

Introduction to Axiomatic Semantics

Lecture 7-8
CS263

CS 263

1

Review

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

CS 263

2

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)
 - Special-purpose specification languages (Z, Larch, JML)

CS 263

3

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 had rules for structured languages
- Since then, there have been axiomatic semantics for substantial languages, and many applications
 - Program verifiers (70s and 80s)
 - PREFIX: Symbolic execution for bug hunting (WinXP)
 - Software validation tools
 - Malware detection
 - Automatic test generation

CS 263

4

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

CS 263

5

Dijkstra Said

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

CS 263

6

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.**"

CS 263

7

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

CS 263

8

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 the precondition and B is the postcondition
- For example:
 $\{y \leq x\} z := x; z := z + 1 \{y < z\}$
is a valid assertion
- These are called Hoare triple or Hoare assertions

CS 263

9

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

CS 263

10

The Assertion Language

- We use first-order predicate logic with 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

CS 263

11

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

CS 263

12

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

CS 263

13

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

- or even better:

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

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

CS 263

14

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 others that are known to be valid
 - We start with validity of first-order formulas

CS 263

15

Derivation Rules

- We write $\vdash A$ when A can be derived from basic axioms
- The derivation rules for $\vdash A$ are the usual ones from first-order logic with arithmetic:
- Natural deduction style axioms:

$$\begin{array}{c}
 \frac{\vdash A \quad \vdash B}{\vdash A \wedge B} \quad \frac{\vdash [a/x]A \quad (a \text{ is fresh})}{\vdash \forall x.A} \quad \frac{\vdash \forall x.A}{\vdash [e/x]A} \\
 \\
 \frac{\vdash A}{\vdash A} \quad \frac{\vdash A}{\vdash A} \quad \frac{\vdash A}{\vdash A} \quad \frac{\vdash A}{\vdash A} \quad \frac{\vdash A}{\vdash A} \\
 \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\
 \frac{\vdash B}{\vdash A \Rightarrow B} \quad \frac{\vdash A \Rightarrow B \quad \vdash A}{\vdash B} \quad \frac{\vdash [e/x]A}{\vdash \exists x.A} \quad \frac{\vdash \exists x.A \quad \vdash B}{\vdash B}
 \end{array}$$

CS 263

16

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 rule of consequence

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

CS 263

17

Derivation Rules for Hoare Logic

- One rule for each syntactic construct:

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

CS 263

18

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)

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

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

CS 263

19

Example: Assignment

- Assume that x does not appear in e
Prove that $\{ \text{true} \} x := e \{ x = e \}$

- But

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

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

- Assignment + consequence:

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

CS 263

20

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."

- Caveats are sometimes needed for languages with aliasing:

- If x and y are aliased then
 $\{ \text{true} \} x := 5 \{ x + y = 10 \}$
is true

CS 263

21

Example: Conditional

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

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

- D_1 is obtained by consequence and assignment

$$\vdash \{ 1 > 0 \} x := 1 \{ x > 0 \}$$

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

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

- D_2 is also obtained by consequence and assignment

$$\vdash \{ y > 0 \} x := y \{ x > 0 \}$$

$$\vdash \text{true} \wedge y > 0 \Rightarrow y > 0$$

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

CS 263

22

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 $x \leq 6$

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

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

- Then finish-off with consequence

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

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

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

CS 263

23

Another Example

- Verify that

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

holds for any A , B and c

- We must construct a derivation tree

$$\frac{\vdash A \Rightarrow \text{true} \quad \vdash \{ \text{true} \wedge \text{true} \} c \{ \text{true} \}}{\vdash \text{true} \wedge \text{false} \Rightarrow B \quad \{ \text{true} \} \text{while true do } c \{ \text{true} \wedge \text{false} \}}$$

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

- We need an additional lemma:

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

- How do you prove this one?

CS 263

24

Using Hoare Rules. Notes

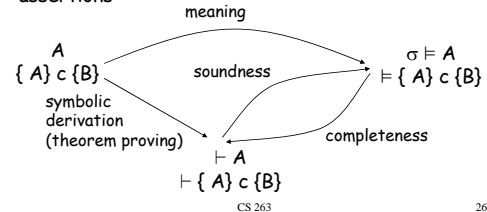
- Hoare rules are mostly syntax directed
- There are three wrinkles:
 - When to apply the rule of consequence ?
 - What invariant to use for while ?
 - How do you prove the implications involved in consequence ?
- The last one is how theorem proving gets in the picture
 - This turns out to be doable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

CS 263

25

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



CS 263

26

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

CS 263

27

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' !

