

Introduction to Denotational Semantics (2/2)



Simple Domain Theory

- Consider programs in an eager, deterministic language with one variable called “x”
 - All these restrictions are just to simplify the examples
- A state σ is just the value of x
 - Thus we can use \mathbb{Z} instead of Σ
- The semantics of a command give the value of final x as a function of input x

$$C[[c]] : \mathbb{Z} \rightarrow \mathbb{Z}_{\perp}$$

Examples - Revisited

- Take C `[[while true do skip]]`
 - Unwinding equation reduces to $W(x) = W(x)$
 - Any function satisfies the unwinding equation
 - Desired solution is $W(x) = \perp$
- Take C `[[while $x \neq 0$ do $x := x - 2$]]`
 - Unwinding equation:
 $W(x) = \text{if } x \neq 0 \text{ then } W(x - 2) \text{ else } x$
 - Solutions (for all values $n, m \in \mathbb{Z}_{\perp}$):
 $W(x) = \text{if } x \geq 0 \text{ then}$
 if x even then 0 else n
 else m
 - Desired solution: $W(x) = \text{if } x \geq 0 \wedge x \text{ even then } 0 \text{ else } \perp$

An Ordering of Solutions

- The desired solution is the one in which all the arbitrariness is replaced with **non-termination**
 - The arbitrary values in a solution are not uniquely determined by the semantics of the code
- We introduce an ordering of semantic functions
- Let $f, g \in \mathbb{Z} \rightarrow \mathbb{Z}_\perp$
- Define $f \sqsubseteq g$ as
$$\forall x \in \mathbb{Z}. f(x) = \perp \text{ or } f(x) = g(x)$$
 - A “smaller” function terminates *at most as often*, and when it terminates it produces the same result

Alternative Views of Function Ordering

- A semantic function $f \in \mathbb{Z} \rightarrow \mathbb{Z}_\perp$ can be written as $S_f \subseteq \mathbb{Z} \times \mathbb{Z}$ as follows:

$$S_f = \{ (x, y) \mid x \in \mathbb{Z}, f(x) = y \neq \perp \}$$

- set of “terminating” values for the function
- If $f \sqsubseteq g$ then
 - $S_f \subseteq S_g$ (and vice-versa)
 - We say that g refines f
 - We say that f approximates g
 - We say that g provides more information than f

The “Best” Solution

- Consider again $C[\text{while } x \neq 0 \text{ do } x := x - 2]$
 - Unwinding equation:
 $W(x) = \text{if } x \neq 0 \text{ then } W(x - 2) \text{ else } x$
- Not all solutions are comparable:
 $W(x) = \text{if } x \geq 0 \text{ then if } x \text{ even then } 0 \text{ else } 1 \text{ else } 2$
 $W(x) = \text{if } x \geq 0 \text{ then if } x \text{ even then } 0 \text{ else } \perp \text{ else } 3$
 $W(x) = \text{if } x \geq 0 \text{ then if } x \text{ even then } 0 \text{ else } \perp \text{ else } \perp$
(last one is least and best)
- Is there **always a least solution?**
- How do we find it?
- *If only we had a general framework* for answering these questions ...

Fixed-Point Equations

- Consider the general unwinding equation for **while**
 $\text{while } b \text{ do } c \equiv \text{if } b \text{ then } c; \text{ while } b \text{ do } c \text{ else skip}$
- We define a context **C** (command with a hole)
 $C = \text{if } b \text{ then } c; \bullet \text{ else skip}$
 $\text{while } b \text{ do } c \equiv C[\text{while } b \text{ do } c]$
 - The grammar for **C** does not contain “while b do c”
- We can find such a (recursive) context for any looping construct
 - Consider: **fact** $n = \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact } (n - 1)$
 - $C(n) = \text{if } n = 0 \text{ then } 1 \text{ else } n * \bullet (n - 1)$
 - $\text{fact} = C[\text{fact}]$

