Lecture 12: Reasoning about Programs using Hoare Logic II

Yu Feng

Fall 2019

Summary of previous lecture

- 2nd homework is due now
- Reasoning about (partial) correctness with Hoare Logic

Simple Imperative Programming Language

Expression E

- Z | V | $E_1 + E_2$ | $E_1 * E_2$

Conditional C

True | False | $E_1 = E_2$ | $E_1 \leq E_2$

Statement S

- skip (Skip)
- abort. (Abort)
- V := E (Assignment)
- S_1; S_2. (Composition)
- **if** C **then** S_1 **else** S_2 (If)
- **while** C **do** S (While)

Simple Imperative Programming Language

Expression E

- $Z \mid V \mid E_1 + E_2 \mid E_1 * E_2$

Conditional C

True | False | $E_1 = E_2 \mid E_1 \leq E_2$

A minimalist programming language for demonstrating key features of Hoare logic.

Statement S

- skip (Skip)
- abort. (Abort)
- V := E (Assignment)
- $S_1; S_2$. (Composition)
- **if** C **then** S_1 **else** S_2 (If)
- **while** C **do** S (While)

Hoare logic rules

$$\frac{}{\vdash \{P\}\ \mathrm{Skip}\ \{P\}}$$

$$\frac{\vdash \{P\}\ S_1\ \{R\} \qquad \vdash \{R\}\ S_2\ \{Q\}}{\vdash \{P\}\ S_1; S_2\ \{Q\}}$$

$$\frac{}{\vdash \{\mathrm{true}\}\ \mathrm{abort}\ \{\mathrm{false}\}}$$

$$\frac{\vdash \{P \wedge C\}\ S_1\ \{Q\} \qquad \vdash \{P \wedge \neg C\}\ S_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ C\ \textbf{then}\ \mathbf{S_1}\ \textbf{else}\ \mathbf{S_2}\ \{Q\}}$$

$$\frac{}{\vdash \{Q[E/x]\}\ x := E\ \{Q\}}$$

$$\frac{\vdash \{I \wedge C\}\ S\ \{I\}}{\vdash \{I\}\ \textbf{while}\ C\ \textbf{do}\ S\ \{I \wedge \neg C\}}$$

$$\frac{\vdash \{P_1\}\ S\ \{Q_1\} \quad P \Rightarrow P_1 \quad Q_1 \Rightarrow Q}{\vdash \{P\}\ S\ \{Q\}}$$

Outline of this lecture

- Loop invariant
- Automating Hoare logic with verification condition (VC)
- A brief introduction about the Dafny tool

Proof rule for loop

$$\frac{\vdash \{I \wedge C\}\ S\ \{I\}}{\vdash \{I\}\ \textbf{while}\ C\ \textbf{do}\ S\ \{I \wedge \neg C\}}$$

- A loop invariant I has following properties:
  - I holds before the loop
  - I holds after each iteration of the loop
- Suppose I is a loop invariant for this loop. What is guaranteed to hold after loop terminates?
- This rule simply says "If I is a loop invariant, then $I \wedge \neg C$ must hold after loop terminates"

Proof rule for loop

$$\frac{\vdash \{I \wedge C\}\ S\ \{I\}}{\vdash \{I\}\ \textbf{while}\ C\ \textbf{do}\ S\ \{I \wedge \neg C\}}$$

Consider the statement S= while x<n do x=x+1

Let's prove validity of $\{x \leq n\}$ S $\{x \geq n\}$

What is the appropriate loop invariant?

First, let's prove $x \leq n$ is loop invariant. What do we need to show?

Proof rule for loop

$$\frac{\vdash \{I \wedge C\}\ S\ \{I\}}{\vdash \{I\}\ \textbf{while}\ C\ \textbf{do}\ S\ \{I \wedge \neg C\}}$$

Consider the statement S= while x<n do x=x+1

Let's prove validity of $\{x \leq n\}$ S $\{x \geq n\}$

What is the appropriate loop invariant?

