



# Proof Techniques for Operational Semantics

# Small-Step Contextual Semantics

- In small-step contextual semantics, derivations are not tree-structured
- A contextual semantics derivation is a sequence (or list) of atomic rewrites:

$$\langle x+(7-3), \sigma \rangle \rightarrow \langle x+(4), \sigma \rangle \rightarrow \langle 5+4, \sigma \rangle \rightarrow \langle 9, \sigma \rangle$$

$$\sigma(x)=5$$

If  $\langle r, \sigma \rangle \rightarrow \langle e, \sigma' \rangle$

then  $\langle H[r], \sigma \rangle \rightarrow \langle H[e], \sigma' \rangle$

$r$  = redex

$H$  = context (has hole)

# Context Decomposition

- Decomposition theorem:

If **c** is not “skip” then there exist unique **H** and **r** such that **c** is **H[r]**

- “Exist” means progress
- “Unique” means determinism



# Short-Circuit Evaluation

- What if we want to express **short-circuit** evaluation of  $\wedge$  ?
  - Define the following **contexts**, **redexes** and **local reduction rules**
$$H ::= \dots \mid H \wedge b_2$$
$$r ::= \dots \mid \text{true} \wedge b \mid \text{false} \wedge b$$
$$\langle \text{true} \wedge b, \sigma \rangle \rightarrow \langle b, \sigma \rangle$$
$$\langle \text{false} \wedge b, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle$$
  - the local reduction kicks in **before**  $b_2$  is **evaluated**

# Contextual Semantics Summary

- Can view • as representing the **program counter**
- Contextual semantics is **inefficient to implement directly**
- The major advantage of contextual semantics: it allows a mix of local and global reduction rules
  - For IMP we have only local reduction rules: only the redex is reduced
  - Sometimes it is useful to work on the context too
  - We'll do that when we study **memory allocation**, etc.

# Cunning Plan for Proof Techniques

- Why Bother?
- Mathematical Induction
- Well-Founded Induction
- Structural Induction
  - “Induction On The Structure Of The Derivation”

# One-Slide Summary

- **Mathematical Induction** is a **proof technique**: If you can prove  $P(0)$  and you can prove that  $P(n)$  implies  $P(n+1)$ , then you can conclude that for all natural numbers  $n$ ,  $P(n)$  holds.
- Induction works because the natural numbers are **well-founded**: there are no **infinite descending chains**  $n > n-1 > n-2 > \dots > \dots$ .
- **Structural induction** is induction on a formal structure, like an AST. The base cases use the leaves, the inductive steps use the inner nodes.
- **Induction on a derivation** is structural induction applied to a derivation  $D$  (e.g.,  $D::\langle c, \sigma \rangle \Downarrow \sigma'$ ).

# Why Bother?

- I am loathe to teach you anything that I think is a **waste of your time**.
- Thus I must convince you that inductive opsem proof techniques are useful.
  - Recall class goals: **understand PL research techniques and apply them to your research**
- This motivation should also highlight where you might use such techniques in your own research.



# Never Underestimate

“Any counter-example posed by the Reviewers against this proof would be a useless gesture, no matter what technical data they have obtained. **Structural Induction is now the ultimate proof technique in the universe. I suggest we use it.**” --- Admiral Motti, *A New Hope*

# Classic Example (Schema)

- “A well-typed program cannot go wrong.”
  - Robin Milner
- When you design a new type system, you must show that it is **safe** (= that the type system is sound with respect to the operational semantics).
- *A Syntactic Approach to Type Soundness*. Andrew K. Wright, Matthias Felleisen, 1992.
  - Type preservation: “if you have a well-typed program and apply an opsem rule, the result is well-typed.”
  - Progress: “a well-typed program will never get stuck in a state with no applicable opsem rules”
- Done for real languages: SML/NJ, SPARK ADA, Java
  - PL/I, plus basically every toy PL research language ever.

# Classic Examples

- **CCured Project (Berkeley)**

- A program that is instrumented with CCured run-time checks (= “adheres to the CCured type system”) will not segfault (= “the x86 opsem rules will never get stuck”).

- **Vault Language (Microsoft Research)**

- A well-typed Vault program does not leak any tracked resources and invokes tracked APIs correctly (e.g., handles IRQI correctly in asynchronous Windows device drivers, cf. Capability Calculus)

- **RC - Reference-Counted Regions For C (Intel Research)**

- A well-typed RC program gains the speed and convenience of region-based memory management but need never worry about freeing a region too early (run-time checks).

