Hoare Logic, Part II

CS560: Reasoning About Programs

Roopsha Samanta

Partly based on slides by Isil Dillig
Roadmap

Previously
- Hoare logic: Hoare triples, inference rules for partial correctness

Today
- (Semi-)automating Hoare logic
- Verification condition (VC) generation
- Predicate transformers
Hoare logic proofs can be tedious!

- What is a good loop invariant?
- What rule to apply when?

Let’s assume an oracle provides loop invariants for now and automate the rest!
Program + Specification = Verification Condition Generator

Verification condition (FOL formula) → Automatic Theorem prover (SMT Solver)

Valid ✓ Not valid ✗

Verification condition is a formula that is valid iff program is correct
Inductive assertion method

Verification condition is a formula that is valid iff program is partially correct

Verification condition (FOL formula)

Automatic Theorem prover (SMT Solver)

Valid ✓
Not valid ✗
Inductive assertion method

{pre}
x := 0
y := 1
while {I} (y <= n) do
  if (z == 0) then
    x := x + 1
  else
    x := x + y
  y := y + 1
{post}
Inductive assertion method

- Represent program as control-flow graph with annotations/inductive assertions

An annotation can be a precondition, postcondition, loop invariant or assertion.
Inductive assertion method

- Represent program as control-flow graph with annotations/inductive assertions
- Identify basic paths

- Basic path starts at a precondition/loop invariant, and ends at a postcondition/loop invariant/assertion.
- Loop invariants only at the start/end of basic paths
Inductive assertion method

- Represent program as control-flow graph with annotations/inductive assertions
- Identify basic paths
- For each basic path: check if corresponding Hoare triple is valid
Inductive assertion method

- Represent program as control-flow graph with annotations/inductive assertions
- Identify basic paths
- For each basic path: check if corresponding Hoare triple is valid

\[
\begin{align*}
\{\text{pre}\} x & := 0; y := 1 \{I\} \\
\{I\} & \text{ assume } y > n \{\text{post}\} \\
\{I\} & \text{ skip } \{I\} \\
\{I\} & \text{ assume } y \leq n; \text{ assume } z \neq 0; x := x + y; y := y + 1 \{I\} \\
\{I\} & \text{ assume } y \leq n; \text{ assume } z = 0; x := x + 1; y := y + 1 \{I\}
\end{align*}
\]
Generating VCs: Forwards vs. Backwards

Forwards Analysis
- Starts from precondition and tries to prove postcondition
- Computes strongest postconditions (sp)

Backwards Analysis
- Starts from postcondition and tries to prove precondition
- Computes weakest liberal preconditions (wp)

Predicate transformers: FOL x stmts → FOL
Incorporate effects of program statements into FOL formulas
WP and SP

\[ wp(S, Q) : \text{the weakest predicate that guarantees } Q \text{ will hold after executing } S \text{ from any state satisfying the predicate} \]

\[ sp(S, P) : \text{the strongest predicate that holds after executing } S \text{ from any state satisfying } P \]

\[ \{P\} S \{Q\} \text{ is valid iff:} \]

\[ P \Rightarrow wp(S, Q) , \text{ or, } sp(S, P) \Rightarrow Q \]

“largest” set of states

“smallest” set of states
Computing $sp(S, P)$

- $sp(\text{skip}, P) = P$
- $sp(\text{assume } C, P) = C \land P$
- $sp(S_1; S_2, P) = sp(S_2, sp(S_1, P))$
- $sp(x := E, P) = \exists x^0. x = E[x^0/x] \land P[x^0/x]$

$sp(x := x + 1, x > 0) = \exists x^0. x = x^0 + 1 \land x > 0$

$= x > 1$ (Quantifier Elimination)
Computing \( wp(S, Q) \)

- \( wp(\text{assume } C, Q) = C \rightarrow Q \)
- \( wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q)) \)
- \( wp(x := E, Q) = Q[E/x] \)

\[ + Q[E/x] \]

\[ x := e \{ Q \} \]
Roadmap

Today
- (Semi-)automating Hoare logic
- Verification condition (VC) generation
- Predicate transformers

Next
- How to fully automate Hoare logic: automatic invariant generation