Lecture 2: Program Semantics

Rohan Padhye and Jonathan Aldrich

Program Analysis

isr institute for SOFTWARE RESEARCH

Learning goals

- Use big- and small-step semantics to show how While programs evaluate
- Use small-step semantics to show how While3Addr programs evaluate
- Use structural induction to prove things about program semantics

WHILE (concrete) syntax

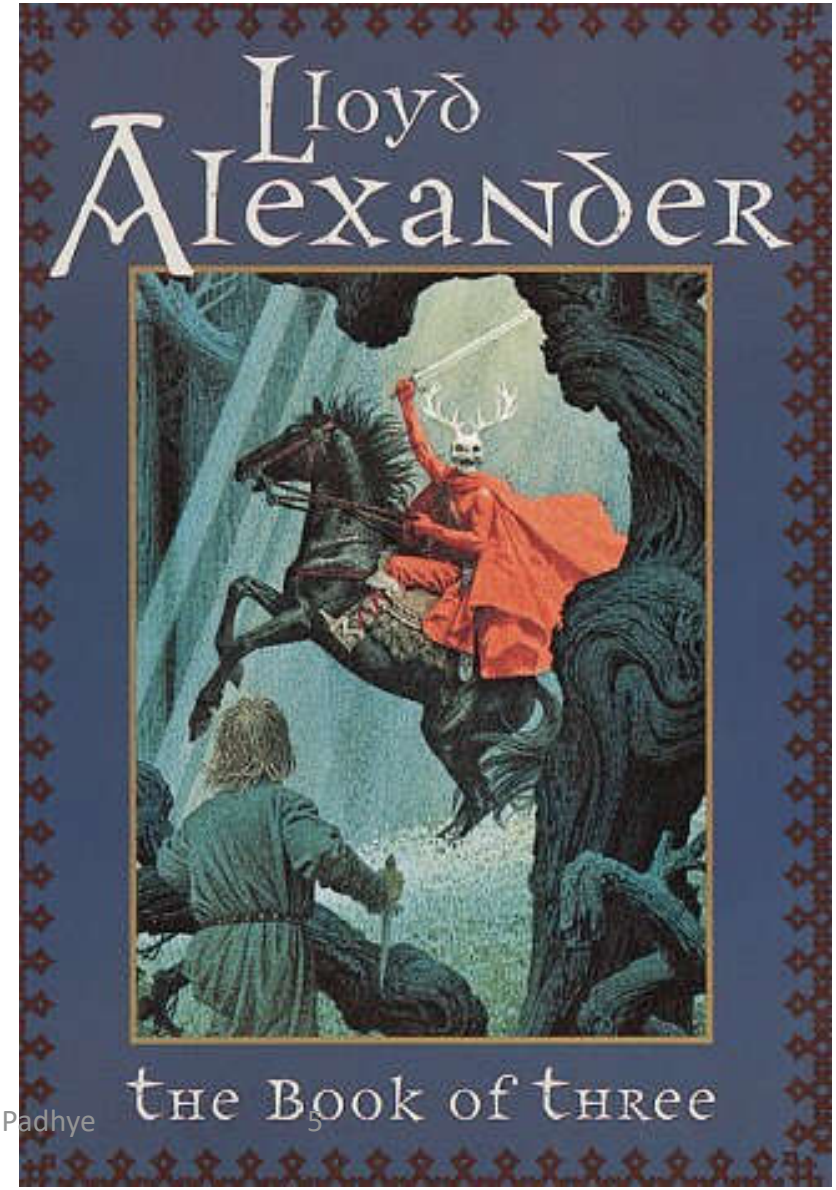
- Categories:
  - $S \in$ **Stmt** statements
  - $a \in$ **Aexp** arithmetic expressions
  - $x, y \in$ **Var** variables
  - $n \in$ **Num** number literals
  - $P \in$ **BExp** boolean predicates
  - $l \in$ **labels** statement addresses (line numbers)
- Syntax:
  - S ::= x := a | skip | S_1 ; S_2
    | if P then S_1 else S_2 | while P do S
  - a ::= x | n | a_1 op_a a_2
  - op_a ::= + | - | * | / | …
  - P ::= true | false | not P | P_1 op_b P_2 | $a1$ op_r $a2$
  - op_b ::= and | or | …
  - op_r ::= < | ≤ | = | > | ≥ | ...