CS 263

28

Soundness of the Consequence Rule

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

$$\frac{\vdash A \Rightarrow A' \quad H_1 :: \vdash \{ A' \} c \{ B' \} \quad \vdash B' \Rightarrow B}{\vdash \{ A \} c \{ B \}}$$
- From soundness of the first-order logic derivations we have $\sigma \models A \Rightarrow A'$, hence $\sigma \models A'$
- From IH with D and H_1 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.

CS 263

29

Soundness of the Assignment Axiom

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

$$\frac{\vdash \{ [e/x] B \} x := e \{ B \}}{\vdash \{ A \} c \{ B \}}$$
- 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 $\sigma \models [e/x]B$ and $\langle \sigma, e \rangle \Downarrow n$ then $\sigma[x := n] \models B$

CS 263

30

Soundness of the While Rule

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

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

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

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

CS 263

31

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 D_2 and H_1 we get $\sigma' \models A$
- By IH on D_3 and H 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

CS 263

32

Completeness of Axiomatic Semantics Weakest Preconditions

CS 263

33

Completeness of Axiomatic Semantics

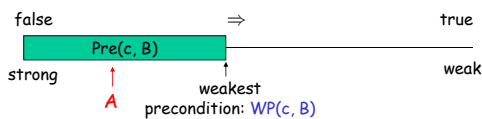
- Is it true that whenever $\models \{ A \} c \{ B \}$ we can also derive $\vdash \{ A \} c \{ B \}$?
- If it isn't then it means that there are valid properties of programs that we cannot verify with Hoare rules
- Good news: for our language the Hoare triples are complete
- Bad news: this is true only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

CS 263

34

Proof Idea

- Dijkstra's idea: To verify that $\models \{ A \} c \{ B \}$
 - Find out all predicates A' such that $\models \{ A' \} c \{ B \}$
 - call this set $Pre(c, B)$
 - Verify that $A \in Pre(c, B)$
- Assertions can be ordered:



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

CS 263

35

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:
 - wp is the weakest precondition
 - If $\models \{ A \} c \{ B \}$ then $\models A \Rightarrow wp(c, B)$
 - wp is a precondition (according to the Hoare rules)
 - $\vdash \{ wp(c, B) \} c \{ B \}$
$$\frac{\vdash A \Rightarrow wp(c, B) \quad \vdash \{ wp(c, B) \} c \{ B \}}{\vdash \{ A \} c \{ B \}}$$
- We also need that whenever $\models A$ then $\vdash A$!

CS 263

36

Weakest Preconditions

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

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

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

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

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

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

CS 263

37

Weakest Preconditions for Loops

- We start from the equivalence $\text{while } b \text{ do } c = \text{if } b \text{ then } c; \text{ while } b \text{ do } c \text{ else skip}$

- Let $w = \text{while } b \text{ do } c$ 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!

- Just like for denotational semantics
- We could use domain theory to solve these (supplemental slides at end of deck)
- Or, we derive the result using finite approximations of while

CS 263

38

Weakest Preconditions - Finite approximations

- Consider the family of predicates W_k defined:
 - A state $\sigma \models W_k$ iff either the execution of while in state σ requires at least k iterations, or it terminates in fewer than k iterations in a final state that satisfies B
 - As for a family of finite unrollings: while_k
 - $\text{while}_0 b \text{ do } c \equiv \text{forever}$
 - $\text{while}_k b \text{ do } c \equiv \text{if } b \text{ then } c; \text{ while}_{k-1} b \text{ do } c \text{ else skip}$
- This gives us the (non-recursive) equations:
 - $W_0 \equiv \text{true}$
 - $W_k \equiv b \Rightarrow wp(c, W_{k-1}) \wedge \neg b \Rightarrow B$

CS 263

39

Weakest Preconditions - Finite approximations

- One can show $W_k \Rightarrow W_{k-1}$
 - Need to prove a monotonicity lemma:
 - If $\models B \Rightarrow B'$ then $\models wp(c, B) \Rightarrow wp(c, B')$
- We can define W as the limit:

$$W(\sigma) = \begin{cases} \text{false} & \text{if } \exists k. W_k(\sigma) \text{ is false} \\ \text{true} & \text{Otherwise} \end{cases}$$