Fixed-Point Equations

- The meaning of a context is a semantic functional $F : (\mathbb{Z} \rightarrow \mathbb{Z}_\perp) \rightarrow (\mathbb{Z} \rightarrow \mathbb{Z}_\perp)$ such that

$$F \llbracket C[w] \rrbracket = F \llbracket w \rrbracket$$

- For “while”: $C = \text{if } b \text{ then } c; \bullet \text{ else skip}$

$$F w x = \text{if } \llbracket b \rrbracket x \text{ then } w (\llbracket c \rrbracket x) \text{ else } x$$

- F depends only on $\llbracket c \rrbracket$ and $\llbracket b \rrbracket$
- We can rewrite the unwinding equation for while
 - $W(x) = \text{if } \llbracket b \rrbracket x \text{ then } W(\llbracket c \rrbracket x) \text{ else } x$
 - or, $W x = F W x$ for all x ,
 - or, $W = F W$ (by function equality)

Fixed-Point Equations

- The meaning of “while” is a solution for $W = F W$
- Such a W is called a fixed point of F
- We want the least fixed point
 - We need a general way to find least fixed points
- Whether such a least fixed point exists depends on the properties of function F
 - Counterexample: $F w x = \text{if } w x = \perp \text{ then } 0 \text{ else } \perp$
 - Assume W is a fixed point
 - $F W x = W x = \text{if } W x = \perp \text{ then } 0 \text{ else } \perp$
 - Pick an x , then $\text{if } W x = \perp \text{ then } W x = 0 \text{ else } W x = \perp$
 - Contradiction. This F has no fixed point!

Can We Solve This?

- Good news: the functions F that *correspond to contexts in our language* have least fixed points!
- The only way $F w x$ uses w is by invoking it
- If any such invocation diverges, then $F w x$ diverges!
- It turns out: F is monotonic, continuous
 - Not shown here!

New Notation: λ

- $\lambda x. e$
 - an anonymous function with body e and argument x
- Example: $\text{double}(x) = x+x$
 $\text{double} = \lambda x. x+x$
- Example: $\text{allFalse}(x) = \text{false}$
 $\text{allFalse} = \lambda x. \text{false}$
- Example: $\text{multiply}(x,y) = x*y$
 $\text{multiply} = \lambda x. \lambda y. x*y$

The Fixed-Point Theorem

- If F is a semantic function corresponding to a context in our language

- F is monotonic and continuous (we assert)
- For any fixed-point G of F and $k \in \mathbb{N}$

$$F^k(\lambda x. \perp) \sqsubseteq G$$

- The least of all fixed points is

$$\sqcup_k F^k(\lambda x. \perp)$$

- Proof (not detailed in the lecture):

1. By mathematical induction on k .

Base: $F^0(\lambda x. \perp) = \lambda x. \perp \sqsubseteq G$

Inductive: $F^{k+1}(\lambda x. \perp) = F(F^k(\lambda x. \perp)) \sqsubseteq F(G) = G$

- Suffices to show that $\sqcup_k F^k(\lambda x. \perp)$ is a fixed-point

$$F(\sqcup_k F^k(\lambda x. \perp)) = \sqcup_k F^{k+1}(\lambda x. \perp) = \sqcup_k F^k(\lambda x. \perp)$$

WHILE Semantics

- We can use the fixed-point theorem to write the denotational semantics of while:

$$\llbracket \text{while } b \text{ do } c \rrbracket = \sqcup_k F^k (\lambda x. \perp)$$

where $F f x = \text{if } \llbracket b \rrbracket x \text{ then } f (\llbracket c \rrbracket x) \text{ else } x$

- Example: $\llbracket \text{while true do skip} \rrbracket = \lambda x. \perp$
- Example: $\llbracket \text{while } x \neq 0 \text{ then } x := x - 1 \rrbracket$
 - $F (\lambda x. \perp) x = \text{if } x = 0 \text{ then } x \text{ else } \perp$
 - $F^2 (\lambda x. \perp) x = \text{if } x = 0 \text{ then } x \text{ else if } x-1 = 0 \text{ then } x-1 \text{ else } \perp$
 $= \text{if } 1 \geq x \geq 0 \text{ then } 0 \text{ else } \perp$
 - $F^3 (\lambda x. \perp) x = \text{if } 2 \geq x \geq 0 \text{ then } 0 \text{ else } \perp$
 - $\text{LFP}_F = \text{if } x \geq 0 \text{ then } 0 \text{ else } \perp$
- Not easy to find the closed form for general LFPs!