First, let's prove $x \leq n$ is loop invariant. What do we need to show?

$$\cfrac{\cfrac{\cfrac{\vdash \{x+1 \leq n\} x = x+1 \{x \leq n\} \quad x \leq n \wedge x < n \Rightarrow x+1 < n}{\vdash \{x \leq n \wedge x < n\} x = x+1 \{x \leq n\}}}{\vdash \{x \leq n\} S \{x \leq n \wedge \neg(x < n)\}} \quad x \leq n \wedge \neg(x < n) \Rightarrow x \geq n}{\{x \leq n\} S \{x \geq n\}}$$

Invariant vs. Inductive Invariant

- Suppose I is a loop invariant for "while C do S"
- Does it always satisfy $\{I \wedge C\}\ S\ \{I\}$?
- Consider $I = j \geq 1$ and the code:

  i:=1; j:=1; while i<n do{j:=j+i; i:=i+1}

- Strengthened invariant $j \geq 1 \wedge i \geq 1$
- Key challenge in verification is finding inductive loop invariants

Manual proof construction is tedious

$\{x \leq n\}$ // precondition

while $(x < n)$ **do**

$\{x \leq n \wedge x < n\}$ // loop invariant

$x := x + 1$

$\{x = n\}$ // postcondition

Manual proof construction is tedious

$\{x \leq n\}$ // precondition

while $(x < n)$ **do**

$\{x \leq n \wedge x < n\}$ // loop invariant

$x := x + 1$

$\{x = n\}$ // postcondition

Hoare Logic proofs are highly manual:

Manual proof construction is tedious

$\{x \leq n\}$ // precondition

while $(x < n)$ **do**

$\{x \leq n \wedge x < n\}$ // loop invariant

$x := x + 1$

$\{x = n\}$ // postcondition

Hoare Logic proofs are highly manual:

- When to apply the rule of consequence?

Manual proof construction is tedious

$\{x \leq n\}$ // precondition

while $(x < n)$ **do**

$\{x \leq n \wedge x < n\}$ // loop invariant

$x := x + 1$

$\{x = n\}$ // postcondition

Hoare Logic proofs are highly manual:

- When to apply the rule of consequence?
- What loop invariants to use?

Manual proof construction is tedious

$\{x \leq n\}$ // precondition

while $(x < n)$ **do**

$\{x \leq n \wedge x < n\}$ // loop invariant

$x := x + 1$

$\{x = n\}$ // postcondition

Hoare Logic proofs are highly manual:

- When to apply the rule of consequence?
- What loop invariants to use?

We can automate much of the proof process with **verification condition generation**!

