# Finding Loops Invariants by a Backward Method Using Inductive Assertions and Proving them Correct Using Mathematical Induction

Ramon A. Mata-Toledo[64]

**Abstract**

The method of proving programs correct has been an endeavor of computer scientists for decades now. However, little progress, in comparison, with some other aspects of computing, has been made on this respect. The current method using Hoare' Triple and the Dikjstra' pre- and post-conditions are not easy to follow by most students. The method of Inductive assertions has been tried too to find loop invariants using the so-called 'forward method." The author has found that this 'forward method" is still difficult and, for some program, even more difficult to follow than the Hoare's Triple and Dijkstra' pre and post conditions. In this paper the author proposes a "backward method" using Inductive assertion which starts at the end of the program and works its way up to the beginning of the program. This method has been proven successful the author's classes where the student has found the method easier to use and understand.

**Keywords:** inductive assertions, Hoare triple, Dijkstra conditions

## 1. Introduction

Because an incorrect program can lead to disastrous results, a large amount of methodology has been constructed trying to verify the correctness of the programs. Ideally, we would like to write program the ways engineers can put a building, it is not necessary to put it up first to verify that it would stay that way. During the last decades there here have been many attempts to write programs to emulate the way the engineers design a building. However, whereas engineers have been putting up building for centuries, computer programmers have been writing programs for a few decades now, so the science is not mature enough to be able to emulate the work centuries of the engineering experience. All these programming attempts generally fall under the umbrella of program verification or proof of correctness. Many approaches have been used such as Hoare's Triples and Dijkstra Pre and Post conditions [1, 2] Additional efforts have been devoted to automating program verification so that it can be carried out using a computer. However, only limited progress has been made toward this goal. Indeed, some mathematicians and theoretical computer scientists argue that it will never be realistic to [automate] the proof of correctness of complex programs." [3]

## 2. Defining the Correctness of a program

[64] Ph.D., Rollins College, U.S.A. - rmatatoledo@rollins.edu

A program is said to be **correct** if it produces the correct output for every possible input. The usual way of proving a program correct is subjecting it to intensive testing. Notice that definition of correctness by testing the program using input values seem futile even for a simple program like adding two integer numbers. How many pair of integers can be used to be sure that the program works correctly? If we represent each integer in its 2's complement notation and use 32 bits internally for their representation, then we will need almost $2^{64}$ possible combinations to test that the addition produces the right results. Even with a very fast computer this would take an excessive amount of time. So, obviously, a better way is needed to even prove that a program as the one just mentioned is correct. It is the purpose of this document to use Inductive Assertions [1] along with the method of Mathematical Induction to prove small loops correct. Although the programs used as examples seem elementary, it is the method used what the author is trying to explain.

## 3. How to prove a loop correct?

To prove a program that uses a loop to perform a task is necessary to show that:

- First, that the program produces the correct answers (partial correctness).
- Second, prove that the program terminates. (Loop termination)

The author will use the method of Mathematical Induction to prove partial correctness once the invariant has been determined [5,6]. Next, he will use the Archimedean Principle applied to integer numbers for loop termination. All problems presented here only use variables and constant of type integer, hence the adaptation of the Archimedean Principle only to integer numbers instead of reals [3]. Although the programs used as examples could be made more complex by the addition of conditional of different types within the loop, we have avoided them because the author is interested in showing how an invariant can be found and proved. In a later paper, the author intends to show that the methods explained here also wells with conditional statements of different types within a loop.

## 4. What is needed to prove a loop?

The key to proving a loop is discovering the "loop invariant", although most authors use Tony Hoare's definition of a loop invariant [1], we will define a "**loop invariant**" as an "*expression that it is true when the execution of the program reaches the loop for the very first time and true when the loop reaches its last statement and goes back to test the termination condition of the loop.*" Somewhere, between the first statement and the last one, the loop invariant may be false but what is important is that is true at the beginning and at the end of the loop. Notice that proving that the loop terminates is a different task that needs to be performed following the proof of partial correctness. The notation of Tony Hoare's assumes all what I said about the partial correctness of a loop, but it is presented more

formally using the notion of triples (his notation) and an inference-rule like notation which student seem not to understand very well.

