Computer-Aided Reasoning for Software

CSE507

Reasoning about Programs

courses.cs.washington.edu/courses/cse507/14au/

Emina Torlak
emina@cs.washington.edu

Today

Today

Last lecture

- Finite model finding for first-order logic with quantifiers, relations, and transitive closure

Today

Last lecture

- Finite model finding for first-order logic with quantifiers, relations, and transitive closure

Today

- Reasoning about (partial) correctness of programs
  - Hoare Logic
  - Verification Condition Generation

Today

Last lecture

- Finite model finding for first-order logic with quantifiers, relations, and transitive closure

Today

- Reasoning about (partial) correctness of programs
  - Hoare Logic
  - Verification Condition Generation

Based on lectures by Isil Dillig, Daniel Jackson, and Viktor Kuncak

Program verification & checking (L10–L15)

Program verification & checking (L10–L15)

Classic verification (L10, L11)

- Checking that all (terminating) executions satisfy an FOL property on all inputs

Program verification & checking (L10–L15)

Classic verification (L10, L11)

- Checking that all (terminating) executions satisfy an FOL property on all inputs

Bounded verification (L12)

- Scope-complete checking of FOL properties

Program verification & checking (L10–L15)

Classic verification (L10, L11)

- Checking that all (terminating) executions satisfy an FOL property on all inputs

Bounded verification (L12)

- Scope-complete checking of FOL properties

Symbolic execution (L13)

- Systematic checking of FOL properties

Program verification & checking (L10–L15)

Classic verification (L10, L11)

- Checking that all (terminating) executions satisfy an FOL property on all inputs

Bounded verification (L12)

- Scope-complete checking of FOL properties

Symbolic execution (L13)

- Systematic checking of FOL properties

Model checking (L14, L15)

- Exhaustive checking of temporal properties of abstracted programs

Program verification & checking (L10–L15)

Active research topic for 45 years

Classic verification (L10, L11)

- Checking that all (terminating) executions satisfy an FOL property on all inputs

Bounded verification (L12)

- Scope-complete checking of FOL properties

Symbolic execution (L13)

- Systematic checking of FOL properties

Model checking (L14, L15)

- Exhaustive checking of temporal properties of abstracted programs

Program verification & checking (L10–L15)

Classic verification (L10, L11)

- Checking that all (terminating) executions satisfy an FOL property on all inputs

Bounded verification (L12)

- Scope-complete checking of FOL properties

Symbolic execution (L13)

- Systematic checking of FOL properties

Model checking (L14, L15)

- Exhaustive checking of temporal properties of abstracted programs

Active research topic for 45 years

Classic ideas every computer scientist should know

Program verification & checking (L10–L15)

Classic verification (L10, L11)

- Checking that all (terminating) executions satisfy an FOL property on all inputs

Bounded verification (L12)

- Scope-complete checking of FOL properties

Symbolic execution (L13)

- Systematic checking of FOL properties

Model checking (L14, L15)

- Exhaustive checking of temporal properties of abstracted programs

Active research topic for 45 years

Classic ideas every computer scientist should know

Understanding the ideas can help you become a better programmer

Classic verification: seminal papers

Classic verification: seminal papers

1967: *Assigning Meaning to Programs* (Floyd)

Classic verification: seminal papers

1967: *Assigning Meaning to Programs* (Floyd)

1969: *An Axiomatic Basis for Computer Programming* (Hoare)

Classic verification: seminal papers

1967: *Assigning Meaning to Programs* (Floyd)

1969: *An Axiomatic Basis for Computer Programming* (Hoare)

1975: *Guarded Commands, Nondeterminacy and Formal Derivation of Programs* (Dijkstra)

Classic verification: seminal papers

1967: *Assigning Meaning to Programs* (Floyd)

- 1978 Turing Award

1969: *An Axiomatic Basis for Computer Programming* (Hoare)

- 1980 Turing Award

1975: *Guarded Commands, Nondeterminacy and Formal Derivation of Programs* (Dijkstra)

- 1972 Turing Award

Specifying correctness in Hoare logic

$$\{P\}\ S\ \{Q\}$$

Specifying correctness in Hoare logic

{P} S {Q}

Hoare triple

- S is a program statement (or fragment).
- P is an FOL formula called the *precondition*.
- Q is an FOL formula called the *postcondition*.

