Lecture 2: Abstract Syntax and Program Semantics

17-355/17-665/17-819: Program Analysis

Rohan Padhye

August 28 and September 2, 2025

* Course materials developed with Jonathan Aldrich and Claire Le Goues

Administrivia

- HW1 will be out tonight – CodeQL. Due next Thursday (Sep 4).
  - Lots of references online
  - Recitation will have some practice problems
  - Submit via Gradescope.
- Office hours are up on website
- Lecture notes/slides on website
  - Read after class; useful for HW and exams (won't always have slides)
  - Text PDF updates frequently (usually before class); get latest copy
  - For now, ignore 2.2, 2.4, 3.1.3 (WHILE3ADDR) – We'll cover it next week
- Please bring paper/pen for in-class exercises

Learning Goals

- Recognize the basic WHILE demonstration language and define its abstract syntax.
- Describe the function of an AST and outline the principles behind AST walkers for simple bug-finding analyses
- Define the meaning of programs using operational semantics
- Read and write inference rules and derivation trees
- Use big- and small-step semantics to show how WHILE programs evaluate
- Use structural induction to prove things about program semantics

Recap: Concrete vs. Abstract Syntax

- A tree representation of source code based on the language grammar.
- **Concrete syntax**: The rules by which programs can be expressed as strings of characters
  - E.g. "if (x * (a + b)) { foo(a); }"
  - Use finite automata and context-free grammars, automatic lexer/parser generators
- **Abstract syntax**: a subset of the parse tree of the program.
  - Only care about statements, expressions and their relationship with constituent operands.
  - Don't care about parenthesis, semicolons, keywords, etc.
- (The intuition is fine for this course; take compilers if you want to learn how to parse for real.)

The WHILE language – Example program

```
y := x;
z := 1;
if y > 0 then
   while y > 1 do
      z := z * y;
      y := y - 1
else
   skip
```

- Sample program computes z = *x!* using *y* as a temp variable.
- WHILE uses assignment statements, if-then-else, while loops.
- All vars are integers.
- Expressions only arithmetic (for vars) or relational (for conditions).
- No I/O statements. Inputs and outputs are implicit.
  - Later on, we may use extensions with explicit `read x` and `print x`.

WHILE abstract syntax

S	statements
a	arithmetic expressions (AExp)
x, y	program variables (Vars)
n	number literals
b	boolean expressions (BExp)

We'll use these meta-variables frequently for ease of notation

$$
\begin{array}{lcl}
S & ::= & x := a \\
 & | & \text{skip} \\
 & | & S_1;\ S_2 \\
 & | & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & | & \text{while } b \text{ do } S
\end{array}
\qquad
\begin{array}{lcl}
b & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \text{not } b \\
 & | & b_1\ op_b\ b_2 \\
 & | & a_1\ op_r\ a_2
\end{array}
\qquad
\begin{array}{lcl}
a & ::= & x \\
 & | & n \\
 & | & a_1\ op_a\ a_2
\end{array}
\qquad
\begin{array}{lcl}
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \mid \leqslant \mid = \\
 & | & > \mid \geqslant \\
op_a & ::= & + \mid - \mid * \mid /
\end{array}
$$

Exercise: Building an AST

```
y := x;
z := 1;
if y > 0 then
  while y > 1 do
    z := z * y;
    y := y - 1
else
  skip
```

S	statements
a	arithmetic expressions (AExp)
x, y	program variables (Vars)
n	number literals
b	boolean expressions (BExp)

$$
\begin{array}{rcl}
S & ::= & x := a \\
 & | & \text{skip} \\
 & | & S_1;\ S_2 \\
 & | & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & | & \text{while } b \text{ do } S
\end{array}
\qquad
\begin{array}{rcl}
b & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \text{not } b \\
 & | & b_1\ op_b\ b_2 \\
 & | & a_1\ op_r\ a_2
\end{array}
\qquad
\begin{array}{rcl}
a & ::= & x \\
 & | & n \\
 & | & a_1\ op_a\ a_2
\end{array}
\qquad
\begin{array}{rcl}
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \mid \leqslant \mid = \\
 & | & > \mid \geqslant \\
op_a & ::= & + \mid - \mid * \mid /
\end{array}
$$

Our first static analysis: AST walking

- One way to find “bugs” is to walk the AST, looking for particular patterns.
  - Traverse the AST, look for nodes of a particular type
  - Check the neighborhood of the node for the pattern in question.
  - Basically, a glorified “grep” that knows about the syntax but not semantics of a language.