- **Typed Assembly Language (Cornell)**

- Reasonable C programs (e.g., device drivers) can be translated to TALx86. Well-typed TALx86 programs are type- and memory-safe.

- **Secure Information Flow (Many, e.g., Volpano et al. ‘96)**

- Lattice model of secure flow analysis is phrased as a type system, so type soundness = noninterference.

# Recent Examples

- “We prove soundness (Theorem 6.8) by mutual **induction on the derivations** of ...”
  - An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C, POPL 2014
- “The proof goes by **induction on the structure** of  $p$ .”
  - NetKAT: Semantic Foundations of Networks, POPL 2014
- “The operational semantics is given as a big-step relation, on which our compiler correctness proofs can all proceed by **induction** ...”
  - CakeML: A Verified Implementation of ML, POPL 2014
- Method: Chose 4 papers from POPL 2014, 3 of them use structural induction.

# Decade-Old Examples

- “The proof proceeds by rule induction over the target term producing translation rules.”
  - Chakravarty et al. '05
- “Type preservation can be proved by standard induction on the derivation of the evaluation relation.”
  - Hosoya et al. '05
- “Proof: By induction on the derivation of  $N \Downarrow W$ .”
  - Sumi and Pierce '05
- Method: chose four POPL 2005 papers at random, the three above mentioned structural induction.  
(emphasis mine)

# Induction

- **Most important technique** for studying the formal semantics of prog languages
  - If you want to perform or understand PL research, **you must grok this!**
- Mathematical Induction (simple)
- Well-Founded Induction (general)
- Structural Induction (widely used in PL)

# Mathematical Induction

- Goal: prove  $\forall n \in \mathbb{N}. P(n)$
- Base Case: prove  $P(0)$
- Inductive Step:
  - Prove  $\forall n > 0. P(n) \Rightarrow P(n+1)$
  - “Pick arbitrary  $n$ , assume  $P(n)$ , prove  $P(n+1)$ ”
- Why does induction work?

# Why Does It Work?

- There are no infinite descending chains of natural numbers
- For any  $n$ ,  $P(n)$  can be obtained by **starting from the base** case and **applying  $n$  instances of the inductive step**



**BUT I WANT HIM TO COME WITH US!!**





# Well-Founded Induction

- A relation  $\preceq \subseteq A \times A$  is well-founded if there are no infinite descending chains in  $A$ 
  - Example:  $\prec_1 = \{ (x, x+1) \mid x \in \mathbb{N} \}$ 
    - aka the predecessor relation
  - Example:  $< = \{ (x, y) \mid x, y \in \mathbb{N} \text{ and } x < y \}$
- Well-founded induction:
  - To prove  $\forall x \in A. P(x)$  it is enough to prove  $\forall x \in A. [\forall y \preceq x \Rightarrow P(y)] \Rightarrow P(x)$
- If  $\preceq$  is  $\prec_1$  then we obtain mathematical induction as a special case

# Structural Induction

- Recall  $e ::= n \mid e_1 + e_2 \mid e_1 * e_2 \mid x$
- Define  $\preceq \subseteq \text{Aexp} \times \text{Aexp}$  such that
  - $e_1 \preceq e_1 + e_2$        $e_2 \preceq e_1 + e_2$
  - $e_1 \preceq e_1 * e_2$        $e_2 \preceq e_1 * e_2$
- no other elements of  $\text{Aexp} \times \text{Aexp}$  are  $\preceq$ -related
- To prove  $\forall e \in \text{Aexp}. P(e)$ 
  - $\vdash \forall n \in \mathbb{Z}. P(n)$
  - $\vdash \forall x \in \mathbb{L}. P(x)$
  - $\vdash \forall e_1, e_2 \in \text{Aexp}. P(e_1) \wedge P(e_2) \Rightarrow P(e_1 + e_2)$
  - $\vdash \forall e_1, e_2 \in \text{Aexp}. P(e_1) \wedge P(e_2) \Rightarrow P(e_1 * e_2)$

# Notes on Structural Induction

- Called structural induction because the proof is guided by the **structure** of the expression
- One proof case per form of expression
  - Atomic expressions (with no subexpressions) are all **base cases**
  - Composite expressions are the **inductive case**
- This is the *most useful form of induction* in the study of PL

