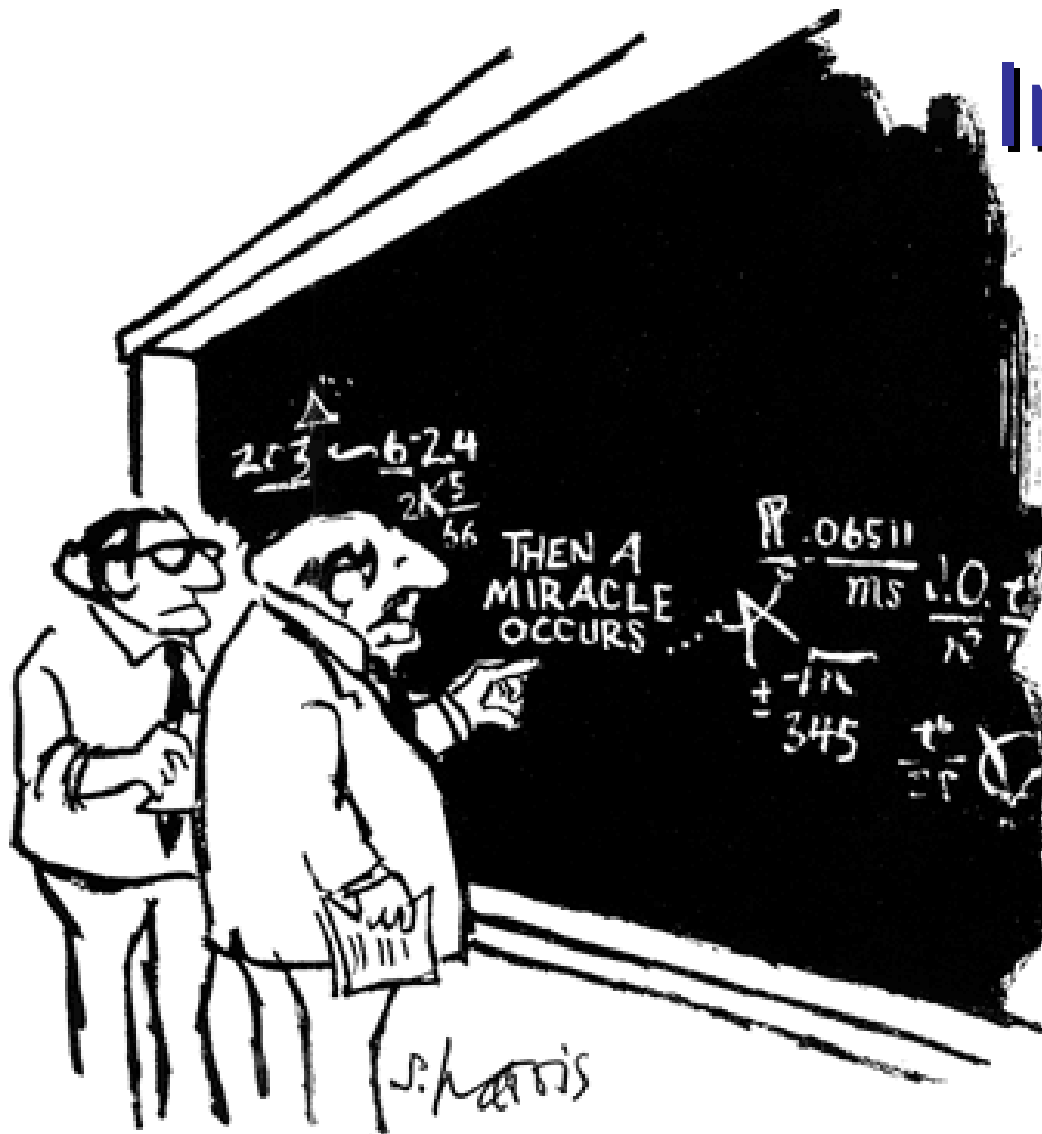


Introduction to Axiomatic Semantics

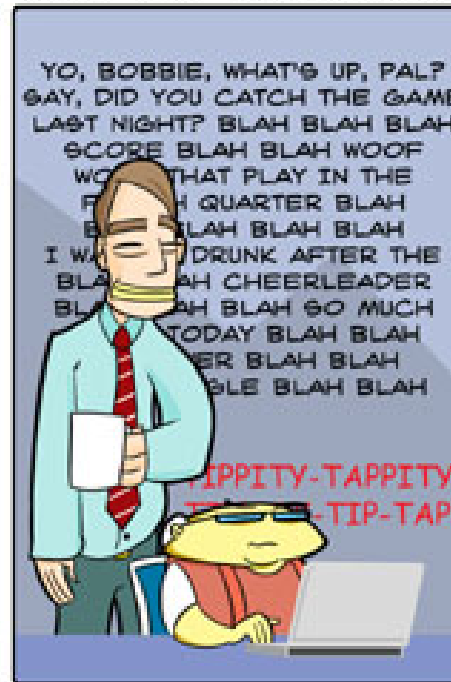


"I think you should be more explicit here in step two."

