CS 2700 Programing and Data Structures.

Slot C (Mon 10.00am, Tues 9.00am, Wed 8.00am, Fri 12.00pm)

Instructor: Meghana Nasre (meghana@cse.iitm.ac.in)

Week 1: Correctness of Programs.

Tools for Two Aspects

1. Correctness
2. Complexity

Program Correctness

Testing

- Can quickly find obvious bugs
- Cases we do not test still hide bugs
- Testing is exhaustive only if number of inputs is finite

Formal methods should be used in conjunction with testing, not as a replacement

Formal Methods

- Treat programs as mathematical objects
- Use mathematical notation to precisely specify what a program does
- Use rules to mathematically prove the correctness
- Can be expensive

Floyd-Hoare Logic

- More commonly known as Hoare Logic
- Method for mathematically reasoning about programs
- Basis of “automated” program verification systems
- Works with Hoare Triples

  {pre-condition} Statement {post-condition}

Tony Hoare

Robert Floyd

Pre and Post-conditions

- Constraints that MUST be satisfied at a program point.
- Constraints are simple Boolean expressions.
- Pre-conditions : Prior to the statement.
- Post-conditions: After the statement.
- Having the conditions as precise as possible helps.
- Types of statements:
  - Assignments, conditionals, loops

Assignments

```
{ x > 0 }
x = x+1;
{  E1  }
```

```
{ x > 0 }
 x = 2*x;
{  E2  }
x = x+y;
{  E3  }
```

Assumptions:

- Values do not overflow
- x is of integer data type

Observe:

- Assuming x is of type int, E2: $x > 0$.
- E2 : x %2 == 0
- If we had an additional pre-condition that $y >= 0$, then E3: $x > 1$

Conditionals

```
{ x > 0 }
 if (x > 5) {
     {  x > 5   }
      x = 4;            **
      { x == 4 }
}
{   E3  }
```

E3 when x = 10 at Line **:

- (x > 0 && x <= 5) || x == 10

E3 when x = 4 at Line **:

- (x > 0 && x <= 5) || x == 4
- (x > 0 && x <= 5)

Question: Can we replace the || by XOR?

Conditionals

```
{ E1 }
 if (B) {
  { E1 && B  }
     S1
   {  E2      }
 }
 { E3 }
```

if (B) then S1

- Either the effect of S1 is visible as E2

  OR

- E1 and NOT(B) hold

E3:

E2 || (E1 && NOT(B))

observe the ||

Conditionals

```
{ x > 0 }
 if (x > 5) {
    { x > 5  }
      x = 4;
    {  x == 4  }
 } else  {
     {  E1  }
     x = x – 10;
     {  E2  }
 }
{  E3  }
```

E1: x > 0 && x <= 5

E2: x > -10 && x <= -5

E3: x == 4 || (-9 <= x <= -5)

if (B) then S1 else S2

- Note the && in E1 and E2
- Note the || in E3

- Relative updates (x = x – 10) modify the earlier expressions
- Absolute updates (x = 4) generate new expressions

How does this relate to my programs?

$((x == 3) \land (y == 0) \land (ptr == NULL)) \;||\; ((x == 3) \land (y != 0) \land (ptr == \&x))$

```
#include<stdio.h>
int main() {
	int x = 3;
	int y;   // read y from user.
	int *ptr = &x;
	if (y == 0) ptr = NULL;
	if (*ptr < 5) printf(" x<5");
	else printf(" x >= 5");
}
```

- What is the output of the program?
- How do we address it?
- How do we address it with our new learning about pre-conditions and post-conditions?

Loops

- **while (B) { S }**
- Loops are interesting since we do not know how many times the loop executes.
- We want a condition which holds true irrespective of the times the loop executed.

```
{ x >= 0 && y >= 0 }
while (x >= y) {
    { x >= y && y >= 0 }
      x = x – y;
    { x >= 0 && y >=0 }
}
{ x >= 0 && y >= 0 }
```

Loop Invariant

- { I }
- while (B) {
- {I && B }
- S1; S2; S3;
- { I }
- }
- {I && NOT(B) }

We call an expression I a loop invariant if:

- It holds just before the loop.
- It holds just after the test B. We assume test B does not have side effects.
- It holds at the end of the loop.
- It need NOT hold at intermediate steps.

Loop Invariant : example 1

Program to find the sum of first n positive integers

- Some trivial invariants:
  - k == 1 || k == 2 || k == 3…
  - sum == 1 || sum == 3 || …
  - Combining the above..

```
int k = 1; sum = 0;
while ( k <= n )  {
    sum = sum + k;
    k = k + 1;
}
```

Invariant: sum = 1 + 2 + 3 + .. + k ✗

Is this correct?

Correct Invariant: sum = 1 + 2 + 3 + .. + k-1

Loop Invariant : example 2

```
A: array indexed 0 .. n-1
int k = n;
while ( k != 0 )  {
    k = k - 1;
    A[k] = 0;
}
```

{ 0 <= j <= n-1 &&
for all j >= k,   A[j] == 0 }

What does the loop do?
Sets A[0] … A[n-1] equal to 0.
What should be the post condition at the end of the loop?

{for j = 0 .. n-1: A[j] == 0}

Guess a loop invariant.

Assume n > 0 is a pre-condition.
{k <= n && k > = 0}
This is a loop invariant but not useful one.

