Reasoning about Programs II

Emina Torlak
emina@cs.washington.edu
Overview

Last two lectures

• Reasoning about (partial) correctness with Hoare Logic
• Reasoning about total correctness with Dafny

Today

• Automating Hoare Logic with verification condition generation—how tools like Dafny work
Recap: Imperative Programming Language (IMP)

**Expression** $E$
- $Z | V | E_1 + E_2 | E_1 \times E_2$

**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)
Recap: inference rules for Hoare logic

\[\vdash \{P\} \text{skip} \{P\}\]

\[\vdash \{\text{true}\} \text{abort} \{\text{false}\}\]

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

\[\vdash \{P\} \text{if } C \text{ then } S_1 \text{ else } S_2 \{Q\}\]

\[\vdash \{P \land C\} \text{if } C \text{ then } S_1 \{Q\} \vdash \{P \land \neg C\} S_2 \{Q\}\]

\[\vdash \{P\} \text{while } C \text{ do } S \{P \land \neg C\}\]

loop invariant
Challenge: manual proof construction is tedious!

Hoare Logic proofs are highly manual:

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

{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
Challenge: manual proof construction is tedious!

\[
\{x \leq n\} \quad \text{// precondition}
\]

\begin{verbatim}
while (x < n) do
  \{x \leq n\} \quad \text{// loop invariant}
  x := x + 1
\end{verbatim}

\[
\{x = n\} \quad \text{// postcondition}
\]

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 with VC generation

Program annotated with pre/post conditions, loop invariants

Verification Condition Generator (VCG)

verification condition (VC)

SMT solver
Automating Hoare logic with VC generation

Program annotated with pre/post conditions, loop invariants

Verification Condition Generator (VCG)

A formula $\varphi$ generated automatically from the annotated program.
The program satisfies the specification if $\varphi$ is valid.
Automating Hoare logic with VC generation

Program annotated with pre/post conditions, loop invariants

Intermediate Verification Language (IVL)

Verification Condition Generator (VCG)

verification condition (VC)

SMT solver
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

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).
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.

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

Today, we’ll see how to compute weakest liberal preconditions (WP) for IMP.
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 liberal preconditions (WP) for IMP. This lets us verify partial correctness properties.
VC generation with WP

Program annotated with pre/post conditions, loop invariants

Intermediate Verification Language (IVL)

Verification Condition Generator (VCG)

verification condition (VC)

SMT solver
VC generation with WP

- Program annotated with pre/post conditions, loop invariants
- Intermediate Verification Language (IVL)
- Verification Condition Generator (VCG)
- Verification condition (VC)
- SMT solver
VC generation with WP: from IMP to IVL

Program annotated with pre/post conditions, loop invariants (IMP)

Intermediate Verification Language (IVL)

Verification Condition Generator (VCG)

verification condition (VC)

SMT solver

\[
E ::= \mathbb{Z} | V | E + E | E \cdot E
\]

\[
C ::= \text{true} | \text{false} | E = E | E \leq E
\]

\[
S ::= \text{skip} | \text{abort} | V := E | S; S |
\]

\[
\text{if } C \text{ then } S \text{ else } S |
\]

\[
\text{while } C \{!\} \text{ do } S
\]

\[
\{P\} S \{Q\}
\]
VC generation with WP: from IMP to IVL

Program annotated with pre/post conditions, loop invariants (IMP)

Intermediate Verification Language (IVL)

Verification Condition Generator (VCG)

verification condition (VC)

SMT solver

E ::= Z | V | E + E | E * E
C ::= true | false | E = E | E ≤ E
S ::= skip | abort | V := E | S; S |
    if C then S else S |
    while C {I} do S

E ::= Z | V | E + E | E * E
C ::= true | false | E = E | E ≤ E
S ::= skip | abort | V := E | S; S |
    if C then S else S |
    assert C | assume C | havoc V
VC generation with WP: loop-free code

E ::= Z | V | E + E | E * E
C ::= true | false | E = E | E ≤ E
S ::= skip | abort | V := E | S; S |
    if C then S else S |
    while C {I} do S

{P} S {Q}
VC generation with WP: loop-free code
VC generation with WP: loop-free code

wp(S, Q):
VC generation with WP: loop-free code

\[ wp(S, Q): \]
\[ \cdot \ wp(\text{skip}, Q) = Q \]
VC generation with WP: loop-free code

wp(S, Q):

• \( wp(\text{skip}, Q) = Q \)
• \( wp(\text{abort}, Q) = \text{true} \)
VC generation with WP: loop-free code

wp(S, Q):
  • wp(skip, Q) = Q
  • wp_abort, Q) = true
  • wp(x := E, Q) = Q[E / x]
VC generation with WP: loop-free code

wp(S, Q):

