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.
  - For example, we may want prove that a program segment or method gets the right answer.

Isn't testing enough?

- Assuming the program compiles, we can go ahead and perform some amount of testing.
- Testing shows that for specific examples (test cases) the program is doing what was intended.
- Testing can only show existence of some bugs but cannot exhaustively identify all of them.
- Program verification can be used to prove the correctness of the program with any input.

Software Testing

- Methods
  - Black-box, white-box
- Levels
  - Unit (Method), Module (Class), Integration, System
- Types
  - Functionality, Configuration, Usability, Reliability, Performance, Compatibility, Error, Localization, …
- Processes
  - Regression, Automation, Test-Driven Development, 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? 2^{32}

```
int divide (int operand1, int operand2) {
    return operand1 / operand2;
}
```

- $2^{32} * 2^{32} = 2^{64}$, a large number, so we clearly cannot test exhaustively!
- Instead we formally specify program behavior, then use logic techniques to infer (prove) program correctness.

Program Correctness Proofs

- Part 1 - Prove program produces correct answer when (if) it terminates.
- Part 2 - Prove that the program does indeed terminate at some point.
- We can only Part 1, because Part 2 has been proven to be undecidable:
  - Thus we try to prove that a method is correct, assuming that it terminates (partial correctness).

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 (i.e., locations in the program)
  - Associate the predicate with specific program points
    - Immediately before or after a statement

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
    - For Example: Method that determines the square root 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
    - For Example: Can we specify that the output or result is exactly correct after a call to the method?

Pre and Post Conditions

- ***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.

Pre-condition
before code executes
x = 1

↓

```
{
    // Program segment
}
```

↓

Post-condition
after code executes
z = 3

Hoare Triple

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

Pre-condition (p)
before code executes

↓

```
{
// Program segment: (S)
}
```

↓

Post-condition (q)
after code executes

Program Verification

Example #1: Assignment Statements

- Assume that our proof system already includes rules of arithmetic, and theorems about divisibility …
- Consider the following code:

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

- Pre-condition: $p(x), x = 1$
- Post-condition: $q(z), z = 3$

What is true BEFORE code executes

What is true AFTER code executes

Program Verification

Example #1: Assignment Statements

- Prove that the program segment:

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

- Is correct with respect to:
  - pre-condition: x = 1
  - post-condition: 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 x + y = 1 + 2 = 3
- Thus, the program segment is correct with regards to the pre-condition that x = 1 and post-condition z = 3.

Program Verification
Example #2: Assignment Statements

- Prove that the program segment:

```
y = 2;
z = x * y;
```

- Is correct with respect to:
  pre-condition: x >= 1
  post-condition: 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 = 2 * (y >= 1), which makes z >= 2
- Thus, the program segment is correct for pre-condition y >= 1 and post-condition z >= 2.

Program Verification
Example #3: Assignment Statements

- Prove that the program segment, given integer variables:

```
y = x * x + 2 * x – 5;
```

- Is correct with respect to: pre-condition: -4<= x <= 1, and post-condition: -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 post-condition -6 <= y <= 3, or more precisely y belongs to the set {-6, -5, -2, 3}

Program Verification
Example #4: Assignment Statements

- Given the following segment, x and y are integer variables:

```
// pre-condition: -3 < x <= 3
y = x * x - 3 * x + 4;
// post-condition: ?? <= y <= ??
```

- Suppose -3 < x and x <= 3 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 = 3 then y is assigned (3)*(3) - 3*(3) + 4 = 4
- Thus, the post-condition for y is 2 <= y <= 14.

So far only propositions, what about predicates?

- What if the data type was float or double, or the interval was unbounded?
- Now we need to use predicates – universally quantified over a range of values.
- Actually this is what we did, but simply enumerated all the values in the range since they were integers.
- Revisit Example #3: with floating point values:
  - Need to use more math
  - Is the function increasing?
  - In what intervals?

```
float x, y;
// code to initialize x
y = x * x – 2 * x - 5;
```

Redo with floating point

Example #3: Assignment Statements

- Given that the polynomial below is an increasing function in the interval [-1, 4], prove conditions of the program segment:

$$f(x) = x^2 + 2x - 5$$

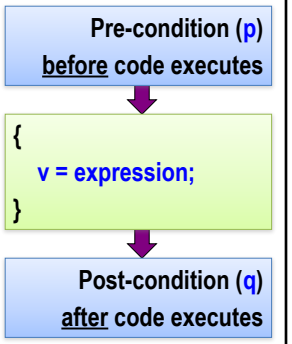
```
float x, y;
// code to initialize x
y = x * x – 2 * x - 5;
```

