

Proofs about Programs

How can we know that a program is correct?

Test it

--A program with just two integers  
has  $> 10^{18}$  states

Prove it

--That is the goal of work in the area of  
Program Verification, started in 1967  
by Robert Floyd at Stanford



Robert W. Floyd  
(1936 - 2001)

What do we mean by "the program is correct"?

Given a correct set of inputs, it produces the  
correct outputs.

Precondition      properties of the inputs  
S                    the program that executes  
Postcondition      properties of the outputs

One problem is that some programs never terminate.

S is **partially correct** with respect to precondition P and  
postcondition Q if whenever P is true of the inputs and  
S terminates, then Q is true of the outputs.

We denote this by saying that the Hoare Triple

$P \{S\} Q$   
is true.

S is **totally correct** with respect to precondition P and  
postcondition Q if whenever P is true of the inputs, then  
S terminates and Q is true of the outputs.

What we would like are rules of inference that allow us  
to draw conclusions about programs and their pre- and  
postconditions.



Sir Charles Antony Richard Hoare  
(1934 - )

Weakest precondition

There might be a lot of preconditions such that  
 $P \{S\} Q$  is true.

$P \{a := b + 1\} (a > 1)$

The postcondition is satisfied if

P is  $b > 10$   
P is  $b > 9$   
etc.

The precondition that describes the maximal set of  
inputs such that executing P leads to the satisfaction  
of Q is called the **weakest precondition**.

For the example above, P is  $b > 0$

Weakest precondition

$$P \{y := x * x\} (y \geq 4)$$

$x \geq 2$  guarantees the postcondition, but the weakest precondition is  $(x \geq 2) \vee (x \leq -2)$

Note that if  $P$  is  $(x \geq 2) \vee (x \leq -2)$  and  $P'$  is  $x \geq 2$  then  $P' \rightarrow P$

We say that  $P'$  is **stronger** (more restrictive) than  $P$ .

The reason that we are interested in weakest preconditions is

- we can often reason backwards from the postcondition to the weakest precondition that guarantees it
- if that happens to be the specification for the program, then the program is correct.

Rule of Consequence

Suppose  $P \{S\} Q$

If  $P' \rightarrow P$ , then  $P' \{S\} Q$  is also true.

We can strengthen the precondition and maintain the truth value of the Hoare Triple.

Examples:

If we can prove that a program correctly sorts any array of integers, then we know it sorts an array of positive integers.

If  $(b > 0) \{a := b + 1\} (a > 1)$  is true, then  $(b > 10) \{a := b + 1\} (a > 1)$  is also true

Rule of Consequence

We also maintain the truth value of a Hoare Triple if we weaken the postcondition.

A way to express these rules is

$$\frac{P' \rightarrow P \quad P \{S\} Q \quad Q \rightarrow Q'}{P' \{S\} Q'}$$

Suppose we want to prove  $P' \{S\} Q'$ .

One way to do this is to find  $WP(S, Q)$  and then show that  $P' \rightarrow WP(S, Q)$ .

Rule of Sequential Composition

$$\frac{P \{S_1\} Q \quad Q \{S_2\} R}{P \{S_1; S_2\} R}$$

This rule means that if we can find ways to show correctness for the individual parts of a program, we can show it for the program as a whole.

Assignment

Suppose we have  $\{x := e\} P(x)$

where  $P(x)$  is a postcondition depending on  $x$ .

What has to be the precondition for  $P(x)$  to be true?

Answer:  $P(e)$

We find the weakest precondition by substituting  $e$  for  $x$  in  $P(x)$ , thus "undoing" the assignment.

Assignment

$(y > 0) \{x := y\} (x > 0)$

$(x - 1 > 0) \{x := x - 1\} (x > 0)$

$(1 > 0) \{x := 1\} (x > 0)$

True  $\{x := 1\} (x > 0)$

Assignment

Suppose we are asked to verify  $P \{S\} Q$ , and  $S$  is just an assignment statement.

From  $Q$ , we figure out the weakest precondition  $R$ .

If  $R$  turns out to be  $P$ , or if  $P \rightarrow R$ , then the program is correct.

The "proof" comes in determining  $R$ , which is not difficult for assignments. We might also have to provide an argument that shows that  $P \rightarrow R$ , using ordinary mathematical logic.

Conditionals

Suppose we want to prove the correctness of

*if* condition *then*  
 $S_1$   
*else*  
 $S_2$

with respect to precondition  $P$  and postcondition  $Q$ .

