

Program correctness

“Computer Science is no more about computers than astronomy is about telescopes.” - E. W. Dijkstra

Now that we can prove things, wouldn't it be nice if we could prove things about our programs? Things like, do they work? Are they right? Trivial things like that.

1. A program is correct iff it produces the correct output for every possible input.
2. Therefore, one possible technique is to try out every possible input to see if each gives the right answer.
3. Unfortunately, that can take a long time. Imagine a program that, given a valid position in a game of chess, generates a list of legal moves. It's a straightforward problem, and easy to verify, but it has a large number of different inputs. There are, roughly, 10^{40} different legal positions in chess. If it takes, say, one hundredth of a second to generate output from input, it would still take 10^{38} seconds to test all of the inputs. This is roughly 10^{30} years. I don't know about you, but that is longer than I'm willing to wait.
4. Instead, we'll have to use some other technique to try to prove program correctness.
5. We break our proof into 2 parts:
 - (a) The answer is correct, if the program terminates. If a program meets this condition, it is called “partially correct.”
 - (b) The program always terminates.
6. We will introduce a new notation to indicate partial correctness: $p\{S\}q$ says that a piece of program S (called the segment) is partially correct with respect to the initial assertion p and the final assertions q .
 - (a) p is the set of properties that the program segment needs to be true. If S is an entire program, p is the input. If S is a program piece, p is the set of things true when the program gets to segment S .
 - (b) q is the set of properties that must be true after S runs, given p .

7. So far this is pretty simple. Show that the following is partially correct:

$$\begin{aligned} p &: x = 1 \\ S &: y \Leftarrow 2; \\ & \quad z \Leftarrow x + y; \\ q &: z = 3 \end{aligned}$$

Well, if x starts out as 1, and y is assigned 2, then z is assigned 1+2, or 3, therefore z is 3, and $p\{S\}q$ is true.

8. So here's the basic plan: we'll start by trying to prove that some property q is true at the end of the program, if some other property p is true at the start. We'll develop ways to break the program into smaller chunks: if property r is true and the end of chunk one and property s is true at the end of chunk two, then we'll know that property q is true at the end of the program. We just keep doing that until the chunks are so small they are trivially true. Let's see.

9. In order to prove things with this system, we need some rules of inference:

(a) Composition:

$$p\{S_1\}q \wedge q\{S_2\}r \rightarrow p\{S_1; S_2\}r$$

If we start with p being true, and at the end of S_1 q is true, *and* if q is true, then at the end of S_2 r is true, *then* we can conclude that if we start with p and run S_1 followed by S_2 , then at the end r will be true.

We use this when we want to prove that some $p\{S_1; S_2\}r$ is true. All we do is break the code into the two parts S_1 and S_2 , and prove that both $p\{S_1\}q$ and $q\{S_2\}r$ are true.

This enables us to split programs in to pieces, prove that the pieces are correct, and glue them back together.

For example, lets prove the previous code example. Let S_1 be $y \Leftarrow 2$; and S_2 be $z \Leftarrow x + y$;. To prove that $p\{S\}q$ is true, we can apply composition: prove $p\{S_1\}r$ and $r\{S_2\}q$ are true, where r is $y = 2 \wedge x = 1$. $p\{S_1\}r$ is true by the definition of $y \Leftarrow 2$; and since x was not changed. $r\{S_2\}q$ is true, by the definition of $+$, and the fact that $y = 2 \wedge x = 1$. There, we're done.

(b) Conditional. If we have a bit of code that says, if *condition* then S , we use the conditional rule:

$$[(p \wedge \text{condition})\{S_1\}q] \wedge [(p \wedge \neg \text{condition}) \rightarrow q] \rightarrow p\{S\}q,$$

where, $S = \mathbf{if\ condition\ then\ } S_1$

This says that if we're trying to prove a bit of code containing an if statement is correct, then we need to prove the body is correct, with the addition of the if-condition to the initial assertions, and that when the negation of the if-condition

is added to the initial assertions, this implies that the final assertions would still be true. That is, we check to see if it works either way. For example:

$$\begin{array}{l}
 p : \quad T \\
 S \left\{ \begin{array}{l} \mathbf{if } x > y \mathbf{ then} \\ S_1 : \quad y \leftarrow x \end{array} \right. \\
 q : \quad y \geq x
 \end{array}$$

Prove:

$$\begin{array}{l}
 T \wedge (x > y)\{y \leftarrow x\}(y \geq x)\square \quad \text{definition of assignment} \\
 T \wedge \neg(x > y) \rightarrow (y \geq x) \\
 (x \leq y) \rightarrow (y \geq x)\square
 \end{array}$$

Since p is True, there are no other initial assertions to worry about, and both x and y can be anything. If $x > y$, then we execute S , and y is assigned x . Thus, $y = x$, and $y \geq x$. If $\neg(x > y)$, then $x \leq y$, therefore $y \geq x$.

