

(The derivation of)
The meaning of programs

CMSC631 Guest Lecture
Saurabh Srivastava

Part of this lecture is based
on George Necula's slides on
Automated Deduction.

The meaning of imperative programs

- Axiomatic Semantics
 - Importance of Loop Invariants
- Verification Conditions
- Soundly (over-)approximating loop invariants
 - Abstract Interpretation
 - Domains of approximation
 - Constraint-based Analysis
- Applications/Project Ideas

Automation



Assumptions



Axiomatic Semantics

Or... how to specify the meaning of
imperative programs

Programs → Theorems. Axiomatic Semantics

- Consists of:
 - A language for making assertions about programs
 - Rules for establishing when assertions hold
- Typical assertions:
 - During the execution, only non-null pointers are dereferenced
 - This program terminates with $x = 0$
- Partial vs. total correctness assertions
 - Safety vs. liveness properties
 - Usually focus on safety (partial correctness)

Partial Correctness Assertions

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

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

with the meaning that:

- Whenever we start the execution of c in a state that satisfies A , the program either does not terminate or it terminates in a state that satisfies B
- A is called precondition and B is called 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

Total Correctness Assertions

- $\{A\} c \{B\}$ is a partial correctness assertion. It does not imply termination
- $[A] c [B]$ is a total correctness assertion meaning that
Whenever we start the execution of c in a state that satisfies A the program does terminate in a state that satisfies B
- 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

Languages for Assertions

- A specification language
 - Must be easy to use and expressive (conflicting needs)
 - Most often the latter, not the former ☹
 - Syntax: how to construct assertions
 - Semantics: what assertions mean
- Typical examples
 - First-order logic
 - Temporal logic (used in protocol specification, hardware specification)
 - Special-purpose languages: Z, Larch, Java ML

State-Based Assertions

- Assertions that characterize the state of the execution
 - state = state of locals + state of memory
- Our assertions will need to be able to refer to
 - Variables
 - Contents of memory
- What are not state-based assertions
 - Variable x is live, lock L will be released
 - There is no correlation between the values of x and y

An Assertion Language

- We'll use a fragment of first-order logic

Formulas $A ::= O \mid T \mid \perp \mid P_1 \wedge P_2 \mid \forall x.P \mid P_1 \Rightarrow P_2 \mid \dots$

Atoms $O ::= f(O_1, \dots, O_n) \mid E_1 \leq E_2 \mid E_1 = E_2 \mid \dots$

Exprs $E ::= n \mid \text{true} \mid \text{false} \mid \dots$

- We can also have an arbitrary assortment of function symbols
 - $\text{ptr}(E, t)$ - expression E denotes a pointer to a t
 - $E:\text{ptr}(t)$ - same in a different notation
 - $E_1 \rightsquigarrow E_2$ - list cell E_2 is reachable from E_1
 - these can be built-in or defined

Semantics of Assertions

- We introduced a language of assertions, we need to assign meanings to assertions.
 - We ignore for now references to memory
- Notation $\rho, \sigma \models A$ to say that an assertion holds in a given state.
 - This is well-defined when ρ is defined on all variables occurring in A and σ is defined on all memory addresses referenced in A
- The \models judgment is defined inductively on the structure of assertions.

Semantics of Assertions

- Formal definition (we drop σ for simplicity):

$\rho \models \text{true}$	always
$\rho \models e_1 = e_2$	iff $\rho \vdash e_1 \Downarrow n_1$ and $\rho \vdash e_2 \Downarrow n_2$ and $n_1 = n_2$
$\rho \models e_1 \geq e_2$	iff $\rho \vdash e_1 \Downarrow n_1$ and $\rho \vdash e_2 \Downarrow n_2$ and $n_1 \geq n_2$
$\rho \models A_1 \wedge A_2$	iff $\rho \models A_1$ and $\rho \models A_2$
$\rho \models A_1 \vee A_2$	iff $\rho \models A_1$ or $\rho \models A_2$
$\rho \models A_1 \Rightarrow A_2$	iff $\rho \models A_1$ implies $\rho \models A_2$
$\rho \models \forall x.A$	iff $\forall n \in \mathbb{Z}. \rho[x:=n] \models A$
$\rho \models \exists x.A$	iff $\exists n \in \mathbb{Z}. \rho[x:=n] \models A$

Semantics of Assertions

- Now we can define formally the meaning of a partial correctness assertion

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

$$\forall \rho \sigma . \forall \rho' \sigma' . (\rho, \sigma \models A \wedge \rho, \sigma \vdash c \Downarrow \rho', \sigma') \Rightarrow \rho', \sigma' \models B$$

- ... and the meaning of a total correctness assertion

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

$$\forall \rho \sigma . \forall \rho' \sigma' . (\rho, \sigma \models A \wedge \rho, \sigma \vdash c \Downarrow \rho', \sigma') \Rightarrow \rho', \sigma' \models B$$

\wedge

$$\forall \rho \sigma . \rho, \sigma \models A \Rightarrow \exists \rho' \sigma' . \rho, \sigma \vdash c \Downarrow \rho', \sigma'$$

Why Isn't This Enough?

