Software Analysis

Hoare Logic: Proving Programs Correct

Presenter: Jonathan Aldrich

Session Objectives

- Motivate the study of formal verification
- Formalize specifications using pre- and post-conditions
- Apply Hoare Logic techniques to prove that implementations meet their specification

Testing and Proofs

- Testing
  - Observable properties
  - Verify program for one execution
  - Manual development with automated regression
  - Most practical approach now
- Proofs
  - Any program property
  - Verify program for all executions
  - Manual development with automated proof checkers
  - May be practical for small programs in the future
- So why study proofs if they aren't (yet) practical?
  - Proofs tell us how to ***think*** about program correctness
    - Important for development, inspection
  - Foundation for static analysis tools
    - These are just simple, automated theorem provers
    - Many are practical today!

How would you argue that this program is correct?

```
float sum(float *array, int length) {
    float sum = 0.0;
    int i = 0;
    while (i < length) {
        sum = sum + array[i];
        i = i + 1;
    }
    return sum;
}
```

Function Specifications

- Predicate: a boolean function over program state
  - i.e. an expression that returns a boolean
  - We often use mathematical symbols as well as program text
- Examples
  - x=3
  - y > x
  - $(x \neq 0) \Rightarrow (y+z = w)$
  - $s = \Sigma_{(i \in 1..n)} a[i]$
  - $\forall i \in 1..n \; . \; a[i] > a[i-1]$
  - true

Function Specifications

- Contract between client and implementation
  - Precondition:
    - A predicate describing the condition the function relies on for correct operation
  - Postcondition:
    - A predicate describing the condition the function establishes after correctly running
- Correctness with respect to the specification
  - If the client of a function fulfills the function's precondition, the function will execute to completion and when it terminates, the postcondition will be true
- What does the implementation have to fulfill if the client violates the precondition?
  - A: Nothing. It can do anything at all.

Function Specifications

```
/*@ requires len >= 0 && array.length == len
  @
  @ ensures \result ==
  @             (\sum int j;  0 <= j && j < len;  array[j])
  @*/
float sum(int array[], int len) {
      float sum = 0.0;
      int i = 0;
      while (i < length) {
            sum = sum + array[i];
            i = i + 1;
      }
      return sum;
}
```

Notation from the Java Modeling Language (JML)

Quick Quiz

Assume the specification for sum given in the lecture slides:

requires len >= 0 && array.length == len

ensures \result == (\sum int j; 0 <= j && j < len; array[j])

Assume the following input and outputs for sum, where a 3 element array is written as [1, 2, 3]. For which of the inputs and outputs is the implementation of sum correct according to the specification given?

- Input: array = [1, 2, 3, 4], len = 4
  Output: 10
- Input: array = [0, 0, 3, -7], len = 4
  Output: none (the program does not terminate)
- Input: array = [1, 2, 3, 4], len = 3
  Output: 7
- Input: array = [1, 2, -3, 4], len = 4
  Output: 7

Hoare Triples

- Formal reasoning about program correctness using pre- and postconditions
- Syntax: {P} S {Q}
  - P and Q are predicates
  - S is a program
- Semantics
  - If we start in a state where P is true and execute S, then S will terminate in a state where Q is true

Hoare Triple Examples

- { true } x := 5 { x=5 }
- { x = y } x := x + 3 { x = y + 3 }
- { x > -1 } x := x * 2 + 3 { x > 1 }
- { x=a } if (x < 0) then x := -x { x=|a| }
- { false } x := 3 { x = 8 }
- { x < 0 } while (x!=0) x := x-1 { }
  - no such triple!

Strongest Postconditions

- Here are a number of valid Hoare Triples:
  - {x = 5} x := x * 2 { true }
  - {x = 5} x := x * 2 { x > 0 }
  - {x = 5} x := x * 2 { x = 10 || x = 5 }
  - {x = 5} x := x * 2 { x = 10 }
    - All are true, but this one is the most *useful*
    - x=10 is the *strongest postcondition*
- If {P} S {Q} and for all Q' such that {P} S {Q'}, Q $\Rightarrow$ Q', then Q is the strongest postcondition of S with respect to P
  - check: x = 10 $\Rightarrow$ true
  - check: x = 10 $\Rightarrow$ x > 0
  - check: x = 10 $\Rightarrow$ x = 10 || x = 5
  - check: x = 10 $\Rightarrow$ x = 10