Example: shifting by more than 31 bits.

Assume we want to find code patterns of the following form:

```
x << -3
z >> 35
```

For 32-bit integer vars, these operations may signal unintended typos, since it doesn't makes sense to shift by a number outside the range (0, 32).

Example: shifting by more than 31 bits.

```
For each instruction I in the program
  if I is a shift instruction
      if (type of I's left operand is int
            && I's right operand is a constant
            && value of constant < 0 or > 31)
         warn("Shifting by less than 0 or more
              than 31 is meaningless")
```

Our first static analysis: AST walking

- One way to find “bugs” is to walk the AST, looking for particular patterns.
  - Traverse the AST, look for nodes of a particular type
  - Check the neighborhood of the node for the pattern in question.
- Various frameworks, some more language-specific than others.
  - Tradeoffs between language agnosticism and semantic information available.
  - Consider “grep”: very language agnostic, not very smart.
  - Python’s ”astor” package designed for Python ASTs. Clean API; highly specific.
- One common architecture based on Visitor pattern:
  - class Visitor has a visitX method for each type of AST node X
  - Default Visitor code just descends the AST, visiting each node
  - To do something interesting for AST element of type X, override visitX
- Other more recent approaches based on semantic search, declarative logic programming, or query languages.

CodeQL

- A language for querying code. Developed by GitHub.
- Supports many common languages.
- Library of common programming patterns and optimizations.

CodeQL queries 1.23

Dashboard / Java queries

Inefficient empty string test

Created by Documentation team, last modified on Mar 28, 2019

```
from MethodAccess ma
where
    ma.getMethod().hasName("equals") and
    ma.getArgument(0).(StringLiteral).getValue() = ""
select ma, "This comparison to empty string is inefficient, use isEmpty()
instead."
```

Query: InefficientEmptyStringTest.ql › Expand source

When checking whether a string `s` is empty, perhaps the most obvious solution is to write something like `s.equals("")` (or `"".equals(s)`). However, this actually carries a fairly significant overhead, because `String.equals` performs a number of type tests and conversions before starting to compare the content of the strings.

Recommendation

The preferred way of checking whether a string `s` is empty is to check if its length is equal to zero. Thus, the condition is `s.length() == 0`. The `length` method is implemented as a simple field access, and so should be noticeably faster than calling `equals`.

Note that in Java 6 and later, the `String` class has an `isEmpty` method that checks whether a string is empty. If the codebase does not need to support Java 5, it may be better to use that method instead.

Back to WHILE

S	statements
a	arithmetic expressions (AExp)
x, y	program variables (Vars)
n	number literals
b	boolean expressions (BExp)

$$
\begin{array}{lll}
S & ::= & x := a \\
  & \mid & \text{skip} \\
  & \mid & S_1;\ S_2 \\
  & \mid & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
  & \mid & \text{while } b \text{ do } S
\end{array}
\qquad
\begin{array}{lll}
b & ::= & \text{true} \\
  & \mid & \text{false} \\
  & \mid & \text{not } b \\
  & \mid & b_1\ op_b\ b_2 \\
  & \mid & a_1\ op_r\ a_2
\end{array}
\qquad
\begin{array}{lll}
a & ::= & x \\
  & \mid & n \\
  & \mid & a_1\ op_a\ a_2
\end{array}
\qquad
\begin{array}{lll}
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \mid \leqslant \mid = \\
  & \mid & > \mid \geqslant \\
op_a & ::= & + \mid - \mid * \mid /
\end{array}
$$

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 semantics
  - How would I execute this?
  - Interpreter
- Axiomatic semantics
  - What is true after I execute this?
  - Symbolic Execution
- Denotational semantics
  - What function is this trying to compute?
  - Mathematical modeling

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.
  - 4 + 2 is evaluated by adding integers 4 and 2 to get 6.
    - Rule can be generalized for an expression containing only literals: $n_1 + n_2$
  - $a_1 + a_2$ is evaluated by:
    - First evaluating expression a_1 to value n_1
    - Then evaluating expression a_2 to integer n_2
    - The result of the evaluation is the literal representing $n_1 + n_2$
    - Here, evaluation order is being defined as left-to-right (post-order AST traversal)
- Operational semantics *abstracts the execution of a concrete interpreter*.

