Hoare style program verification

Abhik Roychoudhury
National University of Singapore

Remarks

- SW Model Checking
  - Automated
  - Reason about transition systems.
  - Abstractions can make the reasoning imprecise albeit conservative.
- Theorem Proving
  - User-guided
  - Reason about programs.
  - Exact reasoning.

Remarks

- SW Model Checking
  - Abstractions designed for a given program and/or property.
  - Path-sensitive.
    - False alarms, but because abstraction was coarse
  - Property required.
- Static Analysis
  - Abstract domain fixed for all programs in a PL, depending on the analysis performed.
  - Usually analysis results of diff paths are merged.
    - Hence false alarms
  - Abst. domain all imp.

Remarks

- The approach of developing proof rules for reasoning about language constructs is radically different from model checking
  - Reason about programs (not transition systems)
  - Non-mechanized.
  - Notion of distinguished control locations ingrained
    - Reason about pre- and post-conditions holding before and after execution of a block of code.
- Consider sequential programs in this lecture
  - Can extend to develop proof rules for multi-threaded programs.

Hoare triple

- {Pre} P {Post}
  - If P is run from a state where Pre holds and P terminates, then Post holds in the end-state [Partial correctness]
  - If P is run from a state where Pre holds, then P terminates and Post holds in the end-state [Total correctness]
  - A Hoare triple involving program P is a specification about P.

Trivial example

- Say P is while true do x = 0 endwhile
- P is partially correct w.r.t. any specification of the form {Pre} P {Post}
- P is not totally correct w.r.t. any specification of the form {Pre} P {Post}
- We will develop a proof system for reasoning about partial correctness
  - First step to reasoning about total correctness

Notations

- $|-_{par}$ {Pre} P {Post}
  - The Hoare triple can be shown to be partially correct in our proof system
- $|=_{par}$ {Pre} P {Post}
  - The Hoare triple is partially correct.
- $|-_{tot}$ {Pre} P {Post}, $|=_{tot}$ {Pre} P {Post}
  - Similar
  - Standard notions of soundness/completeness

IISc Summer Course 2007 by Abhik Roychoudhury

Factorial program

{ x ≥ 0 }	**{ x ≥ 0 }**
/* x is input */	/* x is input */
y = 1; z = 0;	y = 1;
while (z != x) do	while (x != 0) do
z = z + 1;	y = y *x;
y = y *z;	x = x – 1;
endwhile	endwhile
{ y = x! }	**{ ??? }**

IISc Summer Course 2007 by Abhik Roychoudhury

The problem

- x was destructively updated in Program2
  - In the end-state, we cannot say y = x!
  - To state correctness conditions, not enough to use program variables
  - Need to remember the original value of x
  - **{x =x0/\ x≥0}** Program2 **{y = x0!}**
  - **x0 is a universally quantified logical variable.**

IISc Summer Course 2007 by Abhik Roychoudhury

Logical variables

- **{x =x0 ∧ x≥0}** Program2 **{y = x0!}**
  - For all x0, if x = x0 and x ≥ 0 and we run Program2 such that it terminates, we will have y = x0! in the end state.
  - These variables appear only in the logical formulae of pre- and post-conditions.
  - Never appear in the program being verified.
- We now present the proof rules of our proof system.

IISc Summer Course 2007 by Abhik Roychoudhury

Proof Rules

$$\frac{\text{Premises}}{\text{Conclusion}}$$

Both premises and conclusion are Hoare triples.

If premises specify properties about programs C1, C2, …, Cn

-- the conclusion specifies a property about a bigger program C typically containing C1,C2,…,Cn

IISc Summer Course 2007 by Abhik Roychoudhury

Rule for Assignment

$$\frac{}{\{\psi[x \to E]\}\ x = E\ \{\psi\}}$$

No premises in this rule.

To prove ψ after the assignment, **ψ [x → E]** should hold before the assignment.

IISc Summer Course 2007 by Abhik Roychoudhury

Why not forwards ?