Weakest Preconditions

- Here are a number of valid Hoare Triples:
  - {x = 5 && y = 10} z := x / y { z < 1 }
  - {x < y && y > 0} z := x / y { z < 1 }
  - {y ≠ 0 && x / y < 1} z := x / y { z < 1 }
    - All are true, but this one is the most *useful* because it allows us to invoke the program in the most general condition
    - y ≠ 0 && x / y < 1 is the *weakest precondition*
- If {P} S {Q} and for all P' such that {P'} S {Q}, P' $\Rightarrow$ P, then P is the weakest precondition *wp*(S,Q) of S with respect to Q

Hoare Triples and Weakest Preconditions

- {P} S {Q} holds if and only if P $\Rightarrow$ *wp*(S,Q)
  - In other words, a Hoare Triple is still valid if the precondition is stronger than necessary, but not if it is too weak
- Question: Could we state a similar theorem for a strongest postcondition function?
  - e.g. {P} S {Q} holds if and only if *sp*(S,P) $\Rightarrow$ Q
  - A: Yes, but it's harder to compute

Quick Quiz

Consider the following Hoare triples:

A) { z = y + 1 } x := z * 2 { x = 4 }
B) { y = 7 } x := y + 3 { x > 5 }
C) { false } x := 2 / y { true }
D) { y < 16 } x := 2 / y { x < 8 }

- Which of the Hoare triples above are valid?
- Considering the valid Hoare triples, for which ones can you write a stronger postcondition? (Leave the precondition unchanged, and ensure the resulting triple is still valid)
- Considering the valid Hoare triples, for which ones can you write a weaker precondition? (Leave the postcondition unchanged, and ensure the resulting triple is still valid)

Hoare Logic Rules

- Assignment
  - { P } x := 3 { x+y > 0 }
  - What is the weakest precondition P?
    - What is most general value of y such that 3 + y > 0?
    - y > -3

Hoare Logic Rules

- Assignment
  - { P } x := 3*y + z { x * y - z > 0 }
  - What is the weakest precondition P?

Hoare Logic Rules

- Assignment
  - { P } x := 3 { x+y > 0 }
  - What is the weakest precondition P?
- Assignment rule
  - *wp*(x := E, P) = [E/x] P
    - Resulting triple: { [E/x] P } x := E { P }
  - [3 / x] (x + y > 0)
  - = (3) + y > 0
  - = y > -3

Hoare Logic Rules

- Assignment
  - { P } x := 3*y + z { x * y - z > 0 }
  - What is the weakest precondition P?
- Assignment rule
  - *wp*(x := E, P) = [E/x] P
  - [3*y+z / x] (x * y – z > 0)
  - = (3*y+z) * y - z > 0
  - = $3*y^2 + z*y - z > 0$

Hoare Logic Rules

- Sequence
  - { P } x := x + 1; y := x + y { y > 5 }
  - What is the weakest precondition P?

Hoare Logic Rules

- Sequence
  - { P } x := x + 1; y := x + y { y > 5 }
  - What is the weakest precondition P?
- Sequence rule
  - *wp*(S;T, Q) = *wp*(S, *wp*(T, Q))
  - *wp*(x:=x+1; y:=x+y, y>5)
  - = *wp*(x:=x+1, *wp*(y:=x+y, y>5))
  - = *wp*(x:=x+1, x+y>5)
  - = x+1+y>5
  - = x+y>4

Hoare Logic Rules

- Conditional
  - { P } if x > 0 then y := z else y := -z { y > 5 }
  - What is the weakest precondition P?

Hoare Logic Rules

- Conditional
  - { P } if x > 0 then y := z else y := -z { y > 5 }
  - What is the weakest precondition P?
- Conditional rule
  - *wp*(if B then S else T, Q)
    = B $\Rightarrow$ *wp*(S,Q) && $\neg$B $\Rightarrow$ *wp*(T,Q)
  - *wp*(if x>0 then y:=z else y:=-z, y>5)
  - = x>0 $\Rightarrow$ *wp*(y:=z,y>5) && x≤0 $\Rightarrow$ *wp*(y:=-z,y>5)
  - = x>0 $\Rightarrow$ z > 5 && x≤0 $\Rightarrow$ -*z* > 5
  - = x>0 $\Rightarrow$ z > 5 && x≤0 $\Rightarrow$ z < -5

Quick Quiz

Fill in the missing pre- or post-conditions with predicate that make each Hoare triple valid.