## 5. How do I go about finding and the proving the loop invariant?

There are several questions and actions that we need to ask and perform to find the loop invariant of a loop. These questions and actions are:

1) What does the program do? That is, what is to be produced when the program ends. Express this result in terms of the outputs of the program.
2) What variable holds the output of the program?
3) What variable controls execution of the loop?
4) What is the final value of this variable?
5) Find an expression that ties up the variable that holds the final result and the one that control the loop. This expression should be such that when you substitute the final variable of the variable that controls the loop produces the expression that you determined in step 1
6) See what variables change within the loop and add the necessary subscripts to prove using mathematical induction the expression of step 1.
7) Once the partial correctness of the loop is found, show that the loop ends using the Archimedean Property adapted for integer numbers.

## 6. What is the Archimedean's Property?

According to Reference [4], the Archimedean property is a theorem for the real number system which modified for the integer numbers may be stated as follows:

"If $x > 0$ and if y is an arbitrary integer number, there exists another positive integer n such that $n*x > y$"

What this property means in geometric terms is that any line segment (y), no matter now long, may be covered by a finite number of line segments of a given positive length. In other words, a small ruler (x) used often enough (n) can measure arbitrarily large distances (y). Archimedes realized that this was a fundamental property of the straight line and stated it explicitly as one of the axioms of geometry.

For us, when proving a loop, the application of the Archimedean property can be applied as follows: Let's say that a loop is controlled by a Boolean condition involving a variable M such as ($M < 100$). In this case, the current value of M is what controls the execution of the loop; by extension we will say that the variable M controls the loop. Let's also assume that M is increased by a fixed positive amount every time the loop is executed. If this is the case, then, M eventually, assuming no other errors will stop the execution of the program, will

reach the value of M or surpass it. This last statement depends on whether M is increased by 1 or some other positive quantity. In this document, we will assume that the Archimedean property has been proved true elsewhere as shown in Reference 3.

## 7. Two Examples

For this first example we will take a program snippet from Reference No. 1 to find the loop invariant, its proof, and the loop termination. The programs are presented in pseudo-code using # to indicate inline comments. The statements in the snippet are basically self-explanatory.

***Example No. 1***

Find the loop invariant to prove that the program snippet shown below computes the $n^{th}$ power of real number x for a given positive integer n.

```
input n # a positive integer

input x # a real number

power := 1

i := 1

while i ≤ n ← I

  power := power * x

  i  := i  + 1

endwhile
```

Following the steps indicated in Section 5 of this document we will have that:

1) the first question to ask is what does the program do? In this case the answer is trivial because the program snippet calculates $x^n$.
2) The output of the snippet is hold by the variable *power*. That is, at the end of the program power = $x^n$.
3) What variable controls the execution of the loop? Here, the variable *i* is used to determine the number of iterations of the loop.

4) What is the final value of *i,* the variable that controls the loop? Whatever the value of *n* is, the loop will terminate when $i > n$. Because *i* is incremented by 1 every time the loop is executed its final is $i = n + 1$.
5) We now need to find an expression that involves the variable that holds the final result, *power*, and the final value of *i*, the variable that controls the loop, *i*, in such a way that, when we replace *i* by its final value, we obtain the output of the program. We claim that the invariant is: power = $x^{i-1}$.

   Notice that if we replace in the previous expression *i* by its final value *n*+1, we obtain the output of the program. That is,

$$power = x^n$$

6) The variables that change within the loop and which are of our interest are *power* and *i*. Therefore, let's add the corresponding subscripts and claim that the invariant is:

$$power_n = x^{i_n}$$

We will prove this invariant using Mathematical induction on *n*. The invariant must be true the first time the execution of the program reaches the top of the loop. This is indicated in the snippet with (I). Therefore, for $n = 0$ (the basis of induction) we will have the following:

$power_0 = 1^0 = 1$ (Notice that by the assignment statement *power* = 1 and any real raised to 0 is equal to the unity)

Therefore, the invariant is true the first-time execution reaches the top of the loop.

Now as the Hypothesis of induction, we will assume that $n = k$. We will interpret this as having gone through the loop *k* times already. That is, we are assuming that $power_k = x^{i_k}$ is true. (II)

To prove that the invariant is true for $n = k + 1$, we will have to assume that we go around the loop "one more time."