Loop Invariant: example 3

```
t = 1; u = xy[0];
while ( t < r)  {
     if (xy[t] > u)
         u = xy[t];
     t++;
}
```

Take away: avoid writing such cryptic programs.

- Finding elegant and useful loop invariants needs a high level understanding of the code.
- It is non-trivial to do it automatically.
- The programmer (you) should state them as precisely as possible.

Testing whether a given condition is a valid invariant is much simpler than coming up with the condition.

Program Correctness (partial)

Overall strategy:

- Write your algorithm / program
- Write down the pre-conditions at the beginning and post-conditions at the end of the program.
- For each statement show that its post-condition follows from the pre-condition.

Notes:

- Axioms or rules of Hoare logic are simple, but we can select too strong a pre-condition or too weak a post-condition.
- Need to achieve the right trade-off.
- We are NOT proving termination via this method. We assume that the program terminates.

Axioms of Hoare logic

- Empty Statement :

$$\frac{}{\{P\}\ \text{no-op}\ \{P\}}$$

- Assignment Statement:

$$\frac{}{\{P[x\ /\ t]\}\ x = t\ \{P\}}$$

If P is true when x is replaced by t **before** the assignment, then P is true after the assignment.

- Rule of Composition:

$$\frac{\{P\}\ S1\ \{Q\}\ \&\&\ \{Q\}\ S2\ \{R\}}{\{P\}\ S1;\ S2\ \{R\}}$$

Axioms of Hoare logic

- Strengthening pre-cond:

$$\frac{\{P\}\ S\ \{Q\} \quad \&\&\ R \rightarrow P}{\{R\}\ S\ \{Q\}}$$

example: {practiced 10 problems} write exam {90+ marks}
{practiced 20 problems} write exam {90+ marks}

- Weakening post-cond:

$$\frac{\{P\}\ S\ \{Q\} \quad \&\&\ Q \rightarrow R}{\{P\}\ S\ \{R\}}$$

example: {practiced 10 problems} write exam {90+ marks}
{practiced 10 problems} write exam {80+ marks}

Axioms of Hoare logic

- Conditional:

{P && B} S1 {Q} && {P && NOT(B) S2 {Q}

{ P} if (B) then S1 else S2 { Q}

- Loops:

{P && B} S {P}

{ P} while (B) S { P && NOT(B) }

Using Hoare Triples

{ x and y are int
x == t1, y == t2 }
x = x + y;
y = x – y;
x = x – y;

{ P }

{y == t2 && x + y – y == t1 }

x = x + y;

{x – (x – y) == t2 && x -y == t1 }

y = x - y;

{x - y == t2 && y == t1 }

x = x - y;

{x == t2 && y == t1 }

REPLACE x BY x+y

REPLACE y BY x-y

REPLACE x BY x-y

Using Hoare Triples

```
int fun (int n) {
    int k, j;
    k = 0; j = 1;
    while ( k < n ) {
        k = k+1;   j = 2 *j;
    }
    { R }
    return j;
}
```

GUESS: (1) POST CONDITION

(2) LOOP INVARIANT

POST CONDITION: $j == 2^n$

LOOP INVARIANT: $j == 2^k$

NOTE THAT

$$(\text{LOOP INV} \wedge k == n) \Rightarrow j == 2^n$$

Example continued..

```
(1)  {j == 2^k}
     while ( k < n ) {
  (2)    {j == 2^k && k < n}
         k = k+1;
        {2j == 2^k}
         j = 2 *j;
  (3)    {j == 2^k}
     }
```

WE GUESSED LOOP INV AS $j == 2^k$

∴ IT NEEDS TO HOLD AT (1), (2), (3)

CHECK WHETHER IT HOLDS.

BY USING ASSIGNMENT AXIOM TWICE, WE CAN VERIFY THAT LOOP INV. IS INDEED CORRECT.

THE ($k < n$) part comes from the check of while loop.

Example continued..

```
int k, j;
{1 == 1}
k = 0;
{1 == 2^k}
j = 1;
{j == 2^k}
while ( k < n ) {
        k = k+1;   j = 2 *j;
  }
}
```

NOTE THAT WITH THE GUESSED INVARIANT AND POST COND. WE GET TRUE AS THE PRECONDITION.

IF WE STRENTHEN OUR POST CONDITION THEN WE WILL OBTAIN $n \geq 0$. as precondition

One last example..

- Input: An array A of integers indexed from 0 to n-1
- Goal: set max = largest element in the array.
- Write the post-condition.

NOTE: $m \geq A[k]$ for $0 \leq k \leq n-1$

IS NOT SUFFICIENT.

```
// A : indexed from 0 .. n-1
int m = A[0];
int k = 1;
while ( k < n ) {
     if (A[k] > m)
          m = A[k];
    } else {
        // do nothing.
    }
    k = k+1;
}
```

To Summarize..

- Hoare logic and Hoare triples provide an automated way of proving (partial) program correctness.
- Hoare style proofs can become very lengthy much more detailed.
- How detailed should our proofs be?
  - We should be able to write the detailed Hoare style proofs (if needed).
  - Most proofs that we will write will be compact (example: prove invariant of the loop).
  - Be ready to expand a compact proof to a detailed proof if necessary!
- See list of incomplete proofs on wikipedia

Does this terminate?

```
void fun (int n ) {
    print n;
    while (n != 1) {
        if (n % 2 == 0)
            n = n / 2; print n;
        else
            n = 3*n+1;  print n;
    }
    print n;
}
```