A) { x = y } x := y * 2 { }
B) { } x := x + 3 { x = z }
C) { } x := x + 1; y := y * x { y = 2 * z }
D) { } if (x > 0) then y := x else y := 0 { y > 0 }

Hoare Logic Rules

- Loops
  - { P } while (i < x) f=f*i; i := i + 1 { f = x! }
  - What is the weakest precondition P?
- Intuition
  - Must prove by induction
    - Only way to generalize across number of times loop executes
  - Need to guess induction hypothesis
    - Base case: precondition P
    - Inductive case: should be preserved by executing loop body

Proving loops correct

- First consider *partial correctness*
  - The loop may not terminate, but if it does, the postcondition will hold
- {P} while B do S {Q}
  - Find an invariant Inv such that:
    - P $\Rightarrow$ Inv
      - The invariant is initially true
    - { Inv && B } S {Inv}
      - Each execution of the loop preserves the invariant
    - (Inv && $\neg$B) $\Rightarrow$ Q
      - The invariant and the loop exit condition imply the postcondition

Loop Example

- Prove array sum correct

```
{ N ≥ 0 }
j := 0;
s := 0;

while (j < N) do

    j := j + 1;
    s := s + a[j];

end
{ s = (Σi | 0≤i<N • a[i]) }
```

Replace N with j
Add information on range of j

Loop Example

- Prove array sum correct

```
{ N ≥ 0 }
j := 0;
s := 0;
{ 0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }
while (j < N) do
    {0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) && j < N}
    j := j + 1;
    s := s + a[j];
    {0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }
end
{ s = (Σi | 0≤i<N • a[i]) }
```


Quick Quiz

Consider the following program:

```
{ N >= 0 }
i := 0;
while (i < N) do
   i := N
{ i = N }
```

Which of the following loop invariants are correct? For those that are incorrect, explain why.

A) i = 0
B) i = N
C) N >= 0
D) i <= N

Proof Obligations

- Invariant is initially true
  { N ≥ 0 }
  j := 0;
  s := 0;
  { 0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }
- Invariant is maintained
  {0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) && j < N}
  j := j + 1;
  s := s + a[j];
  {0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }
- Invariant and exit condition implies postcondition
  0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) && j ≥ N
  ⇒ s = (Σi | 0≤i<N • a[i])

Proof Obligations

- Invariant is initially true
  { N ≥ 0 }
  { 0 ≤ **0** ≤ N && 0 = (Σi | 0≤i<**0** • a[i]) } *// by assignment rule*
  j := 0;
  { 0 ≤ j ≤ N && **0** = (Σi | 0≤i<j • a[i]) } *// by assignment rule*
  s := 0;
  { 0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }

- Need to show that:
  (N ≥ 0) ⇒ (0 ≤ 0 ≤ N && 0 = (Σi | 0≤i<0 • a[i]))
= (N ≥ 0) ⇒ (0 ≤ N && 0 = **0**) *// 0 ≤ 0 is true, empty sum is 0*
= (N ≥ 0) ⇒ (0 ≤ N) *// 0=0 is true, P && true is P*
= **true**

Proof Obligations

- Invariant is maintained

  {0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) && j < N}

  {0 ≤ **j +1** ≤ N && s+a[**j+1**] = (Σi | 0≤i<**j+1** • a[i]) } *// by assignment rule*

  j := j + 1;

  {0 ≤ j ≤ N && **s+a[j]** = (Σi | 0≤i<j+1 • a[i]) } *// by assignment rule*

  s := s + a[j];

  {0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }
- Need to show that:

  (0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) && j < N)

  ⇒ (0 ≤ j +1 ≤ N && s+a[j+1] = (Σi | 0≤i<j+1 • a[i]))

= (0 ≤ j < N && s = (Σi | 0≤i<j • a[i]))

  ⇒ (**-1 ≤ j < N** && s+a[j+1] = (Σi | 0≤i<j+1 • a[i])) *// simplify bounds of j*

= (0 ≤ j < N && s = (Σi | 0≤i<j • a[i]))

  ⇒ (-1 ≤ j < N && s+a[j+1] = (Σi | 0≤i<j • a[i]) **+ a[j]**) *// separate last element*

// we have a problem – we need a[j+1] and a[jj] to cancel out

Where's the error?