Analysis of WHILE

- Questions to answer:
  - What is the “meaning” of a given While expression/statement ?
  - How would we go about evaluating While expressions and statements?
  - How are the evaluator and the meaning related?

Three Canonical Approaches

- Operational
  - How would I execute this?
- Axiomatic
  - What is true after I execute this?
  - Symbolic Execution
- Denotational
  - What function is this trying to compute?



An Operational Semantics

- Specifies how expressions and statements should be evaluated
- Depending on the form of the expression
  - 0, 1, 2, . . . don't evaluate any further.
    - They are normal forms or values.
  - $a_1 + a_2$ is evaluated by first evaluating a_1 to n_1, then evaluating a_2 to n_2. (post-order traversal)
    - The result of the evaluation is the literal representing $n_1 + n_2$.
  - Similarly for $a_1 * a_2$
- Operational semantics abstracts the execution of a concrete interpreter

Semantics of WHILE

- The meanings of WHILE expressions depend on the values of variables
  - What does "x+5" mean? It depends on "x"!
- The value of integer variables at a given moment is abstracted as a function from Var to **Z** (a state)
  - If x = 8 in our state, we expect "x+5" to mean 13
- We use E, a state, to denote a map from variables to values

Notation: Judgment

- We write:

$$\langle E, a \rangle \Downarrow n$$

- To mean that a evaluates to n in state E.
- This is a judgment. It asserts a relation between E, a, and n.
- In this case we can view $\Downarrow$ as a function with two arguments (E and a).

Operational Semantics

- This formulation is called big-step operational semantics
  - or natural operational semantics
  - the $\Downarrow$ judgment relates the expression and its "meaning"
- How should we define

$$< E, a_1 + a_2 > \Downarrow \ldots ?$$

Notation: Rules of Inference

- We express the evaluation rules as rules of inference for our judgment
  - called the derivation rules for the judgment
  - also called the evaluation rules (for operational semantics)
- Typically, we have a rule per language construct:

$$\frac{\langle E, a_1 \rangle \Downarrow n_1 \quad \langle E, a_2 \rangle \Downarrow n_2}{\langle E, a_1 + a_2 \rangle \Downarrow n_1 + n_2}$$

This is the *only* rule for $a_1 + a_2$

Rules of Inference

$$\frac{\text{Hypothesis}_1 \ \ldots \ \text{Hypothesis}_N}{\text{Conclusion}}$$

$$\frac{\langle E, a_1 \rangle \Downarrow n_1 \quad \langle E, a_2 \rangle \Downarrow n_2}{\langle E, a_1 + a_2 \rangle \Downarrow n_1 + n_2}$$

- For any given proof system, a finite number of rules of inference (or schema) are listed somewhere
- Rule instances should be easily checked

Evaluation Rules (for Aexp)

$$\frac{}{\langle E, n\rangle \Downarrow n} \qquad \frac{}{\langle E, x\rangle \Downarrow E(x)}$$

$$\frac{\langle E, a_1\rangle \Downarrow n_1 \quad \langle E, a_2\rangle \Downarrow n_2}{\langle E, a_1 + a_2\rangle \Downarrow n_1 + n_2} \qquad \frac{\langle E, a_1\rangle \Downarrow n_1 \quad \langle E, a_2\rangle \Downarrow n_2}{\langle E, a_1 - a_2\rangle \Downarrow n_1 - n_2}$$

$$\frac{E, a_1\rangle \Downarrow n_1 \quad \langle E, a_2\rangle \Downarrow n_2}{\langle E, a_1 - a_2\rangle \Downarrow n_1 * n_2}$$

- This is called structural operational semantics
  - rules defined based on the structure of the expression
- These rules do not impose an order of evaluation!