# Example of Induction on Structure of Expressions

- Let
  - $L(e)$  be the # of literals and variable occurrences in  $e$
  - $O(e)$  be the # of operators in  $e$
- Prove that  $\forall e \in Aexp. L(e) = O(e) + 1$
- Proof: by induction on the structure of  $e$ 
  - Case  $e = n$ .  $L(e) = 1$  and  $O(e) = 0$
  - Case  $e = x$ .  $L(e) = 1$  and  $O(e) = 0$
  - Case  $e = e_1 + e_2$ .
    - $L(e) = L(e_1) + L(e_2)$  and  $O(e) = O(e_1) + O(e_2) + 1$
    - By induction hypothesis  $L(e_1) = O(e_1) + 1$  and  $L(e_2) = O(e_2) + 1$
    - Thus  $L(e) = O(e_1) + O(e_2) + 2 = O(e) + 1$
  - Case  $e = e_1 * e_2$ . Same as the case for  $+$

# Other Proofs by Structural Induction on Expressions

- Most proofs for Aexp sublanguage of IMP
- **Small-step** and **natural** semantics obtain equivalent results:

$$\forall e \in \text{Exp}. \forall n \in \mathbb{N}. \quad e \rightarrow^* n \Leftrightarrow e \Downarrow n$$

- Structural induction on expressions works here because all of the semantics are **syntax directed**

# Stating The Obvious (With a Sense of Discovery)

- You are given a concrete state  $\sigma$ .
- You have  $\vdash \langle x + 1, \sigma \rangle \Downarrow 5$
- You also have  $\vdash \langle x + 1, \sigma \rangle \Downarrow 88$
- Is this possible?



# Why That Is Not Possible

- Prove that IMP is deterministic

$$\forall e \in \text{Aexp}. \forall \sigma \in \Sigma. \forall n, n' \in \mathbb{N}. \langle e, \sigma \rangle \Downarrow n \wedge \langle e, \sigma \rangle \Downarrow n' \Rightarrow n = n'$$

$$\forall b \in \text{Bexp}. \forall \sigma \in \Sigma. \forall t, t' \in \mathbb{B}. \langle b, \sigma \rangle \Downarrow t \wedge \langle b, \sigma \rangle \Downarrow t' \Rightarrow t = t'$$

$$\forall c \in \text{Comm}. \forall \sigma, \sigma', \sigma'' \in \Sigma. \langle c, \sigma \rangle \Downarrow \sigma' \wedge \langle c, \sigma \rangle \Downarrow \sigma'' \Rightarrow \sigma' = \sigma''$$

- No immediate way to use *mathematical* induction
- For commands we cannot use *induction on the structure of the command*
  - while's evaluation does *not* depend only on the evaluation of its strict subexpressions

$$\frac{\langle b, \sigma \rangle \Downarrow \text{true} \quad \langle c, \sigma \rangle \Downarrow \sigma' \quad \langle \text{while } b \text{ do } c, \sigma' \rangle \Downarrow \sigma''}{\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma''}$$

$$\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma''$$

Q: Movies (292 / 842)

- From the 1981 movie **Raiders of the Lost Ark**, give either the protagonist's phobia xor the composer of the musical score.