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.

The PC Weenies®



Observations

- A key part of doing research is noticing when something is incongruous or when something changes - or otherwise spotting patterns.

Observations

- A key part of doing research is noticing when something is incongruous or when something changes - or otherwise spotting patterns.
- suffix === state
- r1 r2 === c1 ; c2
- r1* === while ? do r1
- r1 | r2 === if ? then r1 else r2

Aujourd'hui, nous ferons ...

- **Assertions**
- **Validity**
- **Derivation Rules**
- **Soundness**
- **Completeness**

Assertions for IMP

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

If A holds in state σ

Then **there exists** σ' such that $\langle c, \sigma \rangle \Downarrow \sigma'$

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 \models

- We need to assign meanings to our assertions
- New judgment $\sigma \models A$ 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 \models 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

$\sigma \models \text{true}$	always
$\sigma \models e_1 = e_2$	iff $\llbracket e_1 \rrbracket \sigma = \llbracket e_2 \rrbracket \sigma$
$\sigma \models e_1 \geq e_2$	iff $\llbracket e_1 \rrbracket \sigma \geq \llbracket e_2 \rrbracket \sigma$
$\sigma \models A_1 \wedge A_2$	iff $\sigma \models A_1$ and $\sigma \models A_2$
$\sigma \models A_1 \vee A_2$	iff $\sigma \models A_1$ or $\sigma \models A_2$
$\sigma \models A_1 \Rightarrow A_2$	iff $\sigma \models A_1$ implies $\sigma \models A_2$
$\sigma \models \forall x.A$	iff $\forall n \in \mathbb{Z}. \sigma[x:=n] \models A$
$\sigma \models \exists x.A$	iff $\exists n \in \mathbb{Z}. \sigma[x:=n] \models A$

Hoare Triple Semantics

- Now we can define formally the meaning of a **partial correctness** assertion $\models \{ A \} c \{ B \}$

$$\forall \sigma \in \Sigma. \forall \sigma' \in \Sigma. (\sigma \models A \wedge \langle c, \sigma \rangle \Downarrow \sigma') \Rightarrow \sigma' \models B$$

- ... and a **total correctness** assertion $\models [A] c [B]$

$$\forall \sigma \in \Sigma. \sigma \models A \Rightarrow \exists \sigma' \in \Sigma. \langle c, \sigma \rangle \Downarrow \sigma' \wedge \sigma' \models B$$

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

$$\forall \sigma \in \Sigma. \forall \sigma' \in \Sigma. (\sigma \models A \wedge \langle c, \sigma \rangle \Downarrow \sigma') \Rightarrow \sigma' \models B$$

\wedge

$$\forall \sigma \in \Sigma. \sigma \models A \Rightarrow \exists \sigma' \in \Sigma. \langle c, \sigma \rangle \Downarrow \sigma'$$

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 ($\vdash A \iff$ “we can prove A ”)
- The derivation rules for $\vdash A$ are the usual ones from first-order logic with arithmetic:

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

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

$$\frac{\begin{array}{c} \vdash A \\ \dots \\ \vdash B \end{array}}{\vdash A \Rightarrow B}$$

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

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

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

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

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:

$$\vdash \{A\} \text{ skip } \{A\}$$
$$\vdash \{[e/x]A\} x := e \{A\}$$
$$\vdash \{A\} c_1 \{B\} \quad \vdash \{B\} c_2 \{C\}$$
$$\vdash \{A\} c_1; c_2 \{C\}$$
$$\vdash \{A \wedge b\} c_1 \{B\} \quad \vdash \{A \wedge \neg b\} c_2 \{B\}$$
$$\vdash \{A\} \text{ if } b \text{ then } c_1 \text{ else } c_2 \{B\}$$
$$\vdash \{A \wedge b\} c \{A\}$$
$$\vdash \{A\} \text{ while } b \text{ do } c \{A \wedge \neg b\}$$

Alternate Hoare Rules

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

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

(This one is called the “forward” axiom for assignment)

$$\vdash A \wedge b \Rightarrow C \quad \vdash \{C\} c \{A\} \quad \vdash A \wedge \neg b \Rightarrow B$$

$$\vdash \{A\} \text{ while } b \text{ do } c \{B\}$$

(C is the loop invariant)

Example: Assignment

- (Assuming that x does not appear in e)

Prove that $\{\text{true}\} x := e \{x = e\}$

- Assignment Rule:

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

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

- Use Assignment + Consequence:

$$\vdash \text{true} \Rightarrow e = e$$

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

$$\vdash \{\text{true}\} x := e \{x = e\}$$

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
$$\{ \text{true} \} x := 5 \{ x + y = 10 \}$$
is true

Example: Conditional

$$D_1 :: \vdash \{\text{true} \wedge y \leq 0\} x := 1 \{x > 0\}$$
$$D_2 :: \vdash \{\text{true} \wedge y > 0\} x := y \{x > 0\}$$

