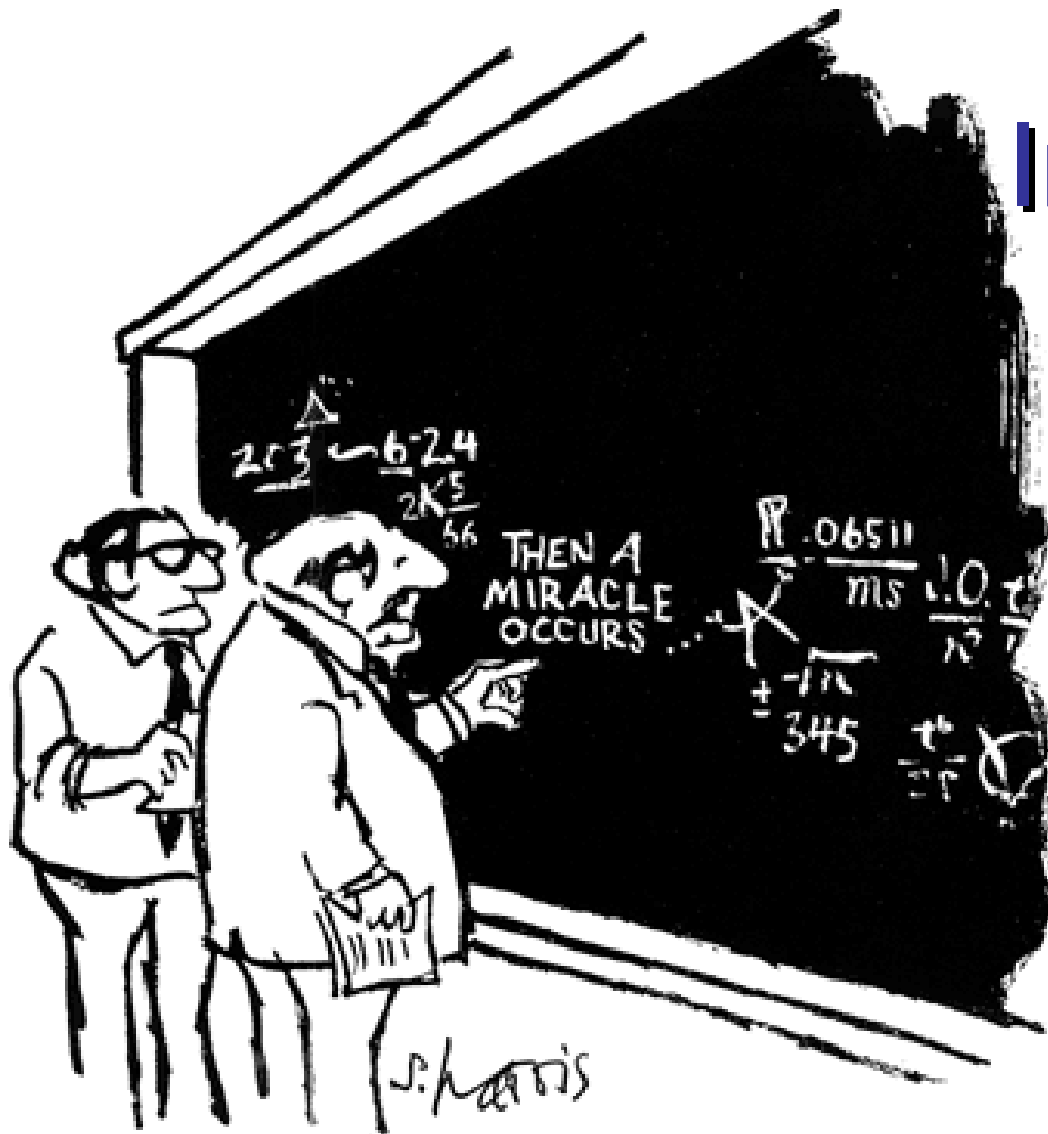# Introduction to Axiomatic Semantics (1/2)
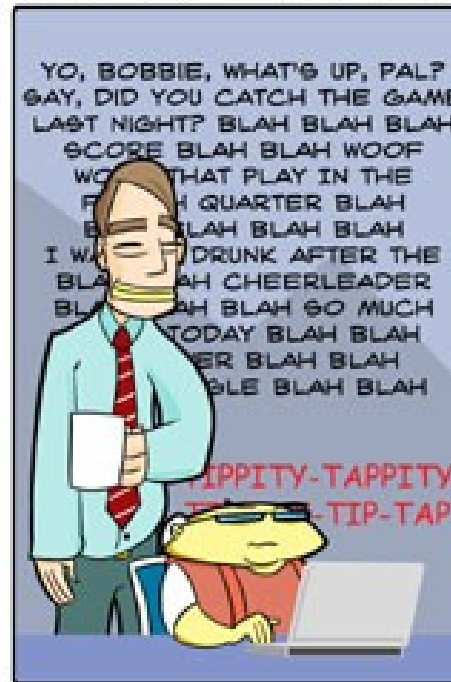