Specifying correctness in Hoare logic

{P} S {Q}

Hoare triple

- S is a program statement (or fragment).
- P is an FOL formula called the *precondition*.
- Q is an FOL formula called the *postcondition*.

Partial correctness (Hoare triple semantics)

- If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Specifying correctness in Hoare logic

Hoare triple

{P} S {Q}

- S is a program statement (or fragment).
- P is an FOL formula called the *precondition*.
- Q is an FOL formula called the *postcondition*.

Partial correctness (Hoare triple semantics)

- If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Total correctness

[P] S [Q]

- If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Specifying correctness in Hoare logic

Hoare triple

{P} S {Q}

safety

- S is a program statement (or fragment).
- P is an FOL formula called the *precondition*.
- Q is an FOL formula called the *postcondition*.

Partial correctness (Hoare triple semantics)

- If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

Total correctness

[P] S [Q]

- If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Specifying correctness in Hoare logic

Hoare triple

- S is a program statement (or fragment).
- P is an FOL formula called the ***precondition***.
- Q is an FOL formula called the ***postcondition***.

{P} S {Q}

safety

Partial correctness (Hoare triple semantics)

- If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

liveness

[P] S [Q]

Total correctness

- If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Specifying correctness in Hoare logic

Hoare triple

- S is a program statement (or fragment).
- P is an FOL formula called the *precondition*.
- Q is an FOL formula called the *postcondition*.

{P} S {Q}

safety

Partial correctness (Hoare triple semantics)

- If S executes from a state satisfying P, and if its execution terminates, then the resulting state satisfies Q.

liveness

[P] S [Q]

Total correctness

- If S executes from a state satisfying P, then its execution terminates and the resulting state satisfies Q.

Examples of Hoare triples

Examples of Hoare triples

{false} S {Q}

Examples of Hoare triples

{false} S {Q}

- Valid for all S and Q.

Examples of Hoare triples

{false} S {Q}

- Valid for all S and Q.

{P} while (true) do skip {Q}

Examples of Hoare triples

{false} S {Q}

- Valid for all S and Q.

{P} while (true) do skip {Q}

- Valid for all P and Q.

Examples of Hoare triples

{false} S {Q}

- Valid for all S and Q.

{P} while (true) do skip {Q}

- Valid for all P and Q.

{true} S {Q}

Examples of Hoare triples

{false} S {Q}

- Valid for all S and Q.

{P} while (true) do skip {Q}

- Valid for all P and Q.

{true} S {Q}

- If S terminates, then Q must hold.

Examples of Hoare triples

{false} S {Q}

- Valid for all S and Q.

{P} while (true) do skip {Q}

- Valid for all P and Q.

{true} S {Q}

- If S terminates, then Q must hold.

{P} S {true}

Examples of Hoare triples

{false} S {Q}

- Valid for all S and Q.

{P} while (true) do skip {Q}

- Valid for all P and Q.

{true} S {Q}

- If S terminates, then Q must hold.

{P} S {true}

- Valid for all P and S.

Proving partial correctness in Hoare logic

A simple imperative 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)
  - V := E (Assignment)
  - S_1; S_2 (Composition)
  - **if** C **then** S_1 **else** S_2 (If)
  - **while** C **do** S (While)

Proving partial correctness in Hoare logic

A simple imperative language

- **Expression** E
  - $Z \mid V \mid E_1 + E_2 \mid E_1 * E_2$
- **Conditional** C
  - $\text{true} \mid \text{false} \mid E_1 = E_2 \mid E_1 \leq E_2$
- **Statement** S
  - **skip** (Skip)
  - $V := E$ (Assignment)
  - $S_1; S_2$ (Composition)
  - **if** C **then** S_1 **else** S_2 (If)
  - **while** C **do** S (While)

One inference rule for every statement in the language:

$$\frac{\vdash\{P_1\}S_1\{Q_1\} \;\ldots\; \vdash\{P_n\}S_n\{Q_n\}}{\vdash\{P\}S\{Q\}}$$

If the Hoare triples $\{P_1\}$ $S_1\{Q_1\}$ … $\{P_n\}S_n\{Q_n\}$ are provable, then so is $\{P\}S\{Q\}$.