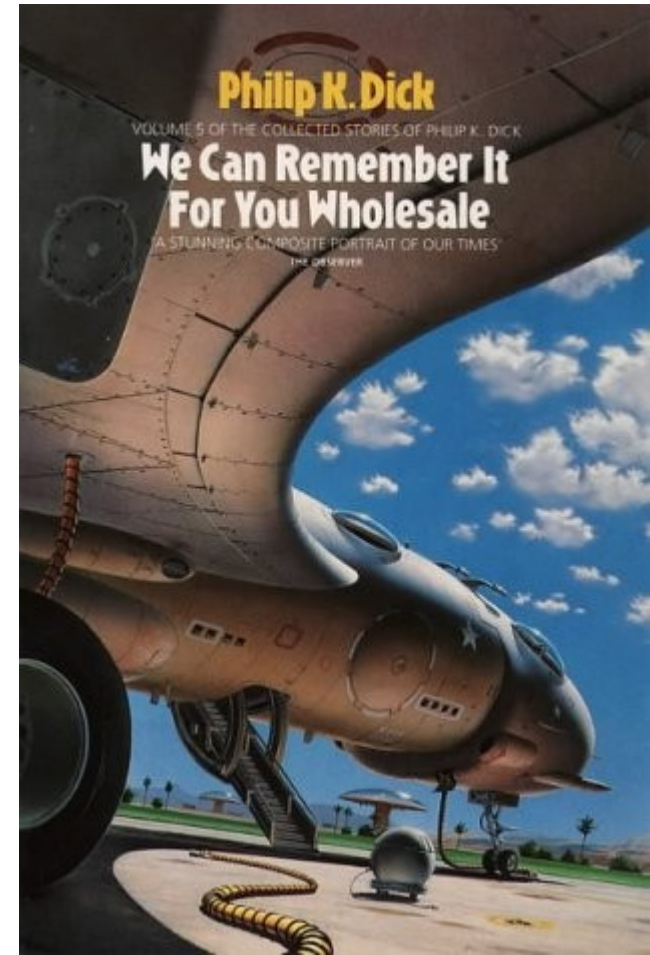


# Computer Science

- This Dutch Turing-award winner is famous for the semaphore, “THE” operating system, the Banker's algorithm, and a shortest path algorithm. He favored structured programming, as laid out in the 1968 article *Go To Statement Considered Harmful*. He was a strong proponent of formal verification and correctness by construction. He also penned *On The Cruelty of Really Teaching Computer Science*, which argues that CS is a branch of math and relates provability to correctness.

# Recall Opsem

- Operational semantics assigns meanings to programs by listing rules of inference that allow you to prove judgments by making derivations.
- A derivation is a tree-structured object made up of valid instances of inference rules.



# We Need Something New

- Some **more powerful** form of induction ...
- With all the bells and whistles!



# Induction on the Structure of Derivations

- Key idea: The hypothesis does not just assume a  $c \in \text{Comm}$  but the **existence of a derivation of  $\langle c, \sigma \rangle \Downarrow \sigma'$**
- Derivation trees are also defined inductively, just like expression trees
- A derivation is built of subderivations:

$$\begin{array}{c}
 \frac{\langle x, \sigma_{i+1} \rangle \Downarrow 5 - i \quad 5 - i \leq 5}{\langle x \leq 5, \sigma_{i+1} \rangle \Downarrow \text{true}} \quad \frac{\frac{\langle x + 1, \sigma_{i+1} \rangle \Downarrow 6 - i}{\langle x := x + 1, \sigma_{i+1} \rangle \Downarrow \sigma_i} \quad \langle W, \sigma_i \rangle \Downarrow \sigma_0}{\langle x := x + 1; W, \sigma_{i+1} \rangle \Downarrow \sigma_0} \\
 \hline
 \langle \text{while } x \leq 5 \text{ do } x := x + 1, \sigma_{i+1} \rangle \Downarrow \sigma_0
 \end{array}$$

- Adapt the structural induction principle to work on the structure of derivations