"I think you should be more explicit here in step two."

How's The Homework Going?

- Remember that you can't just define a meaning function in terms of itself – you must use some fixed point machinery.
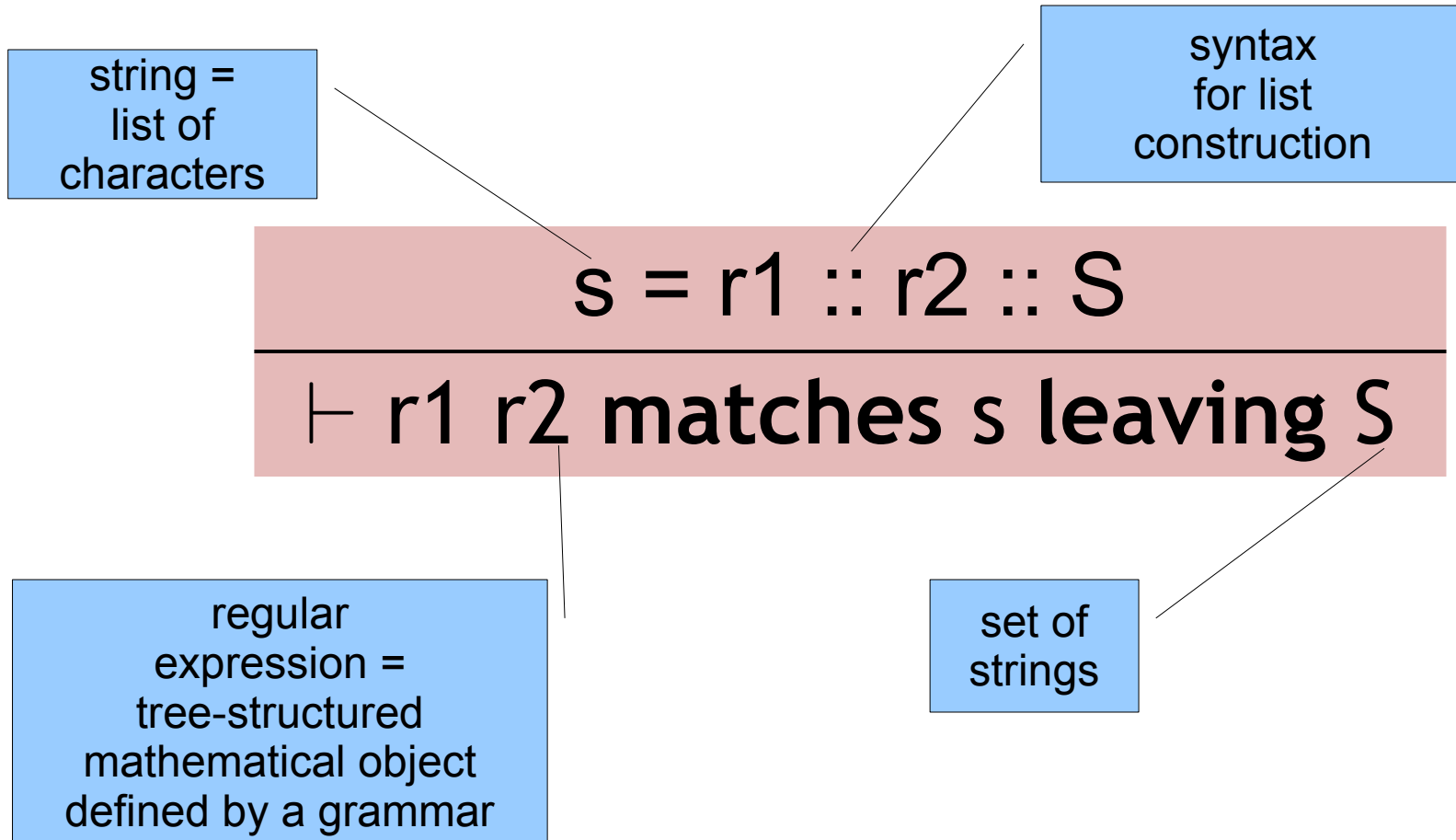
Observations