Inference rules for Hoare logic

$$\frac{}{\vdash \{P\}\ \textbf{skip}\ \{P\}}$$

Inference rules for Hoare logic

$$\frac{}{\vdash \{P\}\ \textbf{skip}\ \{P\}}$$

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

Inference rules for Hoare logic

$$\frac{}{\vdash \{P\}\ \textbf{skip}\ \{P\}}$$

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

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

Inference rules for Hoare logic

$$\dfrac{}{\vdash \{P\}\ \textbf{skip}\ \{P\}}$$

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

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

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

Inference rules for Hoare logic

$$\frac{}{\vdash \{P\}\ \textbf{skip}\ \{P\}}$$

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

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

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

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

Inference rules for Hoare logic

$$\frac{}{\vdash \{P\}\ \textbf{skip}\ \{P\}}$$

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

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

$$\frac{\vdash \{P \land C\}\ S_1\ \{Q\} \quad \vdash \{P \land \neg C\}\ S_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

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

$$\frac{\vdash \{P \land C\}\ S\ \{P\}}{\vdash \{P\}\ \textbf{while}\ C\ \textbf{do}\ S\ \{P \land \neg C\}}$$

loop invariant

Example: proof outline

```
{x ≤ n}
while (x < n) do
  {x ≤ n ∧ x < n}
  {x+1 ≤ n}                 // consequence
  x := x + 1
  {x ≤ n}                   // assignment
{x ≤ n ∧ x ≥ n}             // while
{x ≥ n}                     // consequence
```

Example: proof outline with auxiliary variables

$\{x = X \wedge y = Y\}$
$\{y = Y \wedge x = X\}$
t := x
$\{y = Y \wedge t = X\}$ // assignment
x := y
$\{x = Y \wedge t = X\}$ // assignment
y := t
$\{x = Y \wedge y = X\}$ // assignment

