Loop Invariants

Overview

A loop invariant is a condition that is necessarily true immediately before and immediately after each iteration of a loop. (Note that this says nothing about its truth or falsity part way through an iteration.)

For example, in Java, a `while` loop has the following form, where `B` is a boolean expression (that we shall call the **guard** of the loop) and `S` is a sequence of commands/instructions (that we shall call the **body** of the loop).

```
while (B)
   { S }
```

A flow chart that indicates the steps taken in executing such a loop is as follows:

```
                        +-------+
                        | start |
                        +-------+
                            |
                            | <--- Q (the precondition) holds here (assumption)
                            |
+-------------------------> | <--- P (the invariant) holds here (every time)
|                           |
|                           |
|                          / \
|                         /   \
|                 true   /  B  \   false
|               +<------ \     / -------->+
|    P & B ---->|         \   /           |
|    holds      |          \ /            | <---- P & !B  holds here
|    here       v                         |
|             +---+                       | <---- R (postcondition)
|             | S |                       v       is to hold here
|             +---+                    +------+
|               |                      | stop |
|               |                      +------+
+<--------------+
```

Note: In what follows, we assume that evaluation of `B` has no "side effects" (i.e., results in no changes to values of variables in the program). **End of note.**

The flow chart is annotated to indicate the "locations" at which

1. we assume that the precondition Q holds,
2. we want the postcondition R to hold, and
3. we want the loop invariant P to hold.

It also indicates locations at which `P&!B` holds and at which `P&B` holds, assuming that `P` holds every time execution reaches the "critical location" on the graph (i.e., the spot immediately above the diamond-shaped box indicating the evaluation of `B`). It is easy to choose a `P` that is an invariant of the loop, i.e., that always holds at the critical location. (For example, choosing `P` to be **true** works.) However, it is not always easy to choose a `P` that, in addition, is useful in demonstrating that `R` must hold when the loop terminates. To be useful, the invariant `P` that we seek should be such that

```
P & !B ==> R
```

(i.e., the truth of `P` together with the falsity of the loop's guard, `B`, guarantees the truth of `R`). Why? Because then we can be sure that, when the loop terminates (at which point `P&!B` must hold), `R`, too, must hold, as desired.

How do you show that an invariant really is one?
A question that may have occurred to you is this: How do you show that a chosen `P` is actually true every time execution reaches the critical location? Usually, it is done as follows:

First, you show that `Q ==> P` (i.e., the truth of the precondition guarantees the truth of `P`), from which you may conclude that `P` holds the first time execution reaches the critical location.

Second, you show that `P` holds each subsequent time the critical location is reached. To accomplish this, it suffices to show that, if `S` (the loop body) is executed beginning in a state in which `P&B` holds, then `P` will necessarily hold when `S` is finished executing. (Of course, this corresponds to showing that `{P & B} S {P}` is a valid Hoare Triple.)

In effect, this amounts to a proof by mathematical induction on the number of loop iterations executed. The first step is the base case, which shows that `P` holds the first time execution reaches the critical location (i.e., after zero iterations). The second step is the inductive step, in which it is shown that, if, for any positive number *n*, `P` holds the *n*-th time execution reaches the critical spot (i.e., after the *(n-1)*-st iteration), then it must also hold the (*n+1*)-st time (i.e., after the *n*-th iteration).

How do you show that a loop terminates?

To show that a loop terminates after finitely many iterations, we show that each iteration *makes progress towards termination* in some way. Technically, this amounts to showing that there is some integer function of the program variables that

1. decreases on each iteration and
2. is such that the function's value must be greater than zero in order for another loop iteration to occur (or, equivalently, the truth of `P&B` guarantees that the function's value is greater than zero).

Note that, at the beginning of each loop iteration, the value of such a function is an upper bound on the number of iterations remaining to be executed.

For example, take the following loop:

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

Then, taking *g* to be the function defined by the expression `n-i`, we have that the value of *g* decreases on each iteration (due to the fact that `i` increases on each iteration but `n` stays the same). We also have, as required, that the truth of the loop guard, `i<n`, guarantees that the value of *g* exceeds zero.

To illustrate the usefulness of the loop invariant concept, we discuss a few amusing puzzles.