# Induction on Derivations

- To prove that **for all derivations D of a judgment, property P holds**

- For each derivation rule of the form

$$\frac{H_1 \dots H_n}{C}$$

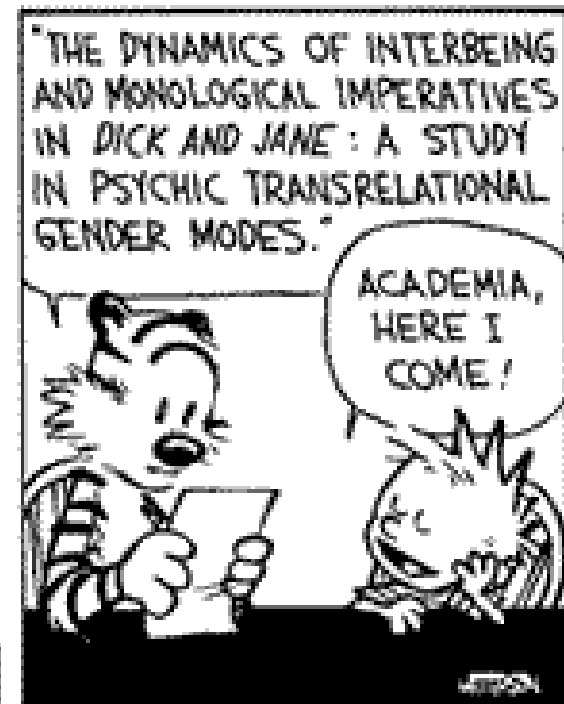
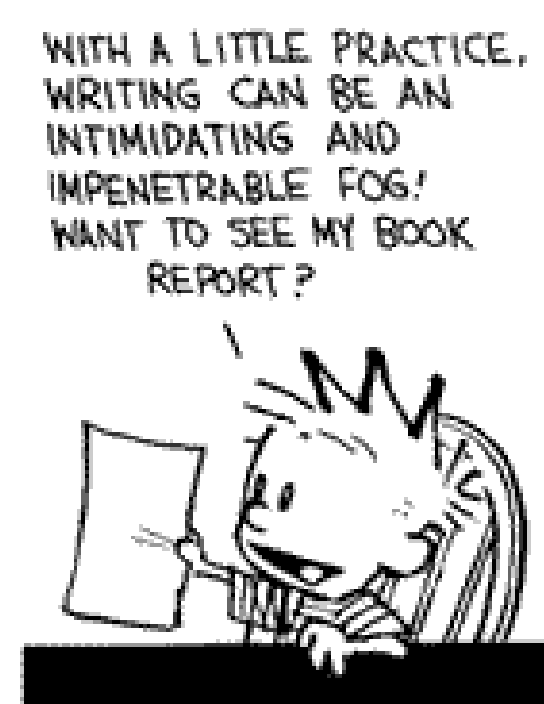
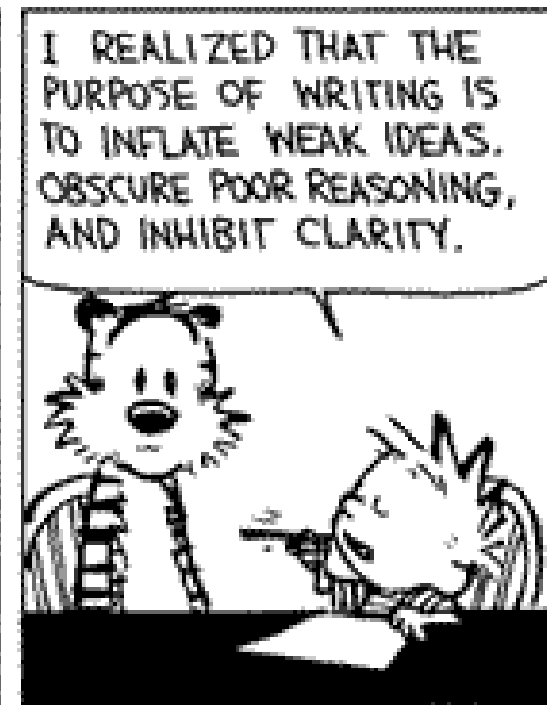
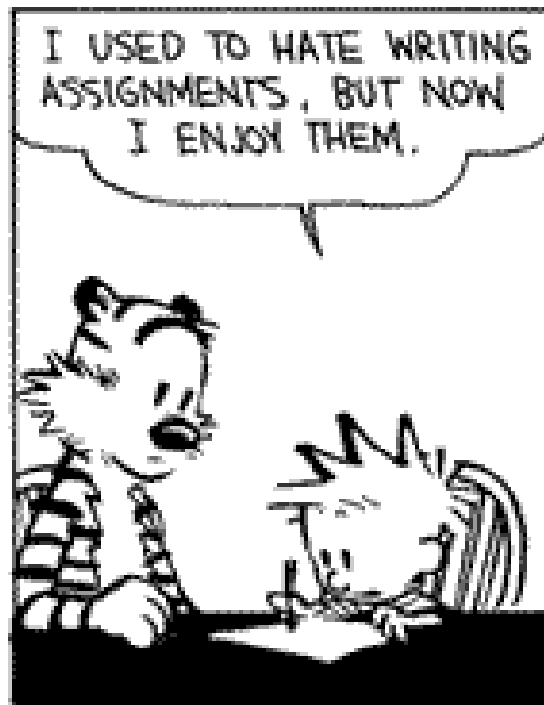
- **Assume** P holds for derivations of  $H_i$  ( $i = 1..n$ )
- **Prove** the the property holds for the derivation obtained from the derivations of  $H_i$  using the given rule

