B, m) \downarrow \text{true}\]
Binary Relations

\[(E, m) \downarrow U \quad (E', m) \downarrow V \quad U \ rop \ V = b\]

\[(E \ rop \ E', m) \downarrow b\]

- 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

\[(E, m) \downarrow U \quad (E', m) \downarrow V \quad U \text{ op } V = N\]

\[ (E \text{ op } E', m) \downarrow N\]

where \(N\) is the specified value for (mathematical) \(U \text{ op } V\)
Commands

Skip: \((\text{skip}, m) \Downarrow m\)

Assignment: \((E, m) \Downarrow V\)

\[ (k := E, m) \Downarrow m \left[ k \leftarrow V \right] \]

Sequencing: \((C, m) \Downarrow m'\)

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

\[ (C; C', m) \Downarrow m'' \]
If Then Else Command

\[
\begin{align*}
(B,m) \downarrow \text{true} & \quad (C,m) \downarrow m' \\
\hline
\text{(if B then C else C' fi, m)} & \downarrow m'
\end{align*}
\]

\[
\begin{align*}
(B,m) \downarrow \text{false} & \quad (C',m) \downarrow m' \\
\hline
\text{(if B then C else C' fi, m)} & \downarrow m'
\end{align*}
\]
Example: If Then Else Rule

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

\[
\Downarrow \ ?
\]
Example: If Then Else Rule

\[(x > 5, \{x \rightarrow 7\}) \downarrow ?\]
\[(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \downarrow ?\]
Example: Arith Relation

? > ? = ?

\[(x, \{x \rightarrow 7\}) \downarrow ? \quad (5, \{x \rightarrow 7\}) \downarrow ?\]

\[(x > 5, \{x \rightarrow 7\}) \downarrow ?\]

\[(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \downarrow ?\]
Example: Identifier(s)

\[ 7 > 5 = \text{true} \]

\[
\begin{align*}
(x, \{x \rightarrow 7\}) & \downarrow 7 \\
(5, \{x \rightarrow 7\}) & \downarrow 5
\end{align*}
\]

\[
(x > 5, \{x \rightarrow 7\}) \downarrow ?
\]

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

\[
\downarrow ?
\]
Example: Arith Relation

7 > 5 = true

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

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

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

\Downarrow ?
Example: If Then Else Rule

\[
7 > 5 = \text{true} \\
(\mathbf{x},\{\times \rightarrow 7\}) \Downarrow 7 \\
(\mathbf{5},\{\times \rightarrow 7\}) \Downarrow 5 \\
(\mathbf{x} > 5, \{\times \rightarrow 7\}) \Downarrow \text{true} \\
(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{\times \rightarrow 7\}) \Downarrow ?
\]

\[
(\mathbf{y} := 2 + 3, \{\times \rightarrow 7\}) \Downarrow ?
\]
Example: Assignment

7 > 5 = true

(7, \{x\rightarrow 7\}) \Downarrow 7
(5, \{x\rightarrow 7\}) \Downarrow 5

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

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

\Downarrow ?
Example: Arith Op

\[ ? + ? = ? \]

\[
\begin{align*}
7 > 5 & = \text{true} \\
(x, \{x \rightarrow 7\}) & \downarrow 7 \\
(5, \{x \rightarrow 7\}) & \downarrow 5 \\
\hline
(x > 5, \{x \rightarrow 7\}) & \downarrow \text{true} \\
(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) & \downarrow ?
\end{align*}
\]
Example: Numerals

2 + 3 = 5

(2, {x -> 7}) \Downarrow 2  (3, {x -> 7}) \Downarrow 3

7 > 5 = true

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

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

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

\Downarrow ?
Example: Arith Op

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

2 + 3 = 5
\[ (2, \{x \rightarrow 7\}) \Downarrow 2 \quad (3, \{x \rightarrow 7\}) \Downarrow 3 \]
\[ (2 + 3, \{x \rightarrow 7\}) \Downarrow 5 \]
\[ (y := 2 + 3, \{x \rightarrow 7\}) \Downarrow ? \]