- Prove array sum correct

{ N ≥ 0 }
j := 0;
s := 0;

while (j < N) do

  j := j + 1;
  s := s + a[j];

end
{ s = (Σi | 0≤i<N • a[i]) }

Need to add element ***before*** incrementing j

Corrected Code

- Prove array sum correct

{ N ≥ 0 }
j := 0;
s := 0;

while (j < N) do

    s := s + a[j];
    j := j + 1;

end
{ s = (Σi | 0≤i<N • a[i]) }

Proof Obligations

- Invariant is maintained

  {0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) && j < N}
  {0 ≤ j +1 ≤ N && **s+a[j]** = (Σi | 0≤i<j+1 • a[i]) } *// by assignment rule*
  s := s + a[j];
  {0 ≤ **j +1** ≤ N && s = (Σi | 0≤i<**j+1** • a[i]) } *// by assignment rule*
  j := j + 1;
  {0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }

- Need to show that:

  (0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) && j < N)
  ⇒ (0 ≤ j +1 ≤ N && s+a[j] = (Σi | 0≤i<j+1 • a[i]))
  = (0 ≤ j < N && s = (Σi | 0≤i<j • a[i]))
  ⇒ (**-1 ≤ j < N** && s+a[j] = (Σi | 0≤i<j+1 • a[i])) *// simplify bounds of j*
  = (0 ≤ j < N && s = (Σi | 0≤i<j • a[i]))
  ⇒ (-1 ≤ j < N && s+a[j] = (Σi | 0≤i<j • a[i]) **+ a[j]**) *// separate last part of sum*
  = (0 ≤ j < N && s = (Σi | 0≤i<j • a[i]))
  ⇒ (-1 ≤ j < N && s = (Σi | 0≤i<j • a[i])) *// subtract a[j] from both sides*
  = **true** *// 0 ≤ j ⇒ -1 ≤ j*

Proof Obligations

- Invariant and exit condition implies postcondition

  $0 \le j \le N \;\&\&\; s = (\Sigma i \mid 0 \le i < j \bullet a[i]) \;\&\&\; j \ge N$

  $\Rightarrow s = (\Sigma i \mid 0 \le i < N \bullet a[i])$

= $0 \le j \;\&\&\; \mathbf{j = N} \;\&\&\; s = (\Sigma i \mid 0 \le i < j \bullet a[i])$

  $\Rightarrow s = (\Sigma i \mid 0 \le i < N \bullet a[i])$

  // because $(j \le N \;\&\&\; j \ge N) = (j = N)$

= $0 \le \mathbf{N} \;\&\&\; s = (\Sigma i \mid 0 \le i < \mathbf{N} \bullet a[i]) \Rightarrow s = (\Sigma i \mid 0 \le i < N \bullet a[i])$

  // by substituting N for j, since j = N

= **true** *// because P && Q* $\Rightarrow$ *Q*

Quick Quiz

- For the program below and the invariant i <= N, write the proof obligations. The form of your answer should be three mathematical implications.

```
{ N >= 0 }
i := 0;
while (i < N) do
   i := N
{ i = N }
```

- Invariant is initially true:
- Invariant is preserved by the loop body:
- Invariant and exit condition imply postcondition:

Invariant Intuition

- For code without loops, we are simulating execution directly
  - We prove one Hoare Triple for each statement, and each statement is executed once
- For code with loops, we are doing *one* proof of correctness for *multiple* loop iterations
  - Don't know how many iterations there will be
  - Need our proof to cover all of them
  - The invariant expresses a *general* condition that is true for every execution, but is still strong enough to give us the postcondition we need
  - This tension between generality and precision can make coming up with loop invariants hard

Session Summary

- While testing can find bugs, formal verification can assure their absence
- Verification requires formalizing interfaces as pre- and post-conditions
- Hoare Logic is a mechanical approach for verifying software
  - Creativity is required in finding loop invariants, however

Further Reading

- C.A.R. Hoare. **An Axiomatic Basis for Computer Programming.** *Communications of the ACM* 12(10):576-580, October 1969.