- $\{\varphi\}$ x = E $\{\psi\}$
  - How to define ψ in terms of φ ?
  - Cannot be achieved mechanically in general
  - The backwards formulation of the rule allows deducing Hoare triple by mechanically substituting x.
  - Instead define φ in terms of ψ

Sequential Composition

$$\frac{\{\varphi\}\ C1\ \{\psi1\} \qquad \{\psi1\}\ C2\ \{\psi\}}{\{\varphi\}\ C1\ ;\ C2\ \{\psi\}}$$

Need assertion for end-state of C1 and begin-state of C2.

If-statement

$$\frac{\{\varphi \wedge b\}\ C1\ \{\psi\} \qquad \{\varphi \wedge \neg b\}\ C2\ \{\psi\}}{\{\varphi\}\ \text{if } b\ \text{ then } C1\ \text{else } C2\ \{\psi\}}$$

Involves a case-split.

Pre-condition typically does not say anything about b

Needs to augmented with truth/falsehood of b.

While statement

$$\frac{\{\psi \wedge b\}\ C\ \{\psi\}}{\{\psi\}\ \text{while } b\ \text{do } c\ \{\psi \wedge \neg b\}}$$

ψ is the loop invariant.

Rule for partial correctness (number of times the loop executes/ termination is not known/ not guaranteed).

Implications

$$\frac{\varphi' \Rightarrow \varphi \qquad \{\varphi\}\ C\ \{\psi\} \qquad \psi \Rightarrow \psi'}{\{\varphi'\}\ C\ \{\psi'\}}$$

1. Strengthening the pre-condition
2. Weakening the post-condition

Why do we need this rule ?

Example 1

- $\{y < 2\}$ y = y +1 $\{y < 5\}$
- **Proof:**
- $\{y < 2\}$
- $\{y +1 < 5\}$ *implication rule*
- y = y+1
- $\{y < 5\}$ *assignment rule*

Example 2

- {true} z=x; z = z+y; u = z {u = x +y}
- Proof:
  - {true}
  - {x+y = x + y}
  - z = x;
  - {z+y = x +y}
  - z = z + y;
  - {z = x + y}
  - u = z
  - { u = x + y}

Push up the assertions starting from the post-condition of the code fragment being verified.

Example 3

- {true} if (x>y) z = y else z = x {z = min(x,y)}
- **Proof:**
- {x = min(x,y)} z = x { z = min(x,y)}
- {y = min(x,y)} z = y { z = min(x,y)}
- How to combine these triples using the rule for if-statements in our proof system ?
- Use the rule of implications

Example 3

- $x \leq y \Rightarrow x = \min(x,y)$
- $x > y \Rightarrow y = \min(x,y)$
- {true /\ x > y} z = y { z = min(x,y)}
- {true /\ ¬(x>y)} z = x { z = min(x,y)}
- Combine using the if-rule
- {true} if (x>y) z=y else z=x { z =min(x,y)}

Reasoning about loops

- To prove {φ} while b do c {ψ}
- We must
  - Find a loop invariant η i.e. {η /\ b} c {η}
  - this means
    - {η} while b do c {η /\ ¬b}
  - Show that $\varphi \Rightarrow \eta$
  - Show that $(\eta \wedge \neg b) \Rightarrow \psi$
  - Use rule of impl. to prove {φ} while b do c {ψ}

Loop invariant

- Synthesis involves human ingenuity
  - No unique inv. for a given loop
  - The formula True is an invariant for any loop
  - But our inv. η should satisfy
    - $\varphi \Rightarrow \eta$
    - $(\eta \wedge \neg b) \Rightarrow \psi$
  - Usually choose invariants which capture relationships between variables whose values are modified at each iteration.