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

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

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

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

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 $x \leq 6$

$$\frac{\vdash x \leq 6 \wedge x \leq 5 \Rightarrow x+1 \leq 6 \quad \vdash \{x+1 \leq 6\} x := x+1 \{x \leq 6\}}{\vdash \{x \leq 6 \wedge x \leq 5\} x := x+1 \{x \leq 6\}}$$

$$\vdash \{x \leq 6 \wedge x \leq 5\} x := x+1 \{x \leq 6\}$$

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

- Then finish-off with consequence

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

$$\vdash x \leq 6 \wedge x > 5 \Rightarrow x = 6 \quad \vdash \{x \leq 6\} \text{ while } \dots \{x \leq 6 \wedge x > 5\}$$

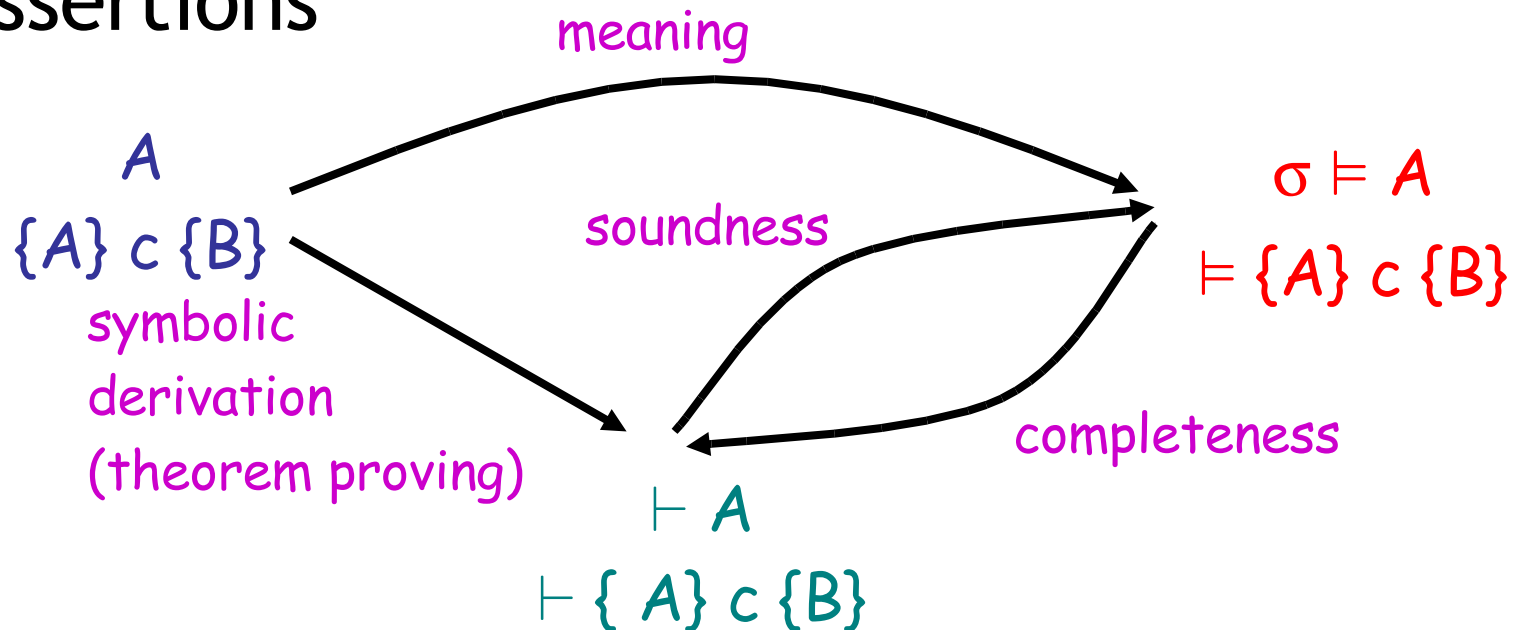
$$\vdash \{x \leq 0\} \text{ while } \dots \{x = 6\}$$

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



Soundness and Completeness



Soundness of Axiomatic Semantics

- Formal statement of soundness:

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

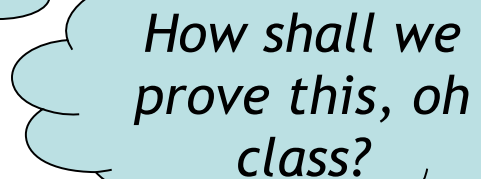
or, equivalently

For all σ , if $\sigma \models A$

and $Op :: \langle c, \sigma \rangle \Downarrow \sigma'$

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

then $\sigma' \models B$



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 O_p and P_r
 - Assume that $x < y$ iff x is a substructure of y
- Define the ordering
$$(o, p) \prec (o', p') \text{ iff } o < o' \text{ or } o = o' \text{ and } p < p'$$
 - Called **lexicographic (dictionary) ordering**
- This \prec is a well-founded order and leads to simultaneous induction
- If $o < o'$ then h can actually be larger than h' !
- It can even be unrelated to h' !