Manual proof construction is tedious

$\{x \leq n\}$ // precondition

while $(x < n)$ **do**

$\{x \leq n \wedge x < n\}$ // loop invariant

$x := x + 1$

$\{x = n\}$ // postcondition

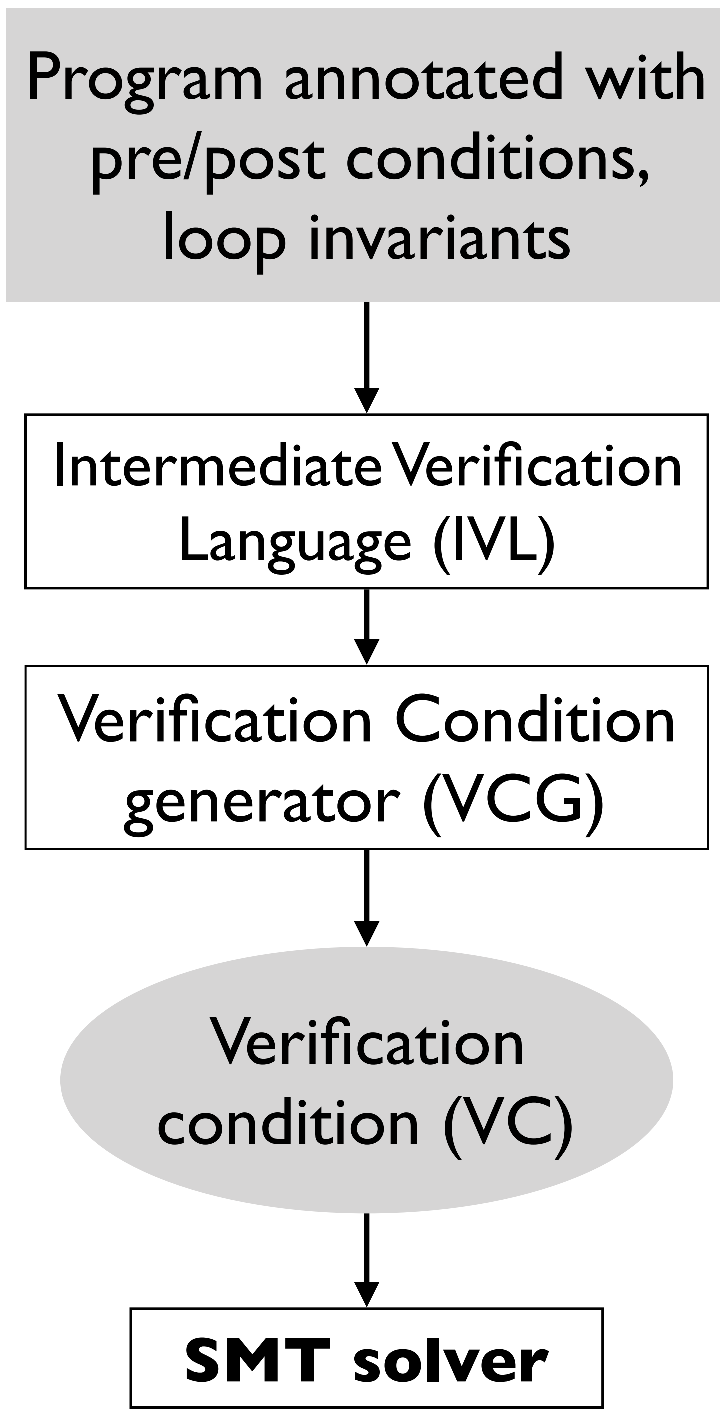
Hoare Logic proofs are highly manual:

- When to apply the rule of consequence?
- What loop invariants to use?

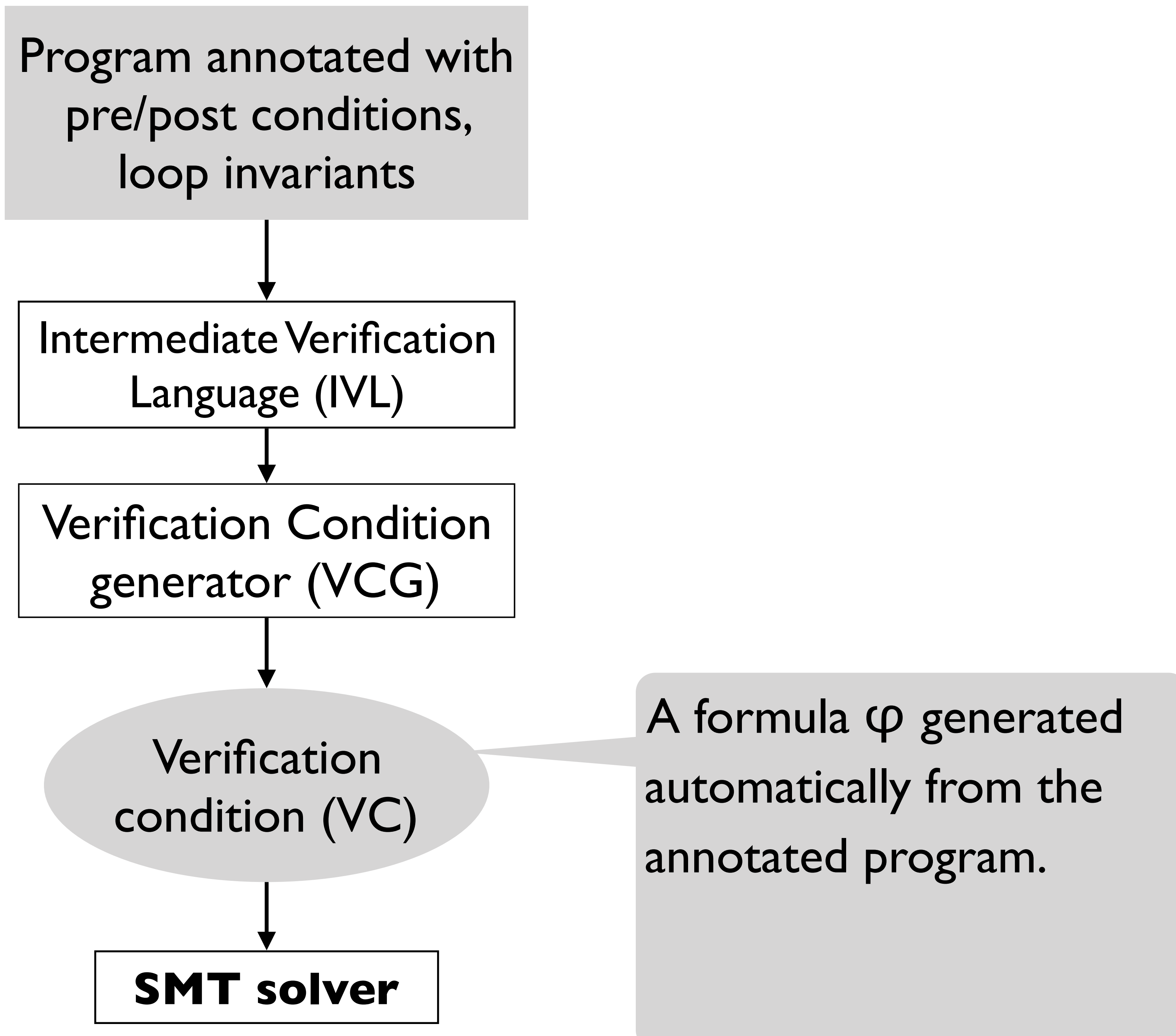
We can automate much of the proof process with **verification condition generation**!

