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

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

• $S ::= \text{skip} \mid S; S \mid I ::= E$
  
  \quad | \quad \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:

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

```c
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)

```llvm
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 \( n \) ("\( d \) 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 \( n \) ("\( d \) 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 "\( d \) 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** — $d_1 \text{ dom } d_2 \land d_2 \text{ dom } d_3 \Rightarrow d_1 \text{ dom } d_3$

• **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

**Source Program**

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

**Mathematical Program Environment**

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

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)
  • ...
  • mₙ₊₁ = [[Sₙ]](mₙ)

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

We can define the sequence (s₁, m₁), (s₂, m₂), (s₃, m₃), ... ,(sₙ, mₙ), (., mₙ₊₁) or its projection (m₁, ... mₙ) 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 ∈ Identifiers
• N ∈ 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:**  
  \[(\text{true},m) \downarrow \text{true}\]  
  \[(\text{false},m) \downarrow \text{false}\]
Booleans:

\[
\begin{align*}
(B, m) \downarrow \text{false} & \quad & (B, m) \downarrow \text{true} & \quad & (B', m) \downarrow b \\
(B \& B', m) \downarrow \text{false} & \quad & (B \& B', m) \downarrow b \\
(B, m) \downarrow \text{true} & \quad & (B, m) \downarrow \text{false} & \quad & (B', m) \downarrow b \\
(B \text{ or } B', m) \downarrow \text{true} & \quad & (B \text{ or } B', m) \downarrow b \\
(B, m) \downarrow \text{true} & \quad & (B, m) \downarrow \text{false} & \quad & (\text{not } B, m) \downarrow \text{false} \\
(\text{not } B, m) \downarrow \text{false} & \quad & (\text{not } B, m) \downarrow \text{true}
\end{align*}
\]
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: \[
\frac{(E,m) \downarrow V}{(k := E,m) \downarrow m [k \leftarrow V]}
\]

Sequencing: \[
\frac{(C,m) \downarrow m' \quad (C',m') \downarrow m''}{(C; C', m) \downarrow m''}
\]
If Then Else Command

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

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

↓ ?
Example: If Then Else Rule

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

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

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

7 > 5 = true

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

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

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

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

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

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x, \{x \rightarrow 7\}) \downarrow 7 &\quad (5, \{x \rightarrow 7\}) \downarrow 5 \\
(x \rightarrow 7) \downarrow \text{true} &\quad (2+3, \{x \rightarrow 7\}) \downarrow ? \\
(y := 2 + 3, \{x \rightarrow 7\}) &\quad (y := 3 + 4, \{x \rightarrow 7\}) \downarrow ?
\end{align*}
\]

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

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

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

\[ 2 + 3 = 5 \]
\[ (2, \{x \rightarrow 7\}) \downarrow 2 \quad (3, \{x \rightarrow 7\}) \downarrow 3 \]

\[ 7 > 5 = \text{true} \]
\[ (x, \{x \rightarrow 7\}) \downarrow 7 \quad (5, \{x \rightarrow 7\}) \downarrow 5 \]
\[ (x > 5, \{x \rightarrow 7\}) \downarrow \text{true} \]

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

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

\[
2 + 3 = 5
\]

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

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

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

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

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

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

\[
\downarrow \{x \rightarrow 7, y \rightarrow 5\}
\]
While Command

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

\[1\]

\[(B, m) \downarrow \text{true} \quad 2 \quad (C, m) \downarrow m' \quad 3 \quad (\text{while } B \text{ do } C \text{ od}, m') \downarrow m'' \]

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

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

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

3. \( (x > 5, \{x \rightarrow 2\}) \downarrow \text{false} \)

while \( x > 5 \) do \( x := x - 5 \) od;

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

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

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

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

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

*The rule assumes the loop terminates!*
While Command

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

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

_The rule assumes the loop terminates!_

\[\text{while } (x>0) \text{ do } x:=x+1\text{ od, } \{x \rightarrow 1\} \Downarrow ? ? ?\]
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:

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

\[
\begin{align*}
\text{compute}_\text{exp} & (\text{Var}(v), m) = \text{look}_\text{up} \ v \ m \\
\text{compute}_\text{exp} & (\text{Int}(n), _) = \text{Num} \ (n)
\end{align*}
\]

...  
\[
\begin{align*}
\text{compute}_\text{com} & (\text{IfExp}(b, c1, c2), m) = \\
\text{if } & \text{compute}_\text{exp} (b, m) = \text{Bool}(\text{true}) \\
\text{then } & \text{compute}_\text{com} (c1, m) \\
\text{else } & \text{compute}_\text{com} (c2, m)
\end{align*}
\]
Natural Semantics Interpreter Implementation

• Loop: \[
\begin{align*}
(B, m) \downarrow \text{false} & \quad \Rightarrow \quad (\text{while } B \text{ do } C \text{ od}, m) \downarrow m \\
(B,m) \downarrow \text{true} & \quad \Rightarrow \quad (C,m) \downarrow m' \quad \text{(while } B \text{ do } C \text{ od, } m') \downarrow m'' \\
\end{align*}
\]

\[
\text{compute_com (While(b,c), m) =}
\]

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

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