- \( wp(\text{skip}, Q) = Q \)
- \( wp(\text{abort}, Q) = \text{true} \)
- \( wp(x := E, Q) = Q[E / x] \)
- \( wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q)) \)
VC generation with WP: loop-free code

wp(S, Q):
• \(wp(\text{skip}, Q) = Q\)
• \(wp(\text{abort}, Q) = \text{true}\)
• \(wp(x := E, Q) = Q[E / x]\)
• \(wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))\)
• \(wp(\text{if } C \text{ then } S_1 \text{ else } S_2, Q) = (C \rightarrow wp(S_1, Q)) \land (\neg C \rightarrow wp(S_2, Q))\)
VC generation with WP: what about loops?

\[
wp(S, Q): \\
\begin{align*}
\text{wp(skip, } Q\text{) } &= Q \\
\text{wp(abort, } Q\text{) } &= \text{true} \\
\text{wp(x := E, } Q\text{) } &= Q[E / x] \\
\text{wp}(S_1; S_2, Q) &= \text{wp}(S_1, \text{wp}(S_2, Q)) \\
\text{wp(if } C \text{ then } S_1 \text{ else } S_2, Q) &= (C \rightarrow \text{wp}(S_1, Q)) \land (\neg C \rightarrow \text{wp}(S_2, Q)) \\
\text{wp(while } C \text{ {I} do } S, Q) &= ?
\end{align*}
\]
VC generation with WP: what about loops?

\( \text{wp}(S, Q): \)

- \( \text{wp}(\text{skip}, Q) = Q \)
- \( \text{wp}(\text{abort}, Q) = \text{true} \)
- \( \text{wp}(x := E, Q) = Q[E / x] \)
- \( \text{wp}(S_1; S_2, Q) = \text{wp}(S_1, \text{wp}(S_2, Q)) \)
- \( \text{wp}(\text{if } C \text{ then } S_1 \text{ else } S_2, Q) = (C \rightarrow \text{wp}(S_1, Q)) \land (\neg C \rightarrow \text{wp}(S_2, Q)) \)
- \( \text{wp}(\text{while } C \{I\} \text{ do } S, Q) = \chi \)

A fixpoint! In general, cannot be expressed as a syntactic construction in terms of the postcondition.
VC generation with WP: what about loops?

\[ wp(S, Q): \]

- \[ wp(\text{skip}, Q) = Q \]
- \[ wp(\text{abort}, Q) = \text{true} \]
- \[ wp(x := E, Q) = Q[E / x] \]
- \[ wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q)) \]
- \[ wp(\text{if } C \text{ then } S_1 \text{ else } S_2, Q) = (C \rightarrow wp(S_1, Q)) \land (\neg C \rightarrow wp(S_2, Q)) \]
- \[ wp(\text{while } C \{ I \} \text{ do } S, Q) = \chi \]

A fixpoint! In general, cannot be expressed as a syntactic construction in terms of the postcondition.

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.
VC generation with WP: what about loops?

while C {I} do S

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.
VC generation with WP: what about loops?

while C {l} do S

Cut the loop.

assert l;
havoc x; ... // for each loop target x
assume l;
if C then S; assert l; assume false;
else skip;

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.
VC generation with WP: what about loops?

while C {I} do S

Cut the loop.

assert I;

havoc x; ... // for each loop target x

assume I;

if C then S; assert I; assume false;
else skip;

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.
**VC generation with WP: what about loops?**

```plaintext
while C {I} do S

Cut the loop.

assert I;
havoc x; ... // for each loop target x
assume I;
if C then S; assert I; assume false;
else skip;
```

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.

$$wp(S, Q):$$

- $$wp(assert C, Q) = C \land Q$$
VC generation with WP: what about loops?

\[
\text{while } C \{I\} \text{ do } S
\]

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.

**wp(S, Q):**
- \( wp(\text{assert } C, Q) = C \land Q \)
- \( wp(\text{assume } C, Q) = C \rightarrow Q \)
VC generation with WP: what about loops?

while C {I} do S

Cut the loop.

assert I;
havoc x; ... // for each loop target x
assume I;
if C then S; assert I; assume false; else skip;

wp(S, Q):

• wp(assert C, Q) = C ∧ Q
• wp(assume C, Q) = C → Q
• wp(havoc x, Q) = ∀ x . Q

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.
VC generation with WP: putting it all together

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

1. Translate IMP to IVL by cutting loops.
2. Compute WP for IVL.
Verifying a Hoare triple

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

\[ P \rightarrow \text{wp}(S_{IVL}, Q) \]
Verifying a Hoare triple

**Theorem:** \( \{P\} S \{Q\} \) is valid if the following formula is valid

\[ P \rightarrow \text{wp}(S_{IVL}, Q) \]

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

• Automating Hoare Logic with VCG based on WPs

Next lecture

• Symbolic execution