Introduction to Axiomatic Semantics

Lecture 10-11
ECS 240

Review

- Operational semantics
  - relatively simple
  - many flavors
  - adequate guide for an implementation of the language
  - not compositional
- Denotational semantics (didn't cover)
  - mathematical
  - canonical
  - compositional
- Operational $\Leftrightarrow$ denotational
- We would also like a semantics that is appropriate for arguing program correctness

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)

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

Hoare Said

- "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

Dijkstra Said

- "Program testing can be used to show the presence of bugs, but never to show their absence!"

Hoare Also Said

- "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 (and praying)
- 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)
  - Documentation of programs and interfaces

Assertions for IMP

- The assertions we make about IMP programs are of the form:

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

  with the meaning that:
  - If A holds in state σ and $\langle c, \sigma \rangle \Downarrow \sigma'$
  - then B holds in σ'
- A is called <u>precondition</u> and B is called <u>postcondition</u>
- For example:

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

  is a valid assertion
- These are called <u>Hoare triple</u> or <u>Hoare assertions</u>

Assertions for IMP (II)

- {A} c {B } is a partial correctness assertion. It does not imply termination
- [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's be more formal
  - 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 and mix the logical variables and the program variables

- Implicitly, for us all IMP variables range over integers

- All IMP boolean expressions are also assertions

Semantics of Assertions

- We introduced a language of assertions, we need to assign meanings to assertions.
- Notation $\sigma \models A$ to say that an assertion holds in a given state .
  - 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.
- 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 $[\![e_1]\!]\,\sigma = [\![e_2]\!]\sigma$

$\sigma \models e_1 \geq e_2$ iff $[\![e_1]\!]\,\sigma \geq [\![e_2]\!]\sigma$

$\sigma \models A_1 \wedge A_2$ 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 $\exists n \in \mathbb{Z}.\sigma[x:=n] \models A$

Semantics of Assertions

- 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 the meaning of a total correctness assertion

  $\models [A]\ c\ [B]$ iff

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

  $$\wedge$$

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

Deriving Assertions

- Now we have the formal mechanism to decide when $\{A\}\ c\ \{B\}$
  - 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
  - And also it is impossible to effectively verify the truth of a $\forall x.\ A$ assertion (by using the definition of validity)

- So we define a symbolic technique for deriving valid assertions from other valid assertions

Derivation Rules for Hoare Triples

- We write $\vdash \{A\}\ c\ \{B\}$ when we can derive the triple using derivation rules
- One derivation rule for each command in the language
- Plus, the rule of <u>consequence</u>

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

Derivation Rules for Hoare Logic

- One rule for each syntactic construct:

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

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

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

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

Hoare Rules

- For some constructs multiple rules are possible:

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

(This was the "forward" axiom for assignment)

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

- Exercise: these rules can be derived from the previous ones using the consequence rules

Example: Assignment

- Assume that x does not appear in e

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

- First the assignment rule

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

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

- Then with the consequence rule:

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

The Assignment Axiom (Cont.)

- Hoare said: "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."

- How about aliasing?
  - If x and y are aliased then

    { true } x := 5 { x + y = 10}

    is true

Example: Conditional

$$\frac{D_1 :: \vdash \{\text{true} \wedge y \leq 0\}\ x := 1\ \{x > 0\} \qquad D_2 :: \vdash \{\text{true} \wedge y > 0\}\ x := y\ \{x > 0\}}{\vdash \{\text{true}\}\ \text{if } y \leq 0 \text{ then } x := 1 \text{ else } x := y\ \{x > 0\}}$$

- D_1 is obtained by consequence and assignment

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

- D_2 is also obtained by consequence and assignment

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

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 $x \leq 6$

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

- Then finish-off with consequence

$$\dfrac{\dfrac{\vdash x \leq 0 \Rightarrow x \leq 6 \qquad \vdash x \leq 6 \wedge x > 5 \Rightarrow x = 6 \qquad \vdash \{x \leq 6\} \text{ while } \ldots \{ x \leq 6 \wedge x > 5\}}{\vdash \{x \leq 0\} \text{ while } \ldots \{x = 6\}}}{}$$