- A key part of doing research is noticing when something is incongruous. This is related to spotting patterns.

Observations

- A key part of doing research is noticing when something is incongruous. This is related to spotting patterns.

- suffix   === state
- r1 r2   === c1 ; c2
- r1*   === while ? do r1
- r1 | r2   === if ? then r1 else r2

What's Wrong Here?

- Look closely at this "opsem rule"

string =
list of
characters

syntax
for list
construction

$$\frac{s = r1 :: r2 :: S}{\vdash r1\ r2\ \textbf{matches}\ s\ \textbf{leaving}\ S}$$

regular
expression =
tree-structured
mathematical object
defined by a grammar

set of
strings

Review via Class Participation

- Tell Me About Operational Semantics
- Tell Me About Structural Induction
- Tell Me About Denotational Semantics

- We would also like a semantics that is appropriate for arguing program correctness
- "Axiomatic Semantics", we'll call it.

Aujourd'hui, nous ferons …

- History
- Assertions
- Validity
- Derivation Rules
- Soundness
- Completeness

Axiomatic Semantics

- An <u>axiomatic semantics</u> consists of:
  - A language for stating assertions about programs,
  - Rules for establishing the truth of assertions
- Some typical kinds of assertions:
  - This program terminates
  - If this program terminates, the variables x and y have the same value throughout the execution of the program
  - The array accesses are within the array bounds
- Some typical languages of assertions
  - First-order logic
  - Other logics (temporal, linear, pointer-assertion)
  - Special-purpose specification languages (SLIC, Z, Larch)

History

- Program verification is almost as old as programming (e.g., *Checking a Large Routine*, Turing 1949)

- In the late '60s, Floyd had rules for flow-charts and Hoare for structured languages

- Since then, there have been axiomatic semantics for substantial languages, and many applications
 - ESC/Java, SLAM, PCC, SPARK Ada, …

Tony Hoare Quote

- "Thus the practice of proving programs would seem to lead to solution of three of the most pressing problems in software and programming, namely, reliability, documentation, and compatibility. However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs."