- Alternatively we can write: $W = \bigwedge_{k \geq 0} W_k$
- Can $\bigwedge A_i$ be expressed in our language of assertions?
 - In many cases yes (see Winskel), we'll assume yes for now

CS 263

40

Weakest Preconditions

- We proved that weakest preconditions exist
- See document on the web page for the proof of completeness with weakest preconditions
- But they are impossible to compute (in general)
- WP are needed when you try to prove the strongest fact about your program.
 - Often it is sufficient to prove weaker facts
 - Is there an alternative to WP that is easier to compute yet sufficient in most cases?

CS 263

41

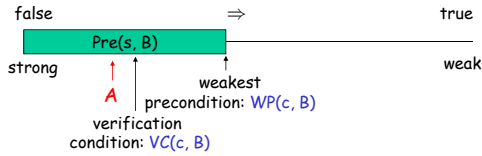
Verification Conditions

CS 263

42

Not Quite Weakest Preconditions

- Recall what we are trying to do:



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

CS 263

43

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:
 - The invariant formula must hold every time before b is evaluated

CS 263

44

Verification Condition Generation (1)

- Mostly follows the definition of the wp function

$VC(\text{skip}, B) = B$
 $VC(c_1; c_2, B) = VC(c_1, VC(c_2, B))$
 $VC(\text{if } b \text{ then } c_1 \text{ else } c_2, B) = b \Rightarrow VC(c_1, B) \wedge \neg b \Rightarrow VC(c_2, B)$
 $VC(x := e, B) = [e/x]B$
 $VC(\text{let } x = e \text{ in } c, B) = [e/x]VC(c, B)$

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

CS 263

45

Verification Condition Generation for WHILE

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

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

CS 263

46

Example of VC

- Compute the VC for the following subprogram

```

x = x0; y = y0;
while [inv ???] x != y do
  if x < y then
    y := y - x
  else
    x := x - y
    
```

CS 263

47

Soundness of VCGen

- Simple form
 - $\models \{VC(c, B)\} c \{B\}$
- Or equivalently
 - $\models VC(c, B) \Rightarrow wp(c, B)$
- Proof is by induction on the structure of c
 - Try it!
- Soundness holds for any choice of invariants!
- We'll look now at properties and extensions of VCs

CS 263

48

VC and Invariants

- Consider the Hoare triple:
 $\{x \leq 0\} \text{ while } x \leq 5 \text{ do } x := x + 1 \{x = 6\}$
- 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

CS 263

49

Forward Verification Condition Generation

- Traditionally VC is computed backwards
 - Works well for structured code
- But it can also be computed in a forward direction
 - Works even for un-structured languages (e.g., assembly language)
 - Uses symbolic evaluation, a technique that has broad applications in program analysis
 - e.g. the PREFIX tool (Intrinsa, Microsoft) works this way

CS 263

50

Forward VC Gen. Idea

- 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 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 $x_2 := e_2$

CS 263

51

Forward VC Generation by Symbolic Evaluation

Details

CS 263

52

Symbolic Evaluation

- Consider the language of instructions:
 $x := e \mid f() \mid \text{if } e \text{ goto } L \mid \text{goto } L \mid L: \mid \text{return} \mid \text{inv } e$
- The "inv e" instruction is an annotation
 - Says that boolean expression e holds at that point
- Notation: I_k is the instruction at address k

CS 263

53

Symbolic Evaluation. The State.

- We set up a symbolic evaluation 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 shall use states also as substitutions:

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

So far this is pretty much like the operational semantics

CS 263

54

Symbolic Evaluation. The Invariants.

- The symbolic evaluator keeps track of the encountered invariants
- A new element of evaluation 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
 - And one more time around an arbitrary iteration

CS 263

55

Symbolic Evaluation. Rules.