But loop invariants still need to be provided...

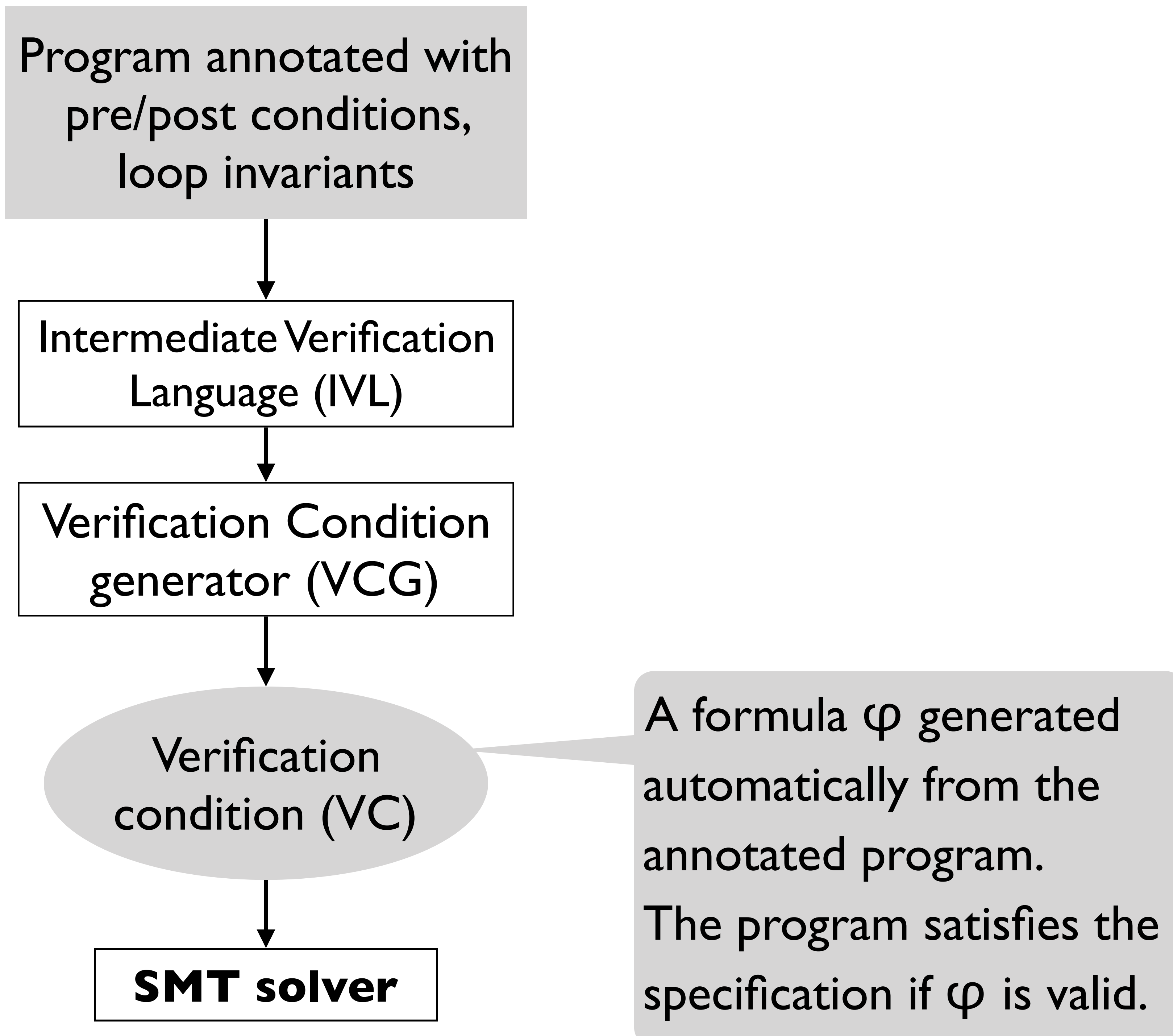
Automating Hoare Logic via VC generation