    -- C.A.R Hoare, *An Axiomatic Basis for Computer Programming*,1969

Edsger Dijkstra Quote

- "Program testing can be used to show the presence of bugs, but never to show their absence!"
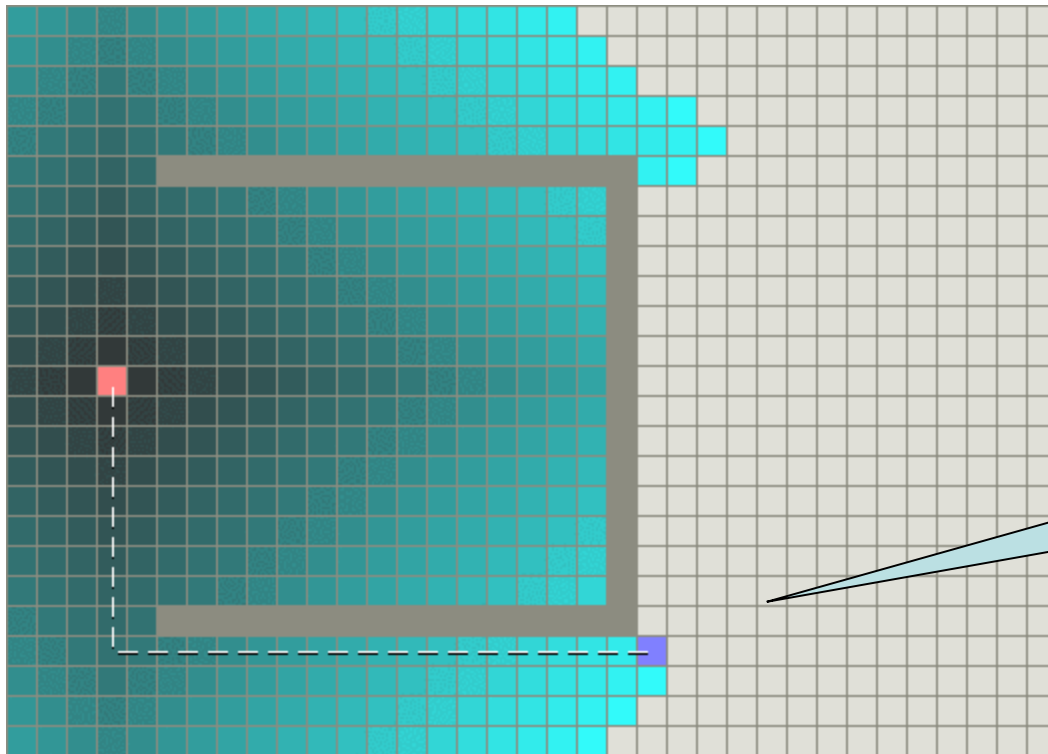
Qu'est-ce que c'est?

Tony Hoare Quote, Mark 2

- "It has been found a serious problem to define these languages [ALGOL, FORTRAN, COBOL] with sufficient rigor to ensure compatibility among all implementations. ... one way to achieve this would be to insist that all implementations of the language shall satisfy the axioms and rules of inference which underlie proofs of properties of programs expressed in the language. In effect, this is equivalent to accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language."

Other Applications of Axiomatic Semantics

- The project of defining and proving everything formally has not succeeded (at least not yet)

- Proving has not replaced testing and debugging

- Applications of axiomatic semantics:
  - Proving the correctness of algorithms (or finding bugs)
  - Proving the correctness of hardware descriptions (or finding bugs)
  - "extended static checking" (e.g., checking array bounds)
  - Proof-carrying code
  - Documentation of programs and interfaces

Assertion Notation

$$\{A\} \ c \ \{B\}$$

with the meaning that:

- – if A holds in state σ and if <c, σ> ⇓ σ'
- – then B holds in σ'

- A is the <u>precondition</u>
- B is the <u>postcondition</u>
- For example:

$$\{\ y \le x\ \}\ z := x;\ z := z + 1\ \{\ y < z\ \}$$

is a valid assertion

- These are called <u>Hoare triples</u> or <u>Hoare assertions</u>

Assertions for IMP

- {A} c {B} is a <u>partial correctness assertion</u>.
  - Does not imply termination (= it is valid if c diverges)
- [A] c [B] is a <u>total correctness assertion</u> meaning that

  If A holds in state σ

  Then there exists σ' such that $<c, \sigma> \Downarrow \sigma'$

  and B holds in state σ'

- Now let us be more formal (you know you want it!)
  - Formalize the language of assertions, A and B
  - Say when an assertion holds in a state
  - Give rules for deriving Hoare triples

The Assertion Language

- We use first-order predicate logic on top of IMP expressions

  $A ::= \text{true} \mid \text{false} \mid e_1 = e_2 \mid e_1 \geq e_2$

  $\mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid A_1 \Rightarrow A_2 \mid \forall x.A \mid \exists x.A$

- Note that we are somewhat sloppy in mixing logical variables and the program variables

- All IMP variables implicitly range over integers
- All IMP boolean expressions are also assertions

Assertion Judgment ⊨

- We need to assign meanings to our assertions
- New judgment $\sigma \vDash A$ to say that an assertion holds in a given state (= "A is true in σ")
  - This is well-defined when σ is defined on all variables occurring in A
- The $\vDash$ judgment is defined inductively on the structure of assertions (surprise!)
- It relies on the denotational semantics of arithmetic expressions from IMP