Another Example

- Verify that

  $\vdash \{A\}$ while true do c $\{ B\}$

  holds for any A, B and c
- We must construct a derivation tree

$$\dfrac{\vdash A \Rightarrow \text{true} \qquad \dfrac{\vdash \{\text{true} \wedge \text{true}\}\ c\ \{\text{true}\}}{\{\text{true}\}\ \text{while true do c}\ \{\text{true} \wedge \text{false}\}} \qquad \vdash \text{true} \wedge \text{false} \Rightarrow B}{\vdash \{A\}\ \text{while true do c}\ \{ B\}}$$

- We need an additional lemma:

  $\forall A.\forall c. \vdash \{ A \}\ c\ \{\text{true}\}$

  - How do you prove this one?

Using Hoare Rules. Notes

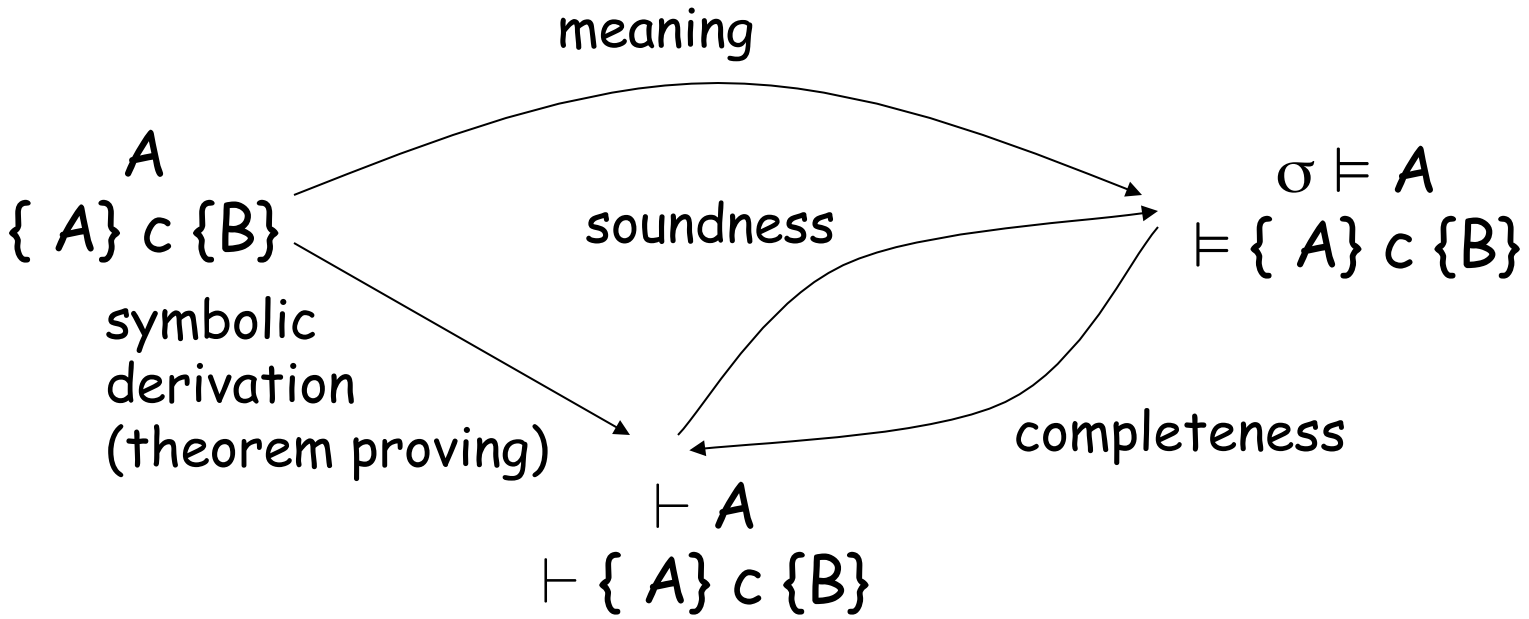
- Hoare rules are mostly syntax directed

- There are three wrinkles:
  - When to apply the rule of consequence ?
  - What invariant to use for while ?
  - How do you prove the implications involved in consequence ?