- Now we have the formal mechanism to decide when $\{A\} c \{B\}$
 - Start the program in all states that satisfies A
 - Run the program
 - Check that each final state satisfies B
- This is exhaustive testing
- Not enough
 - Can't try the program in all states satisfying the precondition
 - Can't find all final states for non-deterministic programs
 - And also it is impossible to effectively verify the truth of a $\forall x.A$ postcondition (by using the definition of validity)

Derivations as Proxies for Validity

- 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
- We write $\vdash A$ when we can derive (prove) assertion A
 - We wish that $(\forall \rho \sigma. \rho, \sigma \models A) \text{ iff } \vdash A$
- We write $\vdash \{A\} c \{B\}$ when we can derive (prove) the partial correctness assertion
 - We wish that $\models \{A\} c \{B\} \text{ iff } \vdash \{A\} c \{B\}$

Derivation Rules for Assertions

- The derivation rules for $\vdash A$ are the usual ones from first-order logic with
- Natural deduction style axioms:

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

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

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

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

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

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

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

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 rule of consequence

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

Derivation Rules for Hoare Logic

- One rule for each syntactic construct:

$$\frac{}{\vdash \{A\} \text{ skip } \{A\}}$$

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

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

Derivation Rules for Hoare Logic (II)

- The rule for while is not syntax directed
 - It needs a loop invariant

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

- Exercise: try to see what is wrong if you make changes to the rule (e.g., drop " $\wedge b$ " in the premise, ...)

Hoare Rules: Assignment

- Example: $\{ A \} x := x + 2 \{ x \geq 5 \}$. What is A ?
 - A has to imply $x \geq 3$
- General rule:

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

- Surprising how simple the rule is !
- But try $\{ A \} *x = 5 \{ *x + *y = 10 \}$
 - A is $"*y = 5 \text{ or } x = y"$
 - How come the rule does not work?

Example: Assignment

- Assume that x does not appear in e
Prove that $\{\text{true}\} x := e \{x = e\}$
- Because $[e/x](x = e) \equiv (e = [e/x]e) \equiv (e = e)$, we have

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

- Assignment + consequence:

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

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."

- Caveats are sometimes needed for languages with aliasing:
 - 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 is obtained by consequence and assignment

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

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

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

- D_2 is also obtained by consequence and assignment

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

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

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

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 $x \leq 6$

$$\vdash x \leq 6 \wedge x \leq 5 \Rightarrow x + 1 \leq 6$$

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

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

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

- Then finish-off with consequence

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

$$\vdash x \leq 6 \wedge x > 5 \Rightarrow x = 6$$

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

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

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

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

- We need an additional lemma:
$$\forall c. \vdash \{\text{true}\} c \{\text{true}\}$$
 - How do you prove this one?

GCD Example

- Let c be the program:

```
while ( $x \neq y$ ) do
  if ( $x \leq y$ )
    then  $y := y - x$ 
    else  $x := x - y$ 
```

- We'll derive that

$$\vdash \{x = m \wedge y = n\} c \{x = \text{gcd}(m, n)\}$$

GCD Example (2)

- Crucial to select good loop invariant
 - Let the precondition Pre be
$$x = m \wedge y = n$$
 - Let the postcondition $Post$ be
$$x = \text{gcd}(m, n)$$

We use the loop invariant

$$I \stackrel{def}{=} \text{gcd}(x, y) = \text{gcd}(m, n)$$

GCD Example (3)

We first use the rule of consequence to obtain the subgoal

$$\vdash \{ I \} c \{ I \wedge \neg(x \neq y) \} \quad (1)$$

But we also need to prove

$$\vdash Pre \Rightarrow I \quad (2)$$

$$\vdash I \wedge \neg(x \neq y) \Rightarrow Post \quad (3)$$

Subgoal 2 reduces to

$$x = m \wedge y = n \Rightarrow \text{gcd}(x, y) = \text{gcd}(m, n)$$

Subgoal 3 reduces to

$$\text{gcd}(x, y) = \text{gcd}(m, n) \wedge x = y \Rightarrow x = \text{gcd}(m, n)$$

GCD Example (4)

Now we still have to derive subgoal 1:

$$\vdash \{I\} c \{I \wedge \neg(x \neq y)\}$$

We can apply the rule for `while` and we get the subgoal

$$\vdash \{I \wedge x \neq y\} d \{I\} \quad (4)$$

where d is the body of the loop:

```
if( $x \leq y$ )
  then  $y := y - x$ 
  else  $x := x - y$ 
```

GCD Example (5)