©2005 Krishna M. Sadasivam

“The Real Deal” Axiomatic Semantics

“NOBODY UNDERSTANDS ME.”

Soundness of Axiomatic Semantics

- Formal statement of soundness:

If $\vdash \{ A \} c \{ B \}$ then $\models \{ A \} c \{ B \}$

or, equivalently

For all σ , if $\sigma \models A$

and $Op :: \langle c, \sigma \rangle \Downarrow \sigma'$

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

then $\sigma' \models B$

- “Op” = “Opsem Derivation”
- “Pr” = “Axiomatic Proof”

Simultaneous Induction

- Consider two structures O_p and P_r
 - Assume that $x < y$ iff x is a substructure of y
- Define the ordering
$$(o, p) \prec (o', p') \text{ iff}$$
$$o < o' \quad \text{or} \quad o = o' \text{ and } p < p'$$
 - 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 $\text{Pr} : \vdash \{A\} c \{B\}$ was the while rule:

$$\text{Pr}_1 :: \vdash \{A \wedge b\} c \{A\}$$

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

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

$$\text{Op}_1 :: \langle b, \sigma \rangle \Downarrow \text{true} \quad \text{Op}_2 :: \langle c, \sigma \rangle \Downarrow \sigma' \quad \text{Op}_3 :: \langle \text{while } b \text{ do } c, \sigma' \rangle \Downarrow$$

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

Assume that $\sigma \models A$

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

- By soundness of booleans and Op_1 we get $\sigma \models b$
 - Hence $\sigma \models A \wedge b$
- By IH on Pr_1 and Op_2 we get $\sigma' \models A$
- By IH on Pr and Op_3 we get $\sigma'' \models A \wedge \neg b$, q.e.d.
 - This is the tricky bit!

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

Completeness of Axiomatic Semantics

- If $\models \{A\} c \{B\}$ can we always derive $\vdash \{A\} c \{B\}$?
- If so, axiomatic semantics is complete
- If not then there are valid properties of programs that we cannot verify with Hoare rules :-)
- Good news: for our language the Hoare triples are complete
- Bad news: only if the underlying logic is complete
(whenever $\models A$ we also have $\vdash A$)
 - this is called relative completeness

Examples, General Plan

- OK, so:

$$\models \{x < 5 \wedge z = 2\} y := x + 2 \{y < 7\}$$

- Can we prove it?

$$?\vdash? \{x < 5 \wedge z = 2\} y := x + 2 \{y < 7\}$$

- Well, we *could* easily prove:

$$\vdash \{x+2 < 7\} y := x + 2 \{y < 7\}$$

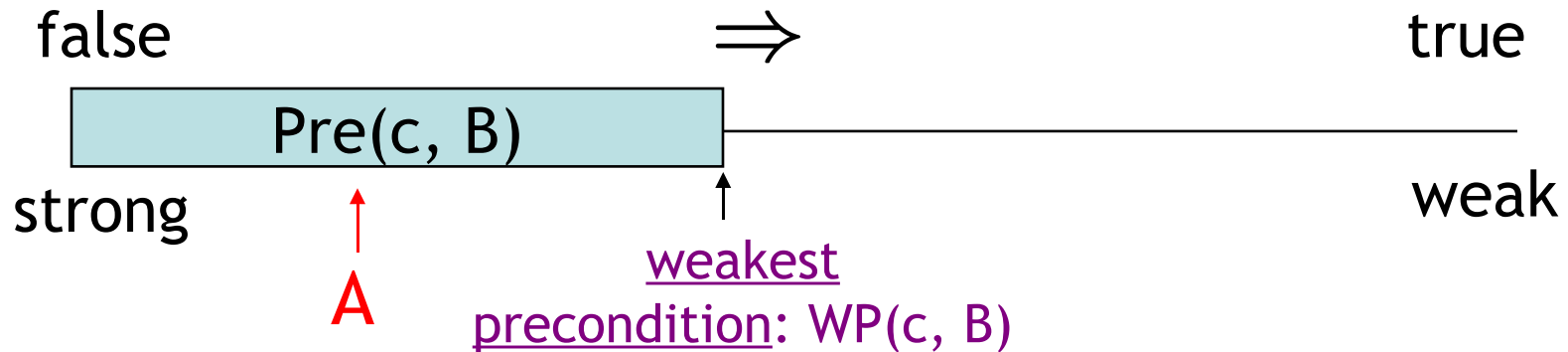
- And we know ...

$$\vdash x < 5 \wedge z = 2 \Rightarrow x+2 < 7$$

- Shouldn't those two proofs be enough?

Proof Idea

- Dijkstra's idea: To verify that $\{ A \} c \{ B \}$
 - a) Find out all predicates A' such that $\models \{ A' \} c \{ B \}$
 - call this set $Pre(c, B)$ (Pre = “pre-conditions”)
 - b) Verify for one $A' \in Pre(c, B)$ that $A \Rightarrow A'$
- Assertions can be ordered:



- Thus: compute $WP(c, B)$ and prove $A \Rightarrow WP(c, B)$

Proof Idea (Cont.)

