Correctness

Rupesh Nasre.
rupesh@iitm.ac.in

July 2019

Program Correctness

- Notions of correctness
  - Proof
  - Testing
- Proofs cover all the cases.
- Testcases are inherently incomplete.
  - except when the number of inputs is finite.
  - In general, testing can prove *presence* of bugs, and not their absence.
- Proofs are guided by computation properties.

Example

```
#include <stdio.h>

int main() {
	int var = 3;
	int flag = 0;
	int *ptr = &var;

	if (flag == 0) ptr = NULL;

	if (*ptr < 5) printf("if\n");
	else printf("else\n");

	return 0;
}
```

```
$ gcc bug.c
$ a.out
Segmentation fault (core dumped)
$
```

Example

```
#include <stdio.h>
#include <assert.h>
int main() {
	int var = 3;
	int flag = 0;
	int *ptr = &var;

	if (flag == 0) ptr = NULL;

	assert(ptr != NULL);

	if (*ptr < 5) printf("if\n");
	else printf("else\n");

	return 0;
}
```

$ gcc bug.c

Error: Undefined reference to **assert**

$ gcc bug.c

$ a.out

a.out: bug.c:12: main: Assertion `ptr != NULL' failed.

Aborted (core dumped)

$

With this knowledge, now the error can be traced backwards.

Example

```
#include <stdio.h>
#include <assert.h>
int main() {
        int var = 3;
        int flag = 0;
        int *ptr = &var;

        assert(ptr != NULL);

        if (flag == 0) ptr = NULL;

        if (*ptr < 5) printf("if\n");
        else printf("else\n");

        return 0;
}
```

```
$ gcc bug.c
$ a.out
Segmentation fault (core dumped)
$
```

With this knowledge, now the error can be traced forward.

Pre and Post-conditions

- Define constraints that **must** be satisfied at a program point.
- Constraints are simple boolean expressions.
  - We will use C-style conditions.
- Pre-condition (post-condition) identifies constraints <u>prior to</u> (<u>after</u>) a statement.
- Ideally, we want the **most precise** conditions.
- Types of statements:
  - assignments, conditionals, loops.

Assignments

$\{x > 0\}$

$x{+}{+};$

$\{??\}$

Assignments

$\{x > 0\}$
x++;
$\{x > 1\}$

Assumptions:

- Values do not overflow (no wrap-around).
- Type information is available.

$\{x > 0\}$
$x = 2 * x;$
$\{ ?? \}$
$x = x + y;$
$\{ ?? \}$
$x = x - y;$
$\{ ?? \}$

Notes

- If x > 0 and it is an int, x is at least 1. Hence, 2 * x > 1. Additionally, x % 2 == 0.
- If y is unknown, x + y can be any value.
- If type of y is known to be unsigned, then x + y would also be above 1.
- If the effect of two statements can be gauged together, we can regain the information that x > 1. Otherwise, we lose track of x's value.

Conditionals

$\{x > 0\}$

$\text{if } (x > 5)\ \{$

$\quad x = 4;$

$\}$

Conditionals

$\{x > 0\}$
if $(x > 5)$ {
    $\{x > 5\}$
    $x = 4;$
    $\{x == 4\}$
}
$\{\ ??\ \}$

Conditionals

$\{x > 0\}$

$\text{if } (x > 5) \; \{$

$\quad \{x > 5\}$

$\quad x = 4;$

$\quad \{x == 4\}$

$\}$

$\{x > 0 \;\&\&\; x <= 5 \;||$

$x == 4\}$

After an $\text{if } (C) \; S$

- either the effect of S is visible;
- or the effect of the statements prior to this if statement falls through. In this case, the condition C is guaranteed to be false.

Conditionals

$\{x > 0\}$

$\text{if } (x > 5)\ \{$

$\quad \{x > 5\}$

$\quad x = 4;$

$\quad \{x == 4\}$

$\}$

$\{x > 0\ \&\&\ x <= 5\}$

Conditionals

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

    x -= 10;

}
```

Conditionals

```
{x > 0}
if (x > 5) {
    {x > 5}
    x = 4;
    {x == 4}
} else {
    {x <= 5}
    x -= 10;


