Program Verification
(Rosen, Sections 5.5)

TOPICS
- Program Correctness
- Preconditions & Postconditions
- Program Verification
  - Assignment Statements
  - Conditional Statements
  - Loops
- Composition Rule

Proofs about Programs

- Why make you study logic?
- Why make you do proofs?
- Because we want to prove properties of *programs*
  – In particular, we want to prove properties of variables at specific points in a program

Isn't testing enough?

- Assuming the program compiles, we perform some amount of testing.
- Testing shows that for specific examples the program seems to be running as intended.
- Testing can only show existence of some bugs but cannot exhaustively identify all of them.
- Verification can be used to prove the correctness of the program with any input.

Software Testing

- Levels
  – Unit, module, integration, system testing
- Methods
  – Black-box, white-box
- Types
  – Functionality, Configuration, Usability, Performance, Compatibility, Error, Localization, …
- Processes
  – Automation, write test code first, code coverage, …

Program Verification

- We consider a program to be *correct* if it produces the expected output **for all possible inputs**.
- Domain of input values can be very large, how many possible values of an integer?

```
int multiply (int operand1, int operand2)
   return operand1 * operand2
 // operand1 = 2^32, operand2 = 2^32
```

- Instead we can formally specify program behavior, then use techniques for inferring correctness.
- For example, we can use logic techniques

Program Correctness Proofs

- Two parts:
  - Correct answer when the program terminates (called *partial correctness)*
  - The program does terminate
- We will only do part 1
  - Prove that a method is correct if it terminates
- Part 2 has been shown to be impossible!

Predicate Logic and Programs

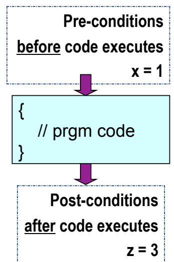
- Variables in programs are like variables in predicate logic:
  - They have a domain of discourse (data type)
  - They have values (drawn from the data type)
- Variables in programs are different from variables in predicate logic:
  - Their values change over time

Assertions

- Two parts:
  - ***Initial Assertion****:* a statement of what must be true about the input values or values of variables at the beginning of the program segment
    - E.g Method that determines the sqrt of a number, requires the input (parameters) to be >= 0
  - ***Final Assertion***: a statement of what must be true about the output values or values of variables at the end of the program segment
    - E.g. What is the output/final result after a call to the method?

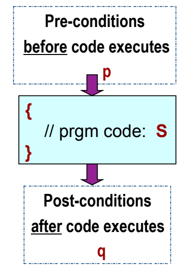
Preconditions and PostConditions

- ***Initial Assertion:*** sometimes called the **pre**condition
- ***Final Assertion***: sometimes called the **post**condition
- **Note**: these assertions can be represented as propositions or predicates. For simplicity, we will write them generally as propositions.



Hoare Triple

- "A program, or program segment, **S**,
  is said to be **partially correct**
  with respect to the
  initial assertion (precondition) **p**
  and the final assertion (postcondition) **q**
  if, whenever **p** is true
  for the input values of **S**
  and **S** terminates,
  then **q** is true for the output values of **S**."
  [Rosen 7th edition, p. 372]
- Notation: **p{S}q**

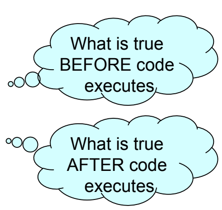


Program Verification
Example #1: Assignment Statements

- Assume that our proof system already includes rules of arithmetic…
- Consider the following code:

```
y = 2;
z = x + y;
```

- Precondition: *p(x), x =1*
- Postcondition: *q(z), z =3*



Program Verification
Example #1: Assignment Statements

- Prove that the program segment:
  y = 2;
  z = x + y;
- Is correct with respect to
  precondition: x = 1
  postcondition: z = 3
- Suppose x = 1 is true as program begins
  - Then y is assigned the value of 2
  - Then z is assigned the value of 3 (x + y = 1 + 2)
  - Thus, the program segment is correct with regards to the precondition that x = 1 & postcondition z = 3