What we need to prove is

$P \{ \textit{if condition then } S_1 \textit{ else } S_2 \} Q$

Conditionals

The rule of inference is

$$\frac{(P \wedge \textit{condition}) \{S_1\} Q \quad (P \wedge \neg \textit{condition}) \{S_2\} Q}{P \{ \textit{if condition then } S_1 \textit{ else } S_2 \} Q}$$

Conditionals

The rule of inference is

$$\frac{(P \wedge \textit{condition}) \{S_1\} Q \quad (P \wedge \neg \textit{condition}) \{S_2\} Q}{P \{ \textit{if condition then } S_1 \textit{ else } S_2 \} Q}$$

If there is no *else* clause, the rule is

$$\frac{(P \wedge \textit{condition}) \{S_1\} Q \quad (P \wedge \neg \textit{condition}) \rightarrow Q}{P \{ \textit{if condition then } S_1 \} Q}$$

Conditionals

$$\frac{(P \wedge \textit{condition}) \{S_1\} Q \quad (P \wedge \neg \textit{condition}) \{S_2\} Q}{P \{ \textit{if condition then } S_1 \textit{ else } S_2 \} Q}$$

To prove: True  $\{ \textit{if } x < 0 \textit{ then } y := -x \textit{ else } y := x \} (y = |x|)$

**Conditionals**

$$\frac{(P \wedge \text{condition}) \{S_1\} Q \quad (P \wedge \neg\text{condition}) \{S_2\} Q}{P \{ \text{if condition then } S_1 \text{ else } S_2 \} Q}$$

**To prove:** True {if  $x < 0$  then  $y := -x$  else  $y := x$ } ( $y = |x|$ )

**Proof for then branch:**

$$\frac{(-x = |x|) \{y := -x\} (y = |x|)}{(\text{True} \wedge x < 0) \{y := -x\} (y = |x|)}$$

(assignment)  
(consequence, since  $x < 0 \rightarrow -x = |x|$ )

**Proof for the else branch:**

$$\frac{(x = |x|) \{y := x\} (y = |x|)}{(\text{True} \wedge x \geq 0) \{y := x\} (y = |x|)}$$

(assignment)  
(consequence, since  $x \geq 0 \rightarrow x = |x|$ )

Since  $x \geq 0$  is  $\neg(x < 0)$ , we have established what is required by the rule.

**Loops**

The complications here are:

- the loop may not terminate
- the body may be executed a number of times
- the body may be executed no times

Our approach:

- prove partial correctness (assume termination)
- try for a separate proof of termination

(there goes the dream of general purpose, automatic verification, since we can't solve the Halting Problem)

**Loops**

**To prove:**  $P \{ \text{while condition do } S \} Q$

We try to find a loop invariant  $I$  that holds after 0, 1, 2, ... iterations of the loop, and then we use the following inductive rule of inference:

If these three conditions hold:

$P \rightarrow I$  (this is the basis of the induction:  $I$  holds on entry to the loop)

$(I \wedge \text{condition}) \{S\} I$  (if  $I$  held after  $k$  iterations, it will hold after  $k+1$  iterations)

$I \wedge \neg\text{condition} \rightarrow Q$  (if  $I$  holds when the loop terminates, the postcondition  $Q$  holds)

then we may conclude:  $P \{ \text{while condition do } S \} Q$

**Loops**

**To prove:**  $P \{ \text{while condition do } S \} Q$

We try to find a loop invariant  $I$  that holds after 0, 1, 2, ... iterations of the loop, and then we use the following inductive rule of inference:

$$\frac{P \rightarrow I \quad (I \wedge \text{condition}) \{S\} I \quad I \wedge \neg\text{condition} \rightarrow Q}{P \{ \text{while condition do } S \} Q}$$

```

program multiply(m, n : integers)
    if n < 0 then a := -n
    else a := n

    k := 0
    x := 0

    while k < a
    {
        x := x + m
        k := k + 1
    }

    if n < 0 then product := -x
    else product := x
    
```

```

program multiply(m, n : integers)
    S1 { if n < 0 then a := -n
        else a := n
    }
    S2 { k := 0
        x := 0
    }
    S3 { while k < a
        {
            x := x + m
            k := k + 1
        }
    }
    S4 { if n < 0 then product := -x
        else product := x
    }
    
```



BUT:

There have been some successes; for example, it has been reported that the control logic of one Paris subway line was verified using logic of the type we have described.

Applying proof techniques to segments of a program makes it better.

Programming with the idea of verification in mind (whether we actually do the proof or not) makes us better programmers.

Program verification gives insights to designers of programming languages on how to create better languages.

[What would we consider significant progress?](#)

- Techniques that apply to programs with millions of lines
- Automated operation
- A useful number of all bugs found, say 10%, are found by automated methods

This was achieved by Microsoft Research for Windows Server 2003.

[Courses that discuss Program Verification](#)

CS156	Calculus of Computation	Prof. Manna
CS295	Software Engineering	Prof. Aiken