- Define a VC function as an interpreter:
 $VC : 1..n \times \text{SymbolicState} \times \text{InvariantState} \rightarrow \text{Assertion}$

$VC(L, \Sigma, Inv)$	if $I_k = \text{goto } L$
$e \Rightarrow VC(L, \Sigma, Inv) \wedge$ $\neg e \Rightarrow VC(k+1, \Sigma, Inv)$	if $I_k = \text{if } e \text{ goto } L$
$VC(k+1, \Sigma[x := \Sigma(e)], Inv)$	if $I_k = x := e$
$VC(k, \Sigma, Inv) = \frac{\Sigma(\text{Post}_{\text{current-function}})}{\Sigma(\text{Pre}_f) \wedge \forall a_1..a_m. \Sigma'(\text{Post}_f) \Rightarrow VC(k+1, \Sigma', Inv)}$ (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 $I_k = \text{return}$ if $I_k = f()$

CS 263

56

Symbolic Evaluation. Invariants.

Two cases when seeing an invariant instruction:

- We see the invariant for the first time
 - $I_k = \text{inv } e$.
 - $k \notin Inv$
 - 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

$$VC(k, \Sigma, Inv) = \frac{\Sigma(e) \wedge \forall a_1..a_m. \Sigma'(e) \Rightarrow VC(k+1, \Sigma', Inv \cup \{k\})}{\text{with } \Sigma' = \Sigma[y_1 := a_1, \dots, y_m := a_m]}$$

(like a function call)

CS 263

57

Symbolic Evaluation. Invariants.

- We see the invariant for the second time
 - $I_k = \text{inv } e$
 - $k \in Inv$

$$VC(k, \Sigma, Inv) = \Sigma(e)$$

(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 (Compaq SRC)
 - Sacrifice completeness for usability

CS 263

58

Symbolic Evaluation. Putting it all together

- Let
 - x_1, \dots, x_n be all the variables and a_1, \dots, a_n fresh parameters
 - Σ_0 be the state $[x_1 := a_1, \dots, x_n := a_n]$
 - \emptyset be the empty *Inv* set
- For all functions f in your program, compute
 $\forall a_1..a_n. \Sigma_0(\text{Pre}_f) \Rightarrow VC(f_{\text{entry}}, \Sigma_0, \emptyset)$
- If all of these predicates are valid then:
 - If you start the program by invoking any f in a state that satisfies Pre_f the program will execute such that
 - At all "inv e " the e holds, and
 - If the function returns then Post_f holds
 - Can be proved w.r.t. a real interpreter (operational semantics)

CS 263

59

VC Generation Example

- Consider the program


```

Precondition:  $x \leq 0$ 

Loop: inv  $x \leq 6$ 
      if  $x > 5$  goto End
       $x := x + 1$ 
      goto Loop

End: return      Postcondition:  $x = 6$ 
            
```

CS 263

60

VC Generation Example (cont.)

$$\begin{aligned} \forall x. & \\ & x \leq 0 \Rightarrow \\ & \quad x \leq 6 \wedge \\ & \quad \forall x'. \\ & \quad \quad (x' \leq 6 \Rightarrow \\ & \quad \quad \quad x' > 5 \Rightarrow x' = 6 \\ & \quad \quad \quad \wedge \\ & \quad \quad \quad x' \leq 5 \Rightarrow x' + 1 \leq 6) \end{aligned}$$

- VC contains both **proof obligations** and assumptions about the control flow

CS 263

61

VC Can Be Large

- Consider the sequence of conditionals
(if $x < 0$ then $x := -x$); (if $x \leq 3$ then $x += 3$)
- With the postcondition $P(x)$
- The VC is
 - $x < 0 \wedge -x \leq 3 \Rightarrow P(-x + 3) \wedge$
 - $x < 0 \wedge -x > 3 \Rightarrow P(-x) \wedge$
 - $x \geq 0 \wedge x \leq 3 \Rightarrow P(x + 3) \wedge$
 - $x \geq 0 \wedge x > 3 \Rightarrow P(x)$
- There is one conjunct for each path
 \Rightarrow exponential number of paths!
- Conjuncts for non-feasible paths have un-satisfiable guard!
- Try with $P(x) = x \geq 3$

CS 263

62

VC Can Be Large (2)

- VCs are exponential in the size of the source because they attempt relative completeness:
 - To handle the case then the correctness of the program must be argued independently for each path
- Remark:
 - It is unlikely that the programmer could write a program by considering an exponential number of cases
 - But possible. Any examples?
- Solutions:
 - Allow invariants even in straight-line code
 - Thus do not consider all paths independently!