- Completeness of axiomatic semantics:

If $\models \{ A \} c \{ B \}$ then $\vdash \{ A \} c \{ B \}$

- Assuming that we can compute $\text{wp}(c, B)$ with the following properties:

- wp is a **precondition** (according to the Hoare rules)

$\vdash \{ \text{wp}(c, B) \} c \{ B \}$

- wp is (*truly*) **the weakest** precondition

If $\models \{ A \} c \{ B \}$ then $\models A \Rightarrow \text{wp}(c, B)$

$\vdash A \Rightarrow \text{wp}(c, B) \quad \vdash \{ \text{wp}(c, B) \} c \{ B \}$

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

- We also need that whenever $\models A$ then $\vdash A$!

Weakest Preconditions

- Define $wp(c, B)$ inductively on c , following the Hoare rules:

- $$wp(c_1; c_2, B) = \frac{\{A\} c_1 \{C\} \quad \{C\} c_2 \{B\}}{\{A\} c_1; c_2 \{B\}}$$

- $$wp(x := e, B) = \frac{}{\{ [e/x]B \} x := E \{B\}}$$

$$\frac{\{A_1\} c_1 \{B\} \quad \{A_2\} c_2 \{B\}}{\{ E \Rightarrow A_1 \wedge \neg E \Rightarrow A_2 \} \text{if } E \text{ then } c_1 \text{ else } c_2 \{B\}}$$

- $$wp(\text{if } E \text{ then } c_1 \text{ else } c_2, B) = E \Rightarrow wp(c_1, B) \wedge \neg E \Rightarrow wp(c_2, B)$$

Weakest Preconditions for Loops

- We start from the unwinding equivalence

while b do c =

if b then c; while b do c else skip

- Let $w = \text{while } b \text{ do } c$ and $W = \text{wp}(w, B)$

- We have that

$$W = b \Rightarrow \text{wp}(c, W) \quad \wedge \quad \neg b \Rightarrow B$$

- But this is a **recursive equation!**
 - We know how to solve these using domain theory
- But we need a domain for assertions

A Partial Order for Assertions

- Which assertion contains the least information?
 - “true” - does not say anything about the state
- What is an appropriate information ordering ?
$$A \sqsubseteq A' \quad \text{iff} \quad \models A' \Rightarrow A$$
- Is this partial order complete?
 - Take a chain $A_1 \sqsubseteq A_2 \sqsubseteq \dots$
 - Let $\bigwedge A_i$ be the infinite conjunction of A_i
$$\sigma \models \bigwedge A_i \quad \text{iff for all } i \text{ we have that } \sigma \models A_i$$
 - I assert that $\bigwedge A_i$ is the least upper bound
- Can $\bigwedge A_i$ be expressed in our language of assertions?
 - In many cases: yes (see Winskel), we'll assume yes for now

Weakest Precondition for WHILE

- Use the fixed-point theorem

$$F(A) = b \Rightarrow wp(c, A) \wedge \neg b \Rightarrow B$$

- (Where did this come from? Two slides back!)
 - I assert that F is both monotonic and continuous
- The least-fixed point (= the weakest fixed point) is

$$wp(w, B) = \bigwedge F^i(\text{true})$$

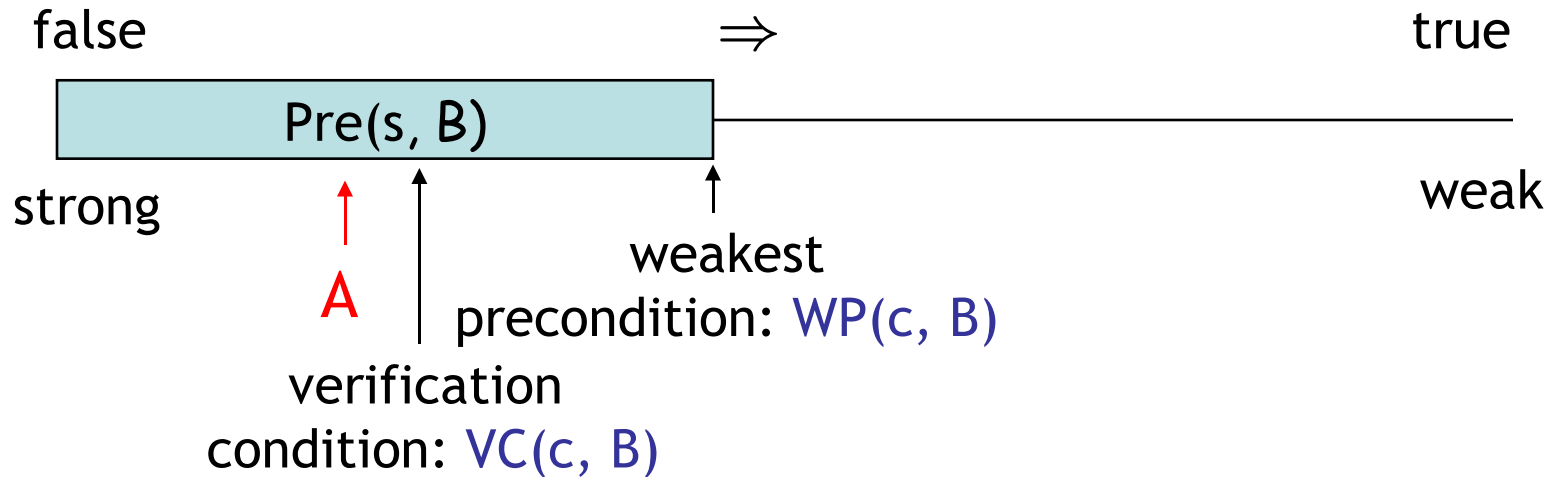
- Notice that unlike for denotational semantics of IMP we are not working on a flat domain!