# New Notation

- Write  $D :: \text{Judgment}$  to mean “ $D$  is the derivation that proves Judgment”

- Example:

$$D :: \langle x+1, \sigma \rangle \Downarrow 2$$



# Induction on Derivations (2)

- Prove that evaluation of commands is deterministic:

$$\langle c, \sigma \rangle \Downarrow \sigma' \Rightarrow \forall \sigma'' \in \Sigma. \langle c, \sigma \rangle \Downarrow \sigma'' \Rightarrow \sigma' = \sigma''$$

- Pick arbitrary  $c, \sigma, \sigma'$  and  $D :: \langle c, \sigma \rangle \Downarrow \sigma'$
- To prove:  $\forall \sigma'' \in \Sigma. \langle c, \sigma \rangle \Downarrow \sigma'' \Rightarrow \sigma' = \sigma''$ 
  - Proof: by induction on the structure of the derivation  $D$
- Case: last rule used in  $D$  was the one for skip

$$D :: \frac{}{\langle \text{skip}, \sigma \rangle \Downarrow \sigma}$$

- This means that  $c = \text{skip}$ , and  $\sigma' = \sigma$
- By inversion  $\langle c, \sigma \rangle \Downarrow \sigma''$  uses the rule for skip
- Thus  $\sigma'' = \sigma$
- This is a base case in the induction

# Induction on Derivations (3)

- Case: the last rule used in  $D$  was the one for **sequencing**

$$D :: \frac{D_1 :: \langle c_1, \sigma \rangle \Downarrow \sigma_1 \quad D_2 :: \langle c_2, \sigma_1 \rangle \Downarrow \sigma'}{\langle c_1; c_2, \sigma \rangle \Downarrow \sigma'}$$

- Pick arbitrary  $\sigma''$  such that  $D'' :: \langle c_1; c_2, \sigma \rangle \Downarrow \sigma''$ .
  - by **inversion**  $D''$  uses the rule for sequencing
  - and has subderivations  $D''_1 :: \langle c_1, \sigma \rangle \Downarrow \sigma''_1$  and  $D''_2 :: \langle c_2, \sigma''_1 \rangle \Downarrow \sigma''$
- By induction hypothesis on  $D_1$  (with  $D''_1$ ):  $\sigma_1 = \sigma''_1$ 
  - Now  $D''_2 :: \langle c_2, \sigma_1 \rangle \Downarrow \sigma''$
- By induction hypothesis on  $D_2$  (with  $D''_2$ ):  $\sigma'' = \sigma'$
- This is a **simple inductive case**



# Induction on Derivations (4)

- Case: the last rule used in  $D$  was `while true`

$$D :: \frac{D_1 :: \langle b, \sigma \rangle \Downarrow \text{true} \quad D_2 :: \langle c, \sigma \rangle \Downarrow \sigma_1 \quad D_3 :: \langle \text{while } b \text{ do } c, \sigma_1 \rangle \Downarrow \sigma'}{\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma'}$$

- Pick arbitrary  $\sigma''$  s.t.  $D'' :: \langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma''$ 
  - by `inversion and determinism of boolean expressions`,  $D''$  also uses the rule for `while true`
  - and has subderivations  $D''_2 :: \langle c, \sigma \rangle \Downarrow \sigma''_1$  and  $D''_3 :: \langle W, \sigma''_1 \rangle \Downarrow \sigma''$
- By induction hypothesis on  $D_2$  (with  $D''_2$ ):  $\sigma_1 = \sigma''_1$ 
  - Now  $D''_3 :: \langle \text{while } b \text{ do } c, \sigma_1 \rangle \Downarrow \sigma''$
- By induction hypothesis on  $D_3$  (with  $D''_3$ ):  $\sigma'' = \sigma'$

# What Do You, The Viewers At Home, Think?

- Let's do *if true* together!
- Case: the last rule in D was *if true*

$$D :: \frac{D_1 :: \langle b, \sigma \rangle \Downarrow \text{true} \qquad D_2 :: \langle c1, \sigma \rangle \Downarrow \sigma_1}{\langle \text{if } b \text{ do } c1 \text{ else } c2, \sigma \rangle \Downarrow \sigma_1}$$

- Try to do this on a piece of paper. In a few minutes I'll have some lucky winners come on down.

