Invariant Assertions for Program Proofs

Mark Ardis
Rose-Hulman Institute of Technology
September 2005

There are a few different methods for formally proving that a program agrees with its specification. The first method we will examine is the *Invariant Assertion* method. The method was first proposed for flowcharts by Robert Floyd [Floyd 67] and then adapted for program code by Tony Hoare [Hoare 69]. Edsger Dijkstra developed another variant of the method a few years later [Dijkstra 75], [Dijkstra 76].

Invariant assertions are statements that are always true (hence the name *invariant*), no matter what values are taken by the variables in the program. Thus, the text of the program can be annotated with these assertions, for they will be true whenever control passes through them. Some programming languages, like C, even have special constructs for including assertions and checking their validity during execution. We will write assertions within comments in this class.

Here is an example of an assignment statement with assertions before and after:

```
        /* n == 0 */
i = 0;
        /* n == i */
```

The first assertion (`/* n == 0 */`) is called the *pre-condition* of the statement, and the second assertion is called the *post-condition*. We can also refer to pre-conditions and post-conditions of blocks of statements.

The invariant assertion method of proof consists of three steps:

1. Write the specification of a program in terms of a pre-condition and post-condition for the program.
2. Show that the program preserves those assertions by applying rules of inference to further annotate the program.
3. Show that any loops or recursive function calls always terminate.

If only the first two steps are performed the program is said to have been proven *partially correct*. It is not deemed *totally correct* unless all three steps are performed. As we will see, some of the steps of this method are fairly mechanical (plug-and-chug), while others require some insight into the meaning of the program. Still other steps do not require any special knowledge of the program, but they do require some facility with logical inference.

Rules of Inference

For each programming language construct there is one rule that explains its meaning, or semantics. There are also rules for composing statements and assertions. We will use only assignment statements and 3 control structures (if-then, if-then-else, and while) in this part of the course. Later we will look at procedure calls.

Here is the first rule:

Statement	Rule	Informal Meaning

Assignment	`/* P(E) */`
	`V = E;`
	`/* P(V) */`
		Whatever is true about the expression "E" before the assignment statement is true about the variable "V" afterward.

The assignment rule allows us to reason about assignment statements. The pre-condition is a predicate `P` about the expression `E` that appears on the right-hand side of the statement. The post-condition is the same predicate `P` about the variable `V` that appears on the left-hand side of the statement. In the example we saw earlier:

```
          /* n == 0 */
i = 0;
          /* n == i */
```

the predicate `P(x)` is `"n == x"`. The first instance of the predicate in the example is `P(0)`, and the second instance is `P(i)`. One way of reading this example is to say that if `n == 0` holds when we reach the assignment statement, then `n == i` will be true after execution of the statement.

The next rule is a little more complicated.

Statement	Rule	Informal Meaning

If-then-else	/* P1 */
	S1
	/* Q */
	and
	/* P2 */
	S2
	/* Q */
	
	imply
	
	/* B => P1 AND
	NOT B => P2 */
	if (B) {
	S1
	} else {
	S2
	}
	/* Q */
		If the body of the then-clause (S1) satisfies its pre- and post-conditions, and if the body of the else-clause (S2) satisfies its pre- and post-conditions, then the if-statement satisfies the given pre- and post-conditions.
		
		The pre-condition for the body includes "B", since it is only executed when "B" holds. Similarly, the pre-condition for the else-clause includes "NOT B". The pre-condition for the if-statement does not include either "B" or "NOT B", since either path may be executed.

This rule subdivides the problem of reasoning about an if-then-else statement into two new problems: 1) show that the body of the then-clause satisfies some pre- and post-condition, and 2) show that the body of the else-clause satisfies a related set of pre- and post-conditions. If both proofs hold, the rule draws a conclusion about the entire if-statement.

Suppose we had proven:

```
     /* x + y <= 2 * x < 10 */
z = 2 * x;
     /* x + y <= z < 10 */
```

and we had proven:

```
     /* x + y <= 2 * y < 10 */
z = 2 * y;
     /* x + y <= z < 10 */
```

Then we would be able to conclude:

```
        /* (x > y) => (x + y <= 2 * x < 10) AND
           (x <= y) => (x + y <= 2 * y < 10) */
if (x > y) {
   z = 2 * x;
} else {
   z = 2 * y;
}
        /* x + y <= z < 10 */
```

Here is how the pieces match up:

```
P1:     x + y <= 2 * x < 10
P2:     x + y <= 2 * y < 10
Q:      x + y <= z < 10
B:      x > y
NOT B:  x <= y
```

The rule for if-then statements is similar, except that there is no else clause.