}
```

Conditionals

```
{x > 0}
if (x > 5) {
    {x > 5}
    x = 4;
    {x == 4}
} else {
    {x <= 5 && x > 0}
    x -= 10;
    {x <= -5 && x > -10}
}
```

Conditionals

$\{x > 0\}$

$\text{if } (x > 5)\ \{$

$\quad \{x > 5\}$

$\quad x = 4;$

$\quad \{x == 4\}$

$\}\ \text{else}\ \{$

$\quad \{x <= 5\ \&\&\ x > 0\}$

$\quad x\ \text{-=}\ 10;$

$\quad \{x <= -5\ \&\&\ x > -10\}$

$\}$

$\{x == 4\ ||\ x \in (-10..-5]\}$

After an $\text{if } (C)\ S1 \text{ else } S2$

- On entering a conditional, the conditions get ANDed (at $S1$, C gets ANDed; at $S2$, $\text{NOT } C$ gets ANDed).
- On exiting a conditional, the conditions get ORed (at the end of if-else).
- Relative updates retain+modify existing conditions (e.g., $x\ \text{-=}\ 10$).
- Absolute updates generate new conditions (e.g., $x = 4$).

Note that the effect of the conditions prior to if-else may not reach end of if-else (unlike in if-without-else)

Getting back to the segfault

```
#include <stdio.h>

int main() {
	int var = 3;
	int flag = 0;
	int *ptr = &var;

	if (flag == 0) ptr = NULL;

	if (*ptr < 5) printf("if\n");
	else printf("else\n");

	return 0;
}
```

Write pre- and post-conditions for this program.

Compare your inference with what happens at program execution time.

Conditionals

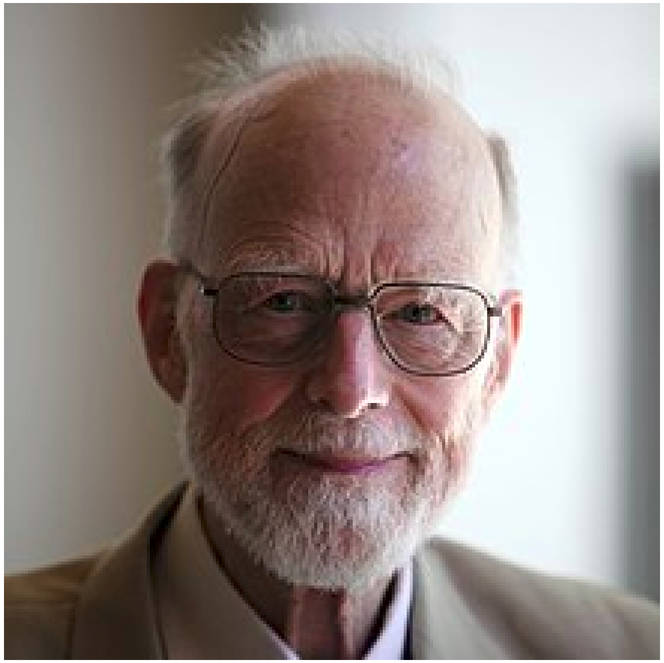
```
{x > 0}
if (...) {
    {x > 5}
    ...
   {x == 4}
} else {
    {x <= 5 && x > 0}
    ...
    {x <= -5 && x > -10}
}
{x == 4 || x ϵ (-10..-5]}
```

Thought exercise: Given a set of pre and post-conditions, can we automatically generate a program?

Tony Hoare

- Inventor of quicksort
- Known for Hoare logic
- Turing Award in 1980

  "fundamental contributions to the definition and design of programming languages"



Loops

```
while (x >= y) {
    x -= y;
}
```

Loops

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

    x -= y;

}
```

We want to check if x becomes negative at the end of the loop.

Loops

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


}
```

Loops

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


}
```

Loops

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

Loops

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

For while (C) S

- On entering the loop for the first time, the conditions get ANDed.
- On reaching C in further iterations, the conditions again get ANDed with C, but the iterative conditions get ORed.
- On exiting a loop, NOT C must be true.
- Similar to if, conditions from within the loop and those from outside would get ORed.

Loops

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

Post-condition indicates that x may be negative at the end of the loop.

Loops

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

Post-condition indicates that x may be negative at the end of the loop; but the program doesn't say so.

Loops

$\{\mathbf{x >= 0}$ && $y >= 0\}$

while $(x >= y)$ {

$\quad\{x >= y$ && $y >= 0\}$

$\quad x$ -= y;

$\quad\{\mathbf{x >= 0}$ && $y >= 0\}$

}

$\{x < y$ && $y >= 0$ && $\mathbf{x >= 0}\}$

x cannot be negative at the end of the loop.

Classwork

- Write the final post-conditions for the following programs.
  - To find the maximum element in an array
  - To compute the sum of the odd elements in an array
  - To compute the sum of the odd index elements
  - To find an anagram of a string (e.g., structures == trust cures, and data structures == custard stature)
- Find a reasonable anagram of annual grit.