- The last one can rely on theorem proving
  - This turns out to be doable
  - Loop invariants turn out to be the hardest problem

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 of Axiomatic Semantics

- Formal statement

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

  or, equivalently

  For all σ, if $\sigma \models A$ and $D :: \langle c, \sigma \rangle \Downarrow \sigma'$

  and $H :: \vdash \{ A \} c \{ B \}$ then $\sigma' \models B$

- How can we prove this?
  - By induction on the structure of c?
    - No, problems with while and rule of consequence
  - By induction on the structure of D?
    - No, problems with rule of consequence
  - By induction on the structure of H?
    - No, problems with while
  - By simultaneous induction on the structure of D and H

Simultaneous Induction

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

  $(d, h) < (d', h')$ iff $d < d'$ or $d = d'$ and $h < h'$
  - Called lexicographic ordering
  - Just like the ordering in a dictionary
- This is a well founded order and leads to simultaneous induction

- If $d < d'$ then h can actually be larger than h'!
- It can even be unrelated to h' !

Soundness of the Consequence Rule

- Case: last rule used in $H :: \vdash \{ A\}\ c\ \{ B\}$ is the consequence rule:

$$\frac{\vdash A \Rightarrow A' \qquad H_1 :: \vdash \{A'\}\ c\ \{B'\} \qquad \vdash B' \Rightarrow B}{\vdash \{A\}\ c\ \{B\}}$$

- From soundness of the first-order logic derivations we have $\sigma \models A \Rightarrow A'$, hence $\sigma \models A'$

- From IH with H_1 and **D** we get that $\sigma' \models B'$

- From soundness of the first-order logic derivations we have that $\sigma' \models B' \Rightarrow B$, hence $\sigma' \models B$, q.e.d.

Soundness of the Assignment Axiom

- Case: the last rule used in H :: $\vdash$ { A } c { B} is the assignment rule

$$\frac{}{\vdash \{[e/x]B\}\ x := e\ \{B\}}$$

- The last rule used in D :: $\langle x := e, \sigma \rangle \Downarrow \sigma'$ must be

$$\frac{D_1 :: \langle e, \sigma \rangle \Downarrow n}{\langle x := e, \sigma \rangle \Downarrow \sigma[x := n]}$$

- We must prove the substitution lemma:

  If $\sigma \vDash [e/x]B$ and $\langle e, \sigma \rangle \Downarrow n$ then $\sigma[x := n] \vDash B$

Soundness of the While Rule

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

$$\frac{H_1 :: \vdash \{A \wedge b\}\ c\ \{A\}}{\vdash \{A\}\ \text{while b do c}\ \{A \wedge \neg\, b\}}$$

- There are two possible rules at the root of D.
  - We do only the complicated case

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

Soundness of the While Rule (Cont.)

Assume that $\sigma \models A$

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

- By property of booleans and D_1 we get $\sigma \models b$
  - Hence $\sigma \models A \wedge b$
- By IH on H_1 and D_2 we get $\sigma' \models A$
- By IH on H and D_3 we get $\sigma'' \models A \wedge \neg b$, q.e.d.