# Induction on Derivations (5)

- Case: the last rule in D was **if true**

$$D :: \frac{D_1 :: \langle b, \sigma \rangle \Downarrow \text{true} \quad D_2 :: \langle c1, \sigma \rangle \Downarrow \sigma'}{\langle \text{if } b \text{ do } c1 \text{ else } c2, \sigma \rangle \Downarrow \sigma'}$$

- Pick arbitrary  $\sigma''$  such that

$$D'' :: \langle \text{if } b \text{ do } c1 \text{ else } c2, \sigma \rangle \Downarrow \sigma''$$

- By **inversion and determinism**,  $D''$  also uses **if true**
- And has subderivations  $D''_1 :: \langle b, \sigma \rangle \Downarrow \text{true}$  and  $D''_2 :: \langle c1, \sigma \rangle \Downarrow \sigma''$

- By induction hypothesis on  $D_2$  (with  $D''_2$ ):  $\sigma' = \sigma''$

# Induction on Derivations

## Summary

- If you must prove  $\forall x \in A. P(x) \Rightarrow Q(x)$ 
  - with A inductively defined and P(x) rule-defined
  - we pick arbitrary  $x \in A$  and  $D :: P(x)$
  - we could do induction on both facts
    - $x \in A$  leads to induction on the structure of x
    - $D :: P(x)$  leads to induction on the structure of D
  - Generally, the induction on the structure of the derivation is more powerful and a safer bet
- Sometimes there are many choices for induction
  - choosing the right one is a trial-and-error process
  - a bit of practice can help a lot

# Equivalence

Optional  
Material  
Begins

- Two expressions (commands) are equivalent if they yield the same result from all states

$e_1 \approx e_2$  iff

$\forall \sigma \in \Sigma. \forall n \in \mathbb{N}.$

$\langle e_1, \sigma \rangle \Downarrow n$  iff  $\langle e_2, \sigma \rangle \Downarrow n$

and for commands

$c_1 \approx c_2$  iff

$\forall \sigma, \sigma' \in \Sigma.$

$\langle c_1, \sigma \rangle \Downarrow \sigma'$  iff  $\langle c_2, \sigma \rangle \Downarrow \sigma'$

# Notes on Equivalence

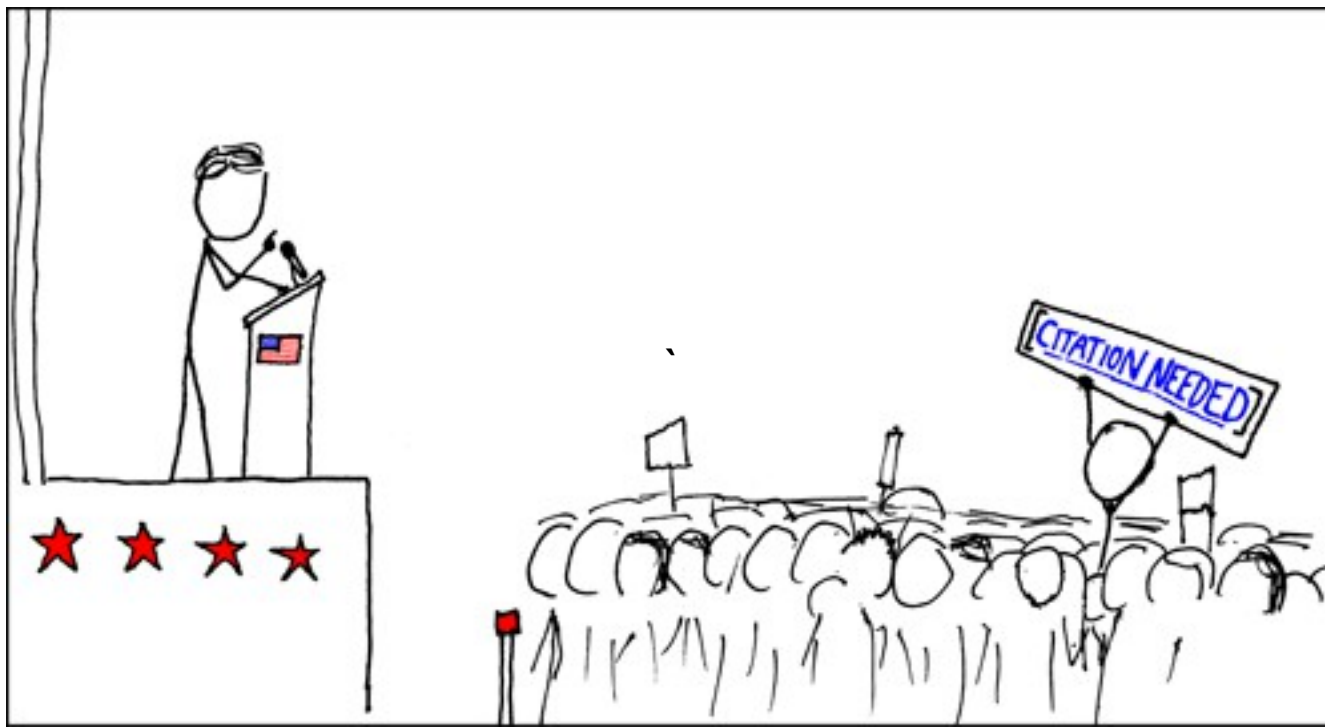
- Equivalence is like logical validity
  - It must hold in all states (= all valuations)
  - $2 \approx 1 + 1$  is like “ $2 = 1 + 1$  is valid”
  - $2 \approx 1 + x$  might or might not hold.
    - So, 2 is not equivalent to  $1 + x$
- Equivalence (for IMP) is undecidable
  - If it were decidable we could solve the halting problem for IMP. *How?*
- Equivalence justifies code transformations
  - compiler optimizations
  - code instrumentation
  - abstract modeling
- **Semantics** is the basis for proving equivalence