We can derive subgoal 4 using the rule for conditionals and we get two subgoals

$$\vdash \{ I \wedge x \neq y \wedge x \leq y \} y := y - x \{ I \} \quad (5)$$

$$\vdash \{ I \wedge x \neq y \wedge x > y \} x := x - y \{ I \} \quad (6)$$

Each of the subgoals 5 and 6 can be derived using the rule of consequence followed by assignment:

$$\vdash I \wedge x \neq y \wedge x \leq y \Rightarrow \text{gcd}(m, n) = \text{gcd}(x, y - x) \quad (7)$$

$$\vdash I \wedge x \neq y \wedge x > y \Rightarrow \text{gcd}(m, n) = \text{gcd}(x - y, y) \quad (8)$$

GCD Example (6)

$$\vdash I \wedge x \neq y \wedge x > y \Rightarrow \text{gcd}(m, n) = \text{gcd}(x - y, y)$$

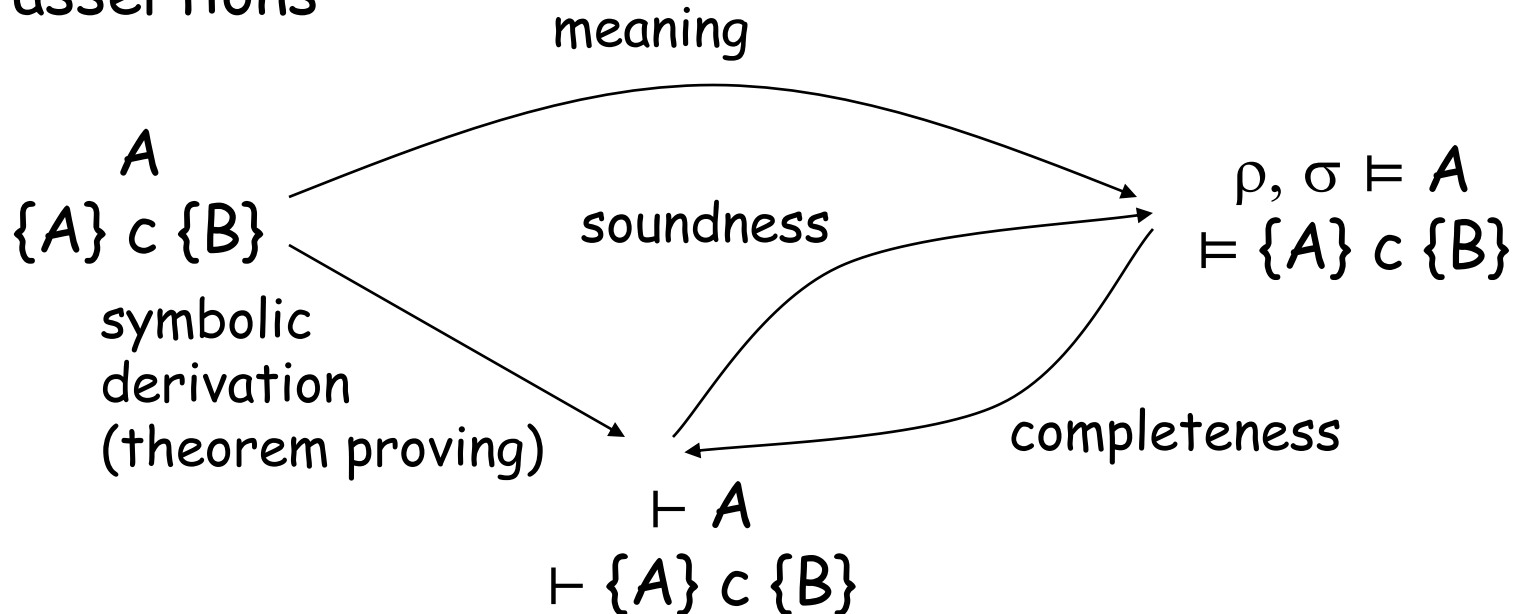
- The above can be proved by realizing that
$$\text{gcd}(x, y) = \text{gcd}(x - y, y)$$
- Q.e.d.
- This completes the proof
- We used a lot of arithmetic
- We had to invent the loop invariants
- What about the proof for total correctness?

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.)