Discussion

- We can write the denotational semantics but we cannot always compute it.
 - Otherwise, we could decide the halting problem
 - H is halting for input 0 iff $\llbracket H \rrbracket 0 \neq \perp$
- We have derived this for programs with one variable
 - Generalize to multiple variables, even to variables ranging over richer data types, even higher-order functions: [domain theory](#)

Can You Remember?

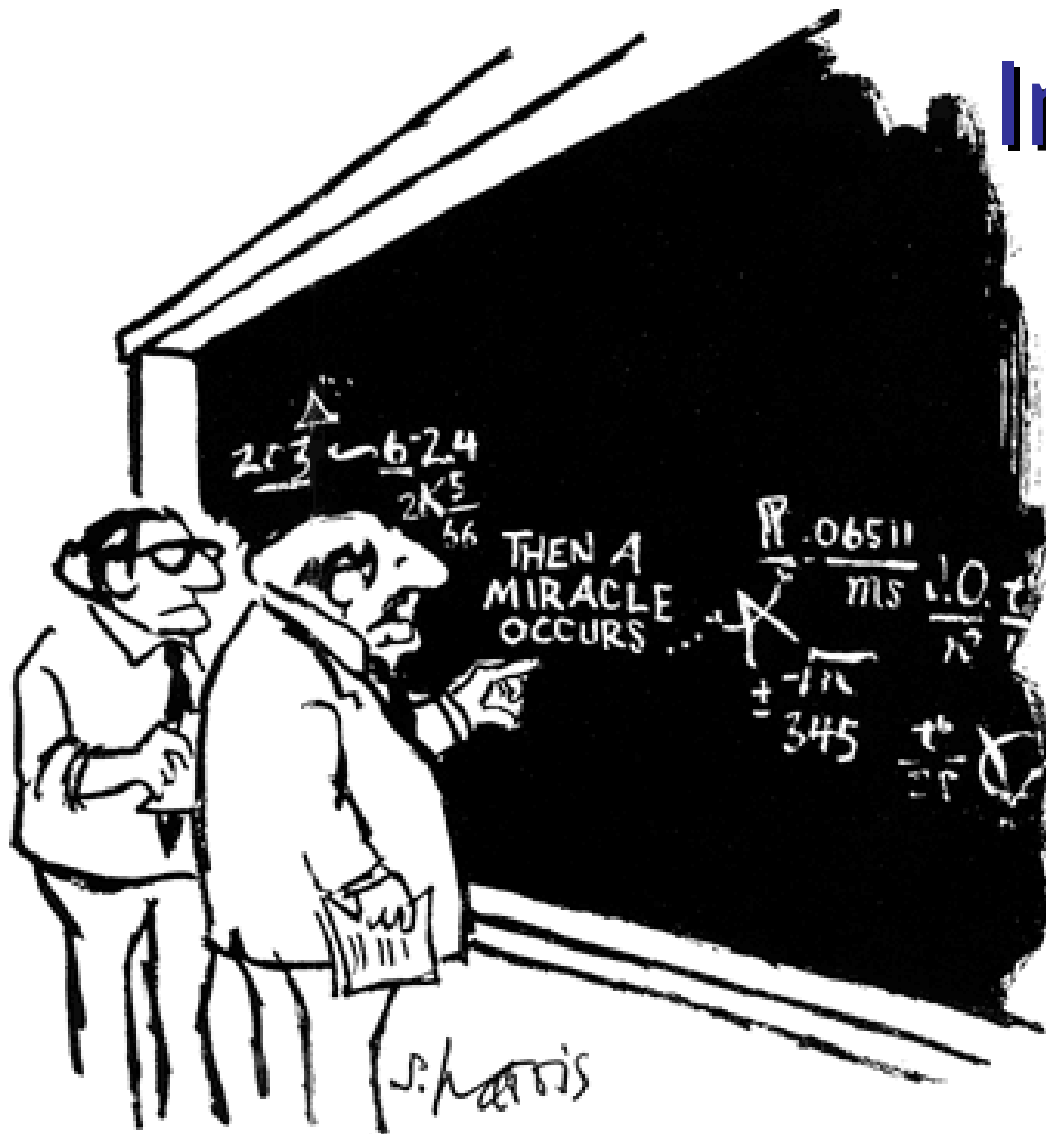
*You just survived the hardest lectures in 615.
It's all downhill from here.*