Semantics of Assertions

Formal definition

$\sigma \vDash$ true   always

$\sigma \vDash e_1 = e_2$ iff $[\![e_1]\!]\ \sigma = [\![e_2]\!]\sigma$

$\sigma \vDash e_1 \geq e_2$ iff $[\![e_1]\!]\ \sigma \geq [\![e_2]\!]\sigma$

$\sigma \vDash A_1 \wedge A_2$ iff $\sigma \vDash A_1$ and $\sigma \vDash A_2$

$\sigma \vDash A_1 \vee A_2$ iff $\sigma \vDash A_1$ or $\sigma \vDash A_2$

$\sigma \vDash A_1 \Rightarrow A_2$ iff $\sigma \vDash A_1$ implies $\sigma \vDash A_2$

$\sigma \vDash \forall x.A$ iff $\forall n \in \mathbb{Z}.\ \sigma[x:=n] \vDash A$

$\sigma \vDash \exists x.A$ iff $\exists n \in \mathbb{Z}.\ \sigma[x:=n] \vDash A$

Hoare Triple Semantics

- Now we can define formally the meaning of a partial correctness assertion ⊨ { A } c { B }

$$\forall\sigma\in\Sigma.\ \forall\sigma'\in\Sigma.\ (\sigma \vDash A \land <c,\sigma> \Downarrow \sigma') \Rightarrow \sigma' \vDash B$$

- … and a total correctness assertion ⊨ [A] c [B]

$$\forall\sigma\in\Sigma.\ \sigma \vDash A \Rightarrow \exists\sigma'\in\Sigma.\ <c,\sigma> \Downarrow \sigma' \land \sigma' \vDash B$$

- or even better yet: (explain this to me!)

$$\forall\sigma\in\Sigma.\ \forall\sigma'\in\Sigma.\ (\sigma \vDash A \land <c,\sigma> \Downarrow \sigma') \Rightarrow \sigma' \vDash B$$

$$\land$$

$$\forall\sigma\in\Sigma.\ \sigma \vDash A \Rightarrow \exists\sigma'\in\Sigma.\ <c,\sigma> \Downarrow \sigma'$$

- In a 1995 Disney movie that has been uncharitably referred to as "Hokey-Hontas", the Stephen Schwartz lyrics *"what I love most about rivers is: / you can't step in the same river twice"* refer to the ideas of which Greek philosopher?

Computer Science

- This American Turing-award winner is known for his work on formal semantics of programming languages, automata theory, modal logic, topology, and category theory. His 1959 paper with Rabin, *Finite Automata and Their Decision Problem*, introduced the idea of *nondeterministic* machines to automata and complexity theory.

- Name the movie described below, its heroine and its star. This 1979 Ridley Scott movie began the first major American film series with a female action hero. Famously, it is the original movie to pass the Bechdel Test.

Deriving Assertions

- Have a formal mechanism to decide $\models \{ A \} c \{ B \}$

  - But it is not satisfactory

  - Because $\models \{A\} c \{B\}$ is defined in terms of the operational semantics, we practically have to run the program to verify an assertion

  - It is impossible to effectively verify the truth of a $\forall x. A$ assertion (check every integer?)

- Plan: define a symbolic technique for deriving valid assertions from others that are known to be valid

  - We start with validity of first-order formulas

Derivation Rules

- We write $\vdash A$ when A can be derived from basic axioms ($\vdash A$ === "we can prove A")
- The derivation rules for $\vdash A$ are the usual ones from first-order logic with arithmetic:

$$\frac{\vdash A \Rightarrow B \quad \vdash A}{\vdash B}$$

$$\frac{\vdash A \quad \vdash B}{\vdash A \wedge B}$$

$$\frac{\vdash [a/x]A \quad (a \text{ is fresh})}{\vdash \forall x.A}$$

$$\frac{\vdash A \\ \dots \\ \vdash B}{\vdash A \Rightarrow B}$$

$$\frac{\vdash \forall x.A}{\vdash [e/x]A}$$

$$\frac{\vdash [e/x]A}{\vdash \exists x.A}$$

$$\frac{\vdash \exists x.A \quad \begin{array}{c}\vdash [a/x]A \\ \dots \\ \vdash B\end{array}}{\vdash B}$$