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 ρ, σ , if $\rho, \sigma \models A$ and $\rho, \sigma \vdash c \Downarrow \rho', \sigma'$
and $\vdash \{ A \} c \{ B \}$ then $\rho', \sigma' \models B$

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

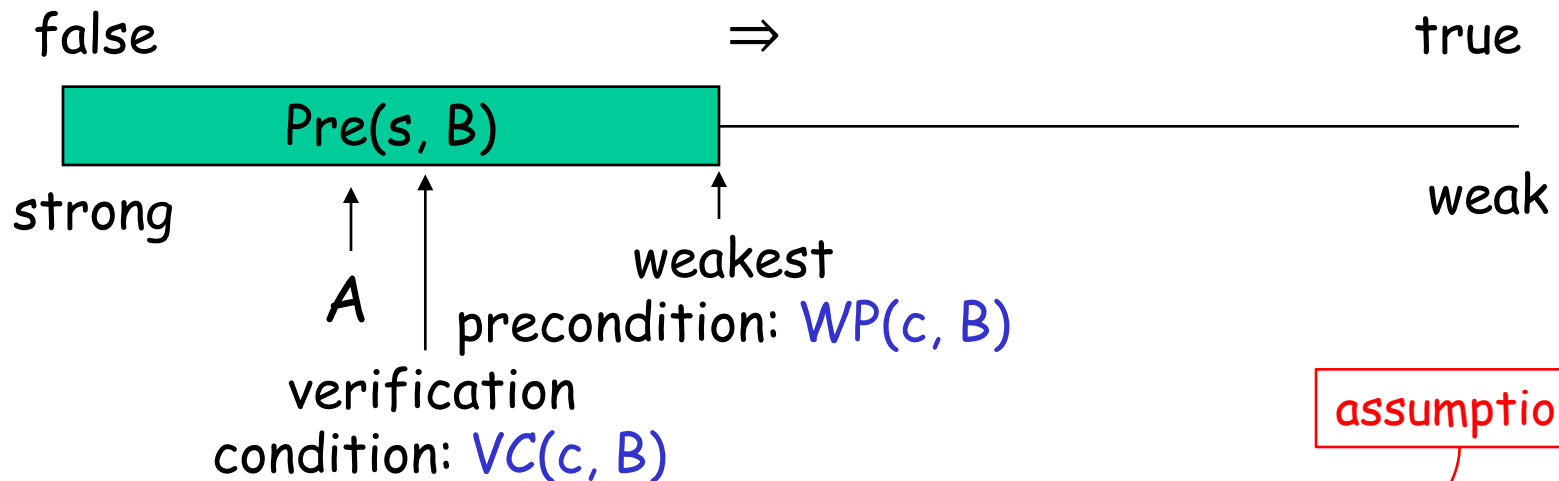
- Good news: for our language the Hoare triples are complete
- Bad news: only if the underlying logic is complete (whenever $\models A$ we also have $\vdash A$)
 - this is called relative completeness

Verification Conditions

Or... how to mechanically verify programs
by making some assumptions

Motivation behind Verification Conditions

- For: $\{A\} c \{B\}$ suppose:
 - Suppose $\models \{X\} c \{B\}$ for any $X \in \text{Pre}(c, B)$; $A \in \text{Pre}(c, B)$
 - But $\vdash \{VC(c, B)\} c \{B\}$



- We shall construct a verification condition: $VC(c, B)$
 - The loops are annotated with loop invariants !
 - VC stronger than the weakest possible $WP(c, B)$
 - 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:
$$\text{while}_I b \text{ do } c$$
 - The invariant formula must hold every time before b is evaluated

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 ?

Verification Condition Generation (1)

- Syntactic construction:

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

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

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

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

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

$$VC(\text{assume } \phi, B) = \phi \Rightarrow B$$

$$VC(\text{assert } \phi, B) = \phi \wedge B$$

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

Notice the connection
to the \vdash Hoare rules

Verification Condition Generation for WHILE

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

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

VC and Invariants

- Consider the Hoare triple:

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

- 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

Abstract Interpretation

Or... how to soundly over-approximate the execution of programs

Invariants as Fixpoints

- Till now we assumed we (magically) knew the invariants
- Suppose invariants are not given
- Invariant Requirements:
 - Holds on entry
 - Holds on arbitrary iteration
 - Optional: Useful, i.e., proves assertion on exit
- Recall: VC involving the invariants is a recursive eq.
$$I \wedge (\forall x_1 \dots x_n. I \Rightarrow (e \Rightarrow VC(c, I) \wedge \neg e \Rightarrow B))$$
- Exactly the notion of a fixpoint solution. Can we compute it mechanically?

Invariants as Fixpoints

- Two systematic strategies for finding fixpoints
- Abstract Interpretation
 - Interpret a program over an abstract domain
 - Find the best possible fixpoint
 - Hope that the best fixpoint is useful on exit
- Later: Constraint-based Analysis
 - Use abstract domain to reduce VC to solvable constraints
 - May not find the best fixpoint
 - Explicitly encodes usefulness

Abstract Domains

- Suppose we make some assumptions
 - About the programs: e.g. manipulates integer valued variables
 - Invariants: e.g. involve only $L \leq x \leq R$ facts
- Given rise to a domain of reasoning
 - Range: $x \in [L,R]$
 - Difference constraints: $x-y \leq c$
 - Linear Arithmetic (LIA): $ax+by \leq c$
 - Uninterpreted functions: $\text{Fib}(n)$
 - Predicate abstraction: set of arbitrary predicates p_i
- Choice of domain implies choice of operators