Soundness and relative completeness

Soundness and relative completeness

Proof rules for Hoare logic are sound

If $\vdash \{P\}\ S\ \{Q\}$ then $\models \{P\}\ S\ \{Q\}$

Soundness and relative completeness

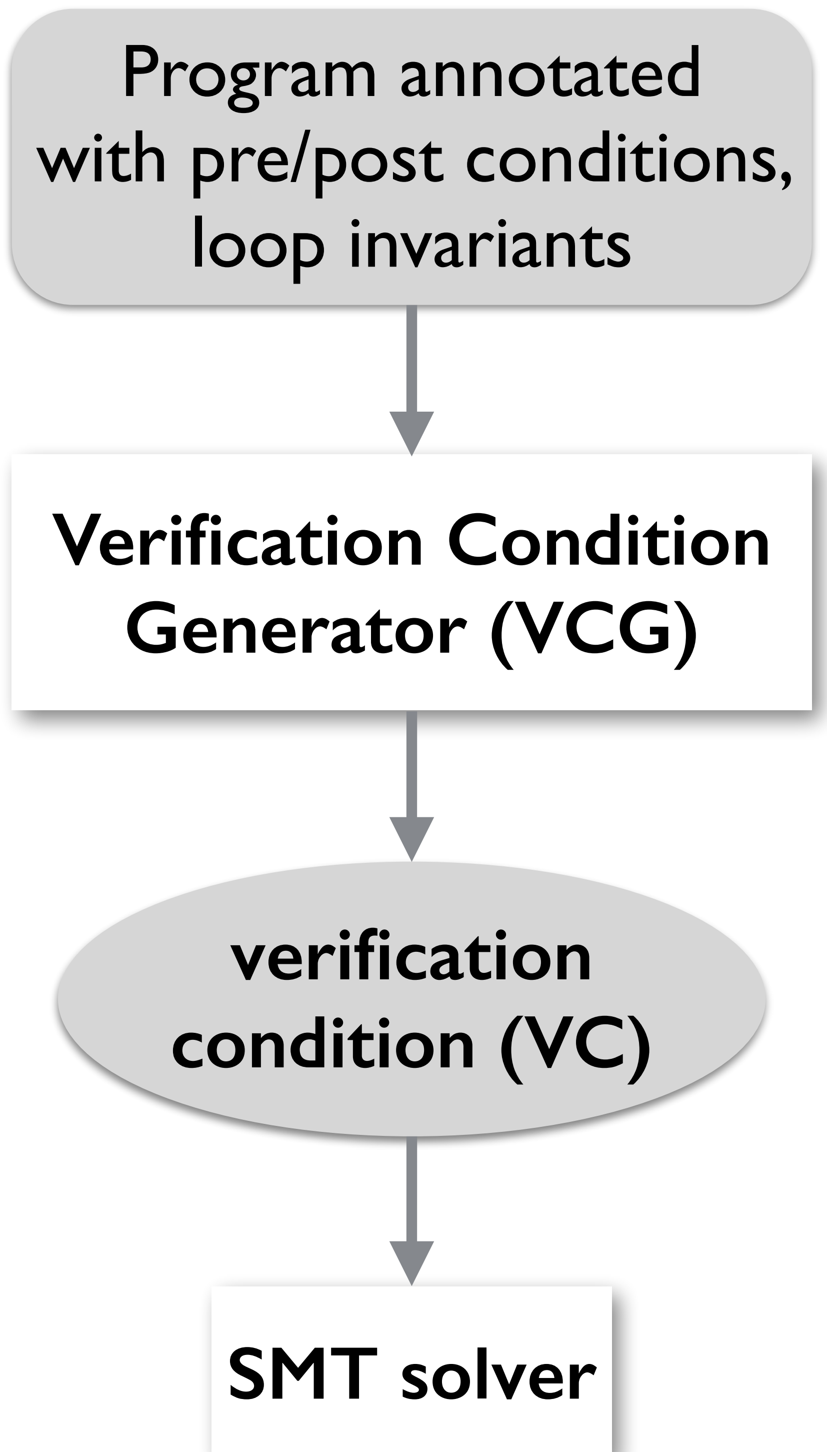