Loops

$\{\mathbf{x >= 0}\ \&\&\ y >= 0\}$

$\text{while } (x >= y)\ \{$

$\quad \{x >= y\ \&\&\ y >= 0\}$

$\quad x\ \text{-=}\ y;$

$\quad \{\mathbf{x >= 0}\ \&\&\ y >= 0\}$

$\}$

$\{x < y\ \&\&\ y >= 0\ \&\&\ \mathbf{x >= 0}\}$

x cannot be negative at the end of the loop.

Such a condition is called a **loop-invariant**.

- It is true prior to entering the loop.
- It is true in every iteration of the loop.
- It is true at the end of the loop.

Loop Invariants

- Properties to be satisfied prior to the loop, after every iteration, and at the end of the loop.
- Abstractly specifies the loop
  - bears potential to hide implementation details
- Similarity with recursion
  - resembles inductive hypothesis
- Often, finding loop invariants is not difficult, but finding useful (elegant) loop invariants is non-trivial.

https://en.wikipedia.org/wiki/Loop_invariant

Loop Invariants

```
i = 0;
while (i < N) {
    sum += i;
    ++i;
}
```

Trivial loop-invariants:

- (i == 0 || i == 1 || ... || i == N – 1 || i == N)
- (sum == 0 || sum == 1 || sum == 3 || sum == 6 || sum == arbitrary)
- (i == 0 && sum == 0 || i == 1 && sum == 1 || i == 2 && sum == 3 || ... || i == N && sum == arbitrary)

Useful loop-invariant:

- sum == $\sum^{0}_{i}$

Now let's check if this invariant is satisfied:

- **After every iteration?**

  The incremented i is not summed up yet. Hence the invariant doesn't hold.

  So the invariant should be sum == $\sum^{0}_{i-1}$

- **At the end of the loop?**

  i == N, and the sum contains the summation of 0..N-1.

- **Just prior to the loop?**

  i == 0, but sum is undefined!

  To satisfy the invariant, we need to add this line: sum = 0;

  Thus, loop invariants can help us identify errors in the program.

Classwork

```
satya = 1;
anand = pm[0];
while (satya < nobel) {
    if (pm[satya] > anand)
        anand = pm[satya];
    ++satya;
}
```

- Finding useful / elegant loop invariants need high-level understanding of the code.
- This is not feasible, in general, automatically.
- Even for humans, loop-invariants are often guessed.
- For your own code, you should be able to state those precisely!

```
i = 1;
max = a[0];
while (i < N) {
    if (a[i] > max)
        max = a[i];
    ++i;
}
```

Loop Invariant:

- max contains the maximum value in a[0]..a[i-1].
- This is satisfied just prior to the loop.
- This is satisfied after every iteration.
- This is satisfied at the end of the loop.
- Hence, this program <u>correctly</u> finds maximum element in an array (having at least one element).

Hoare Logic

- To formally reason about correctness of programs
- Originally seeded by Floyd
- Hence, now called Floyd-Hoare Logic
- Dates back to when neither you nor I was born (1969).
- Works with Hoare Triples
  - {pre-condition} Stmt {post-condition}

Hoare Triples

- Empty statement / no-op
- Assignment
- Rule of composition
- Conditional
- Consequence
- Loop

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

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

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

Try for:

- x = y + 2 {x > 7}
- x = x + 1 {x > 7}
- x = 3 {x == 3}

Classwork

- Use Hoare Logic, prove that the following code correctly swaps two variables.

```
a = a + b;
b = a – b;
a = a – b;
```

Classwork

- Use Hoare Logic, prove that the following code correctly swaps two variables.

$\{b == B \wedge a == A\}$

$\{b == B \wedge a + b - b == A\}$

a = a + b;

$\{b == B \wedge a – b == A\}$

$\{a – (a – b) == B \wedge a – b == A\}$

b = a – b;

$\{a - b == B \wedge b == A\}$

a = a – b;

$\{a == B \wedge b == A\}$

Hoare Triples

- Empty statement / no-op

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

- Assignment

$$\frac{}{\{P[x \rightarrow E]\}\ x = E\ \{P\}}$$

- Rule of composition

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

- Conditional

$$\frac{\{P \land C\}\ S1\ \{Q\},\ \{P \land \neg C\}\ S2\ \{Q\}}{\{P\}\ \text{if}\ (C)\ S1\ \text{else}\ S2\ \{Q\}}$$

- Loop

General form (not useful and incorrect)

$$\frac{\{P \land C\}\ S\ \{Q\}}{\{P\}\ \text{while}\ (C)\ S\ \{Q \land \neg C\}}$$

Loop-invariant form

$$\frac{\{P \land C\}\ S\ \{P\}}{\{P\}\ \text{while}\ (C)\ S\ \{P \land \neg C\}}$$

Classwork

```
int power2(int n) {
      int k, j;

      k = 0;
      j = 1;

      while (k != n) {
            k = k + 1;
            j = 2 * j;
      }

      return j;
}
```

- We want to prove that this function returns 2^n.
  - for $n > 0$
- Using testing, we can *validate* for certain inputs.
- Using a proof, we can *verify*.