Derivation Rules for Hoare Triples

- Similarly we write $\vdash \{A\}\ c\ \{B\}$ when we can derive the triple using derivation rules

- There is one derivation rule for each command in the language

- Plus, the *evil* rule of consequence

$$\frac{\vdash A' \Rightarrow A \quad \vdash \{A\}\ c\ \{B\} \quad \vdash B \Rightarrow B'}{\vdash \{A'\}\ c\ \{B'\}}$$

Derivation Rules for Hoare Logic

- One rule for each syntactic construct:

$$\frac{}{\vdash \{A\} \ \text{skip} \ \{A\}} \qquad \frac{}{\vdash \{[e/x]A\} \ x := e \ \{A\}}$$

$$\frac{\vdash \{A\} \ c_1 \ \{B\} \quad \vdash \{B\} \ c_2 \ \{C\}}{\vdash \{A\} \ c_1; \ c_2 \ \{C\}}$$

$$\frac{\vdash \{A \wedge b\} \ c_1 \ \{B\} \quad \vdash \{A \wedge \neg b\} \ c_2 \ \{B\}}{\vdash \{A\} \ \text{if} \ b \ \text{then} \ c_1 \ \text{else} \ c_2 \ \{B\}}$$

$$\frac{\vdash \{A \wedge b\} \ c \ \{A\}}{\vdash \{A\} \ \text{while} \ b \ \text{do} \ c \ \{A \wedge \neg b\}}$$

Alternate Hoare Rules

- For some constructs multiple rules are possible:
- (Exercise: these rules can be derived from the previous ones using the consequence rules)

$$\frac{}{\vdash \{A\}\ x := e\ \{\exists x_0.[x_0/x]A \land x = [x_0/x]e\}}$$

(This one is called the "forward" axiom for assignment)

$$\frac{\vdash A \land b \Rightarrow C \quad \vdash \{C\}\ c\ \{A\} \quad \vdash A \land \neg b \Rightarrow B}{\vdash \{A\}\ \text{while } b \text{ do } c\ \{B\}}$$

(C is the loop invariant)

Example: Assignment

- (Assuming that x does not appear in e)

  Prove that {true} x := e { x = e }

- Assignment Rule:

$$\frac{}{\vdash \{e = e\}\ x := e\ \{x = e\}}$$

  because $[e/x](x = e) \to e = e$

- Use Assignment + Consequence:

$$\frac{\vdash true \Rightarrow e = e \qquad \dfrac{}{\vdash \{e = e\}\ x := e\ \{x = e\}}}{\vdash \{true\}\ x := e\ \{x = e\}}$$

The Assignment Axiom (Cont.)

- "Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics. It is surprising therefore that the axiom governing our reasoning about assignment is quite as simple as any to be found in elementary logic."   - Tony Hoare

- Caveats are sometimes needed for languages with aliasing (the strong update problem):
  - If x and y are aliased then
    { true } x := 5 { x + y = 10}
    is true

Example: Conditional

$$D_1 :: \vdash \{true \wedge y \leq 0\}\ x := 1\ \{x > 0\}$$

$$D_2 :: \vdash \{true \wedge y > 0\}\ x := y\ \{x > 0\}$$

$$\overline{\vdash \{true\}\ \text{if } y \leq 0 \text{ then } x := 1 \text{ else } x := y\ \{x > 0\}}$$

- D_1 and D_2 were obtained by consequence and assignment. D_1 details:

$$\frac{\vdash \{1 > 0\}\ x := 1\ \{x > 0\} \qquad\qquad \vdash true \wedge y \leq 0 \Rightarrow 1 > 0}{\vdash D_1 :: \{true \wedge y \leq 0\}\ x := 1\ \{x > 0\}}$$

Example: Loop

- We want to derive that

$$\vdash \{x \leq 0\} \text{ while } x \leq 5 \text{ do } x := x + 1 \{ x = 6\}$$