Proof rules for Hoare logic are sound

If $\vdash$ {P} S {Q} then $\models$ {P} S {Q}

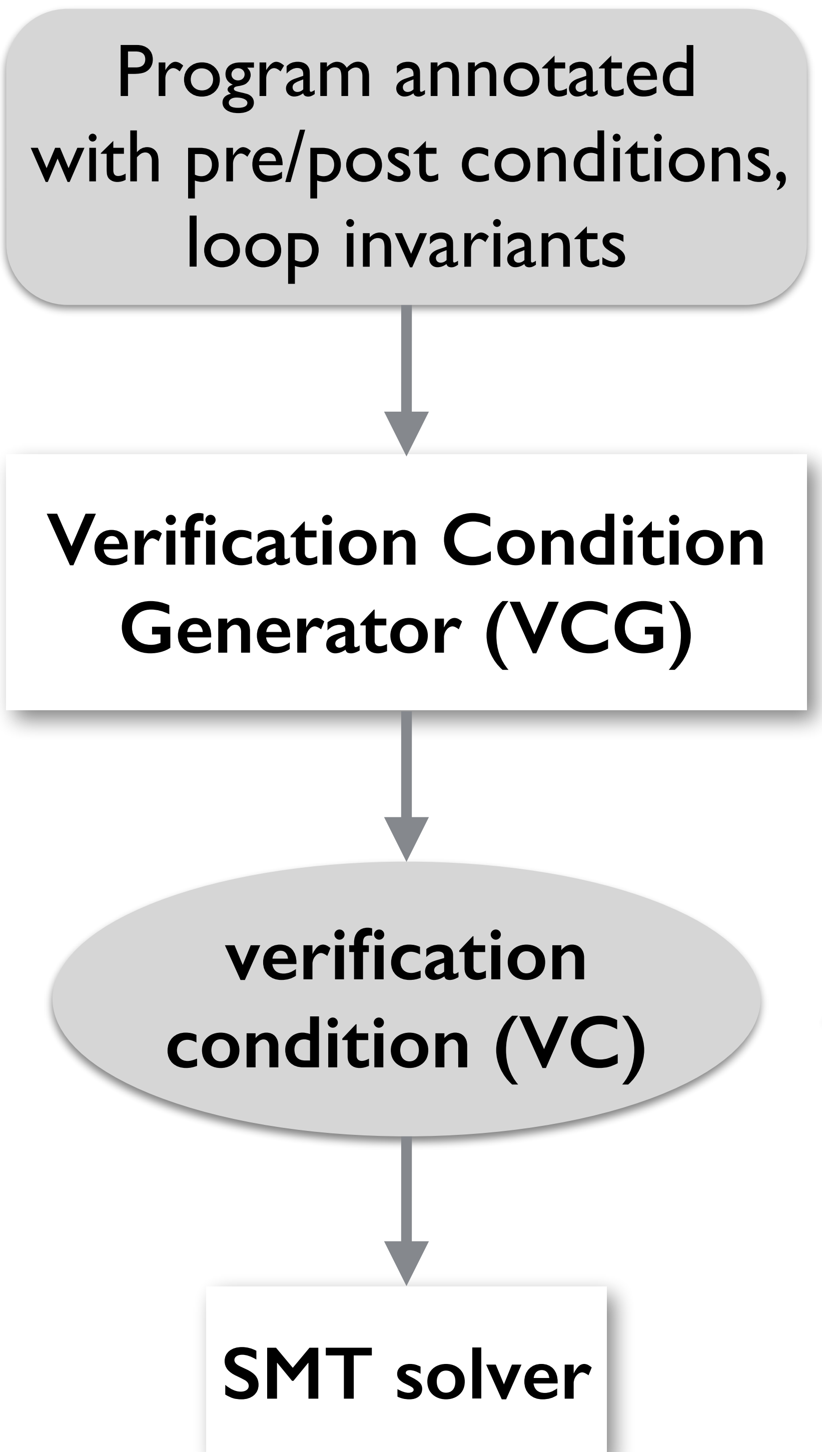
Proof rules for Hoare logic are relatively complete

If $\models$ {P} S {Q} then $\vdash$ {P} S {Q}, assuming an oracle for deciding implications