  - Pre-condition: -1 <= x <= 4
  - Post-condition: ?? <= y <= ??
- Without executing the assignment we know domain of x, so we can prove (using math) the range of y values.
- Q: What is the range of values of f(x)= x * x – 2 * x – 5 that satisfy f(-1) ≤ f(x) ≤ f(4) for values of x in the interval [-1, 4]?
- A: We can prove that, -2 ≤ y ≤ 3 because f(-1)=-2 and f(4)=3

General Rule for Assignments

- To prove the Hoare triple:

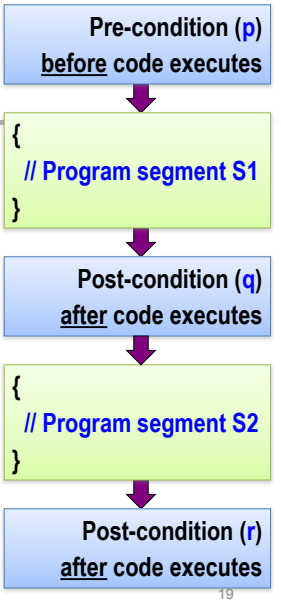
  p {v = expression} q
  - note that **p** and **q** are predicates involving program variables (usually **q** involves **v**)
- We first replace occurrences of v in q by the right hand side expression (expression)
- Then we derive this modified q from p using our rules of inference
- Sometimes we use common sense, e.g., derive first substitute later, as in previous.



Rule 1: Composition Rule

- Once we prove correctness of program segments, we can combine the proofs together to prove correctness of an entire program.

  p {S1} q { S2} r -> p {S1,S2} r
- This is similar to 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

  pre-condition: x = 7, y = 5

  post-condition: 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:
  - // Pre-condition: x = 7, y = 5
    t = x;
  - // Post-condition: t = 7, x = 7, y = 5
  - // Pre-condition: t = 7, x = 7, y = 5
    x = y;
  - // Post-condition: t = 7, x = 5, y = 5
  - // Pre-condition: t = 7, x = 5, y = 5
    y = t;
  - // Post-condition: t = 7, x = 5, y = 7
- The program segment is correct with regards to the pre-condition x = 7 and y =5 and post-condition x = 5 and y = 7.

Rule 2: Conditional Statements

- Given
  if (*c*)
  statement;
  With pre-condition: **p** and post-condition: **q**
- Must show that
  - Case 1: **p && c {S} q**: when **p** is true and **c**, the condition is true then **q** (post-condition) can be derived, when ***S* (*statement*)** terminates,
    AND ALSO THAT
  - Case 2: **p && !c → q**: when ***p*** is true and ***condition*** is false, then ***q*** is true (***S*** does not execute, so we must show that **q** follows directly from **p** and **!c**)

Conditional Rule: Example #1

- Verify that the program segment:

  if (x > y) y = x;

- Is correct with respect to pre-condition T (program state is correct when entering segment) and the post-condition 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 y >= x
- Thus, if pre-condition is true, then y = x or y >= x which means that the post-condition 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 pre-condition T and the post-condition 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 makes x even.
  2. Condition (x % 2 equals 1) is false, then x is already even, and remains even.
- Thus, if pre-condition is true, then either x is even or x is even, so the post-condition 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 pre-condition T and post-condition that abs is the absolute value of x.
- Consider the two cases…

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

  2. Condition (x < 0) is false, x is positive. Assigning abs a positive number means abs is the absolute value of x.
- Thus, if pre-condition is true, then the post-condition 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 pre-condition balance > = 0 and post-condition:

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

  ((balance <= 100) && (nbalance= balance * 1.005))
- Consider the two cases…

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

  2. Condition (balance > 100) is false, 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 post-condition 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

Rule 3: Loop Invariant

```
while (condition)
   S;
```

- Rule:

Note these are both p!

$$\frac{(p \wedge \text{condition})\{S\}p}{\{\text{while condition } S\}(\neg\text{condition} \wedge p)}$$

Note both conclusions

Loop Invariant: Example #1: Simple Assignments

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

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

- Before loop: z = v1
- During loop: z = v1 + y*(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: y=3; z = v1 + y * (x-1)

Loop Invariant: Example #2: More Assignments

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

- 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: Factorial Computation

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

```
i = 1;
factorial = 1;
while (i < n) {
   i++;
   factorial *= i;
}
```

- Before loop: i = 1, factorial = 1 = 1! = i!
- During loop: i < n, and factorial = i!
- After loop: i = n and because i = n, we know i <= n,
  so factorial = i! and because i = n, factorial = i! = n!
- Thus, loop invariant is: i <= n; factorial = i!

So we have proven that the program segment terminates with factorial = n!, i.e. it correctly computes the factorial.