Big-Step Semantics

- Uses down-arrow $\Downarrow$ notation to denote evaluation to normal form.
- $a \Downarrow n$ is a *judgment* that expression a is evaluated to value n
- For example: $(4 + 2) + 9 \ \Downarrow 15$
- You can think of this as a logical proposition.
  - The semantics of a language determines what judgments are provable.

Inference Rules

$$\frac{premise_1 \quad premise_2 \quad \ldots \quad premise_n}{conclusion}$$

- A notation for defining semantics.
- If ALL of the premises above the line can be proved true, then the conclusion holds as well.

Let's Formalize the tiny ADD language

- 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.
  - 4 + 2 is evaluated by adding integers 4 and 2 to get 6.
    - Rule can be generalized for an expression containing only literals
  - $a_1 + a_2$ is evaluated by:
    - First evaluating expression a_1 to value n_1
    - Then evaluating expression a_2 to integer n_2
    - The result of the evaluation is the literal representing $n_1 + n_2$
    - Here, evaluation order is being defined as left-to-right (post-order AST traversal)
- Operational semantics *abstracts the execution of a concrete interpreter*.

Big-step semantics for ADD

$$\frac{}{n \Downarrow n} \; \textit{big-int}$$

$$\frac{a_1 \Downarrow n_1 \quad a_2 \Downarrow n_2}{a_1 + a_2 \Downarrow n_1 + n_2} \; \textit{big-add}$$

Derivation trees

$$\frac{a_1 \Downarrow n_1 \quad a_2 \Downarrow n_2}{a_1 + a_2 \Downarrow n_1 + n_2} \; \textit{big-add}$$

- Let's derive $(4 + 2) + 9 \Downarrow 15$ from the rules

$$\cfrac{\cfrac{4 \Downarrow 4 \quad 2 \Downarrow 2}{4 + 2 \Downarrow 6} \quad 9 \Downarrow 9}{(4 + 2) + 9 \Downarrow 15}$$

- The derivation provides a proof of $(4 + 2) + 9 \Downarrow 15$ using only axioms and inference rules.

Operational Semantics of WHILE

- The meaning of WHILE expressions depend on the values of variables
  - What does x+5 mean? It depends on x.
  - If $x = 8$ at some point, we expect x+5 to mean 13

- The value of integer variables at a given moment is abstracted as a function:

$$E : Var \to Z$$

- We will augment our notation of big-step evaluation to include state:

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

- So, if $\{x \mapsto 8\} \in E$, then $\langle E, x + 5 \rangle \Downarrow 13$

Big-Step Semantics for WHILE expressions

$$\frac{}{\langle E, n \rangle \Downarrow n} \textit{ big-int} \qquad \frac{}{\langle E, x \rangle \Downarrow E(x)} \textit{ big-var}$$

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

- Similarly for other arithmetic and boolean expressions

States propagate in derivations

- Let $E_1 = \{x \mapsto 4\}$. What will $x \ * 2 \ - 6$ evaluate to in this state?

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

$\vdash \langle \mathrm{E}_1, x \ * 2 \ - 6\rangle \Downarrow 2$ (this evaluation is provable via a well-formed derivation)

Big-Step Semantics for WHILE statements

- Statements do not evaluate to values.
- However, statements can have side-effects.
- Notation for statement evaluations: $\langle E, S \rangle \Downarrow E'$

$$\frac{}{\langle E, \texttt{skip} \rangle \Downarrow E} \; \textit{big-skip}$$

$$\frac{\langle E, a \rangle \Downarrow n}{\langle E, x := a \rangle \Downarrow E[x \mapsto n]} \; \textit{big-assign}$$