Automating Hoare logic with VC generation



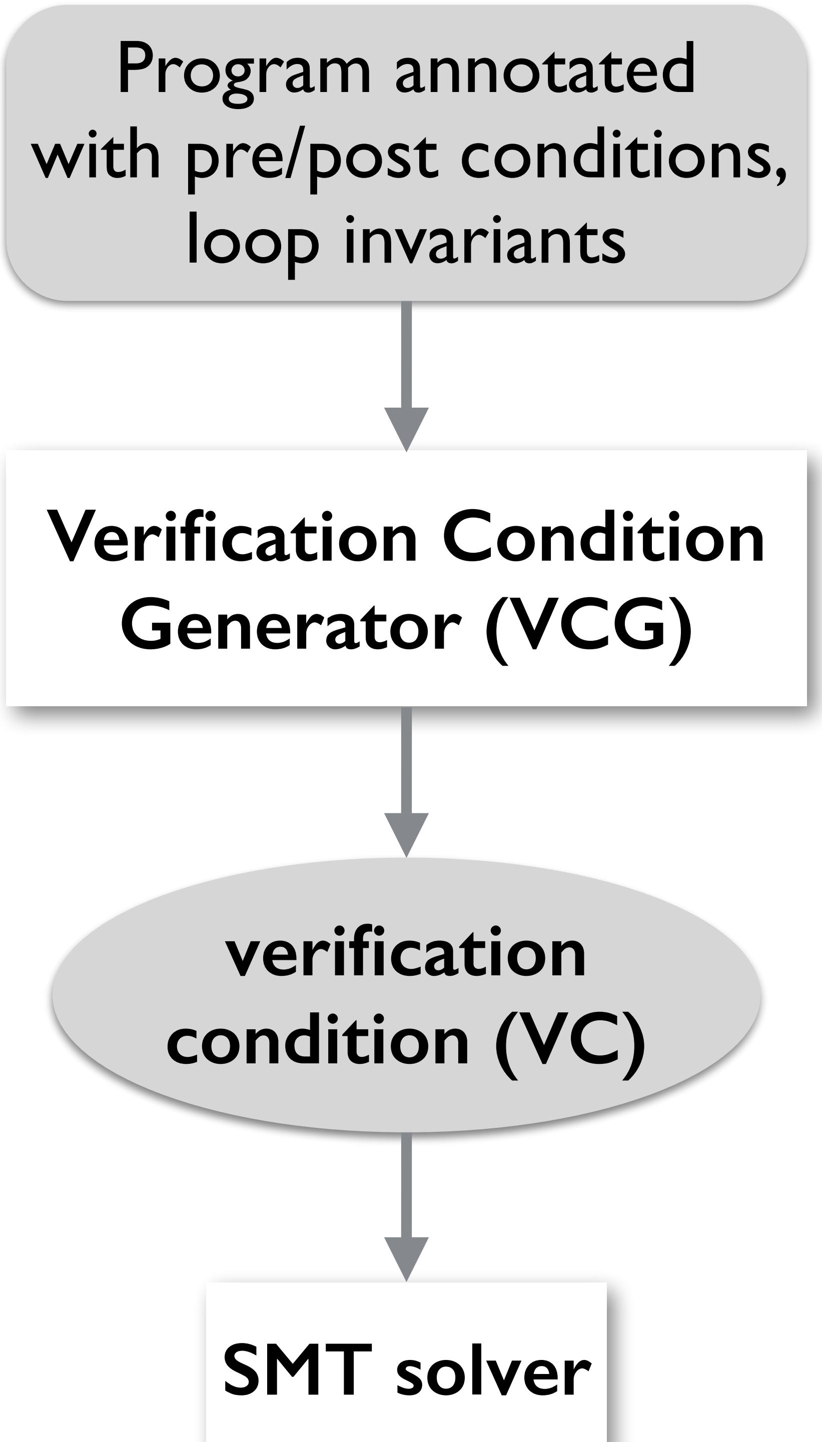
Automating Hoare logic with VC generation



A formula φ generated automatically from the annotated program.

The program satisfies the specification if φ is valid.

Automating Hoare logic with 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.
- Based on computing ***weakest liberal preconditions (wp)***.

VC generation with WP and SP

VC generation with WP and SP

wp(S, Q)

- The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.

VC generation with WP and SP

wp(S, Q)

- The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.

sp(S, P)

- The strongest predicate that holds after S is executed from a state satisfying P.

VC generation with WP and SP

wp(S, Q)

- The weakest predicate that guarantees Q will hold after executing S from a state satisfying that predicate.

sp(S, P)

- The strongest predicate that holds after S is executed from a state satisfying P.

{P} S {Q} is valid iff

- $P \Rightarrow wp(S, Q)$
- $sp(S, P) \Rightarrow Q$

Computing wp(S, Q)

Computing wp(S, Q)

wp(S, Q):

Computing wp(S, Q)

wp(S, Q):

- wp(**skip**, Q) = Q

Computing wp(S, Q)

wp(S, Q):

- wp(**skip**, Q) = Q
- wp(x := E, Q) = Q[E / x]

Computing wp(S, Q)

wp(S, Q):

- wp(**skip**, Q) = Q
- wp(x := E, Q) = Q[E / x]
- wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))

Computing wp(S, Q)

wp(S, Q):

- $wp(\textbf{skip}, Q) = Q$
- $wp(x := E, Q) = Q[E / x]$
- $wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$
- $wp(\textbf{if } C \textbf{ then } S_1 \textbf{ else } S_2, Q) = C \rightarrow wp(S_1, Q) \wedge \neg C \rightarrow wp(S_2, Q)$

Computing wp(S, Q)

wp(S, Q):

- wp(**skip**, Q) = 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 **do** S, Q) = ?