Program Verification
Example #2: Assignment Statements

- Prove that the program segment:
  x = 2;
  z = x * y;
- Is correct with respect to
  precondition: y >= 1
  postcondition: z >= 2
- Suppose y >= 1 is true as program begins
  - Then x is assigned the value of 2
  - Then z is assigned the value of x * y which is 2*(y>=1) which makes z >= 2
  - Thus, the program segment is correct for precondition y >= 1 and postcondition z >= 2

Program Verification
Example #3: Assignment Statements

- Prove that the program segment:
  y = x * x + 2*x - 5
- Is correct with respect to
  precondition: -4 <= x <= 1
  postcondition: -6 <= y <= 3
- Suppose -4 <= x and x <=3 as the program begins
  - If x = -4 then y is assigned (-4)*(-4) + 2*(-4) - 5 = 3
  - If x = -3 then y is assigned (-3)*(-3) + 2*(-3) – 5 = -2
  - If x = -2 then y is assigned (-2)*(-2) + 2*(-2) – 5 = -5
  - If x = -1 then y is assigned (-1)*(-1) + 2*(-1) -5 = -6
  - If x = 0 then y is assigned (0)*(0) + 2*(0) -5 = -5
  - If x = 1 then y is assigned (1)*(1) + 2*(1) – 5 = -2
- Thus, program segment is correct for precondition -6 <= y <= 3
  - - or – {-6, -5, -2, 3}

Program Verification
Example #4: Assignment Statements

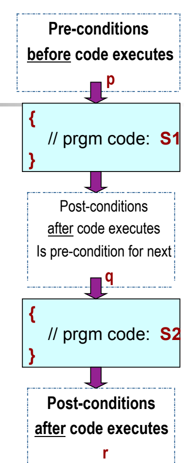
Given the following program segment:
// precondition: -3 < x <= 3
y = x * x - 3*x + 4
What is the postcondition for y?

Suppose -3 <= x and x <=4 as the program begins
- If x = -2 then y is assigned (-2)*(-2) - 3*(-2) + 4 = 14
- If x = -1 then y is assigned (-1)*(-1) - 3*(-1) + 4 = 8
- If x = 0 then y is assigned (0)*(0) - 3*(0) + 4 = 4
- If x = 1 then y is assigned (1)*(1) - 3*(1) + 4 = 2
- If x = 2 then y is assigned (2)*(2) - 3*(2) + 4 = 2
- If x = 2 then y is assigned (3)*(3) - 3*(3) + 4 = 4

Thus, the postcondition for y is 2 <= y <= 14

Rule 1:
Composition Rule

- Once we prove correctness of program segments, we can combine the proofs together to prove correctness of an entire program.
- This is like the hypothetical syllogism inference rule



Program Verification
Example #1: Composition Rule

- Prove that the program segment (swap):
  - t = x;
  - x = y;
  - y = t;
- Is correct with respect to
  - precondition: x = 7, y = 5
  - postcondition: x = 5, y = 7

Program Verification
Example #1 (cont.): Composition Rule

- Program segment:
  - t = x; x = y; y = t;
- Suppose x = 7 and y = 5 is true as program begins
  - // Precondition: x = 7, y = 5
    - t = x
  - // Postcondition: t = 7, x = 7, y = 5
  - // Precondition: t = 7, x = 7, y = 5
    - x = y
  - // Postcondition: t = 7, x = 5, y = 5
  - // Precondition: t = 7, x = 5, y = 5
    - y = t
  - // Postcondition: t = 7, x = 5, y = 7
  - Thus, the program segment is correct with regards to the precondition that x = 7 & y =5 & postcondition x = 5 and y = 7

Rule 2:
Conditional Statements

- Given

  if (*condition*)

  statement;

  With precondition: ***p*** and postcondition: ***q***
- Must show that
  - Case 1: when ***p (precondition)*** is true and ***condition*** is true then ***q (postcondition)*** is true, when ***S (statement)*** terminates

    OR
  - Case 2: when ***p*** is true and ***condition*** is false, then ***q*** is true (***S*** does not execute)

Conditional Rule:
Example #1

Verify that the program segment:

if (x > y) y = x;

Is correct with respect to precondition T and postcondition that y >= x

Consider the two cases…

1. Condition (x >y) is true, then y = x
2. Condition (x >y) is false, then that means x <= y

Thus, if precondition is true, then y =x or x <=y which means that the postcondition that y >=x is true