- Note that in the last use of IH the derivation H did not decrease
- See Winskel, Chapter 6.5 for a soundness proof with denotational semantics

Completeness of Axiomatic Semantics
Weakest Preconditions

Completeness of Axiomatic Semantics

- Is it true that whenever $\models \{A\}\ c\ \{B\}$ we can also derive $\vdash \{A\}\ c\ \{B\}$?
- If it isn't then it means that there are valid properties of programs that we cannot verify with Hoare rules

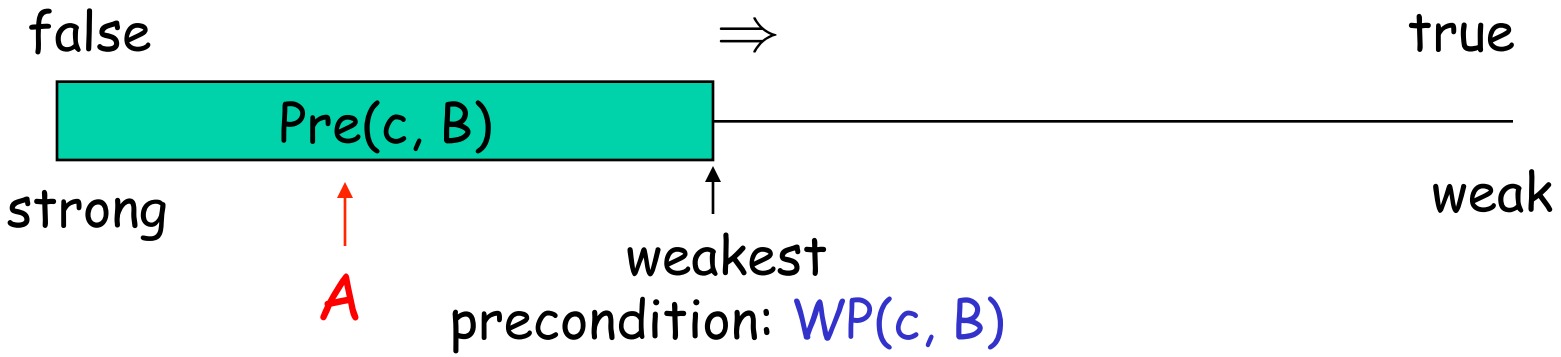
- Good news: for our language the Hoare triples are complete
- Bad news: only if the underlying logic is complete

  (whenever $\models A$ we also have $\vdash A$)

  - this is called relative completeness

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 $Pre(c, B)$
  b) Verify for one $A' \in Pre(c, B)$ that $A \Rightarrow A'$
- Assertions can be ordered:



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

Proof Idea (Cont.)

- Completeness of axiomatic semantics:

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

- Assuming that we can compute wp(c, B) with the following properties:
  1. wp is a precondition (according to the Hoare rules)

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

  2. wp is the weakest precondition

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

$$\frac{\vdash A \Rightarrow wp(c, B) \qquad \vdash \{wp(c, B)\}\ c\ \{B\}}{\vdash \{A\}\ c\ \{B\}}$$

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

Weakest Preconditions

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

$$\frac{\{A\}\ c_1\ \{C\} \qquad \{C\}\ c_2\ \{B\}}{\{\ A\ \}\ c_1;\ c_2\ \{B\}}$$

$wp(c_1;\ c_2,\ B) = wp(c_1,\ wp(c_2,\ B))$

$$\frac{}{\{\ [e/x]B\ \}\ x := e\ \{B\}}$$

$wp(x := e,\ B) = [e/x]B$

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

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

Weakest Preconditions for Loops

- We start from the equivalence

  while b do c = if b then c; while b do c else skip

- Let w = while b do c and W = wp(w, B)

- We have that

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

- But this is a recursive equation !
  - We know how to solve these using domain theory

- We need a domain for assertions

A Partial-Order for Assertions

- What is the assertion that contains least information?
  - true – does not say anything about the state
- What is an appropriate information ordering ?