Computing wp(S, Q)

wp(S, Q):

- wp(**skip**, Q) = 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 **do** S, Q) = ✗

A fixpoint: cannot be expressed as a syntactic construction in terms of the postcondition.

Computing wp(S, Q)

wp(S, Q):

- $wp(\textbf{skip}, Q) = Q$
- $wp(x := E, Q) = Q[E / x]$
- $wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$
- $wp(\textbf{if } C \textbf{ then } S_1 \textbf{ else } S_2, Q) = C \rightarrow wp(S_1, Q) \wedge \neg C \rightarrow wp(S_2, Q)$
- $wp(\textbf{while } C \textbf{ do } S, Q) =$ ✗

Approximate wp(S, Q) with awp(S, Q).

Computing awp(S, Q)

awp(S, Q):

- awp(**skip**, Q) = Q
- awp(x := E, Q) = Q[E / x]
- awp(S_1; S_2, Q) = awp(S_1, awp(S_2, Q))
- awp(**if** C **then** S_1 **else** S_2, Q) = C → awp(S_1, Q) ∧ ¬C → awp(S_2, Q)
- awp(**while** C **do** {I} S, Q) = I

Computing awp(S, Q)

awp(S, Q):

- awp(**skip**, Q) = Q
- awp(x := E, Q) = Q[E / x]
- awp(S_1; S_2, Q) = awp(S_1, awp(S_2, Q))
- awp(**if** C **then** S_1 **else** S_2, Q) = C → awp(S_1, Q) ∧ ¬C → awp(S_2, Q)
- awp(**while** C **do** {I} S, Q) = I

Loop invariant provided by an oracle (e.g., programmer).

Computing awp(S, Q)

awp(S, Q):

- awp(**skip**, Q) = Q
- awp(x := E, Q) = Q[E / x]
- awp(S_1; S_2, Q) = awp(S_1, awp(S_2, Q))
- awp(**if** C **then** S_1 **else** S_2, Q) = C → awp(S_1, Q) ∧ ¬C → awp(S_2, Q)
- awp(**while** C **do** {I} S, Q) = I

For each statement S, also define VC(S,Q) that encodes additional conditions that must be checked.

Computing VC(S, Q)

Computing VC(S, Q)

VC(S, Q):

Computing VC(S, Q)

VC(S, Q):

- VC(**skip**, Q) = true

Computing VC(S, Q)

VC(S, Q):

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

Computing VC(S, Q)

VC(S, Q):

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

Computing VC(S, Q)

VC(S, Q):

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

Computing VC(S, Q)

VC(S, Q):

- VC(**skip**, Q) = true
- VC(x := E, Q) = true
- $VC(S_1; S_2, Q) = VC(S_2, Q) \wedge VC(S_1, awp(S_2, Q))$
- VC(**if** C **then** S_1 **else** S_2, Q) = $VC(S_1, Q) \wedge VC(S_2, Q)$
- VC(**while** C **do** {I} S, Q) = $(I \wedge C \Rightarrow awp(S,I) \wedge VC(S,I)) \wedge (I \wedge \neg C \Rightarrow Q)$

I is an invariant.

I is strong enough.

Verifying a Hoare triple

Theorem: {P} S {Q} is valid if

$$VC(S, Q) \wedge P \rightarrow awp(S, Q)$$

Verifying a Hoare triple

Theorem: {P} S {Q} is valid if

$$VC(S, Q) \wedge P \rightarrow awp(S, Q)$$

The other direction doesn't hold because loop invariants may not be strong enough or they may be incorrect.

Might get false alarms.

Summary

Today

- Reasoning about partial correctness of programs
  - Hoare Logic
  - VCG, WP, SP

Next lecture

- Guest lecture by Rustan Leino!
- Verification with Dafny, Boogie, and Z3.