Weakest Preconditions (Cont.)

- Define a family of wp's
 - $\text{wp}_k(\text{while } e \text{ do } c, B)$ = weakest precondition on which the loop terminates in B if it terminates in k or fewer iterations
- $\text{wp}_0 = \neg E \Rightarrow B$
- $\text{wp}_1 = E \Rightarrow \text{wp}(c, \text{wp}_0) \wedge \neg E \Rightarrow B$
- ...
- $\text{wp}(\text{while } e \text{ do } c, B) = \bigwedge_{k \geq 0} \text{wp}_k = \text{lub} \{ \text{wp}_k \mid k \geq 0 \}$
- See Necula document on the web page for the proof of completeness with weakest preconditions
- Weakest preconditions are
 - Impossible to compute (in general)
 - Can we find something easier to compute yet sufficient?

Not Quite Weakest Preconditions

- Recall what we are trying to do:



- Construct a verification condition: $\text{VC}(c, B)$
 - Our loops will be annotated with loop invariants!
 - VC is guaranteed to be stronger than WP
 - But still weaker than A: $A \Rightarrow \text{VC}(c, B) \Rightarrow \text{WP}(c, B)$

Groundwork

- Factor out the hard work
 - Loop invariants
 - Function specifications (pre- and post-conditions)
- Assume programs are annotated with such specs
 - Good software engineering practice anyway
 - Requiring annotations = Kiss of Death?
- New form of while that includes a loop invariant:
$$\text{while}_{\text{Inv}} \ b \ \text{do} \ c$$
 - Invariant formula Inv must hold every time before b is evaluated
- A process for computing $\text{VC}(\text{annotated_command}, \text{post_condition})$ is called VCGen

Verification Condition Generation

- Mostly follows the definition of the wp function:

$$\text{VC}(\text{skip}, B) = B$$

$$\text{VC}(c_1; c_2, B) = \text{VC}(c_1, \text{VC}(c_2, B))$$

$$\text{VC}(\text{if } b \text{ then } c_1 \text{ else } c_2, B) = b \Rightarrow \text{VC}(c_1, B) \wedge \neg b \Rightarrow \text{VC}(c_2, B)$$

$$\text{VC}(x := e, B) = [e/x] B$$

$$\text{VC}(\text{let } x = e \text{ in } c, B) = [e/x] \text{VC}(c, B)$$

$$\text{VC}(\text{while}_{\text{Inv}} b \text{ do } c, B) = ?$$

VCGen for WHILE

$VC(\text{while}_{Inv} \ e \ \text{do} \ c, \ B) =$

$$\underbrace{Inv}_{\text{Inv holds on entry}} \wedge \left(\forall x_1 \dots x_n. \underbrace{Inv \Rightarrow (e \Rightarrow VC(c, Inv))}_{\text{Inv is preserved in an arbitrary iteration}} \wedge \underbrace{\neg e \Rightarrow B}_{\text{B holds when the loop terminates in an arbitrary iteration}} \right)$$

Inv holds on entry

Inv is preserved in an arbitrary iteration

B holds when the loop terminates in an arbitrary iteration

- Inv is the loop invariant (provided externally)
- x_1, \dots, x_n are all the variables modified in c
- The \forall is similar to the \forall in mathematical induction:
 $P(0) \wedge \forall n \in \mathbb{N}. P(n) \Rightarrow P(n+1)$