Source: Supratik Chakraborty, IITB

Classwork

```
int power2(int n) {
        int k, j;

        k = 0;
        j = 1;

        while (k != n) {
                k = k + 1;
                j = 2 * j;
        }

        return j;
}
```

$\{n > 0\}$

```
k = 0;
j = 1;

while (k != n) {
        k = k + 1;
        j = 2 * j;
}
```

$\{j == 2^n\}$

Hoare Triple

Classwork

$\{n > 0\}$

k = 0;

$\{\varphi_1\}$

j = 1;

$\{\varphi_2\}$

```
while (k != n) {
        k = k + 1;
        j = 2 * j;
}
```

$\{j == 2^n\}$

$\{n > 0\}$

```
k = 0;
j = 1;

while (k != n) {
        k = k + 1;
        j = 2 * j;
}
```

$\{j == 2^n\}$

Hoare Triple

Classwork

$\{n > 0\}$

```
k = 0;
```

$\{\varphi_1\}$

```
j = 1;
```

$\{\varphi_2\}$

```
while (k != n) {
        k = k + 1;
        j = 2 * j;
}
```

$\{j == 2^n\}$

How do we prove

$\{\varphi_2\}$

```
while (k != n) {
        k = k + 1;
        j = 2 * j;
}
```

$\{j == 2^n\}$

?

Let's use the Hoare rule for loops.

But that requires a loop-invariant!

"Guess" the loop-invariant.

Since proof requires $j == 2^n$, we guess that the invariant is $j == 2^k$.

Classwork

$$\frac{\{P \land C\}\ S\ \{P\}}{\{P\}\ \text{while}\ (C)\ S\ \{P \land \neg C\}}$$

$$\frac{\{j == 2^k \land k \mathrel{!=} n\}\ k = k + 1;\ j = 2 * j;\ \{j == 2^k\}}{\{j == 2^k \land k \mathrel{!=} n\}\ \text{while}\ (k \mathrel{!=} n)\ \{\ k = k + 1;\ j = 2 * j;\ \}\ \{j == 2^k \land \neg(k \mathrel{!=} n)\}}$$

To prove the **antecedent**, prove the **precedent**.

How do we prove

$\{\varphi_2\}$

```
while (k != n) {
    k = k + 1;
    j = 2 * j;
}
```

$\{j == 2^n\}$

?

Let's use the Hoare rule for loops.

But that requires a loop-invariant!

"Guess" the loop-invariant.

Since proof requires $j == 2^n$, we guess that the invariant is $j == 2^k$.

Classwork

$$\frac{\{P \land C\}\ S\ \{P\}}{\{P\}\ \text{while (C) S}\ \{P \land \neg C\}}$$

$$\frac{\{j == 2^k \land k \mathrel{!=} n\}\ k = k + 1;\ j = 2 * j;\ \{j == 2^k\}}{\{j == 2^k\}}$$

```
while (k != n) {
    k = k + 1; j = 2 * j;
}
```

$\{j == 2^k \land \neg(k \mathrel{!=} n)\}$

$\{j == 2^k\}$

$\{2 * j == 2^{k+1}\}$

$k = k + 1;$

$\{2 * j == 2^k\}$

$j = 2 * j;$

$\{j == 2^k\}$

To prove the **antecedent**, prove the **precedent**.

We are almost there, except that the precondition is missing $k \mathrel{!=} n$.
This is where we use implication:

$$(j == 2^k) \land (k \mathrel{!=} n) \Rightarrow (j == 2^k)$$

In general, we can strengthen the precondition, and weaken the post-condition.

Classwork

- So far we have proved:

$\{n > 0\}$

k = 0;

$\{\varphi_1\}$

j = 1;

$\{\varphi_2 : j = 2^k\}$

```
while (k != n) {
        k = k + 1;
        j = 2 * j;
}
```

$\{j == 2^n\}$

$\{n > 0\}$

// strengthening the precondition

$\{1 = 2^0\}$

k = 0

$\{\varphi_1 : 1 = 2^k\}$

j = 1

$\{\varphi_2 : j = 2^k\}$

Homework

- Prove that the function computes n!.

```
int fact(int n) {
    k = 1;
    f = 1;

    while (k < n) {
        k = k + 1;
        f = f * k;
    }

    return f;
}
```