Conditional Rule: Example #2

Verify that the program segment:

if (x % 2 == 1) x = x + 1

Is correct with respect to precondition T (state of program is correct as enter this program segment) and postcondition that x is even

Consider the two cases…

1. Condition (x % 2 equals 1) is true, then x is odd. If x is odd, then adding 1 means x is even

2. Condition (x % 2 equals 1) is false, then x is even.

Thus, if precondition is true, then x is even or x is even which means that the postcondition that x is even is true

Rule 2a: Conditional with Else

```
if (condition)
  S1;
else
  S2;
```

- Must show that
  - Case 1: when ***p (precondition)*** is true and ***condition*** is true then ***q (postcondition)*** is true, when ***S1*** ***(statement)*** terminates

    OR
  - Case 2: when ***p*** is true and ***condition*** is false, then ***q*** is true, when ***S2*** (***statement***) terminates

Conditional Rule: Example #3

Verify that the program segment:

if (x < 0) abs = -x;
else abs = x;

Is correct with respect to precondition T and postcondition that abs is the absolute value of x

Consider the two cases…

1. Condition (x < 0) is true, then x is negative. Assigning abs the negative of a negative number, means abs is the absolute value of x

2. Condition (x < 0) is false, then x >= 0 which means x is positive. Assigning abs a positive number, means abs is the absolute value of x

Thus, if precondition is true, abs is absolute value of x or absolute value of x. Thus the postcondition that abs is the absolute value of x is true

Conditional Rule: Example #4

Verify that the program segment:

if (balance > 100) nbalance= balance *1.02
else nbalance= balance * 1.005

Is correct with respect to precondition balance > = 0 and postcondition that

((balance > 100) && (nbalance = balance * 1.02)) | |
((balance <= 100) && (nbalance= balance * 1.005))

Consider the two cases…

1. Condition (balance > 100) is true, then assign nbalance to balance*1.02

2. Condition (balance > 100) is false, then assign nbalance to balance* 1.005

Thus, if precondition of balance > = 0 is true, (balance > 100 and nbalance = balance * 1.02) or (balance <= 100 and nbalance = balance * 1.005). Thus the postcondition is proven

How to we prove loops correct?

- General idea: *loop invariant*
- Find a property that is true before the loop
- Show that it must still be true after every iteration of the loop
- Therefore it is true after the loop

Loop Invariant: Example #1

Given following program segment, what is loop invariant for z?

```
int x = 2; y = 3, z = v1
while (x <= 4) {
        z += y;
        x++;
}
```

What is true about z before, during, and after the loop?

Before loop: z = v1

during loop: z = v1 + 3*(x-1)

- iteration 1: x = 2, z = v1 +3
- iteration 2: x = 3, z = v1 + 6
- iteration 3: x = 4, z = v1 + 9

after loop: z = v1 + 9

Thus, loop invariant is: v1 <= z <= v1 + 9

Loop Invariant: Example #2

Given following program segment, what is loop invariant for x, y, z?

```
int x = 1; y = 2; z = -5
while (x <= 5) {
        z += y;
        x++;
}
```

What is true about x, y, z before, during, and after the loop?

Before loop: x = 1, y = 2, z = -5

during loop: 1 <= x <= 6; y = 2; z = -5 + 2*(x)

Iteration 1: x = 1, z = -3

Iteration 2: x = 2, z = -1

Iteration 3: x = 3, z = 1

Iteration 4: x = 4, z = 3

Iteration 5: x = 5, z = 5

after loop: x = 6; y = 2; z = 5

Thus, loop invariant is: 1 <= x <= 6; y = 2; -5 <=z <= 5

Loop Invariant: Example #3

Given following program segment, what is loop invariant for factorial, i?

```
// precondition: n >= 1
i = 1;
factorial = 1;
while (i < n) {
        i++;
        factorial *= i;
}
```

What is true about i and factorial before, during, and after the loop?

Before loop: i = 1 and because n >= 1, then i <= n

factorial = 1 = 1! = i!

during loop: i < n

factorial = i!

after loop: i = n and because i = n, we know i <= n

factorial = i! and because i = n, factorial = i! = n!

Thus, loop invariant is: i <= n; factorial = i!

Verified that program segment terminates with factorial = n!