  $A \sqsubseteq A'$ iff $\models A' \Rightarrow A$
- Is this partial order complete?
  - Take a chain $A_1 \sqsubseteq A_2 \sqsubseteq \ldots$
  - Let $\bigwedge A_i$ be the infinite conjunction of A_i

    $\sigma \models \bigwedge A_i$ iff for all i we have that $\sigma \models A_i$
  - Verify that $\bigwedge A_i$ is the least upper bound
- Can $\bigwedge A_i$ be expressed in our language of assertions?
  - In many cases yes, we'll assume yes for now

Weakest Precondition for WHILE

- Use the fixed-point theorem

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

  - Verify that F is both monotonic and continuous

- The least-fixed point (i.e. the weakest fixed point) is

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

- 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
  - wp_k(while e do c, B) = weakest precondition on which the loop <u>if</u> it terminates in k or fewer iterations, it terminates in B

  $wp_0 = \neg E \Rightarrow B$

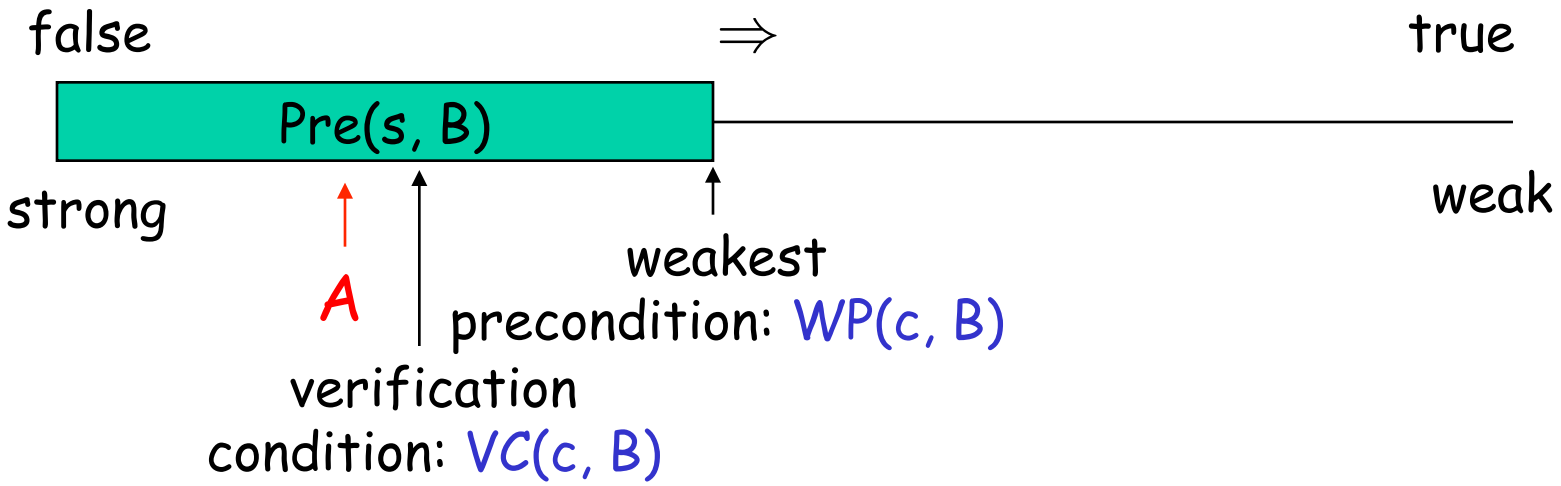
  $wp_1 = E \Rightarrow wp(c, wp_0) \wedge \neg E \Rightarrow B$

  ...
- $wp(\text{while e do c, B}) = \bigwedge_{k \geq 0} wp_k = \text{lub}\{wp_k \mid k \geq 0\}$
- Weakest preconditions are
  - Impossible to compute (in general)
  - Can we find something easier to compute yet sufficient ?

Verification Conditions

Not Quite Weakest Preconditions

- Recall what we are trying to do:



- We shall construct a <u>verification condition</u>: VC(c, B)
  - The loops are annotated with loop invariants !
  - VC is guaranteed stronger than WP
  - But hopefully still weaker than A: $A \Rightarrow VC(c, B) \Rightarrow WP(c, B)$

Verification Conditions

- Factor out the hard work
  - Loop invariants
  - Function specifications
- Assume programs are annotated with such specs.
  - Good software engineering practice anyway
- We will assume that the new form of the while construct includes an invariant:

  while_I b do c
  - The invariant formula must hold every time before b is evaluated

Verification Condition Generation (1)

- Mostly follows the definition of the wp function

VC(skip, B) = B

VC(c_1; c_2, B) = VC(c_1, VC(c_2, B))

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

VC(x := e, B) = [e/x]B

VC(while b do c, B) = ?