(c) Conditional with else. if *condition* then S_1 else S_2 :

$$(p \wedge \text{condition})\{S_1\}q \wedge (p \wedge \neg\text{condition})\{S_2\}q \rightarrow p\{S\}q.$$

$$\text{where, } S = \mathbf{if } \text{condition} \mathbf{ then } S_1 \mathbf{ else } S_2$$

This says that to prove partially correct two segments in an if-else statement, we can add the condition to the initial assertions and prove for the first segment, and separately add the negation of the condition to the initial assertions and prove the second segment:

$$\begin{array}{l}
 p : \quad T \\
 S \left\{ \begin{array}{l} \mathbf{if } x < 0 \mathbf{ then} \\ S_1 : \quad abs \leftarrow -x; \\ \mathbf{else} \\ S_2 : \quad abs \leftarrow x; \end{array} \right. \\
 q : \quad abs = |x|
 \end{array}$$

So we need to break this into 2 parts: $T \wedge x < 0\{abs = -x\}abs = |x|$ and $T \wedge (x \geq 0)\{abs = x\}abs = |x|$. The first is true, since if a number is < 0 , then its opposite is its absolute value. The second case is also true by the definition of absolute value.

Prove:

$$\begin{array}{l}
 T \wedge x < 0\{abs = -x\}abs = |x|\square \quad \text{definition of absolute value} \\
 T \wedge (x \geq 0)\{abs = x\}abs = |x|\square
 \end{array}$$

- (d) Loop invariants. In order to prove things about loops, we have to limit ourselves to certain kinds of loops. Fortunately, almost all of the loops we write of of this kind, or can easily be made into a loop of this kind. The property we need is the presence of a loop invariant. A loop invariant is something that is true before the loop starts, true when it ends, and is what we want to be true at the end of the loop (i.e. it performs the function of q from the earlier rules of inference). The basic rule is:

$$(p \wedge \textit{condition})\{S\}p \rightarrow p\{\mathbf{while} \textit{condition}S\}(\neg\textit{condition} \wedge p)$$

What his says, if we have an invariant that is true both before and after any one pass through the loop, then it will be true after all of the passes.

```

i = 1;
factorial = 1;
p : (factorial = i!) ∧ (i ≤ n)
  while (i < n)
S :   i ++;
        factorial = factorial * i;

```

First we must show that p is indeed a loop invariant. We do this with an inductive proof. In the base case, the first time time through the loop, $i = 1$, and $factorial = 1$, thus $factorial = i!$. For the inductive step: If, before any pass through the loop, $factorial = i!$ and $(i \leq n)$, then after the pass through the loop, $i_{new} = i + 1$, and $factorial_{new} = factorial \cdot (i + 1) = (i + 1)!$ since $i < n$ because of the loop condition, $i_{new} \leq n$. Therefore p is a valid invariant.

The next step is to ensure that the loops terminates (remember that?). Since i starts out $< n$, and i is increased each pass through the loop, eventually, $i \geq n$. Finally note, when $i = n$ then $(factorial = i!) \rightarrow factorial = n!$.

10. Final example. This following program takes 2 numbers n and m and calculates the

product nm by repeated addition. Prove it.

$$\begin{aligned}
 S_1 &= \left\{ \begin{array}{l} \mathbf{if } n < 0 \mathbf{ then} \\ \quad a = -n; \\ \mathbf{else} \\ \quad a = n; \end{array} \right. \\
 S_2 &= \left\{ \begin{array}{l} k = 0; \\ x = 0; \end{array} \right. \\
 S_3 &= \left\{ \begin{array}{l} \mathbf{while } k < a \\ \quad x = x + m; \\ \quad k ++; \end{array} \right. \\
 S_4 &= \left\{ \begin{array}{l} \mathbf{if } n < 0 \mathbf{ then} \\ \quad product = -x; \\ \mathbf{else} \\ \quad product = x; \end{array} \right.
 \end{aligned}$$

$$p : integer(m) \wedge integer(n)$$

$$q : p \wedge (a = |n|)$$

$p\{S_1\}q$ is true by the definition of absolute value

$$r : q \wedge (k = 0) \wedge (x = 0)$$

$q\{S_2\}r$ is trivially true

$$s : x = mk \wedge (k \leq a) \wedge q$$

s is an invariant because A) $0 = m0$ and B) if $x = mk$, then $k_{new} = k + 1$, and $x_{new} = x + m = m(k + 1)$. Since at the end of the loop, $k = a$, the loop terminates with $x = ma$, and $r\{S_3\}s$ is true.

$$t : product = mn$$

$s\{S_4\}t$ is true, because $x = ma$ and $a = |n|$, by the definition of absolute value.