Big-Step Semantics for WHILE statements

$$\frac{\langle E, S_1 \rangle \Downarrow E' \quad \langle E', S_2 \rangle \Downarrow E''}{\langle E, S_1; S_2 \rangle \Downarrow E''} \; \textit{big-seq}$$

$$\frac{\langle E, b \rangle \Downarrow \texttt{true} \quad \langle E, S_1 \rangle \Downarrow E'}{\langle E, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \rangle \Downarrow E'} \; \textit{big-iftrue}$$

$$\frac{\langle E, b \rangle \Downarrow \texttt{false} \quad \langle E, S_2 \rangle \Downarrow E'}{\langle E, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \rangle \Downarrow E'} \; \textit{big-iffalse}$$

Big-Step Semantics for WHILE statements

- Exercise: Write the rule “*big-while*” for

$$\text{while } b \text{ do } S$$

Big-Step Semantics for WHILE statements

$$\frac{\langle E, b\rangle \Downarrow \texttt{false}}{\langle E, \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E}\ \textit{big-whilefalse}$$

$$\frac{\langle E, b\rangle \Downarrow \texttt{true} \qquad \langle E, S; \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E'}{\langle E, \texttt{while}\ b\ \texttt{then}\ S\rangle \Downarrow E'}\ \textit{big-whiletrue}$$

Big-Step Semantics for WHILE statements

$$\frac{\langle E, b\rangle \Downarrow \texttt{false}}{\langle E, \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E}\ \textit{big-whilefalse}$$

Alternate formulation (equivalent to previous slide):

$$\frac{\langle E, b\rangle \Downarrow \texttt{true} \quad \langle E, S \Downarrow E'\rangle \quad \langle E', \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E''}{\langle E, \texttt{while}\ b\ \texttt{then}\ S\rangle \Downarrow E''}\ \textit{big-whiletrue}$$

Big-Step Semantics: Discussion

- Rules suggest an AST interpreter
  - Recursively evaluate operands, then current node (post-order traversal)
- Disadvantages:
  - Cannot reason about non-terminating loops, e.g. while **true** do **skip**
  - Does not model intermediate states
    - Needed for semantics of concurrent execution models (e.g. Java threads)

Small-Step Operational Semantics

- Each step is an atomic rewrite of the program
- Execution is a sequence of (possibly infinite) steps
  - $\langle \mathrm{E}_1, (x * 2) \ - 6 \rangle \rightarrow \langle E_1, (4 * 2) - 6 \rangle \rightarrow \langle E_1, 8 - 6 \rangle \rightarrow 2$

- Small arrow notation for single step:

$$\langle E, a \rangle \rightarrow_a a'$$
$$\langle E, b \rangle \rightarrow_b b'$$
$$\langle E, S \rangle \rightarrow \langle E', S' \rangle$$

(the subscripts on the arrows can be omitted when context is clear)

Small-Step Operational Semantics

- First define a multi-step notation: $\langle E, S\rangle \rightarrow^* \langle E', S'\rangle$

$$\frac{}{\langle E, S\rangle \rightarrow^* \langle E, S\rangle} \; \textit{multi-reflexive}$$

$$\frac{\langle E, S\rangle \rightarrow \langle E', S'\rangle \quad \langle E', S'\rangle \rightarrow^* \langle E'', S''\rangle}{\langle E, S\rangle \rightarrow^* \langle E'', S''\rangle} \; \textit{multi-inductive}$$

- A terminating evaluation of a program P from initial state E_{in} is:

$$\langle E_{in}, P\rangle \rightarrow^* \langle E_{out}, skip\rangle$$

Small-Step Semantics for WHILE expressions

- Axioms are similar:

$$\frac{}{\langle E, x \rangle \rightarrow_a E(x)} \; \textit{small-var}$$

$$\frac{}{\langle E, n \rangle \rightarrow_a n} \; \textit{small-int}$$

Small-Step Semantics for WHILE expressions

- Compound expressions

$$\frac{\langle E, a_1 \rangle \rightarrow_a a_1'}{\langle E, a_1 + a_2 \rangle \rightarrow_a a_1' + a_2} \textit{ small-add-left}$$

$$\frac{\langle E, a_2 \rangle \rightarrow_a a_2'}{\langle E, n_1 + a_2 \rangle \rightarrow_a n_1 + a_2'} \textit{ small-add-right}$$

$$\frac{}{\langle E, n_1 + n_2 \rangle \rightarrow_a n_1 + n_2} \textit{ small-add}$$