- Use the rule for while with invariant $x \leq 6$

$$\frac{\vdash x \leq 6 \wedge x \leq 5 \Rightarrow x+1 \leq 6 \qquad \dfrac{\vdash \{x+1 \leq 6\} \, x := x+1 \, \{ x \leq 6 \}}{\vdash \{x \leq 6 \wedge x \leq 5 \} \, x := x+1 \, \{x \leq 6\}}}{\vdash \{x \leq 6\} \text{ while } x \leq 5 \text{ do } x := x+1 \{ x \leq 6 \wedge x > 5\}}$$

- Then finish-off with consequence

$$\vdash x \leq 0 \Rightarrow x \leq 6$$

$$\frac{\vdash x \leq 6 \wedge x > 5 \Rightarrow x = 6 \qquad \vdash \{x \leq 6\} \text{ while } \dots \{ x \leq 6 \wedge x > 5\}}{\vdash \{x \leq 0\} \text{ while } \dots \{x = 6\}}$$
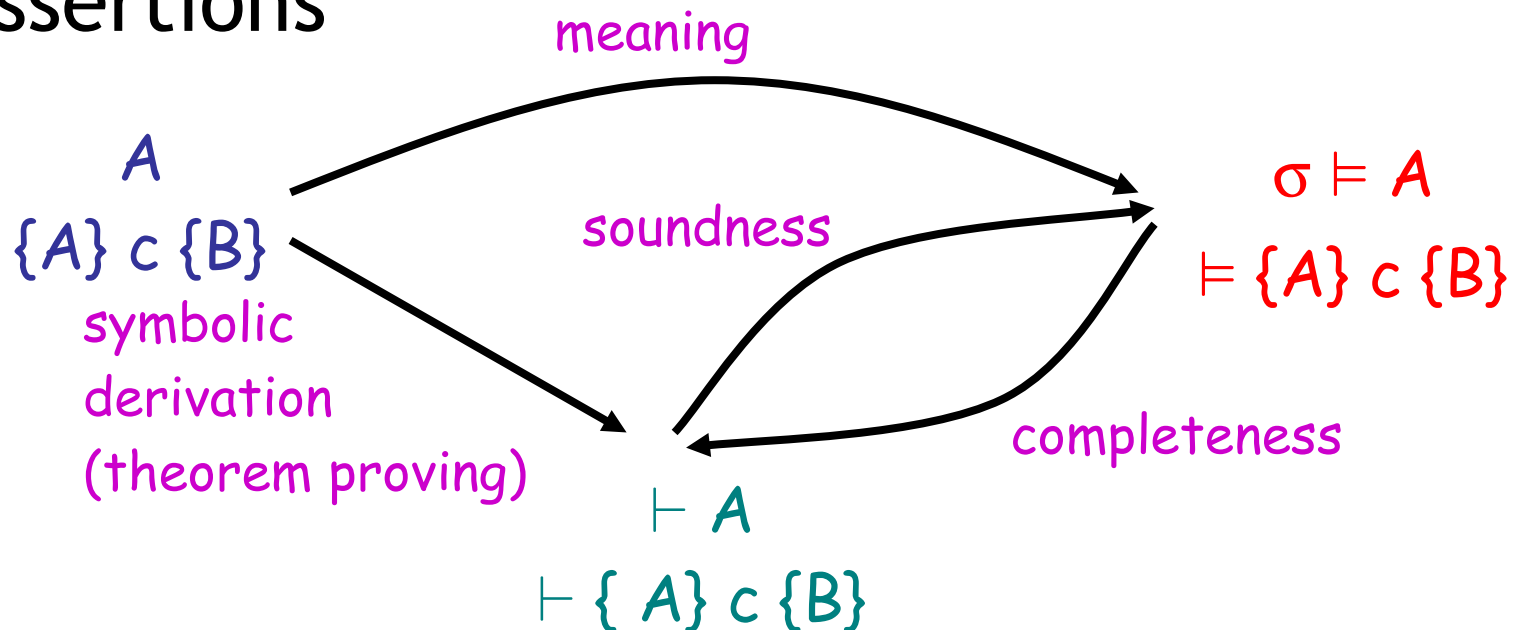
Using Hoare Rules

- Hoare rules are mostly syntax directed
- There are three wrinkles:
  - What invariant to use for while? (fix points, widening)
  - When to apply consequence? (theorem proving)
  - How do you prove the implications involved in consequence? (theorem proving)
- This is how theorem proving gets in the picture
  - This turns out to be doable!
  - The loop invariants turn out to be the hardest problem! (Should the programmer give them? See Dijkstra, ESC.)