Red and Blue Marbles in a Jar:

Suppose you have a jar of one or more marbles, each of which is either RED or BLUE in color. You also have an unlimited supply of RED marbles off to the side. You then execute the following "procedure":

```
while (# of marbles in the jar > 1)  {
   choose (any) two marbles from the jar;
   if (the two marbles are of the same color)  {
      toss them aside;
      place a RED marble into the jar;
   }
   else {   // one marble of each color was chosen
      toss the chosen RED marble aside;
      place the chosen BLUE marble back into the jar;
   }
}
```

By examining the procedure, you can easily verify that the number of marbles in the jar decreases by exactly one on each iteration of the loop. Thus, if initially the jar contained N marbles, then after N-1 iterations of the loop, exactly one marble will remain. What we have just done is to make a persuasive, but informal, argument that the loop terminates after finitely many iterations. To be a bit more precise, we could explicitly define *g* to be the function that, applied to a jar of marbles, yields the number of marbles in that jar. Then we would argue that the value of *g* decreases on each iteration of the loop, and, furthermore, that if the loop guard holds (which is to say that there are at least two marbles in the jar), *g*'s value is greater than zero.

Suppose we know how many RED and BLUE marbles were in the jar, initially. Armed with that information, can we predict, with certainty, the color of the last marble remaining in the jar? (Note that, by virtue of the assumption that the jar was initially non-empty and the fact that the number of marbles decreases by exactly one on each iteration, it is not possible to end up with an empty jar.)

To state it more formally:

Does there exist a function *f*: N × N --> {BLUE, RED} (meaning that the domain of *f* is the set of ordered pairs of natural numbers and the range is the set {BLUE, RED}) that satisfies the following condition:

> For all *K* and *M* (such that at least one of them is non-zero), if we begin with *K* RED marbles and *M* BLUE marbles in the jar, then the last marble remaining in the jar is necessarily of color *f(K,M)*.

It turns out that such a function exists! The key to identifying it is to first identify an invariant of the loop having to do with the number of BLUE marbles in the jar.

Consider the effect that a single iteration of the loop has upon the number of BLUE marbles in the jar. In the case that both marbles chosen were of the same color, the number of BLUE marbles either decreases by two (if both marbles chosen were BLUE) or stays the same (if both marbles chosen were RED). In the case that the marbles chosen were of different colors, the number of BLUE marbles remains the same.

It follows that a single loop iteration has no effect upon whether the number of BLUE marbles is odd or even. (That is, the number of BLUE marbles in the jar either remains odd or remains even as a result of the execution of a single iteration of the loop.) But then the same property holds with respect to any number of iterations of the loop. Letting *K* and *k* denote, respectively, the number of BLUE marbles occupying the jar originally and currently, respectively, we conclude that

k is odd if and only if *K* is odd

is an invariant of the loop. In case you find it easier to comprehend, a different way of expressing this loop invariant is

(both *K* and *k* are odd) or (both *K* and *k* are even)

Suppose that, initially, the number of BLUE marbles in the jar, *K*, were odd, and that the loop has just terminated. (And recall that, because a loop invariant holds at the end of every iteration, it holds, in particular, at the end of the last iteration.) Then the lone marble remaining in the jar must be BLUE, because otherwise we'd have $k = 0$, which is even. By a similar argument, we conclude that, if *K* were even, then the remaining marble must be RED, because otherwise we'd have $k = 1$, which is odd.

Thus, the function *f* mentioned above is as follows:

```
f(K,M) = { RED   if K is even
         { BLUE  otherwise (i.e., if K is odd)
```

Interestingly, the color of the last remaining marble does not depend at all upon the number of RED marbles initially in the jar.

Closed Curve Grid Game:

There are two players, call them Red and Blue. The game is played on a rectangular grid of points, such as illustrated below.

```
6   . . . . . . .
5   . . . . . . .
4   . . . . . . .
3   . . . . . . .
2   . . . . . . .
1   . . . . . . .

    1 2 3 4 5 6 7
```

The rows and columns have been numbered so that we can refer to the point in row i and column j using the ordered pair (i,j).

Red and Blue take alternating turns, with Red going first. Red takes a turn by drawing a red line segment, either horizontal or vertical, connecting any two adjacent points on the grid that are not yet connected by any line segment. Blue takes a turn by doing the same thing, except that the line segment drawn is blue. Red's goal is to form a closed curve (i.e., a sequence of (three or more) distinct line segments starting at some point and returning to that point) comprised entirely of red line segments. Blue's goal is to prevent Red from doing so. The game ends when either Red has formed a closed curve or there are no more line segments to draw.

Making an analogy with a computer program, we can view the game as follows:

```
while (more line segments can be drawn) {
   Red draws line segment;
   Blue draws line segment;
}
```

(For simplicity, we have assumed that the # of possible line segments is even; otherwise the game might end with a RED line being drawn.)

Question: Does either Red or Blue have a "winning strategy", i.e., a game plan that, if followed faithfully, guarantees victory?