Statement	**Rule**	**Informal Meaning**

If-then	`/* P AND B */`
	`S`
	`/* Q */`
	
	and
	
	`(P AND NOT B) => Q`
	
	imply
	
	`/* P */`
	`if ( B ) {`
	`  S`
	`}`
	`/* Q */`
		If the body of the then-clause satisfies its pre- and post-conditions, and if P AND NOT B implies Q, then the if-statement satisfies the given pre- and post-conditions.
		
		The pre-condition for the body includes "B", since it is only executed when "B" holds.
		The pre-condition for the if-statement does not include "B", since either path may be executed.

Note that there are still two sub-problems to solve: one for the then-clause and one for the non-existent else-clause. If both can be solved then the rule draws a conclusion about the entire if-statement.

The while rule is similar to the if-then rule, but it has to handle the cases when the loop body is executed more then once.

Statement	**Rule**	**Informal Meaning**

While	`/* P AND B */`
	`S`
	`/* P */`
	
	and
	
	`P AND NOT B => Q`
	
	imply
	
	`/* P */`
	`while ( B ) {`
	`S`
	`}`
	`/* Q */`
		If the body of the loop satisfies its pre- and post-conditions, and if P AND NOT B implies Q, then the while-statement satisfies the given pre- and post-conditions.
		
		The pre-condition for the body includes "B", since it is only executed when "B" holds.
		The pre-condition for the while-statement does not include "B", since the loop may not be executed in some cases.
		The post-condition of the loop body only contains "P", the loop invariant. For some iterations it may maintain "B", but it must eventually yield "NOT B".

The while rule uses a predicate `P` that remains true for any number of iterations of the loop. For that reason `P` is called the *loop invariant*. `P` should be true when the while statement is first encountered, it should be true after each iteration of the loop, and it should be true after the while statement is finished. Because we also know that the loop condition `B` will be false when the loop finishes, we can strengthen the post-condition of the while statement to include it.

Finally, we have one rule for composing proofs.

Statement	Rule	Informal Meaning
Composition	/* P1 */ S1; /* P2 */ **and** /* P3 */ S2; /* P4 */ **and** P2 => P3 **imply** /* P1 */ S1; /* P2 */ /* P3 */ S2; /* P4 */	If the post-condition of S1 implies the pre-condition of S2 then the proofs can be glued together.

The composition rule formalizes the observation that an assertion that appears above another must imply it logically.

Termination

In order to prove total correctness of a program with its specification, you must show that the program always terminates. There are two programming language elements that could cause trouble: loops and recursive functions. We will defer recursive functions until we build up some notation and rules for functions.

How can you show that a loop always terminates? You must identify some expression, called a *bound function*, whose value:

- changes after each iteration of the loop
- always changes in the same direction
- is bounded by the loop condition

If the expression gets bigger each time through the loop then we say that it is *strictly increasing*, and if it gets smaller each time then we say that it is *strictly decreasing*.

In order to find a bound function you need to first examine the loop condition to see what it can bound. Whatever variables appear in the condition are potential components of the bound function. Next you need to examine the body of the loop to see which variables change every time through the loop. Finally, you need to compose an expression consisting of variables that change and appear in the loop condition. You may need to combine several variables in order to get something that is strictly increasing or strictly decreasing.

Additional Notation

It helps to have some additional notation for describing arrays and simple functions. We will use the expressions in the table below in some of our proofs.

Expression	**Meaning**
`a[s: e]`	The sequence of elements `a[s], ..., a[e]` If `e > s` then the sequence is empty.
`SUM(a[i]: low <= i <= high)`	The sum of all elements `a[i]`, where `i` takes on values `low, ... high`.
`IN(x, a[s: e])`	A predicate that returns true if and only if the value `x` appears in one of the elements a[s], ..., a[e]

Constructing Proofs of Programs

Just as with natural deduction proofs, invariant assertion proofs of programs are developed bottom-up (or backwards from the end). Starting with the post-condition for the last statement in the program, you apply rules of inference to calculate the pre-condition of each statement. Eventually you should reach the pre-condition for the very first statement. If your calculated pre-condition is implied by the pre-condition for the program, then you have constructed the partial proof of the program.