Automating Hoare Logic via VC generation



Automating Hoare Logic via VC generation



Automating Hoare Logic via VC generation

Program annotated with pre/post conditions, loop invariants

↓

Intermediate Verification Language (IVL)

↓

Verification Condition generator (VCG)

↓

Verification condition (VC)

↓

SMT solver

Forwards computation:

A formula φ generated automatically from the annotated program.
The program satisfies the specification if φ is valid.

Automating Hoare Logic via VC generation

Program annotated with pre/post conditions, loop invariants

↓

Intermediate Verification Language (IVL)

↓

Verification Condition generator (VCG)

↓

Verification condition (VC)

↓

SMT solver

Forwards computation:

- Starting from the precondition, generate formulas to prove the postcondition.

A formula φ generated automatically from the annotated program.
The program satisfies the specification if φ is valid.

Automating Hoare Logic via VC generation

Program annotated with pre/post conditions, loop invariants

↓

Intermediate Verification Language (IVL)

↓

Verification Condition generator (VCG)

↓

Verification condition (VC)

↓

SMT solver

Forwards computation:

- Starting from the precondition, generate formulas to prove the postcondition.
- Based on computing strongest postconditions (sp).

A formula φ generated automatically from the annotated program.
The program satisfies the specification if φ is valid.

Automating Hoare Logic via VC generation

Program annotated with pre/post conditions, loop invariants

↓

Intermediate Verification Language (IVL)

↓

Verification Condition generator (VCG)

↓

Verification condition (VC)