When we do this, we find the following conditions based upon how assignment statements work in programming, that is, the new value of *power* is its "old value" multiplied by x.

$$power_{k+1} = power_k * x. \text{ (III)}$$

Replacing (II) in (III) and using the exponential laws we have that

$$power_{k+1} = x^{i_k} * x = x^{i_k+1} \text{ (IV)}$$

However, within the loop we also have that $i_{k+1} = i_k + 1$. (V). Again, the new value of i is its old value increased by 1.

Knowing this and replacing (V) into (IV) we obtain that

$$power_{k+1}, = x^{i_{k+1}}$$

Which proves that the invariant is true.

Now we need to prove that the loop terminates. To do this we need to apply the Archimedean Property noticing that every time we go through the loop, the variable *i* is incremented by 1. Therefore, eventually, according to the Archimedean Property *I* will reach a value that surpass the value of *n*. Notice that it is not necessary to know what the initial value of n is, except that is a positive value.

***Example No. 2***
**(**from Reference No.3**)**

Prove that the following program snippet computes the value of M! for (M > 1).

```
input M  # read positive value M

i := 2

j := 1

while i ≤ M

  j := j * i

  i := i + 1

endwhile

print j
```

The program snippet asks for the calculation of M! and from the snippet itself we can see the j = M! at the end of the program. Therefore, the first two questions are already answered.

The variable that controls the loop execution is i and its final value is i = M + 1 because the loop executes as long as i ≤ M.

Knowing which variable holds the result, j, and the variable that controls the loop, i, we need to look for an expression such that when we replace i by its final value produces the factorial that the program calculates.

Following similar reasoning as on Example No. 1 we can claim that the invariant is $j_n = (i_n - 1)!$ which we need to prove using Mathematical Induction. Notice that if we replace i by its final expression, we get $J_n = M!$

According to the conditions of the program, the first-time execution reaches the top of the loop we have that j = 2 and i = 2. Therefore, the basis of induction for n = 0 is as follows:

$$j_0 = ((i_0 - 1)! = (2\text{-}1)! = 1$$

So, the basis of induction is true.

The Hypothesis of Induction $j_k=(i_k\text{-}1)!$ assumes that we have executed the loop k times. Let's call this expression (I)

To prove that the invariant is true for n = k + 1 we need to go around the loop one more time. Inside the loop we have that the new values of j and i based on the definition of an assignment statement is

$$j_{k+1} = j_k * i_k \quad \text{(II)}$$

And $i_{k+1} = i_k + 1$ (III)

Replacing (I) in (II) we have that $j_{k+1}=(i_k\text{-}1)!*i_k$

According to the definition of factorial we have $j_{k+1} = i_k!$ (IV)

However, from expression (III) we have that $i_k - 1 = i_{k+1}$ (V). Now replacing (V) into (IV) we have that

$$j_{k+1} = (i_{k+1}\text{ -}1)!$$

This final step shows that the invariant is correct.

To prove loop termination, we use again the Archimedean Property. Observe that every time that the loop is executed, the variable i is incremented by 1. Therefore, eventually, i will reach the reach and surpass the value of M.

## 8. Conclusion

These two small examples illustrate that the inductive assertion is easier to initially figure out a possible loop invariant if we work "backwards" starting from the end of the program and reaching the first statement of the program snippet. The fact that Mathematical Induction can be used to prove an invariant is a much powerful tool than any testing that can be done no matter how many numbers we try on.

## References

1. K. H. Rosen. Discrete Mathematics and its Applications.7th Ed. McGraw-Hill, 2012.
2. E. W. Dijkstra. A Discipline of Programming. Prentice Hall, 1976
3. R.B. Anderson. Proving Programs Correct. John Wiley & Sons, 1979.
4. T. M. Apostol. Calculus, Vol. 1. 2nd Ed. John Wiley & Son,1967
5. D. M. Burton. Elementary Number Theory. 5th Ed. McGraw-Hill, New York, 2002.
6. G. Doroféiev et al. Selected Topics of Elementary Mathematics. MIR Publishers, Moscow, 1973.