Then, for each loop, you construct a bound function and demonstrate that it obeys all three properties. That is, either you show that it is strictly increasing and bounded above by the loop condition, or you show that it is strictly decreasing and bounded below.

Example

Here is a simple program that calculates the product of two integers, one of which must be greater than zero.

```
            /* b > 0 */

r = 0;
c = b;
while (c > 0) {
   r = r + a;
   c = c - 1;
}

            /* r == a * b */
```

The first step is to rewrite the post-condition for the program into a post-condition appropriate for the while loop. The inference rule for while loops requires that the post-condition be in the form `P AND (NOT B)` where `P` is the loop invariant and `B` is the loop condition. In this case `B` is `c <= 0` .

From informal study of the loop we can determine that `c` will be equal to `0` at the end of the loop. So, we rewrite `c == 0` as two inequalities: `c >= 0` and `c <= 0` . Note that the second inequality is `NOT B` . The line shown in bold is the new assertion.

```
          /* b > 0 */

r = 0;
c = b;
while (c > 0) {
   r = r + a;
   c = c - 1;
}

          /* r == a * b  AND c >= 0 AND c <= 0 */
          /* r == a * b */
```

Next we try to guess the loop invariant `P` . Again, informal study of the loop tells us that `r` is accumulating increments of `a` each time. The number of increments is the difference between `b` and `c` . So, we rewrite the post-condition to use that fact.

```
        /* b > 0 */

r = 0;
c = b;
while (c > 0) {
   r = r + a;
   c = c - 1;
}
        /* r == a * (b - c) AND c >= 0 AND c <= 0 */
        /* r == a * (b - c) AND c == 0 */
        /* r == a * b */
```

Now we have a post-condition in the form `P AND (NOT B)`. We insert the loop invariant `P` in all the places it should appear according to the rule of inference for while loops. Note that the pre-condition for the body of the loop includes the loop condition.

```
        /* b > 0 */

r = 0;
c = b;
        /* r == a * (b - c) AND c >= 0 */
while (c > 0) {
        /* r == a * (b - c) AND c >= 0 AND c > 0 */

   r = r + a;
   c = c - 1;
        /* r == a * (b - c) AND c >= 0 */
}
        /* r == a * (b - c) AND c >= 0 AND c <= 0 */
        /* r == a * (b - c) AND c == 0 */
        /* r == a * b */
```

We must now show that the loop invariant is preserved by the body of the loop. The first step is to use the rule for assignment statements to derive a pre-condition for the last statement in the loop. We find all instances of the variable `c` in the post-condition and replace them with the expression `c - 1` that appears on the right-hand side of the assignment statement.

```
        /* b > 0 */

r = 0;
c = b;
        /* r == a * (b - c) AND c >= 0 */
while (c > 0) {
        /* r == a * (b - c) AND c >= 0 AND c > 0 */

   r = r + a;
        /* r == a * (b - (c-1)) AND (c-1) >= 0 */
   c = c - 1;
        /* r == a * (b - c) AND c >= 0 */
}
        /* r == a * (b - c) AND c >= 0 AND c <= 0 */
        /* r == a * (b - c) AND c == 0 */
        /* r == a * b */
```

We next calculate the pre-condition for the first statement in the loop. We replace all occurrences of the variable `r` with the expression `r + a`.

```
        /* b > 0 */

r = 0;
c = b;
        /* r == a * (b - c) AND c >= 0 */
while (c > 0) {
        /* r == a * (b - c) AND c >= 0 AND c > 0 */

        /* (r+a) == a * (b - (c-1)) AND (c-1) >= 0 */
   r = r + a;
        /* r == a * (b - (c-1)) AND (c-1) >= 0 */
   c = c - 1;
        /* r == a * (b - c) AND c >= 0 */
}
        /* r == a * (b - c) AND c >= 0 AND c <= 0 */
        /* r == a * (b - c) AND c == 0 */
        /* r == a * b */
```

The pre-condition needs to be simplified in order to see whether it is implied by the loop invariant and the loop condition. We can remove the inner parentheses in the first clause, remembering that a double negative is positive.

```
        /* b > 0 */

r = 0;
c = b;
        /* r == a * (b - c) AND c >= 0 */
while (c > 0) {
        /* r == a * (b - c) AND c >= 0 AND c > 0 */
        /* (r+a) == a * (b - c + 1) AND (c-1) >= 0 */
        /* (r+a) == a * (b - (c-1)) AND (c-1) >= 0 */
   r = r + a;
        /* r == a * (b - (c-1)) AND (c-1) >= 0 */
   c = c - 1;
        /* r == a * (b - c) AND c >= 0 */
}
        /* r == a * (b - c) AND c >= 0 AND c <= 0 */
        /* r == a * (b - c) AND c == 0 */
        /* r == a * b */
```

The first clause is implied, as it is the same as the loop invariant clause with `a` added to each side of the equality. The second clause is implied by the two other clauses available. So, the loop invariant was preserved and we can use it to calculate the pre-condition of the assignment statement immediately preceding the loop.