Small-Step Semantics for WHILE statements

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

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

Small-Step Semantics for WHILE statements

$$\frac{\langle E, b\rangle \rightarrow_b b'}{\langle E, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle \rightarrow \langle E, \texttt{if}\ b'\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle}\ \textit{small-if-congruence}$$

$$\frac{}{\langle E, \texttt{if true then}\ S_1\ \texttt{else}\ S_2\rangle \rightarrow \langle E, S_1\rangle}\ \textit{small-iftrue}$$

Small-Step Semantics for WHILE statements

- Exercise: Write the rule "*small-while*" for

$$\text{while } b \text{ do } S$$

Small-Step Semantics for WHILE statements

$$\frac{}{\langle E, \texttt{while}\ b\ \texttt{do}\ S\rangle \rightarrow \langle \texttt{if}\ b\ \texttt{then}\ S; \texttt{while}\ b\ \texttt{do}\ S\ \texttt{else}\ \texttt{skip}\rangle}\ \textit{small-while}$$

Provability

- Given some operational semantics, $\langle E, a \rangle \Downarrow n$ 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"

  $\vdash \langle E, a \rangle \Downarrow n$ "it is provable that $\langle E, a \rangle \Downarrow n$"

Proofs over semantics

- Once we have defined semantics clearly, we can now reason about programs rigorously via proofs by *structural induction*.
- But first, recall *mathematical induction:*
  - To prove $\forall n : P(n)$ by induction on natural numbers
    - Base case: show that $P(0)$ holds
    - Inductive case: show that $\forall m : P(m) \Rightarrow P(m+1)$

Proofs by Structural Induction

$$a ::= x \mid n \mid a_1\ op_a\ a_2 \qquad op_a ::= + \mid - \mid * \mid /$$

- To prove $\forall a \in Aexp: P(a)$ by induction on structure of syntax
  - Base cases: show that $P(x)$ and $P(n)$ holds
  - Inductive cases: show that
    - $P(a_1) \wedge P(a_2) \Rightarrow P(a_1 + a_2)$
    - $P(a_1) \wedge P(a_2) \Rightarrow P(a_1 * a_2)$
    - $P(a_1) \wedge P(a_2) \Rightarrow P(a_1 / a_2)$

Proofs 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.

Proofs by Structural Induction

- Prove that small-step and big-step semantics of expressions produce equivalent results.

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

- Can be proved via structural induction over syntax. (Exercise)

Proofs by Structural Induction

- Prove that WHILE is *deterministic*. That is, if the program terminates, it evaluates to a unique value.

$$\forall a \in \texttt{Aexp} \;.\; \forall E \;.\; \forall n, n' \in \mathbb{N} \;.\; \langle E, a \rangle \Downarrow n \wedge \langle E, a \rangle \Downarrow n' \Rightarrow n = n'$$

$$\forall P \in \texttt{Bexp} \;.\; \forall E \;.\; \forall b, b' \in \mathcal{B} \;.\; \langle E, P \rangle \Downarrow b \wedge \langle E, P \rangle \Downarrow b' \Rightarrow b = b'$$

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

Rule for while is recursive; doesn't depend only on subexpressions

- Can prove for expressions via induction over syntax, but not for statements.
- But there's still a way.

To prove: $\forall S . \quad \forall E, E', E'' . \quad \langle E, S\rangle \Downarrow E' \wedge \langle E, S\rangle \Downarrow E'' \Rightarrow E' = E''$

Structural Induction over Derivations

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

let $D :: \langle E, S\rangle \Downarrow E'$, and let $D' :: \langle E, S\rangle \Downarrow E''$

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

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:

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

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

By inversion, D' must use either the while−true or the while−false rule. However, having proved that boolean expressions are deterministic (via induction on syntax), and given that D contains the judgment $\langle E, b\rangle \Downarrow \texttt{true}$, we know that D' cannot be the while−false rule, as otherwise it would have to contain a contradicting judgment $\langle E, b\rangle \Downarrow \texttt{false}$.

So, we know that D' is also using while−true rule. In its derivation, D' must also have subderivations $D'_2 :: \langle E, S\rangle \Downarrow E'_1$ and $D'_3 :: \langle E'_1, \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E''$. 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 $E'' = E'$.

Next time

- WHILE3ADDR: A 3-address-code representation of WHILE
- Control-flow graphs
- Introduction to data-flow analysis