Derivations

$$\dfrac{\dfrac{\langle E_1, 4\rangle \Downarrow 4 \quad \langle E_1, 2\rangle \Downarrow 2}{\langle E_1, 4 * 2\rangle \Downarrow 8} \qquad \langle E_1, 6\rangle \Downarrow 6}{\langle E_1, (4 * 2) - 6\rangle \Downarrow 2}$$

Evaluation of Statements

- The evaluation of a Stmt may have side effects but has no direct result
  - What is the result of evaluating a statement?
- The "result" of a Stmt is a new state:

$$< E, S > \Downarrow E'$$

  - Note: the evaluation of S might not terminate! We'll come back to this issue

Stmt Evaluation Rules 1

$$\frac{}{\langle E, \texttt{skip} \rangle \Downarrow E}$$

$$\frac{\langle E, S_1 \rangle \Downarrow E' \qquad \langle E', S_2 \rangle \Downarrow E''}{\langle E, S_1 ; S_2 \rangle \Downarrow E''}$$

How would you write rule(s) for if?

Stmt Evaluation Rules 1

$$\frac{}{\langle E, \text{skip} \rangle \Downarrow E} \qquad \frac{\langle E, S_1 \rangle \Downarrow E' \quad \langle E', S_2 \rangle \Downarrow E''}{\langle E, S_1 ; S_2 \rangle \Downarrow E''}$$

$$\frac{\langle E, P \rangle \Downarrow \text{true} \quad \langle E, S_1 \rangle \Downarrow E'}{\langle \text{if } P \text{ then } S_1 \text{ else } S_2 \rangle \Downarrow E'}$$

$$\frac{\langle E, P \rangle \Downarrow \text{false} \quad \langle E, S_2 \rangle \Downarrow E'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2 \rangle \Downarrow E'}$$

Stmt Evaluation Rules 2

$$\frac{\langle E, a\rangle \Downarrow n}{\langle E, x := a\rangle \Downarrow E[x \to n]}$$

Def: $E[x \to n](x) = n$

$E[x \to n](y) = E(y)$

- Practice: write the rule(s) for while

Stmt Evaluation Rules 3

$$\frac{\langle E, a\rangle \Downarrow n}{\langle E, x := a\rangle \Downarrow E[x \to n]}$$

Def: $E[x \to n](x) = n$

$E[x \to n](y) = E(y)$

$$\frac{\langle E, P\rangle \Downarrow \text{false}}{\langle E, \text{while P do S}\rangle \Downarrow E}$$

$$\frac{\langle E, P\rangle \Downarrow \text{true} \quad \langle E, S; \text{while P do S}\rangle \Downarrow E'}{\langle E, \text{while P do S}\rangle \Downarrow E'}$$

Big-Step Evaluation Issues

- The evaluation rules are not syntax-directed
  - See the rules for while
  - The evaluation might not terminate
- Recall: the evaluation rules suggest an interpreter
- Big-step semantics has two big disadvantages (continued …)

Disadvantages of Natural-Style Operational Semantics

- It is hard to talk about statements whose evaluation does not terminate
  - When there is no E' such that <E, S> $\Downarrow$ E'
  - But that is true also of ill-formed or erroneous statements (in a richer language)!
- It does not give us a way to talk about intermediate states
  - Thus we cannot say that on a parallel machine the execution of two commands is interleaved (= no modeling threads)

Semantics Solution

- **Small-step semantics** addresses these problems
  - Execution is modeled as a (possible infinite) sequence of states
  - Each atomic execution step rewrites the program

Small step semantics

- We will define a relation <E, S> → <E', S'>
  - S' is obtained from S via a rewrite step
  - Evaluation terminates when the program has been rewritten to a terminal program
    - one from which we cannot make further progress
  - For While the terminal command is "skip"
  - As long as the command is not "skip" we can make further progress
    - some commands *never* reduce to skip (e.g., "while true do skip")

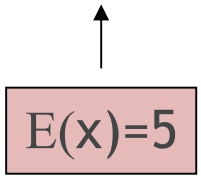
Configurations

- A pair of state and statement: <E, S>.
- Big step judgment relates a configuration to a new state: <E, S> $\Downarrow$ E'
- Small step relates a configuration to a new configuration: <E, S> $\rightarrow$ <E', S'>
- We have one such arrow per grammar production, except Aexp and Bexp do not produce new states (as before).