# Equivalence Examples

- skip;  $c \approx c$
- while b do c  $\approx$   
if b then c; while b do c else skip
- If  $e_1 \approx e_2$  then  $x := e_1 \approx x := e_2$
- while true do skip  $\approx$  while true do  $x := x + 1$
- Let **c** be  
while  $x \neq y$  do  
if  $x \geq y$  then  $x := x - y$  else  $y := y - x$   
then  
 $(x := 221; y := 527; \mathbf{c}) \approx (x := 17; y := 17)$

# Potential Equivalence

- $(x := e_1; x := e_2) \approx x := e_2$
- Is this a valid equivalence?





# Not An Equivalence

- $(x := e_1; x := e_2) \not\approx x := e_2$
- lie. Chigau yo. Dame desu!
- Not a valid equivalence for all  $e_1, e_2$ .
- Consider:
  - $(x := x+1; x := x+2) \not\approx x := x+2$
- But for  $n_1, n_2$  it's fine:
  - $(x := n_1; x := n_2) \approx x := n_2$

# Proving An Equivalence

- Prove that “ $\text{skip}; c \approx c$ ” for all  $c$
- Assume that  $D :: \langle \text{skip}; c, \sigma \rangle \Downarrow \sigma'$
- By **inversion** (twice) we have that

$$D :: \frac{\overline{\langle \text{skip}, \sigma \rangle \Downarrow \sigma} \quad D_1 :: \langle c, \sigma \rangle \Downarrow \sigma'}{\langle \text{skip}; c, \sigma \rangle \Downarrow \sigma'}$$

- Thus, we have  $D_1 :: \langle c, \sigma \rangle \Downarrow \sigma'$
- The other direction is similar

# Proving An Inequivalence


- Prove that  $x := y \not\approx x := z$  when  $y \neq z$
- It suffices to exhibit a  $\sigma$  in which the two commands yield different results

- Let  $\sigma(y) = 0$  and  $\sigma(z) = 1$

- Then

$$\langle x := y, \sigma \rangle \Downarrow \sigma[x := 0]$$

$$\langle x := z, \sigma \rangle \Downarrow \sigma[x := 1]$$



Optional  
Material  
Ends

# Summary of Operational Semantics

- Precise specification of dynamic semantics
  - order of evaluation (or that it doesn't matter)
  - error conditions (sometimes implicitly, by rule applicability; “no applicable rule” = “get stuck”)
- Simple and abstract (vs. implementations)
  - no low-level details such as stack and memory management, data layout, etc.
- Often not compositional (see while)
- Basis for many proofs about a language
  - Especially when combined with type systems!
- Basis for much reasoning about programs
- Point of reference for other semantics

# Homework

- Don't Neglect Your Homework
- Read DPLL(T) and Simplex