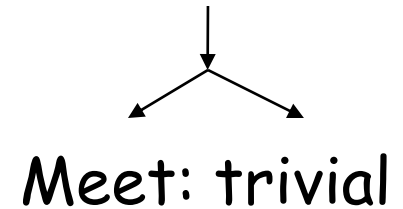
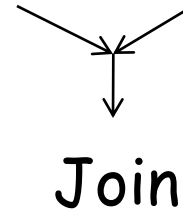
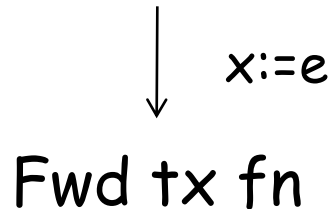
Notice the hierarchy?

Abstract Interpretation

- Interpret the program over an abstract domain
 - I.e., each variable, instead of taking concrete values takes values from an abstract domain
 - E.g., Range Domain, where $x \in [L,R]$; instead of $x=3$ we may have an abstract value as $x=[0,3]$ — an over-approximation
- Run the fwd interpreter over the CFG
 - Ensure that each step we soundly over-approximate
- Keep iterating (around loops) till fixed point reached

Abstract Interpretation: Domain Operators

- Run the fwd interpreter over the CFG
- Need to handle:



- E.g., Range domain:
 - Abstract facts: $\wedge_i x_i = [L_i, R_i]$

$$x=[0,3] \wedge y=[0,3] \xrightarrow{x:=4} x=[4,4] \wedge y=[0,3]$$

$$x=[4,4] \wedge y=[0,3] \rightarrow (x=[4,4] \wedge y=[0,3])$$

$$x=[-4,0] \wedge y=[-3,0] \rightarrow (x=[-4,0] \wedge y=[-3,0])$$

Not in domain

$$\rightarrow x=[-4,4] \wedge y=[-3,3]$$

Lose information:
OK because
over-approximating

Constraint-based analysis

Or... how to express the meaning of
programs as SAT instances

Templates as Domains

- If $[-]$ denote a `hole', then e.g. of templates are:
 - $[-].x + [-].y + [-].z \leq c$
 - $[-] \vee [-]$
 - $\forall k : [-] \Rightarrow A[k] \leq A[k+1]$
 - $\forall x \exists y : [-] \Rightarrow A[x] = B[y] \wedge [-]$
- Inside the holes, $[-]$:
 - Facts from chosen domains, as earlier
 - We will look at Predicate Abstraction

Predicate Abstraction

- Given a fixed finite set of n predicates, associate with each predicate p_i a **boolean indicator** b_i

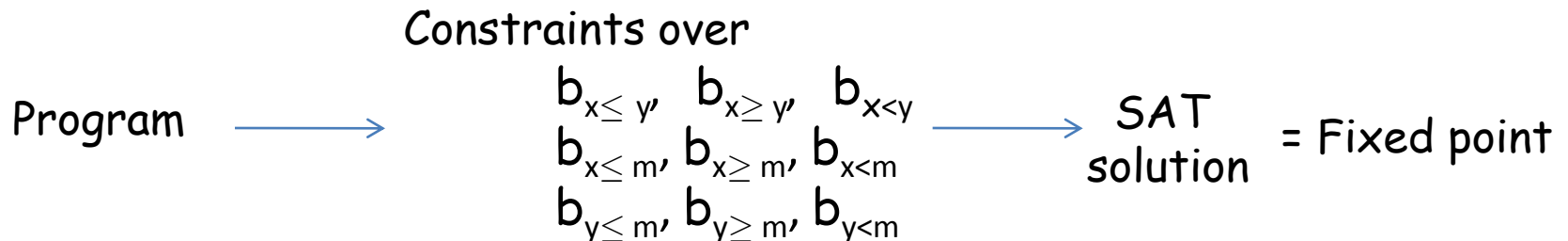
$$\begin{array}{l}
 x \leq y, x \geq y, x < y \\
 x \leq m, x \geq m, x < m \\
 y \leq m, y \geq m, y < m
 \end{array}
 \longrightarrow
 \begin{array}{l}
 b_{x \leq y}, b_{x \geq y}, b_{x < y} \\
 b_{x \leq m}, b_{x \geq m}, b_{x < m} \\
 b_{y \leq m}, b_{y \geq m}, b_{y < m}
 \end{array}$$

- Sound over-approximation of the invariant
 - Boolean expression involving the indicators

$$\mathbf{I} : (x = y \wedge x \leq m) \longrightarrow
 \begin{array}{l}
 b_{x \leq y} = \text{tt} \wedge \\
 b_{x \geq y} = \text{tt} \wedge \\
 b_{x \leq m} = \text{tt}
 \end{array}
 \text{ OR }
 \begin{array}{l}
 b_{x \leq y} = \text{tt} \wedge \\
 b_{x \geq y} = \text{tt} \wedge \\
 b_{y \leq m} = \text{tt}
 \end{array}
 \text{ OR...}$$