Executions

- Key idea: we repeatedly rewrite the program until we reach a final configuration.
- A small step execution is a sequence (or list) of rewrites:

<E,x+(7-3)> → <E,x+(4)> → <E,5+4> → <E,9>



While: Small-step examples

$$\frac{}{\langle E, x\rangle \rightarrow_a E(x)}$$

$$\frac{\langle E, P\rangle \rightarrow_b P'}{\langle E, \text{if } P \text{ then } S_1 \text{ else } S_2\rangle \rightarrow \langle E, \text{if } P' \text{ then } S_1 \text{ else } S_2\rangle}$$

$$\langle E, \text{if true then } S_1 \text{ else } S_2\rangle \rightarrow \langle E, S_1\rangle$$

While: Small-step examples

$$\frac{\langle E, S_1 \rangle \rightarrow \langle E', S_1' \rangle}{\langle E, S_1; S_2 \rangle \rightarrow \langle E', S_1'; S_2 \rangle}\ \textit{small-seq-congruence}$$

$$\frac{}{\langle E, \texttt{skip}; S_2 \rangle \rightarrow \langle E, S_2 \rangle}\ \textit{small-seq}$$

Practice: write small-step rule(s) for a while loop

While3Addr: a different representation

- While3Addr programs are mappings between numbers and instructions.
- So, the configuration needs to include a program counter:

$$c \in E \times \mathcal{N}$$

- And the judgment is slightly different:

$$P \vdash < E, n > \rightsquigarrow < E', n' >$$

$$\frac{P(n) = x := m}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E[x \mapsto m], n+1 \rangle} \textit{ step-const}$$

$$\frac{P(n) = x := m}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto m], n+1\rangle}\ \textit{step-const}$$

$$\frac{P[n] = x := y}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto E(y)], n+1\rangle}\ \textit{step-copy}$$

$$\frac{P(n) = x := y\ op\ z \quad E(y)\ \mathbf{op}\ E(z) = m}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto m], n+1\rangle}\ \textit{step-arith}$$

$$\frac{P(n) = \text{goto } m}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E, m\rangle}\ \textit{step-goto}$$

$$\frac{P(n) = \text{if } x\ op_r\ 0 \text{ goto } m \quad E(x)\ \mathbf{op_r}\ 0 = true}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E, m\rangle}\ \textit{step-iftrue}$$

$$\frac{P(n) = \text{if } x\ op_r\ 0 \text{ goto } m \quad E(x)\ \mathbf{op_r}\ 0 = false}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E, n+1\rangle}\ \textit{step-iffalse}$$

Provability

- Given an opsem system, $\langle E, a \rangle \Downarrow n$ is **provable** *if there exists* a well-formed derivation with $\langle E, a \rangle \Downarrow n$ as its conclusion
  - "well-formed" = "every step in the derivation is a valid instance of one of the rules of inference for this opsem system"
  - "$\langle E, a \rangle \Downarrow n$" = "it is provable that $\langle E, a \rangle \Downarrow n$"

Proof by Mathematical Induction

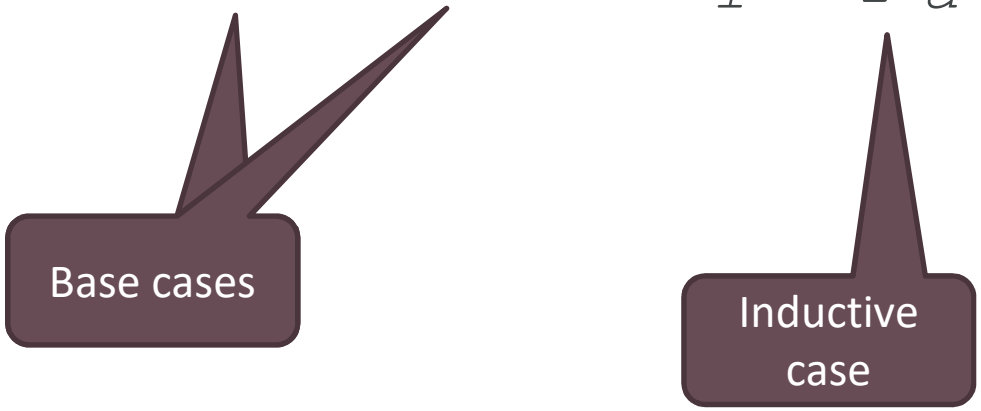
Prove $\forall n.P(n)$ by induction on natural numbers