39
Example: Assignment

2 + 3 = 5

(2,\{x\rightarrow 7\}) \downarrow 2 \quad (3,\{x\rightarrow 7\}) \downarrow 3

7 > 5 = true

(x,\{x\rightarrow 7\}) \downarrow 7 \quad (5,\{x\rightarrow 7\}) \downarrow 5

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

(if x > 5 then y := 2 + 3 else y := 3 + 4 fi, \{x \rightarrow 7\}) \downarrow ?
Example: If Then Else Rule

\[
\begin{align*}
2 + 3 &= 5 \\
(2, \{x \rightarrow 7\}) &\Downarrow 2 \quad (3, \{x \rightarrow 7\}) \Downarrow 3
\end{align*}
\]

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x, \{x \rightarrow 7\}) &\Downarrow 7 \quad (5, \{x \rightarrow 7\}) \Downarrow 5
\end{align*}
\]

\[
\begin{align*}
(x > 5, \{x \rightarrow 7\}) &\Downarrow \text{true} \\
(y := 2 + 3, \{x \rightarrow 7\}) &\Downarrow \{x \rightarrow 7, y \rightarrow 5\}
\end{align*}
\]

\[
\begin{align*}
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi), } \{x \rightarrow 7\} \\
\Downarrow \{x \rightarrow 7, y \rightarrow 5\}
\end{align*}
\]
While Command

\[(B, m) \downarrow \text{false}\]

\[\text{(while } B \text{ do } C \text{ od, m)} \downarrow m\]

\[\text{(while } B \text{ do } C \text{ od, m)} \downarrow m''\]
While Command

1. \((B, m) \downarrow \text{true}\)
2. \((C, m) \downarrow m'\)
3. \((\text{while } B \text{ do } C \text{ od}, m') \downarrow m''\)

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

\(\frac{(B, m) \downarrow \text{true}}{(C, m) \downarrow m'}\)

\(\frac{(\text{while } B \text{ do } C \text{ od}, m') \downarrow m''}{(\text{while } B \text{ do } C \text{ od}, m) \downarrow m''}\)
Example: While Rule

\[(\text{while } x > 5 \text{ do } x := x - 5 \text{ od, } \{x \rightarrow 7\}) \downarrow \{x \rightarrow 2\}\]
Example: While Rule

1. \((x > 5, \{x \rightarrow 7\}) \Downarrow \text{true}\)

2. \((x := x - 5, \{x \rightarrow 7\}) \Downarrow \{x \rightarrow 2\}\)

3. \(\text{while } x > 5 \text{ do } x := x - 5 \text{ od; } \{x \rightarrow 2\}\)

\((\text{while } x > 5 \text{ do } x := x - 5 \text{ od, } \{x \rightarrow 7\}) \Downarrow \{x \rightarrow 2\}\)
While Command and Termination?

\[(B, m) \downarrow false \quad \Rightarrow \quad (\text{while } B \text{ do } C \text{ od, } m) \downarrow m\]

\[(B,m) \downarrow true \quad (C,m) \downarrow m' \quad (\text{while } B \text{ do } C \text{ od, } m') \downarrow m'' \quad \Rightarrow \quad (\text{while } B \text{ do } C \text{ od, } m) \downarrow m''\]

*The rule assumes the loop terminates!*
While Command and Termination?

\[(B, m) \downarrow \text{false} \]

\[(\text{while } B \text{ do } C \text{ od}, m) \downarrow m\]

\[(B, m) \downarrow \text{true} \quad (C, m) \downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m') \downarrow m''\]

\[(\text{while } B \text{ do } C \text{ od}, m) \downarrow m''\]