Where Do We Stand?

- We have a language for asserting properties of programs
- We know when such an assertion is true
- We also have a symbolic method for deriving assertions

meaning

A

$\{A\}\ c\ \{B\}$

symbolic derivation (theorem proving)

soundness

$\sigma \models A$

$\models \{A\}\ c\ \{B\}$

completeness

$\vdash A$

$\vdash \{A\}\ c\ \{B\}$

Soundness
and
Completeness

Soundness of Axiomatic Semantics

- Formal statement of <u>soundness</u>:

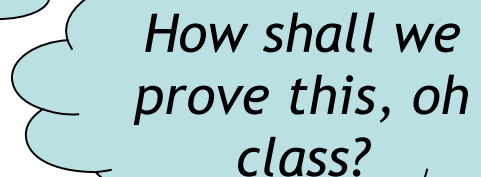 if $\vdash \{ A \} c \{ B \}$ then $\vDash \{ A \} c \{ B \}$

  or, equivalently

    For all σ, if $\sigma \vDash A$

    and Op :: $<c, \sigma> \Downarrow \sigma'$

    and Pr :: $\vdash \{ A \} c \{ B \}$

    then $\sigma' \vDash B$

  How shall we prove this, oh class?

- "Op" === "Opsem Derivation"
- "Pr" === "Axiomatic Proof"

Not Easily!

- By induction on the structure of c?
  - No, problems with while and rule of consequence
- By induction on the structure of Op?
  - No, problems with while
- By induction on the structure of Pr?
  - No, problems with consequence
- By simultaneous induction on the structure of Op and Pr
  - Yes! New Technique!

Simultaneous Induction

- Consider two structures Op and Pr
  - Assume that x < y iff x is a substructure of y
- Define the ordering

  $(o, p) \prec (o', p')$ iff

  $$o < o' \quad \text{or} \quad o = o' \text{ and } p < p'$$

  - Called lexicographic (dictionary) ordering
- This $\prec$ is a well founded order and leads to simultaneous induction
- If o < o' then p can actually be larger than p'!
- It can even be unrelated to p'!

Soundness of the While Rule

(Indiana Proof and the Slide of Doom)

- Case: last rule used in Pr : $\vdash$ {A} c {B} was the while rule:

$$\frac{Pr_1 :: \vdash \{A \wedge b\} \; c \; \{A\}}{\vdash \{A\} \; while \; b \; do \; c \; \{A \wedge \neg b\}}$$

- Two possible rules for the root of Op (*by inversion*)
  - We'll only do the complicated case:

$$\frac{Op_1 :: <b, \sigma> \Downarrow true \qquad Op_2 :: <c,\sigma> \Downarrow \sigma' \qquad Op_3 :: <while \; b \; do \; c, \sigma' > \Downarrow \sigma''}{<while \; b \; do \; c, \sigma > \Downarrow \sigma''}$$

Assume that $\sigma \vDash A$

To show that $\sigma'' \vDash A \wedge \neg b$

- By soundness of booleans and Op_1 we get $\sigma \vDash b$
  - Hence $\sigma \vDash A \wedge b$
- By IH on Pr_1 and Op_2 we get $\sigma' \vDash A$
- By IH on Pr and Op_3 we get $\sigma'' \vDash A \wedge \neg b$, q.e.d. (tricky!)

Soundness of the While Rule

- Note that in the last use of IH the derivation Pr *did not decrease*

- But Op_3 was a sub-derivation of Op

- See Winskel, Chapter 6.5, for a soundness proof with denotational semantics

- To be continued …

Homework

- HW 3 Due Soon
- Axiomatic Reading