↓

SMT solver

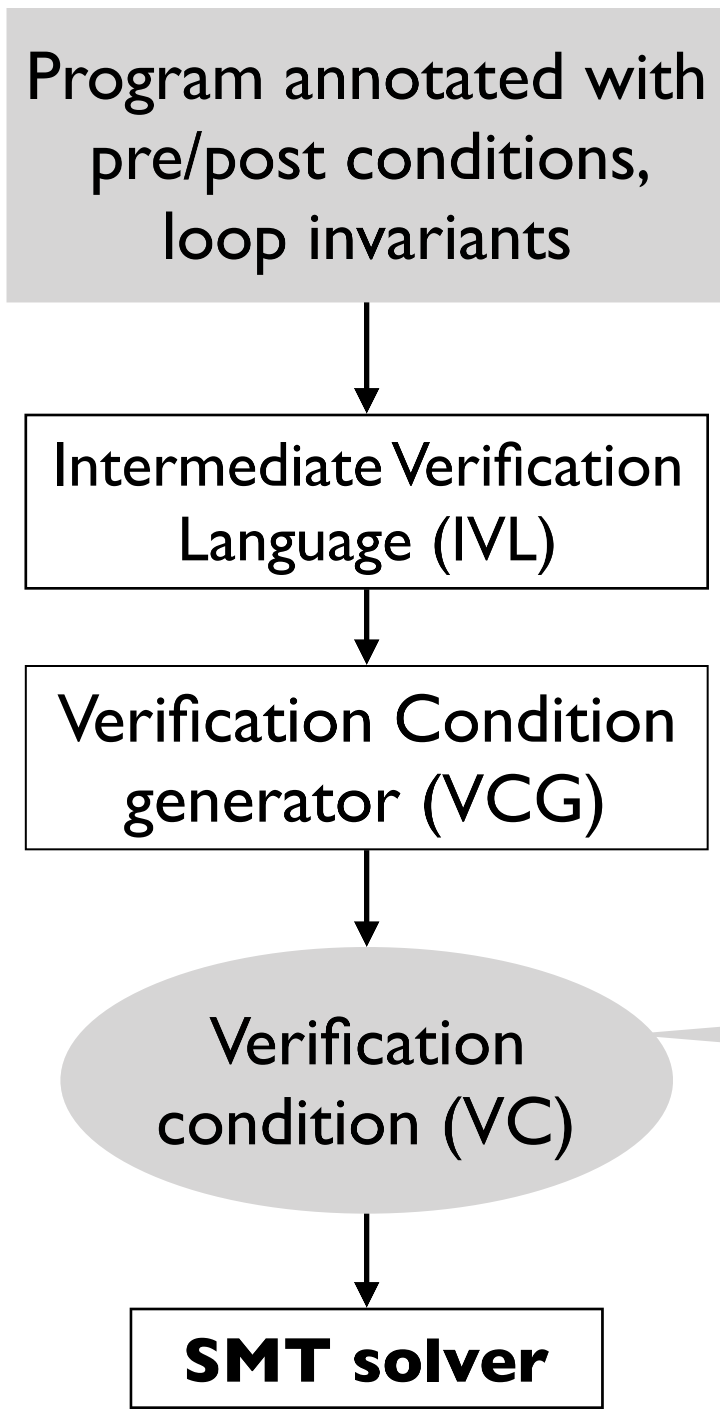
Forwards computation:

- Starting from the precondition, generate formulas to prove the postcondition.
- Based on computing strongest postconditions (sp).

Backwards computation:

A formula φ generated automatically from the annotated program.
The program satisfies the specification if φ is valid.

Automating Hoare Logic via VC generation



Forwards computation:

- Starting from the precondition, generate formulas to prove the postcondition.
- Based on computing strongest postconditions (sp).

Backwards computation:

- Starting from the postcondition, generate formulas to prove the precondition.

A formula φ generated automatically from the annotated program.
The program satisfies the specification if φ is valid.

Automating Hoare Logic via VC generation

Program annotated with pre/post conditions, loop invariants

↓

Intermediate Verification Language (IVL)

↓

Verification Condition generator (VCG)

↓

Verification condition (VC)

↓

SMT solver

Forwards computation:

- Starting from the precondition, generate formulas to prove the postcondition.
- Based on computing strongest postconditions (sp).

Backwards computation:

- Starting from the postcondition, generate formulas to prove the precondition.
- Based on computing weakest preconditions (wp).

A formula φ generated automatically from the annotated program.
The program satisfies the specification if φ is valid.

VC generation with WP and SP

- **sp (S, P)**
  - The strongest predicate that holds for states produced by executing S on a state satisfying P.
- **wp (S, Q)**
  - The weakest predicate that guarantees Q will hold for states produced by executing S on a state satisfying that predicate.
- **{P} S {Q} is valid if**
  - $P \Rightarrow wp(S, Q)$ or $sp(S, P) \Rightarrow Q$

VC generation with WP and SP

- **sp (S, P)**
  - The strongest predicate that holds for states produced by executing S on a state satisfying P.