CS 263

63

Invariants in Straight-Line Code

- Purpose: modularize the verification task
- Add the command "after c establish I"
 - Same semantics as c (I is only for verification purposes)
- $VC(\text{after } c \text{ establish } I, P) =_{\text{def}} VC(c, I) \wedge \forall x_i. I \Rightarrow P$
 - where x_i are the *ModifiedVars*(c)
- Use when c contains many paths
after if $x < 0$ then $x := -x$ establish $x \geq 0$;
if $x \leq 3$ then $x += 3$ { $P(x)$ }
- VC now is (for $P(x) = x \geq 3$)
 $(x < 0 \Rightarrow -x \geq 0) \wedge (x \geq 0 \Rightarrow x \geq 0) \wedge$
 $\forall x. x \geq 0 \Rightarrow (x \leq 3 \Rightarrow P(x+3) \wedge x > 3 \Rightarrow P(x))$

CS 263

64

Dropping Paths

- In absence of annotations drop some paths
- $VC(\text{if } E \text{ then } c_1 \text{ else } c_2, P) =$ choose one of
 - $E \Rightarrow VC(c_1, P) \wedge \neg E \Rightarrow VC(c_2, P)$
 - $E \Rightarrow VC(c_1, P)$
 - $\neg E \Rightarrow VC(c_2, P)$
- We sacrifice soundness!
 - No more guarantees but possibly still a good debugging aid
- Remarks:
 - A recent trend is to sacrifice soundness to increase usability
 - The PREFIX tool considers only 50 non-cyclic paths through a function (almost at random)

CS 263

65

Hoare Rules: Handling Program State

- 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?

CS 263

66

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

CS 263

67

More on Memory

- We allow variables to range over memory states
 - So we can quantify over all possible memory states
- And we use the special pseudo-variable μ in assertions to refer to the current state of memory
- Example:

" $\forall i. i \geq 0 \wedge i < 5 \Rightarrow sel(\mu, A + i) > 0$ " = `allpositive($\mu, A, 0, 5$)`

says that entries 0..4 in array A are positive

CS 263

68

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: model memory as an object**
- And **reason later about memory expressions** with inference rules such as (McCarthy):

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

CS 263

69

Memory Aliasing

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

$$\begin{aligned} A &= [upd(\mu, x, 5)/\mu] (*x + *y = 10) \\ &= [upd(\mu, x, 5)/\mu] (sel(\mu, x) + sel(\mu, y) = 10) \\ &= sel(upd(\mu, x, 5), x) + sel(upd(\mu, x, 5), y) = 10 \quad (*) \\ &= 5 + sel(upd(\mu, x, 5), y) = 10 \\ &= \text{if } x = y \text{ then } 5 + 5 = 10 \text{ else } 5 + sel(\mu, y) = 10 \\ &= x = y \text{ or } *y = 5 \quad (**) \end{aligned}$$

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

CS 263

70

Alternative Handling for Memory

- Reasoning about aliasing can be expensive (NP-hard)
- Sometimes completeness is sacrificed with the following (approximate) rule:

$$sel(upd(M, E_1, E_2), E_3) = \begin{cases} E_2 & \text{if } E_1 = (\text{obviously}) E_3 \\ sel(M, E_3) & \text{if } E_1 \neq (\text{obviously}) E_3 \\ p & \text{otherwise (p is a fresh new parameter)} \end{cases}$$

- The meaning of "obvious" varies:
 - The addresses of two distinct globals are \neq
 - The address of a global and one of a local are \neq
- "PREfix" and GCC use such schemes

CS 263

71

VC Generation Example

- Consider the program


```

1: I := 0      Precondition: B : bool  $\wedge$  A : array(bool, L)
   R := B
3: inv I  $\geq$  0  $\wedge$  R : bool
   if I  $\geq$  L goto 9
   assert saferd(A + I)
   T := *(A + I)
   I := I + 1
   R := T
   goto 3
9: return R      Postcondition: R : bool
            
```

CS 263

72

VC Generation Example (cont.)