Example VCGen Problem

- Let's compute the VC of this program with respect to post-condition $x \neq 0$

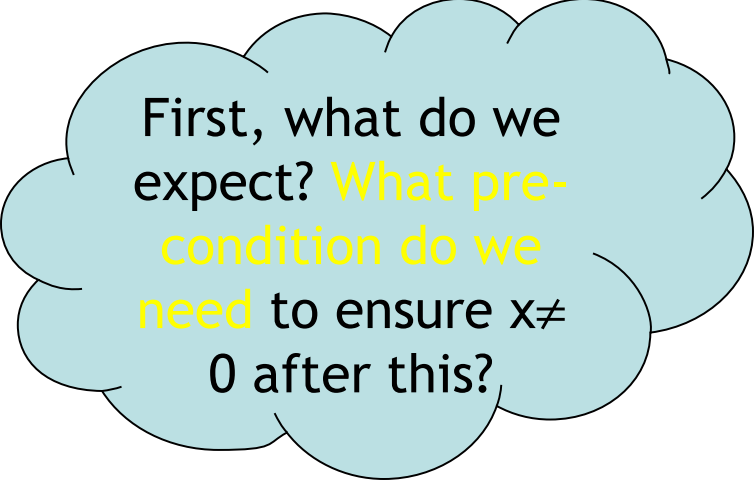
$x = 0;$

$y = 2;$

$\text{while}_{x+y=2} y > 0 \text{ do}$

$y := y - 1;$

$x := x + 1$



First, what do we expect? What pre-condition do we need to ensure $x \neq 0$ after this?

Example of VC

- By the sequencing rule, first we do the while loop (call it **w**):

```
whilex+y=2 y > 0 do  
  y := y - 1;  
  x := x + 1
```

Preserve loop invariant

Ensure post on exit

- $\text{VCGen}(\mathbf{w}, x \neq 0) = x+y=2 \wedge \forall x,y. x+y=2 \Rightarrow (y>0 \Rightarrow \text{VC}(c, x+y=2) \wedge y \leq 0 \Rightarrow x \neq 0)$
- $\text{VCGen}(y:=y-1 ; x:=x+1, x+y=2) = (x+1) + (y-1) = 2$
- **w** Result: $x+y=2 \wedge \forall x,y. x+y=2 \Rightarrow (y>0 \Rightarrow (x+1)+(y-1)=2 \wedge y \leq 0 \Rightarrow x \neq 0)$

Example of VC (2)

- $VC(w, x \neq 0) = x+y=2 \wedge$
 $\forall x, y. x+y=2 \Rightarrow$
 $(y>0 \Rightarrow (x+1)+(y-1)=2 \wedge y \leq 0 \Rightarrow x \neq 0)$
- $VC(x := 0; y := 2; w, x \neq 0) = 0+2=2 \wedge$
 $\forall x, y. x+y=2 \Rightarrow$
 $(y>0 \Rightarrow (x+1)+(y-1)=2 \wedge y \leq 0 \Rightarrow x \neq 0)$
- So now we ask an automated theorem prover to prove it.

Thoreau, Thoreau, Thoreau

```
$ ./Simplify
> (AND (EQ (+ 0 2) 2)
      (FORALL ( x y ) (IMPLIES (EQ (+ x y) 2)
                               (AND (IMPLIES (> y 0)
                                           (EQ (+ (+ x 1) (- y 1)) 2))
                                     (IMPLIES (<= y 0) (NEQ x 0)))))))
```

1: Valid.

- Huzzah!
- Simplify is a non-trivial five megabytes

Can We Mess Up VCGen?

- The invariant is from the user (= the **adversary**, the **untrusted** code base)
- Let's use a loop invariant that is too weak, like "true".
- $VC = \text{true} \wedge \forall x, y. \text{true} \Rightarrow$
 $(y > 0 \Rightarrow \text{true} \wedge y \leq 0 \Rightarrow x \neq 0)$
- Let's use a loop invariant that is false, like "x $\neq 0$ ".
- $VC = 0 \neq 0 \wedge \forall x, y. x \neq 0 \Rightarrow$
 $(y > 0 \Rightarrow x + 1 \neq 0 \wedge y \leq 0 \Rightarrow x \neq 0)$

Emerson, Emerson, Emerson

```
$ ./Simplify  
> (AND TRUE  
  (FORALL ( x y ) (IMPLIES TRUE  
    (AND (IMPLIES (> y 0) TRUE)  
      (IMPLIES (<= y 0) (NEQ x 0))))))
```

Counterexample: context:

```
(AND  
  (EQ x 0)  
  (<= y 0)  
)
```

1: Invalid.

- OK, so we won't be fooled.

Soundness of VCGen

- Simple form

$$\models \{ VC(c, B) \} c \{ B \}$$

- Or equivalently that

$$\models VC(c, B) \Rightarrow wp(c, B)$$

- Proof is by **induction on the structure** of c
 - Try it!
- Soundness holds for **any** choice of invariant!
- Next: properties and extensions of VCs

VC and Invariants

- Consider the Hoare triple:

$$\{x \leq 0\} \text{ while}_{I(x)} x \leq 5 \text{ do } x := x + 1 \{x = 6\}$$

- The VC for this is:

$$x \leq 0 \Rightarrow I(x) \wedge \forall x. (I(x) \Rightarrow (x > 5 \Rightarrow x = 6 \wedge x \leq 5 \Rightarrow I(x+1)))$$

- Requirements on the invariant:

- Holds on entry $\forall x. x \leq 0 \Rightarrow I(x)$
- Preserved by the body $\forall x. I(x) \wedge x \leq 5 \Rightarrow I(x+1)$
- Useful $\forall x. I(x) \wedge x > 5 \Rightarrow x = 6$