**The rule assumes the loop terminates!**

\[\text{while } (x>0) \text{ do } x:=x+1 \text{ od, } \{x->1\} \downarrow ? ? ? \]
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:
  \[
  \frac{(B,m) \downarrow \text{true} \quad (C,m) \downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \downarrow m'}
  \quad \frac{(B,m) \downarrow \text{false} \quad (C',m) \downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \downarrow m'}
  \]

\[
\begin{align*}
\text{compute}\_\text{exp} \ (\text{Var}(v), \ m) &= \text{look\_up} \ v \ m \\
\text{compute}\_\text{exp} \ (\text{Int}(n), \ _) &= \text{Num} \ (n) \\
\ldots \\
\text{compute}\_\text{com} \ (\text{IfExp}(b,c1,c2), \ m) &= \\
&\quad \text{if compute}\_\text{exp} \ (b,m) = \text{Bool}(\text{true}) \\
&\quad \text{then compute}\_\text{com} \ (c1,m) \\
&\quad \text{else compute}\_\text{com} \ (c2,m)
\end{align*}
\]
Natural Semantics Interpreter Implementation

• Loop: \[
\begin{array}{c}
(B, m) \downarrow \text{false} \\
\text{(while } B \text{ do } C \text{ od, } m) \downarrow m
\end{array}
\]

\[
\begin{array}{c}
(B,m) \downarrow \text{true} \\
(C,m) \downarrow m' \\
\text{(while } B \text{ do } C \text{ od, } m') \downarrow m''
\end{array}
\]

\[
\begin{array}{c}
\text{compute_com (While}(b,c), \ m) = \\
\text{if compute_exp } (b,m) = \text{Bool(false)} \\
\text{then } m \\
\text{else compute_com} \\
\text{ (While}(b,c), \ \text{compute_com}(c,m))
\end{array}
\]

• 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

\[(C, m) \rightarrow (C', m') \quad \text{or} \quad (C,m) \rightarrow m'\]

• \(C, C'\) 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 \textbf{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
- $B ::= \text{true} \mid \text{false} \mid B \& B \mid B \text{ or } B \mid \text{not } B \mid E < E \mid E = E$
- $E ::= N \mid I \mid E + E \mid E * E \mid E - E \mid - E$
- $C ::= \text{skip} \mid C;C \mid I ::= E$
  - $\mid \text{if } B \text{ then } C \text{ else } C \text{ fi} \mid \text{while } B \text{ do } C \text{ od}$
Transition Semantics Evaluation

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

\[(C_1,m_1) \rightarrow (C_2,m_2) \rightarrow (C_3,m_3) \rightarrow \ldots \rightarrow \text{(skip, m)} \rightarrow m\]
Transitions for Expressions

- Numerals are values
- Boolean values = \{true, false\}
- Identifiers: \((k,m) \rightarrow (m(k), m)\)
Arithmetic Expressions

\[
\begin{align*}
(E, m) & \rightarrow (E'', m) \\
(E \text{ op } E', m) & \rightarrow (E'' \text{ op } E', m) \\
(E, m) & \rightarrow (E', m) \\
(V \text{ op } E, m) & \rightarrow (V \text{ op } E', m)
\end{align*}
\]

(U op V, m) \rightarrow (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)

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

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

  (B, m) --> (B'', m)

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

  (B, m) --> (B'', m)

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

  (B, m) --> (B', m)

  (not B, m) --> (not B', m)
Relations

\[(E, m) \rightarrow (E'',m)\]

\[(E \text{ rop } E', m) \rightarrow (E'' \text{ rop } E',m)\]

\[(E, m) \rightarrow (E',m)\]

\[(V \text{ rop } E, m) \rightarrow (V \text{ rop } E',m)\]

\[(U \text{ rop } V, m) \rightarrow (\text{true},m) \text{ or } (\text{false},m)\]

depending on whether \(U \text{ rop } V\) 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

\[(\text{skip, } m) \rightarrow m\]

\[(E,m) \rightarrow (E', m)\]

\[\left( k:=E,m \right) \rightarrow \left( k:=E', m \right)\]

\[\left( k:=V,m \right) \rightarrow m[k \leftarrow V]\]

\[\left( C,m \right) \rightarrow \left( C'', m' \right)\]

\[\left( C;C', m \right) \rightarrow \left( C'';C', m' \right)\]

\[\left( C;C', m \right) \rightarrow m'\]

\[\left( C;C', m \right) \rightarrow \left( C', m' \right)\]
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:

  (B,m) --> (B’,m)

  (if B then C else C’ fi, m) --> (if B’ then C else C’ fi, m)
While Command

\[(\text{while } B \text{ do } C \text{ od, } m) \rightarrow \]
\[(\text{if } B \text{ then } ( C ; \text{while } B \text{ do } C \text{ od } ) \]
\[\text{else skip fi, } m)\]

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:

\[
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \rightarrow ?
\]
Example Evaluation

• First step:

\[
(x > 5, \{x \to 7\}) \rightarrow ?
\]

\[
(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \to 7\}) \rightarrow ?
\]
Example Evaluation

- First step:

\[
(x, \{x \rightarrow 7\}) \rightarrow (7, \{x \rightarrow 7\})
\]

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

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

• First step:

\[
\begin{align*}
(x, \{x \rightarrow 7\}) & \rightarrow (7, \{x \rightarrow 7\}) \\
(x > 5, \{x \rightarrow 7\}) & \rightarrow (7 > 5, \{x \rightarrow 7\}) \\
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) & \rightarrow ?
\end{align*}
\]
Example Evaluation

• First step:

\[
\begin{align*}
(x, \{x \rightarrow 7\}) & \rightarrow (7, \{x \rightarrow 7\}) \\
(x > 5, \{x \rightarrow 7\}) & \rightarrow (7 > 5, \{x \rightarrow 7\}) \\
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) & \rightarrow \text{(if } 7 > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\})
\end{align*}
\]
Example Evaluation

• Second Step:

\[
\begin{align*}
(7 > 5, \{x \rightarrow 7\}) \rightarrow (true, \{x \rightarrow 7\}) \\
(if \ 7 > 5 \ then \ y:=2 + 3 \ else \ y:=3 + 4 \ fi, \{x \rightarrow 7\}) \\
\rightarrow (if \ true \ then \ y:=2 + 3 \ else \ y:=3 + 4 \ fi, \{x \rightarrow 7\})
\end{align*}
\]

• Third Step:

\[
(if \ true \ then \ y:=2 + 3 \ else \ y:=3 + 4 \ fi, \{x \rightarrow 7\})
\rightarrow (y:=2+3, \{x\rightarrow7\})
\]
Example Evaluation

• Fourth Step:

\[(2+3, \{x\rightarrow7\}) \rightarrow (5, \{x\rightarrow7\})\]

\[(y:=2+3, \{x\rightarrow7\}) \rightarrow (y:=5, \{x\rightarrow7\})\]

• Fifth Step:

\[(y:=5, \{x\rightarrow7\}) \rightarrow \{y \rightarrow 5, x \rightarrow 7\}\]
Example Evaluation

• Bottom Line:

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

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

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

\[\rightarrow (y := 2 + 3, \quad \{x \rightarrow 7\})\]

\[\rightarrow (y := 5, \quad \{x \rightarrow 7\})\]

\[\rightarrow \quad \{y \rightarrow 5, \ x \rightarrow 7\}\]
Adding Local Declarations

• Add to expressions:

\[ E ::= \ldots \mid \text{let } x = E \text{ in } E' \mid \text{fun } x \rightarrow E \mid E \ E' \]

  • Recall: \( \text{fun } x \rightarrow E \) 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

  \[(C_1,m_1) \rightarrow (C_2,m_2) \rightarrow (C_3,m_3) \rightarrow \ldots \rightarrow (\text{skip, m}) \rightarrow m\]