- $$\forall A. \forall B. \forall L. \forall \mu$$
- $$B : \text{bool} \wedge A : \text{array}(\text{bool}, L) \Rightarrow$$
- $$0 \geq 0 \wedge B : \text{bool} \wedge$$
- $$\forall I. \forall R.$$
- $$I \geq 0 \wedge R : \text{bool} \Rightarrow$$
- $$I \geq L \Rightarrow R : \text{bool}$$
- $$\wedge$$
- $$I < L \Rightarrow \text{saferd}(A + I) \wedge$$
- $$I + 1 \geq 0 \wedge$$
- $$\text{sel}(\mu, A + I) : \text{bool}$$
- VC contains both **proof obligations** and assumptions about the control flow

CS 263

73

Two Memory Models for Records

- Let $r : \text{RECORD } f1 : T1; f2 : T2 \text{ END}$
- Model 1
 - One "memory" for each record
 - One index constant for each field. We postulate $f1 \neq f2$
 - $r.f1$ is $\text{sel}(r, f1)$ and $r.f1 := E$ is $r := \text{upd}(r, f1, E)$
- Model 2
 - One "memory" for each field
 - The record address is the index
 - $r.f1$ is $\text{sel}(f1, r)$ and $r.f1 := E$ is $f1 := \text{upd}(f1, r, E)$

CS 263

74

VC as a "Semantic Checksum"

- Weakest preconditions are an expression of the program's semantics:
 - Two equivalent programs have logically equivalent WP
 - No matter how similar their syntax is!
- VC are almost as powerful

CS 263

75

VC as a "Semantic Checksum" (2)

- Consider the program below
 - In the context of type checking:

```
x := 4
x := x == 5
assert x : bool
x := not x
assert x
```
- High-level type checking is not appropriate here
- The VC is: $4 == 5 : \text{bool} \wedge \text{not}(4 == 5)$
- No confusion because reuse of x with different types

CS 263

76

Invariance of VC Across Optimizations

- VC is so good at abstracting syntactic details that it is syntactically preserved by many common optimizations
 - Register allocation, instruction scheduling
 - Common-subexpression elimination, constant and copy prop.
 - Dead code elimination
- We have identical VC whether or not an optimization has been performed
 - Preserves syntactic form, not just semantic meaning!
- This can be used to verify correctness of compiler optimizations (Translation Validation)

CS 263

77

VC Characterize a Safe Interpreter

- Consider a fictitious "safe" interpreter
 - As it goes along it performs checks (e.g. `saferd`, `validString`)
 - Some of these would actually be hard to implement
- The VC describes all of the checks to be performed
 - Along with their context (assumptions from conditionals)
 - Invariants and pre/postconditions are used to obtain a finite expression (through induction)
- VC is valid \Rightarrow interpreter never fails
 - We enforce same level of "correctness"
 - But better (static + more powerful checks)

CS 263

78

Review

- Verification conditions
 - Capture the semantics of code + specifications
 - Language independent
 - Can be computed backward/forward on structured/unstructured code
 - Can be computed on high-level/low-level code

CS 263

79

Invariants Are Not Easy

- Consider the following code from QuickSort

```
int partition(int *a, int L0, int H0, int pivot) {
    int L = L0, H = H0;
    while(L < H) {
        while(a[L] < pivot) L++;
        while(a[H] > pivot) H--;
        if(L < H) { swap a[L] and a[H] }
    }
    return L
}
```
- Consider verifying only memory safety
- What is the loop invariant for the outer loop ?

CS 263

80

A domain of predicates

Alternative solutions to
weakest precondition equations

CS 263

81

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 $A_1 \sqsubseteq A_2 \sqsubseteq \dots$
 - 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 (see Winskel), we'll assume yes for now

CS 263

82

Weakest Precondition for WHILE

- Use the fixed-point theorem
 $F(A) = b \Rightarrow wp(c, A) \wedge \neg b \Rightarrow B$
 - We can verify that F is both monotonic and continuous
- The least-fixed point (i.e. the weakest fixed point) is
 $wp(w, B) = \bigwedge F(\text{true})$
- Notice that unlike for denotational semantics of IMP we are not working on a flat domain !

CS 263

83

Weakest Preconditions (Cont.)

- Define a family of wp's
 - $wp_k(\text{while } e \text{ do } c, B)$ = weakest precondition on which the loop if it terminates in k or fewer iterations, it terminates in B
 - $wp_0 = \neg E \Rightarrow B$
 - $wp_1 = E \Rightarrow wp(c, wp_0) \wedge \neg E \Rightarrow B$
 - ...
- $wp(\text{while } e \text{ do } c, B) = \bigwedge_{k \geq 0} wp_k = \text{lub} \{wp_k \mid k \geq 0\}$
- See 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 ?

CS 263

84