Verification Condition Generation for WHILE

$$VC(\text{while}_I\ e\ \text{do}\ c, B) = I \wedge (\forall x_1 \ldots x_n.\ I \Rightarrow (e \Rightarrow VC(c, I)) \wedge \neg e \Rightarrow B))$$

- I holds on entry
- I is preserved in an arbitrary iteration
- B holds when the loop terminates in an arbitrary iteration

- I is the loop invariant (provided externally)
- $x_1, \ldots, x_n$ are all the variables modified in c
- The $\forall$ is similar to the $\forall$ in mathematical induction:

$$P(0) \wedge \forall n \in \mathbb{N}.\ P(n) \Rightarrow P(n+1)$$

VC and Invariants

- Consider the Hoare triple:

  $\{x \le 0\}\ \text{while}_I\ x \le 5\ \text{do}\ x := x + 1\ \{x = 6\}$

- The VC for this is:

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

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

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

Memory Aliasing

Hoare Rules: Assignment

- When is the following Hoare triple valid?

  { A } *x = 5 { *x + *y = 10 }

- A ought to be "*y = 5 or x = y"

- The Hoare rule for assignment would give us:

  [5/*x](*x + *y = 10)

  = 5 + *y = 10

  = *y = 5 (we lost one case)

- How come the rule does not work?

Handling Program State

- We cannot have side-effects in assertions
  - While creating the VC we must remove side-effects !
  - But how to do that when lacking precise aliasing information ?

- Important technique: Postpone alias analysis

- Model the state of memory as a symbolic mapping from addresses to values:
  - If E denotes an address and M a memory state then:
  - sel(M,E) denotes the contents of the memory cell
  - upd(M,E,V) denotes a new memory state obtained from M by writing V at address E

Hoare Rules: Side-Effects

- To model writes correctly we use memory expressions
  - A memory write changes the value of memory

$$\frac{}{\{\, B[upd(\mu, E_1, E_2)/\mu] \,\}\ *E_1 := E_2\ \{B\}}$$

- Important technique: treat memory as a whole
- And reason later about memory expressions with inference rules such as (McCarthy):

$$sel(upd(M, E_1, E_2), E_3) = \begin{cases} E_2 & \text{if } E_1 = E_3 \\ sel(M, E_3) & \text{if } E_1 \neq E_3 \end{cases}$$

Memory Aliasing

- Consider again: { A } *x := 5 { *x + *y = 10 }
- We obtain:

  A = [upd(μ, x, 5)/μ] (*x + *y = 10)

  = [upd(μ, x, 5)/μ] (sel(μ, x) + sel(μ, y) = 10)

  = sel(upd(μ, x, 5), x) + sel(upd(μ, x, 5), y) = 10 (*)

  = 5 + sel(upd(μ, x, 5), y) = 10

  = if x = y then 5 + 5 = 10 else 5 + sel(μ, y) = 10

  = x = y or *y = 5

  (**)
- To (*) is theorem generation
- From (*) to (**) is theorem proving

Mutable Records - Two Models

- Let r : RECORD f1 : T1; f2 : T2 END
- Records are reference types
- Method 1
  - One "memory" for each record
  - One index constant for each field. We postulate f1 ≠ f2
  - r.f1 is sel(r,f1) and r.f1 := E is r := upd(r,f1,E)

- Method 2
  - One "memory" for each field
  - The record address is the index
  - r.f1 is sel(f1,r) and r.f1 := E is f1 := upd(f1,r,E)

Next Time

- ESC/Java