Recall: Learning Goals

- DS is compositional
- When should I use DS?
- In DS, meaning is a “math object”
- DS uses \perp (“bottom”) to mean non-termination
- DS uses **fixed points** and **domains** to handle `while`
 - This is the tricky bit

Introduction to Axiomatic Semantics



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

Aujourd'hui, nous ferons ...

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

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.

Axiomatic Semantics

- An axiomatic semantics 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, ...

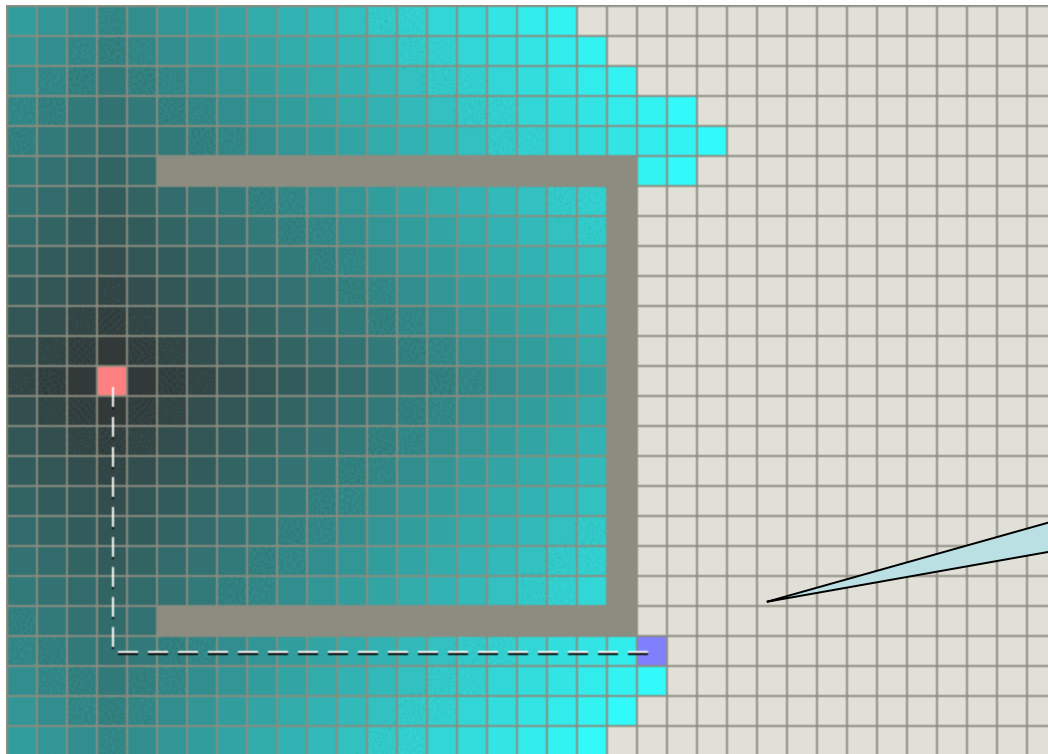
Tony Hoare Quote

- “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 $\langle c, \sigma \rangle \Downarrow \sigma'$
- then B holds in σ'
- A is the precondition
- B is the postcondition
- For example:
$$\{y \leq x\} z := x; z := z + 1 \{y < z\}$$