Symbolic execution, covered in next lecture, computes SPs for finite programs (no unbounded loops).

- **wp (S, Q)**
  - The weakest predicate that guarantees Q will hold for states produced by executing S on a state satisfying that predicate.
- **{P} S {Q} is valid if**
  - $P \Rightarrow wp(S, Q)$ or $sp(S, P) \Rightarrow Q$

VC generation with WP and SP

- **sp (S, P)**
  - The strongest predicate that holds for states produced by executing S on a state satisfying P.
- **wp (S, Q)**
  - The weakest predicate that guarantees Q will hold for states produced by executing S on a state satisfying that predicate.
- **{P} S {Q} is valid if**
  - $P \Rightarrow wp(S, Q)$ or $sp(S, P) \Rightarrow Q$

Symbolic execution, covered in next lecture, computes SPs for finite programs (no unbounded loops).

VC generation with WP and SP

- **sp (S, P)**
  - The strongest predicate that holds for states produced by executing S on a state satisfying P.
- **wp (S, Q)**
  - The weakest predicate that guarantees Q will hold for states produced by executing S on a state satisfying that predicate.
- **{P} S {Q} is valid if**
  - $P \Rightarrow wp(S, Q)$ or $sp(S, P) \Rightarrow Q$

Symbolic execution, covered in next lecture, computes SPs for finite programs (no unbounded loops).

Today, we'll see how to compute weakest preconditions (WP) for IMP. This lets us verify partial correctness properties.

VC generation with WP and SP

- **sp (S, P)**
  - The strongest predicate that holds for states produced by executing S on a state satisfying P.
- **wp (S, Q)**
  - The weakest predicate that guarantees Q will hold for states produced by executing S on a state satisfying that predicate.
- **{P} S {Q} is valid if**
  - $P \Rightarrow wp(S, Q)$ or $sp(S, P) \Rightarrow Q$

Symbolic execution, covered in next lecture, computes SPs for finite programs (no unbounded loops).

Today, we'll see how to compute weakest preconditions (WP) for IMP. This lets us verify partial correctness properties.

VC generation with WP and SP

- **sp (S, P)**
  - The strongest predicate that holds for states produced by executing S on a state satisfying P.
- **wp (S, Q)**
  - The weakest predicate that guarantees Q will hold for states produced by executing S on a state satisfying that predicate.
- **{P} S {Q} is valid if**
  - $P \Rightarrow wp(S, Q)$ or $sp(S, P) \Rightarrow Q$

Symbolic execution, covered in next lecture, computes SPs for finite programs (no unbounded loops).

Today, we'll see how to compute weakest preconditions (WP) for IMP. This lets us verify partial correctness properties.

VC generation with WP

wp (S, Q)

- wp(skip, Q) = Q
- wp(**abort**, Q) = true
- wp(**assert** C,Q) = C ∧ Q
- wp(**assume** C,Q) = C → Q
- wp(**havoc** x,Q) = ∀ x .Q
- wp(x := E, Q) = Q[E / x]
- wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))
- wp(**if** C **then** S_1 **else** S_2,Q) = (C → wp(S_1,Q)) ∧ (¬C → wp(S_2,Q))
- wp(**while** C {I} **do** S, Q) = ?

VC generation with WP

wp (S, Q)

- wp(skip, Q) = Q
- wp(**abort**, Q) = true
- wp(**assert** C,Q) = C $\wedge$ Q
- wp(**assume** C,Q) = C $\rightarrow$ Q
- wp(**havoc** x,Q) = $\forall$ x .Q
- wp(x := E, Q) = Q[E / x]
- wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))
- wp(**if** C **then** S_1 **else** S_2,Q) = (C $\rightarrow$ wp(S_1,Q)) $\wedge$ ($\neg$C $\rightarrow$ wp(S_2,Q))
- wp(**while** C {I} **do** S, Q) = ?

What about loops?