- Check that $I(x) = x \leq 6$ satisfies all constraints

Forward VCGen

- Traditionally the VC is computed backwards
 - That's how we've been doing it in class
 - It works well for **structured code**
- But it can also be computed forward
 - Works even for un-structured languages (e.g., **assembly language**)
 - Uses **symbolic execution**, a technique that has broad applications in program analysis
 - e.g., the PREfix tool (Intrinsa, Microsoft) does this

Forward VC Gen Intuition

- Consider the sequence of assignments

$$x_1 := e_1; x_2 := e_2$$

- The $VC(c, B) = [e_1/x_1]([e_2/x_2]B)$

$$= [e_1/x_1, e_2[e_1/x_1]/x_2] B$$

- We can compute the substitution in a forward way using symbolic execution (aka symbolic evaluation)
 - Keep a symbolic state that maps variables to expressions
 - Initially, $\Sigma_0 = \{ \}$
 - After $x_1 := e_1$, $\Sigma_1 = \{ x_1 \rightarrow e_1 \}$
 - After $x_2 := e_2$, $\Sigma_2 = \{ x_1 \rightarrow e_1, x_2 \rightarrow e_2[e_1/x_1] \}$
 - Note that we have applied Σ_1 as a substitution to right-hand side of assignment $x_2 := e_2$

Simple Assembly Language

- Consider the language of instructions:
 $I ::= x := e \mid f() \mid \text{if } e \text{ goto } L \mid \text{goto } L \mid L: \mid \text{return} \mid \text{inv } e$
- The “ $\text{inv } e$ ” instruction is an annotation
 - Says that boolean expression e holds at that point
- Each function $f()$ comes with Pre_f and Post_f annotations (pre- and post-conditions)
- New Notation (yay!): I_k is the instruction at address k

Symex States

- We set up a symbolic execution state:

$\Sigma : \text{Var} \rightarrow \text{SymbolicExpressions}$

$\Sigma(x)$ = the symbolic value of x in state Σ

$\Sigma[x:=e]$ = a new state in which x 's value is e

- We use states as substitutions:

$\Sigma(e)$ - obtained from e by replacing x with $\Sigma(x)$

- Much like the opsem so far ...

Symex Invariants

- The symbolic executor tracks invariants passed
- A new part of symex state: $Inv \subseteq \{1..n\}$
- If $k \in Inv$ then I_k is an invariant instruction that we have already executed
- Basic idea: execute an **inv** instruction only twice:
 - The **first time** it is encountered
 - Once more time around an arbitrary iteration

Symex Rules

- Define a VC function as an interpreter:

$VC : \text{Address} \times \text{SymbolicState} \times \text{InvariantState} \rightarrow \text{Assertion}$

$VC(L, \Sigma, \text{Inv})$	if $I_k = \text{goto } L$
$e \Rightarrow VC(L, \Sigma, \text{Inv}) \quad \wedge$ $\neg e \Rightarrow VC(k+1, \Sigma, \text{Inv})$	if $I_k = \text{if } e \text{ goto } L$
$VC(k+1, \Sigma[x := \Sigma(e)], \text{Inv})$	if $I_k = x := e$
$\Sigma(\text{Post}_{\text{current-function}})$	if $I_k = \text{return}$
$\Sigma(\text{Pre}_f) \quad \wedge$ $\forall a_1 \dots a_m. \Sigma'(\text{Post}_f) \Rightarrow$ $VC(k+1, \Sigma', \text{Inv})$ (where y_1, \dots, y_m are modified by f) and a_1, \dots, a_m are fresh parameters and $\Sigma' = \Sigma[y_1 := a_1, \dots, y_m := a_m]$	if $I_k = f()$

$VC(k, \Sigma, \text{Inv}) =$

Symex Invariants (2a)

Two cases when seeing an invariant instruction:

2. We see the invariant for the first time

- $I_k = \text{inv } e$
- $k \notin \text{Inv}$ (= “not in the set of invariants we’ve seen”)
- Let $\{y_1, \dots, y_m\}$ = the variables that could be modified on a path from the invariant back to itself
- Let a_1, \dots, a_m be fresh new symbolic parameters

$\text{VC}(k, \Sigma, \text{Inv}) =$

$$\Sigma(e) \wedge \forall a_1 \dots a_m. \Sigma'(e) \Rightarrow \text{VC}(k+1, \Sigma', \text{Inv} \cup \{k\})$$

with $\Sigma' = \Sigma[y_1 := a_1, \dots, y_m := a_m]$

(like a function call)

Symex Invariants (2b)

1. We see the invariant for the second time

- $I_k = \text{inv } E$
- $k \in \text{Inv}$

$$\text{VC}(k, \Sigma, \text{Inv}) = \Sigma(e)$$

(like a function return)

- Some tools take a more simplistic approach
 - Do not require invariants
 - Iterate through the loop a fixed number of times
 - PREFIX, versions of ESC (DEC/Compaq/HP SRC)
 - Sacrifice completeness for usability