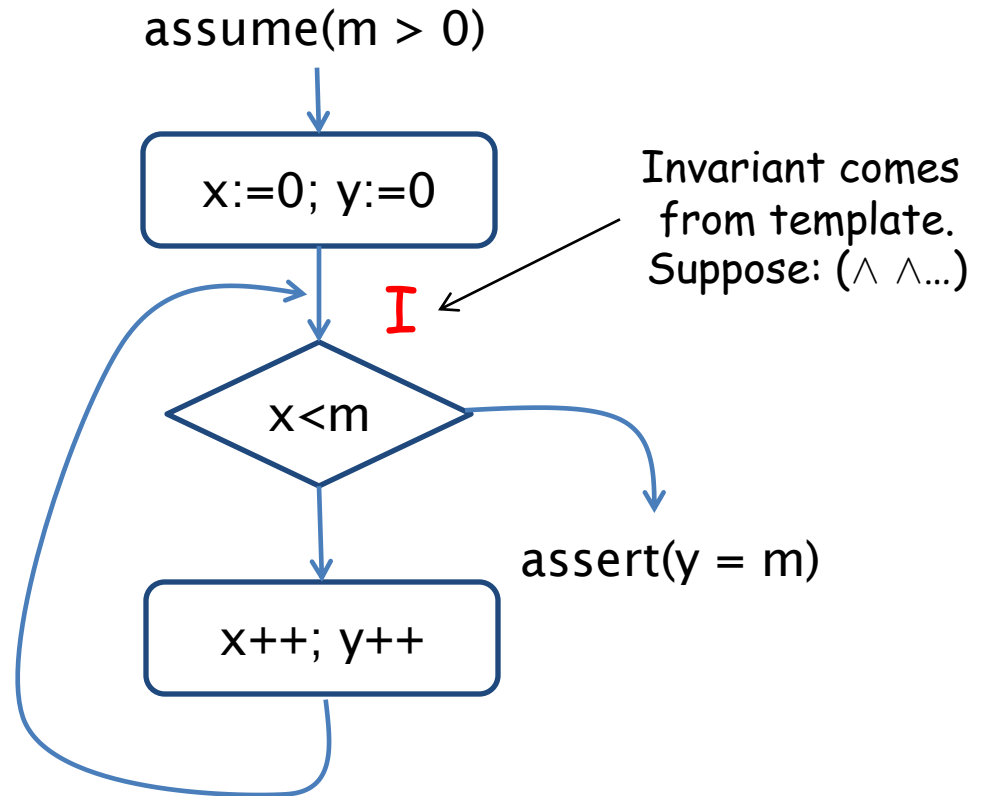
Constraint-based Invariant Inference

- Guess a template: k disjuncts: $(...) \vee (...) \vee (...) : k=3$
- Task: Fill out each disjunct with a boolean monomial (conjunction of indicator literals)
- Approach: Generate boolean constraints over indicators using the program semantics and **directly solve** using off-the-shelf solvers.



Example: Invariants with Templates

```
loop (int m) {  
  assume(m > 0)  
  x:=0; y:=0;  
  
  while (x<m) {  
    x++; y++;  
  }  
  
  assert(y = m)  
}
```



Example: VCs

- 1 $m > 0 \Rightarrow \mathbf{I}[y \rightarrow 0, x \rightarrow 0]$
- 2 $\mathbf{I} \wedge x \geq m \Rightarrow y = m$
- 3 $\mathbf{I} \wedge x < m \Rightarrow \mathbf{I}[y \rightarrow y+1, x \rightarrow x+1]$

Example: Boolean Constraint Generation

Unknown invariant on the LHS;
constrains how **weak** I can be

Unknown invariant on the RHS;
constrains how **strong** I can be

-
- 1 $m > 0 \Rightarrow I[y \rightarrow 0, x \rightarrow 0]$
- 2 $I \wedge x \geq m \Rightarrow y = m$
- 3 $I \wedge x < m \Rightarrow I[y \rightarrow y + 1, x \rightarrow x + 1]$
- The diagram shows three numbered constraints. A red arrow from the 'weak' text points to constraint 1. A red arrow from the 'strong' text points to constraint 2. A red arrow from the 'both sides' text points to the 'I' in constraint 3.