VC generation with WP

wp (S, Q)

- wp(skip, Q) = Q
- wp(**abort**, Q) = true
- wp(**assert** C,Q) = C $\wedge$ Q
- wp(**assume** C,Q) = C $\rightarrow$ Q
- wp(**havoc** x,Q) = $\forall$ x .Q
- wp(x := E, Q) = Q[E / x]
- wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))
- wp(**if** C **then** S_1 **else** S_2,Q) = (C $\rightarrow$ wp(S_1,Q)) $\wedge$ ($\neg$C $\rightarrow$ wp(S_2,Q))
- wp(**while** C {I} **do** S, Q) = ?

What about loops?

VC generation for loops

VC generation for loops

- VC(x := E,Q) = true

VC generation for loops

- $VC(x := E, Q) = true$
- $VC(S_1; S_2, Q) = VC(S_2, Q) \wedge VC(S_1, awp(S_2, Q))$

VC generation for loops

- $VC(x := E, Q) = true$
- $VC(S_1;S_2,Q)=VC(S_2,Q) \wedge VC(S_1,awp(S_2,Q))$
- $VC(\text{if } C \text{ then } S_1 \text{ else } S_2, Q)=VC(S_1,Q) \wedge VC(S_2,Q)$

VC generation for loops

- $VC(x := E, Q) = true$
- $VC(S_1;S_2,Q) = VC(S_2,Q) \wedge VC(S_1, awp(S_2,Q))$
- $VC(\text{if } C \text{ then } S_1 \text{ else } S_2, Q) = VC(S_1,Q) \wedge VC(S_2,Q)$
- To show I is preserved in loop, need:
  - $I \wedge C \Rightarrow awp(S,I) \wedge VC(S,I)$

VC generation for loops

- $VC(x := E, Q) = true$
- $VC(S_1;S_2,Q) = VC(S_2,Q) \wedge VC(S_1, awp(S_2,Q))$
- $VC(\text{if } C \text{ then } S_1 \text{ else } S_2, Q) = VC(S_1,Q) \wedge VC(S_2,Q)$
- To show I is preserved in loop, need:
  - $I \wedge C \Rightarrow awp(S,I) \wedge VC(S,I)$
- To show I is strong enough to establish Q, need:
  - $I \wedge \neg C \Rightarrow Q$

VC generation for loops

- $VC(x := E,Q) = \text{true}$
- $VC(S_1;S_2,Q) = VC(S_2,Q) \wedge VC(S_1,awp(S_2,Q))$
- $VC(\text{if } C \text{ then } S_1 \text{ else } S_2, Q) = VC(S_1,Q) \wedge VC(S_2,Q)$
- To show I is preserved in loop, need:
  - $I \wedge C \Rightarrow awp(S,I) \wedge VC(S,I)$
- To show I is strong enough to establish Q, need:
  - $I \wedge \neg C \Rightarrow Q$
- Putting this together, verification condition for a while loop

VC generation for loops

- $VC(x := E, Q) = true$
- $VC(S_1;S_2,Q)=VC(S_2,Q) \wedge VC(S_1,awp(S_2,Q))$
- $VC(\text{if } C \text{ then } S_1 \text{ else } S_2, Q)= VC(S_1,Q) \wedge VC(S_2,Q)$
- To show I is preserved in loop, need:
  - $I \wedge C \Rightarrow awp(S,I) \wedge VC(S,I)$
- To show I is strong enough to establish Q, need:
  - $I \wedge \neg C \Rightarrow Q$
- Putting this together, verification condition for a while loop

$S' = \text{while } C \text{ do}\{I\}\ S$ is:

  - $VC(S',Q) = (I \wedge C \Rightarrow awp(S,I) \wedge VC(S,I)) \wedge (I \wedge \neg C \Rightarrow Q)$

Verifying a Hoare triple

Verifying a Hoare triple

Theorem: {P} S {Q} is valid if the following formula is valid:

$$P \rightarrow wp(S_{IVL}, Q)$$

TODOs by next lecture

- No class on Monday
- The last reading review will be out
- Start to work on your final report/project! (40%)