is a valid assertion
- These are called Hoare triples or Hoare assertions

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

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

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

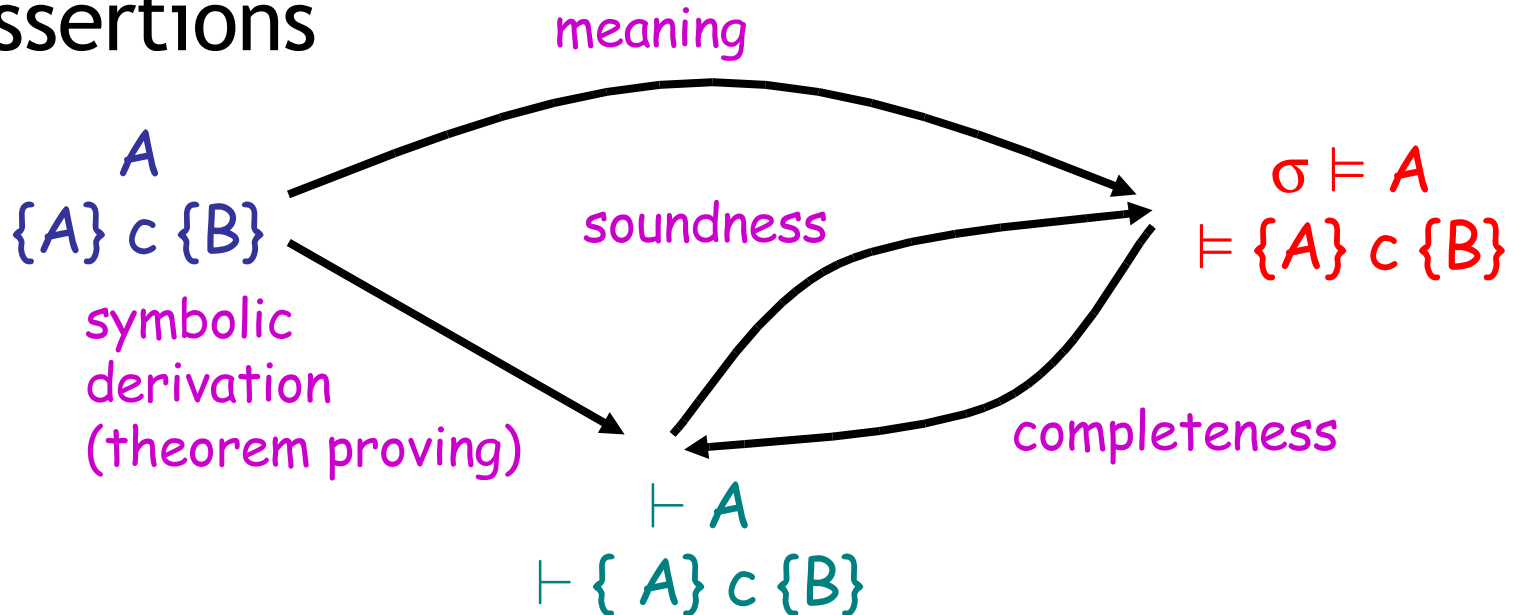
$$\vdash \{x \leq 0\} \text{ while } \dots \{x = 6\}$$

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

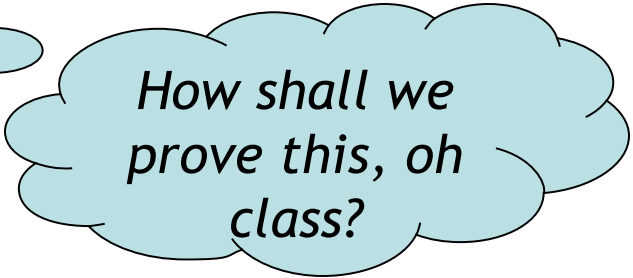
For all σ , if $\sigma \models A$

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

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

then $\sigma' \models B$

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



How shall we prove this, oh class?

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

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®