Guessing invariants

{φ} **Pre-condition**
{??}
While b do
{η /\ b} **Loop invariant**
C
{η }
endwhile
{ψ}

Need to guess η and verify that it is an invariant

Verifying invariants

{φ} **Implication**
{η}
while b do
{η ∧ b} Implication
{η1}
C
{η} Push up based on structure of C
endwhile
{η ∧ ¬b} **Implication**
{ψ}

1. $\eta \wedge \neg b \Rightarrow \psi$
2. $\eta \wedge b \Rightarrow \eta 1$
3. $\varphi \Rightarrow \eta$

{ φ } while b do C { Ψ }
Impl.
{η} while b do C {η ∧ ¬b}
while rule
{η ∧ b} C {η}
φ ⇒ η
η ∧ ¬b ⇒ ψ

Factorial program

- {true}
- y = 1; z = 0;
- {y = z!}
- while (z != x) do
-     z = z + 1; y = y *z
- endwhile
- {y = x!}

Guess the loop invariant
y = z!

Checking the post-loop states

{true}
y= 1; z = 0;
{y = 0!}
while (z != x) do
    z = z +1; y = y*z;
endwhile
{y = z! ∧ ¬(z ≠x)} ***Implication***
{y = x!}

y = z! ∧ ¬(z ≠x)
≡ y = z! ∧ z = x
≡ y = x!

Verifying the invariant

{true}
y= 1; z = 0;
{y = 0!}
while (z != x) do
    { y = z! ∧ z ≠ x}
    z = z +1;
    y = y*z;
    { y = z! }
endwhile

Need to prove this Hoare triple

Verifying the invariant

{true}
y= 1; z = 0;
{y = 0!}
while (z != x) do
    { y = z! ∧ z ≠ x} ***Implication : Easy to check***
    { y*(z+1) = (z+1)! }
    z = z +1; ***Assignment***
    { y*z = z ! }
    y = y*z; ***Assignment***
    { y = z! }
endwhile

So-far

....

{ y = z! }

while (z ≠ x) do

{ y = z! /\ z ≠ x }

z = z +1 ; y = y*z

{ y = z!}

endwhile

{ y = x! }

Does y = z! hold before the while loop ?

Checking pre-loop states

{true}

y = 1;

z = 0;

{ y = z !}

while (z ! = x) do

.....

Need to prove this Hoare triple

Checking pre-loop states

{true} ***Implication***

{ 1 = 0! }

y = 1; ***Assignment***

{ y = 0! }

z = 0; ***Assignment***

{ y = z!}

Proof structure

- {true} Factorial-Program { y = x!}
  - 1. To prove y = x! at the end of the loop, we first guess a loop invariant
    - y = z! is our choice
  - 2. Can the choice of loop invariant in step 1 ensure y = x! at the end of the loop ?
    - Yes
  - 3. Verify that y = z! is indeed a loop invariant
    - This is the premise of the while rule

Proof structure

  - 4. Verify that the loop invariant holds before the loop.
    - Checking pre-loop states
- Steps 3 and 4 constitute a proof of the invariant by induction on # iterations
  - Step 3: induction step
  - Step 4: base case of the proof
  - The loop invariant itself is the ind. Hypothesis, no strengthening involved in this proof.

Choice of Loop Invariant

- The loop invariant must be strong enough to be "proved" an invariant.
  - The while rule is essentially accomplishing induction on # of loop iterations.
  - Often guided by the choice of the post-condition after the loop
    - Our post-condition was y = x!
    - Since z = x at loop exit and z is modified at every loop iteration, choose y = z! as invariant.

Proving total correctness

- Our proof system only shows partial correctness of triples $\{\varphi\}$ P $\{\psi\}$
- To prove total correctness
  - Need to prove termination
  - Only the proof rule for while statement needs to change.
  - To prove termination
    - Find a non-negative integer quantity which decreases in every iteration (call it variant)

Finding variant

- a = x; y = 1;
- while (a > 0) do
-     y = y*a; a = a-1;
- endwhile

- Trivial to find the variant
  - a   in this case

Finding variant

- y= 1; z = 0;
- while (z != x) do
-     z = z + 1; y = y*z
- endwhile
-
- Variant is x – z (lifted from loop guard here)
- In general, finding variant cannot be automated even if the loop is guaranteed to terminate.