- Base case: show that P(0) holds
- Inductive case: show that P(m) implies P(m+1)

Structural Induction

- We can also do induction over syntax

$$a ::= x \mid n \mid a_1 \; op_a \; a_2$$



Example Proof by Structural Induction

Example. Let $L(a)$ be the number of literals and variable occurrences in some expression a and $O(a)$ be the number of operators in a. Prove by induction on the structure of a that $\forall a \in \texttt{Aexp} \,.\, L(a) = O(a) + 1$:

Base cases:

- Case $a = n$. $L(a) = 1$ and $O(a) = 0$
- Case $a = x$. $L(a) = 1$ and $O(a) = 0$

Inductive case 1: Case $a = a_1 + a_2$

- By definition, $L(a) = L(a_1) + L(a_2)$ and $O(a) = O(a_1) + O(a_2) + 1$.
- By the induction hypothesis, $L(a_1) = O(a_1) + 1$ and $L(a_2) = O(a_2) + 1$.
- Thus, $L(a) = O(a_1) + O(a_2) + 2 = O(a) + 1$.

The other arithmetic operators follow the same logic.

Induction on Syntax

- Provable by induction on syntax

$$\forall a \in \texttt{AExp} \,.\, \langle E, a\rangle \rightarrow_a^* n \Leftrightarrow \langle E, a\rangle \Downarrow n$$