• **Definition**: let \(\rightarrow^*\) be the transitive closure of \(\rightarrow\)
  
i.e., the smallest transitive relation containing \(\rightarrow\)
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**
- Example:

  $$E= 2 + 3 + 4$$

  $$E_1 = 5 + 4$$

  $$E_2 = 2 + 7$$

  $$V = 9$$
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 $\lambda$-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 \( \downarrow \))

\[
\begin{align*}
(C, m) &\to (C", m') \\
(C \text{ par } C', m) &\to (C" \text{ par } C', m') \\
(C', m) &\to (C", m') \\
(C \text{ par } C', m) &\to (C \text{ par } C", m') \\
(C, m) &\to (C", m') \\
(C \text{ par skip, } m) &\to (C" , m') \\
(C', m) &\to (C", m') \\
\text{(skip par } C', m) &\to (C", m')
\end{align*}
\]
Extension: Nondeterministic

• E.g., nondeterministic assignment $x = E_1 \[\] E_2$
  • 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 $\alpha$):

$$P ::= \text{true} \mid \text{false}$$
$$\quad \mid \text{not} \ P \mid P_1 \ bop \ P_2 \mid Aexp_1 \ rop \ Aexpr_2$$

$$Aexp ::= \alpha \mid n \mid Aexp_1 + Aexp_2 \mid Aexp_1 \ast Aexp_2$$
$$\quad \mid Aexp_1 - Aexp_2 \mid Aexp_1 / Aexp_2$$

Memory store: $\Sigma: Var \rightarrow Aexp$

Analysis state $(P, \Sigma)$:

- $P$ is called *path condition*, and $\Sigma$ a *symbolic state*. 
Arithmetic And Relational Expressions

\[(E_1, \Sigma) \downarrow A_{\text{exp}1'} \quad (E_2, \Sigma) \downarrow A_{\text{exp}2'}\]

\[\frac{\quad (E_1 \text{ op } E_2, \Sigma) \downarrow A_{\text{exp}1'} \text{ op } A_{\text{exp}2'}\quad }{\quad (E_1, \Sigma) \downarrow A_{\text{exp}1'} \quad (E_2, \Sigma) \downarrow A_{\text{exp}2'} \quad P= A_{\text{exp}1'} \text{ rop } A_{\text{exp}2'}\quad }\]

\[(E \text{ rop } E', \Sigma) \downarrow P\]
Statements

Skip: \[(P, \text{skip}, \Sigma) \downarrow (P, \Sigma)\]

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

Sequencing: \[
\frac{(P, C, \Sigma) \downarrow (P', \Sigma')}{(P', C', \Sigma') \downarrow \Sigma''}
\]
\[
(P, C; C', \Sigma) \downarrow \Sigma''
\]
If Then Else Statement

\[(B, \Sigma) \downarrow Pb \quad \text{SAT}(P \land Pb) \quad (P \land Pb, C, \Sigma) \downarrow (P', \Sigma') \]
\[
\text{(if B then C else C' fi, } \Sigma) \downarrow (P', \Sigma')
\]

\[(B, \Sigma) \downarrow Pb \quad \text{SAT}(P \land \neg Pb) \quad (P \land \neg Pb, C', \Sigma) \downarrow (P', \Sigma') \]
\[
\text{(if B then C else C' fi, } \Sigma) \downarrow (P', \Sigma')
\]

Both are possibly satisfiable (due to symbolic abstraction)!

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

\textbf{Static Symbolic execution:} We “merge” the formulas of both branches and simplify them. This will be clearer after we cover abstract interpretation next!
Example

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

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

// Question: Is y ≥ 0
// after the execution?
```

(See the lecture video)
Another Example

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

// Question: can the code experience an error?

```python
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 Evaluation for Loops: Rule

Together: Let us derive the rule for the finite loop
\[ \text{while}_k (\text{condition}) \quad -- \text{for a constant } k > 0 \]
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