We replace each occurrence of `c` with `b`.

```
        /* b > 0 */

r = 0;
        /* r == a * (b - b) AND b >= 0 */
c = b;
        /* r == a * (b - c) AND c >= 0 */
while (c > 0) {
        /* r == a * (b - c) AND c >= 0 AND c > 0 */
        /* (r+a) == a * (b - c + 1) AND (c-1) >= 0 */
        /* (r+a) == a * (b - (c-1)) AND (c-1) >= 0 */
   r = r + a;
        /* r == a * (b - (c-1)) AND (c-1) >= 0 */
   c = c - 1;
        /* r == a * (b - c) AND c >= 0 */
}
        /* r == a * (b - c) AND c >= 0 AND c <= 0 */
        /* r == a * (b - c) AND c == 0 */
        /* r == a * b */
```

Finally, we calculate the pre-condition of the first statement in the program, replacing every occurrence of `r` with `0`.

```
        /* b > 0 */

        /* 0 == a * (b - b) AND b >= 0 */
r = 0;
        /* r == a * (b - b) AND b >= 0 */
c = b;
        /* r == a * (b - c) AND c >= 0 */
while (c > 0) {
        /* r == a * (b - c) AND c >= 0 AND c > 0 */
        /* (r+a) == a * (b - c + 1) AND (c-1) >= 0 */
        /* (r+a) == a * (b - (c-1)) AND (c-1) >= 0 */
   r = r + a;
        /* r == a * (b - (c-1)) AND (c-1) >= 0 */
   c = c - 1;
        /* r == a * (b - c) AND c >= 0 */
}
        /* r == a * (b - c) AND c >= 0 AND c <= 0 */
        /* r == a * (b - c) AND c == 0 */
        /* r == a * b */
```

We need to simplify the pre-condition to see if it is implied by the pre-condition for the program. The expression `0 == 0` is always true, and we know that `True AND P` is equal to `P`.

```
        /* b > 0 */
        /* b >= 0 */
        /* 0 == a * (b - b) AND b >= 0 */
r = 0;
        /* r == a * (b - b) AND b >= 0 */
c = b;
        /* r == a * (b - c) AND c >= 0 */
while (c > 0) {
        /* r == a * (b - c) AND c >= 0 AND c > 0 */
        /* (r+a) == a * (b - c + 1) AND (c-1) >= 0 */
        /* (r+a) == a * (b - (c-1)) AND (c-1) >= 0 */
   r = r + a;
        /* r == a * (b - (c-1)) AND (c-1) >= 0 */
   c = c - 1;
        /* r == a * (b - c) AND c >= 0 */
}
        /* r == a * (b - c) AND c >= 0 AND c <= 0 */
        /* r == a * (b - c) AND c == 0 */
        /* r == a * b */
```

The program pre-condition implies the pre-condition we calculated, so the program is partially correct with respect to the given pre- and post-conditions.

The last step is to show termination. There is only one loop. The loop condition refers only to the variable `c`, which appears to be strictly decreasing. Since there is only one path through the loop, and that path includes the statement `c = c - 1;`, we can assert that `c` is strictly decreasing. The loop condition bounds `c` below, so the loop terminates.

Note that the process of proving partial correctness is a combination of applying rules and of manipulating expressions into the forms that rules require. Probably the hardest step was the discovery of a loop invariant. Because this step is so difficult, most of your exercises will include loop invariants where needed.

References

[Dijkstra 75] Dijkstra, E.W., Guarded commands, nondeterminancy and the formal derivation of programs. *CACM 18(8)*, 453-457, August 1975.

[Dijkstra 76] Dijkstra, E.W., *A Discipline of Programming*. Prentice Hall, 1976.

[Floyd 67] Floyd, R., Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, XIX American Mathematical Society, 19-32, 1967.

[Hoare 69] Hoare, C.A.R., An axiomatic basis for computer programming. *CACM 12(10)*, 576-580, October 1969.