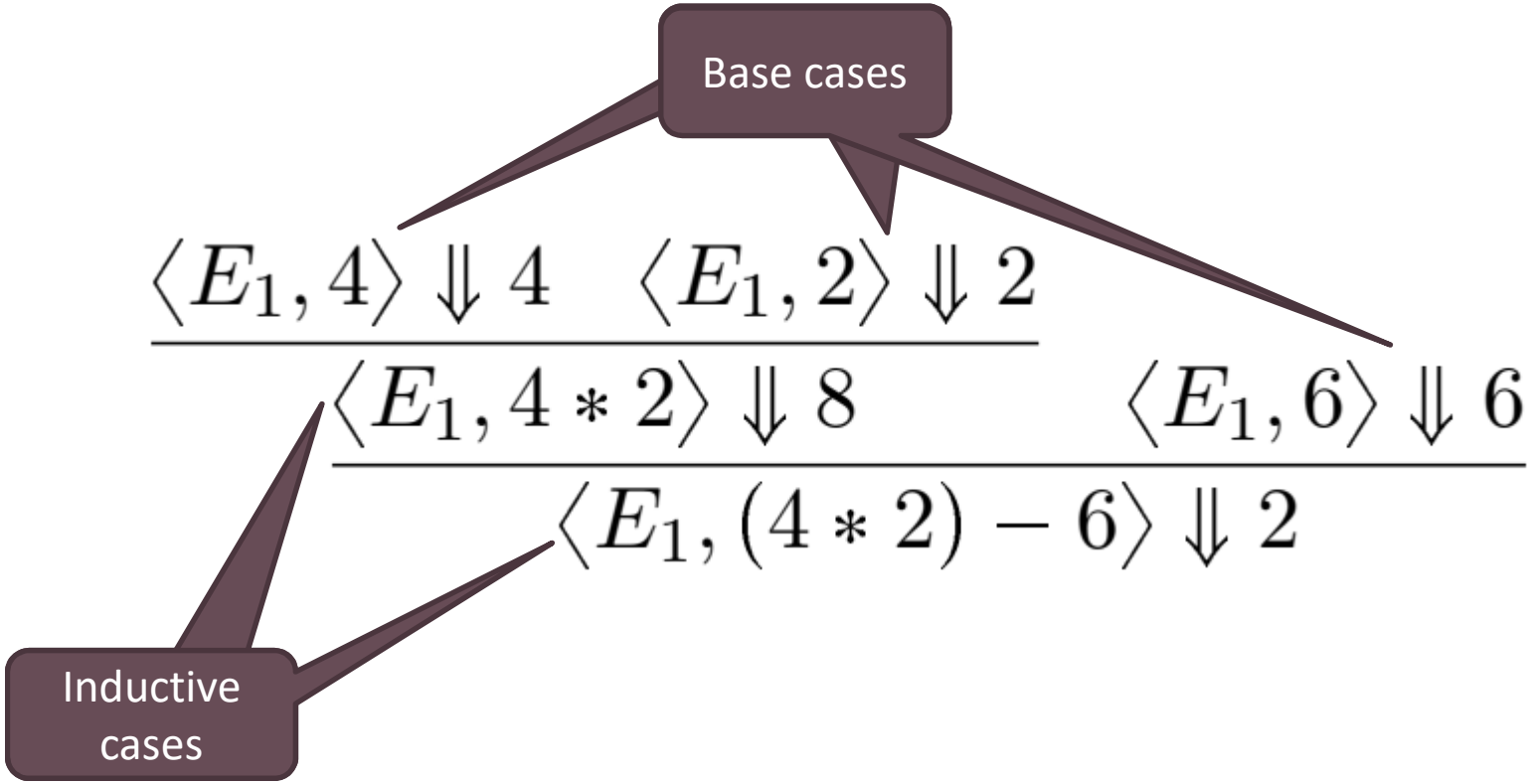
- Not provable by induction on syntax

$$\forall a \in \texttt{Aexp} \,.\quad \forall E \,.\, \forall n, n' \in \mathbb{N} \,.\quad \langle E, a\rangle \Downarrow n \wedge \langle E, a\rangle \Downarrow n' \Rightarrow n = n'$$
$$\forall P \in \texttt{Bexp} \,.\quad \forall E \,.\, \forall b, b' \in \mathcal{B} \,.\quad \langle E, P\rangle \Downarrow b \wedge \langle E, P\rangle \Downarrow b' \Rightarrow b = b'$$
$$\forall S \,.\qquad\quad \forall E, E', E'' \,.\qquad \langle E, S\rangle \Downarrow E' \wedge \langle E, S\rangle \Downarrow E'' \Rightarrow E' = E''$$

Rule for while doesn't just depend on subexpressions

Induction on Derivations

- Derivations have recursive structure



Determinism of Statements

$$\forall S\,. \qquad \forall E, E', E''\,. \qquad \langle E, S\rangle \Downarrow E' \wedge \langle E, S\rangle \Downarrow E'' \Rightarrow E' = E''$$

Proof: by induction of the structure of the derivation D, which we define $D :: \langle E, S\rangle \Downarrow E'$.

Base case: the one rule with no premises, `skip`:

$$D :: \frac{}{\langle E, \texttt{skip}\rangle \Downarrow E}$$

By inversion, the last rule used in D' (which, again, produced E'') must also have been the rule for `skip`. By the structure of the `skip` rule, we know $E'' = E$.

Inductive cases: We need to show that the property holds when the last rule used in D was each of the possible non-skip WHILE commands. I will show you one representative case; the rest are left as an exercise. If the last rule used was the `while-true` statement:

$$D :: \frac{D_1 :: \langle E, P\rangle \Downarrow \texttt{true} \quad D_2 :: \langle E, S\rangle \Downarrow E_1 \quad D_3 :: \langle E_1, \texttt{while}\ P\ \texttt{do}\ S\rangle \Downarrow E'}{\langle E, \texttt{while}\ P\ \texttt{do}\ S\rangle \Downarrow E'}$$

Pick arbitrary E'' such that $D'' :: \langle E, \texttt{while}\ P\ \texttt{do}\ S\rangle \Downarrow E''$

By inversion, and determinism of boolean expressions, D'' must also use the same `while-true` rule. So D'' must also have subderivations $D_2'' :: \langle E, S\rangle \Downarrow E_1''$ and $D_3'' :: \langle E_1'', \texttt{while}\ P\ \texttt{do}\ S\rangle \Downarrow E''$. By the induction hypothesis on D_2 with D_2'', we know $E_1 = E_1''$. Using this result and the induction hypothesis on D_3 with D_3'', we have $E'' = E'$.