New Proof Rule

$$\frac{\{\eta \wedge b \wedge (E = E0 \geq 0)\}\ \ C\ \ \{\eta \wedge (E0 > E \geq 0)\}}{\{\eta \wedge E \geq 0\}\ \text{while b do c}\ \{\eta \wedge \neg b\}}$$

E is the variant.

If it is E0 before the loop, it strictly decreases but remains non-negative.

Of course E should be non-negative before the loop starts.

Factorial program

- { x ≥ 0}
- y = 1; z = 0;
- while (z != x) do
-     z = z + 1;
-     y = y*z;
- endwhile
- { y = x! }

Use the variant x – z to prove termination

Use the loop invariant y =z! as before for proving partial correctness

Reasoning about the loop

$\{ y = z! \wedge x - z \geq 0 \}$

while (x ! = z) do

  z = z +1 ; y = y * z;

endwhile;

$\{ y = z! \wedge x = z\}$

{y = x!}

From the conclusion of the while rule (total correctness)

-- How to show the premise ?

Reasoning about an iteration

{ y = z! ∧ x - z ≥ 0 }
while (x ! = z) do
{ y = z! ∧ x ≠ z ∧ (x – z = E0 ≥ 0) }
z = z +1 ; y = y * z;
{ y = z! ∧ (E0 > x – z ≥ 0) }
endwhile;
{ y = z! ∧ x = z}
{y = x!}

This triple is the premise of the while rule

IISc Summer Course 2007 by Abhik Roychoudhury

Reasoning about an iteration

{ y = z! ∧ x - z ≥ 0 }
while (x ! = z) do
{ y = z! ∧ x ≠ z ∧ E0 = x – z ≥ 0}
{ y*(z+1) = (z+1)! ∧ (E0 > x – (z+1) ≥ 0) }
z = z +1 ;
{ y*z = z! ∧ (E0 > x – z ≥ 0) }
y = y * z;
{ y = z! ∧ (E0 > x – z ≥ 0) }
endwhile;
{y = x!}

Implication: Check it !

IISc Summer Course 2007 by Abhik Roychoudhury

Reasoning about pre-loop states

{ x ≥ 0 }
y = 1; z = 0 ;
{ y = z! ∧ x - z ≥ 0 }
while (x ! = z) do
{ y = z! ∧ x ≠ z ∧ E0 = x – z ≥ 0}
z = z +1 ; y = y * z;
{ y = z! ∧ (E0 > x – z ≥ 0) }
endwhile;
{y = x!}

Need to prove this Hoare triple

IISc Summer Course 2007 by Abhik Roychoudhury

Reasoning about pre-loop states

{ x ≥ 0 }
{ 1 = 0! ∧ x ≥ 0 }
y = 1;
{ y = 0! ∧ x - 0 ≥ 0 }
z = 0 ;
{ y = z! ∧ x - z ≥ 0 }
while (x ! = z) do
z = z +1 ; y = y * z;
endwhile;
{y = x!}

Implication

x ≥ 0 must hold at the initial state of the program for the program to terminate.

This fact is used here in the overall proof.

IISc Summer Course 2007 by Abhik Roychoudhury

Finally, Program Refinement

- {φ} Prog {ψ}
  - Infers properties about pre- and post- states of a program
  - In the flavor of program verification
- Instead, you can treat (φ, ψ) as a specification
  - Correct by construction program synthesis from given pre- and post- conditions.
  - Many choices of implementation possible !
    - Use the specifications to guide the implementation instead of checking the implementation against the specification post-mortem.

IISc Summer Course 2007 by Abhik Roychoudhury

Reading Material

- Chapter 4 of
  - *Logic in Computer Science*
  - By Michael R. A. Huth and Mark D. Ryan
- Additional optional reading
  - See Lesson Plan in course web-page
    - http://drona.csa.iisc.ernet.in/~abhik

IISc Summer Course 2007 by Abhik Roychoudhury