Unknown on both sides;
combination of above cases

Example: Boolean Constraint Generation

Unknown invariant on the LHS;
constrains how **weak** I can be

$$\boxed{2} \quad I \wedge x \geq m \Rightarrow y = m$$

$$1: y \leq m \wedge y \geq m$$

$$2: x < m$$

$$3: x \leq y \wedge y \leq m$$

$$x \leq y, x \geq y, x < y$$

$$x \leq m, x \geq m, x < m$$

$$y \leq m, y \geq m, y < m$$

Unknown invariant on the RHS;
constrains how **strong** I can be

$$\boxed{1} \quad m > 0 \Rightarrow I[y \rightarrow 0, x \rightarrow 0]$$

$$0 \leq 0, 0 \geq 0, 0 < 0$$

$$0 \leq m, 0 \geq m, 0 < m$$

$$0 \leq m, 0 \geq m, 0 < m$$

Maximally-weak ways of satisfying
the constraint using the given preds.

$$\neg \left(\begin{array}{l} x < y \\ x \geq m \\ y \geq m \end{array} \right)$$

$$(b_{x < m}) \vee (b_{y \leq m} \wedge b_{y \geq m}) \vee (b_{x \leq y} \wedge b_{y \leq m})$$

$$\neg b_{x \geq m} \wedge \neg b_{x < y} \wedge \neg b_{y \geq m}$$

Example: Solving using SAT

$$(b_{x < m}) \vee (b_{y \leq m} \wedge b_{y \geq m}) \vee (b_{x \leq y} \wedge b_{y \leq m})$$

$$\neg b_{x \geq m} \wedge \neg b_{x < y} \wedge \neg b_{y \geq m}$$

$$(b_{y \leq m} \Rightarrow (b_{y < m} \vee b_{y \leq x})) \wedge \neg b_{x < m} \wedge \neg b_{y < m}$$

Individual *local* computations

SAT Solver
(fixed point
computation)

tt: $b_{y \leq x}$; $b_{y \leq m}$; $b_{x \leq y}$
ff: rest

$$\mathbf{I}: y = x \wedge y \leq m$$

```
loop (int m) {  
  assume(m > 0)  
  x:=0; y:=0;  
  while (x < m) {  
    x++; y++;  
  }  
  assert(y = m)  
}
```

Sample Projects

Or... how to get a grade in this course. 😊

Verification of heap manipulating programs

- Consider the following program:

```
x := head;  
whileI (x != ⊥) {  
  x->data := 0;  
  x := x->next;  
}
```

$\forall k: (\text{head} \sim k \wedge k \neq \perp) \Rightarrow k \rightarrow \text{data} = 0$

- Verification Conditions:

$\text{true} \Rightarrow I [\text{head}/x]$

$I \wedge x = \perp \Rightarrow \forall k: (\text{head} \sim k \wedge k \neq \perp) \Rightarrow k \rightarrow \text{data} = 0$

$I \wedge x \neq \perp \Rightarrow I[x \rightarrow \text{next}/x, 0/x \rightarrow \text{data}]$

$I: \forall k: (\text{head} \sim k \wedge x \not\sim k) \Rightarrow k \rightarrow \text{data} = 0$

Verification of heap manipulating programs

- Need the semantics of reachability ' $\sim>$ '
- Specified axiomatically:

Reflexivity: $\forall x: x \sim> x$
Transitivity: $\forall x, y, z: x \sim> y \wedge y \sim> z \Rightarrow x \sim> z$
Step: Head: $\forall x: x \neq \perp \Rightarrow x \sim> (x \rightarrow \text{next})$
Step: Tail: $\forall x, y: x \sim> y \Rightarrow x = y \vee (x \rightarrow \text{next}) \sim> y$
End: $\forall x: \perp \sim> x \Rightarrow x = \perp$

$\text{true} \Rightarrow I[\text{head}/x]$

$I \wedge x = \perp \Rightarrow \forall k: (\text{head} \sim> k \wedge k \neq \perp) \Rightarrow k \rightarrow \text{data} = 0$

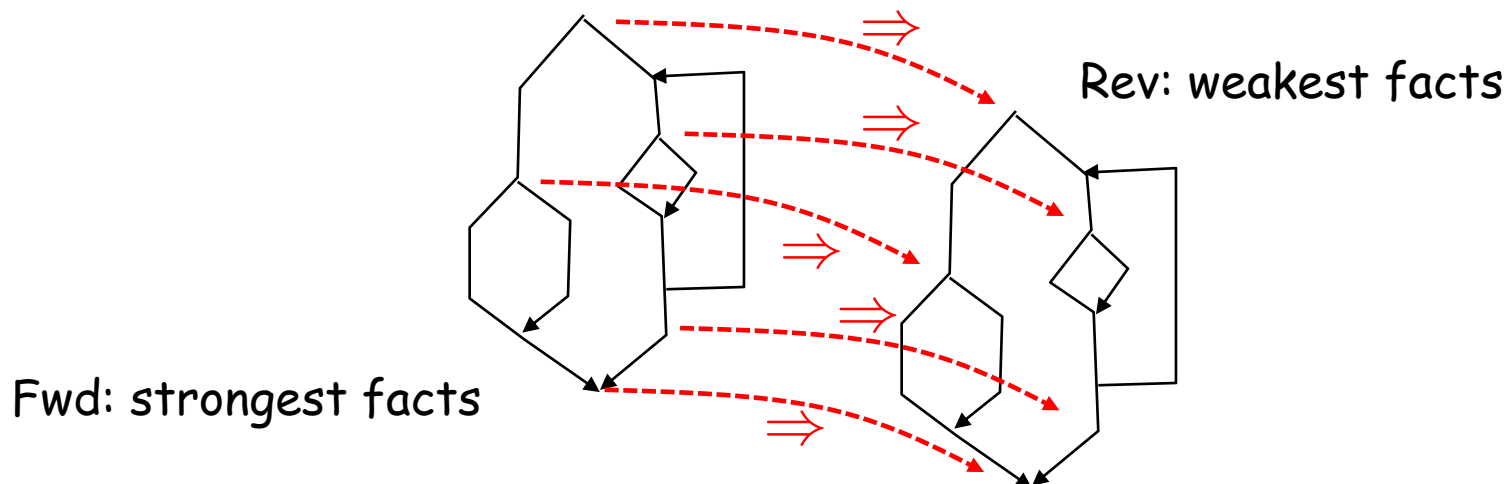
$I \wedge x \neq \perp \Rightarrow I[x \rightarrow \text{next}/x, 0/x \rightarrow \text{data}]$

$I: \forall k: (\text{head} \sim> k \wedge x \sim> k) \Rightarrow k \rightarrow \text{data} = 0$

- Find the semantics of $\sim>$ for the case of trees!
- Use them to verify tree data structures
- Extra: Prove the semantics consistent and sufficient

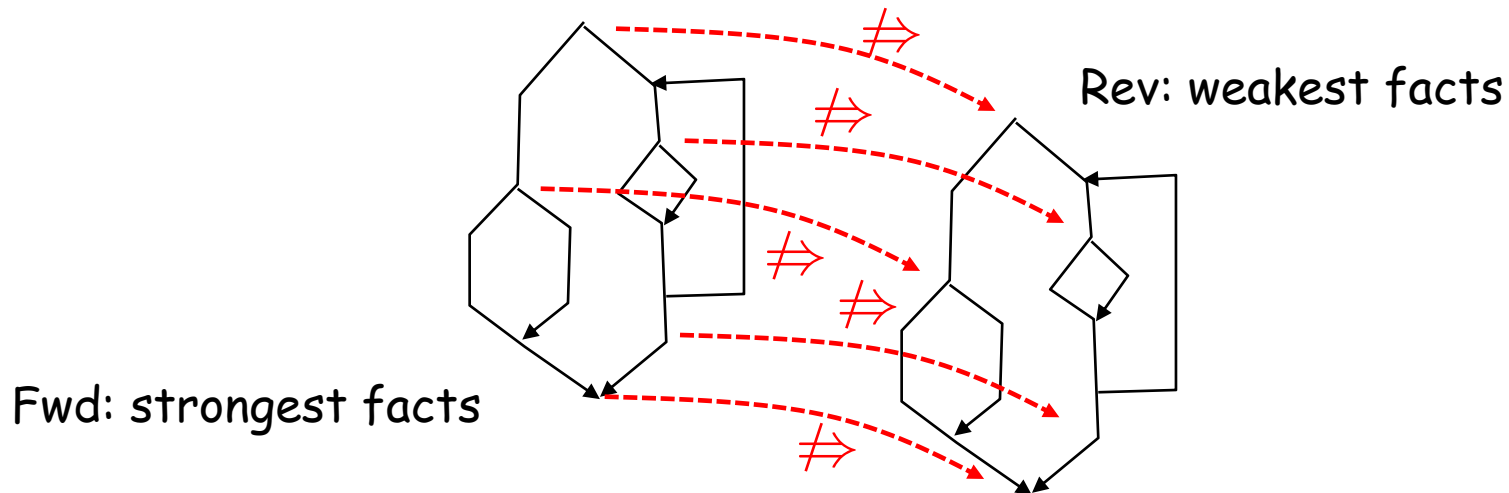
Finding minimal fixes to buggy programs

- Fwd analysis (AI or Constraint-based):
 - We talked of the fwd operators
 - Best fixedpoints are the strongest facts at each prg. loc.
- Rev analysis (AI or Constraint-based):
 - Operators different but possible
 - Best fixedpoints are the weakest facts at each prg. loc.
- In a bug-free program:



Finding minimal fixes to buggy programs

- In a program with some bugs:



- Compute both fwd (F_i) and rev (R_i) facts
- Foreach prg. loc. find minimal explanations E_i
 - $E_i \vdash F_i \Rightarrow R_i$
 - (Abduct operator in the Art. Intell. community)
- Heuristic to choose the best E_i