Answer: Yes! Blue is guaranteed to win the game by responding to each turn by Red in the following manner:

```
if (Red drew a horizontal line segment) {
   let i and j be such that Red's line segment connects (i,j) with (i,j+1)
   if (i>1) {
      draw a vertical line segment connecting (i-1,j+1) with (i,j+1)
   } else {
      draw a line segment anywhere
   }
} else  // Red drew a vertical line segment
   let i and j be such that Red's line segment connects (i,j) with (i+1,j)
   if (j>1) {
      draw a horizontal line segment connecting (i+1,j-1) with (i+1,j)
   } else  {
      draw a line segment anywhere
   }
}
```

Informally, Blue responds to Red by making sure that the line segment just drawn by Red can never occur as one of the two line segments forming an "upper right corner" of a closed curve of red segments. Note that any closed curve of red line segments must include at least one such corner. Thus, if Blue adheres to this strategy, Red can never form a closed curve!

Assuming that Blue follows this strategy, the following statement is true after each step in the playing of the game:

> There does not exist on the grid a pair of red line segments that form an upper right corner.

That is, the above statement is an invariant (assuming Blue follows the specified strategy) with respect to the playing of the game.

Of course, there is nothing special about upper right corners. Blue could have just as easily chosen to prevent Red from forming any of the other three kinds of corners instead. Significantly, when the game ends (i.e., the loop terminates), the invariant will hold, and Red will not have formed an upper right corner. Which means that Red must not have won, so Blue must have won.

Daisy Petal Game

Imagine a daisy (or any other flower) having 16 petals. Two players take alternating moves. A move involves removing either one petal or two adjacent petals from the daisy, at the player's choice. The winner is the one removing the last petal. (Note: By "adjacent" petals, we mean two petals that were adjacent to one another before any petals were removed.)

Question: Does either player have a winning strategy?

Answer: Left as an exercise for the reader.

Now for some examples that come from computer programming.

Sum of 1..N:

We wish to develop a program that, given as input a nonnegative integer `n`, establishes, as its postcondition,

```
R: sum = 1 + 2 + ... + n
```

Very often, an effective approach for writing a program with postcondition `R` is to develop a loop having as an invariant an expression `P`, where `P` is obtained from `R` by replacing a constant by a variable. In this example, we might replace `n` (which is a constant in the sense that it is an input value not subject to being changed) by `k`, thereby obtaining as a (candidate) loop invariant

```
P: sum = 1 + 2 + ... + k
```

The idea is to develop a loop, having `P` as an invariant, in which `k` will be modified so that it eventually reaches the same value as `n`. By making `k!=n` the guard of the loop, we design it so that at the point when `k`'s value becomes equal to `n`, the loop will terminate with both `P` and `k=n` being true, guaranteeing that `R` will be true, too. Why? Because if both

`sum = 1 + 2 + ... + k` and `k=n`

are true, then so must be

```
sum = 1 + 2 + ... + n.
```

(This is just an instance of the general rule that says from `A` and `x=y` we can conclude `A'`, where `A'` is `A` with one or more occurrences of `x` replaced by `y`.)

Before giving the first version of our program, we introduce the notion of **textual substitution**: For expressions `E` and `F` and a variable `x`, we use the notation

```
E(x:=F)
```

to refer to the expression we get by replacing every (free) occurrence of `x` in `E` by `F`. For example,

`(2x + 7 = 14x)(x:=y-2)` denotes `2(y-2) + 7 = 14(y-2)`

Our "program skeleton" is as follows:

```
// {precondition: n >= 0, where n is the input value}
k = 0;
< code to establish P(k:=0): sum == 1 + ... + 0 >

// {loop invariant P: sum == 1 + 2 + ... + k}
while (k != n)  {
   < code to establish P(k := k+1): sum == 1 + ... + k + (k+1) >
   k = k+1;  // increment k to re-establish P : sum == 1 + 2 + ... + k
}
// {postcondition: sum = 1 + 2 + ... + n}
```

We chose to increment `k` at the end of the loop body rather than at the beginning. Sometimes one choice works better than the other; sometimes it makes little difference.

To establish `P(k:=0)` is easy: simply initialize `sum` to `0`.

In order to establish `P(k:=k+1)` (i.e., `sum = 1 + 2 + ... + k + (k+1)`) inside the loop, it suffices to add `k+1` to `sum`. (The invariant tells us that the value of `sum` is already `1 + 2 + ... + k`, so to make `sum` equal to `1 + 2 + ... + k + (k+1)` requires adding `k+1` to it!)

Having established `P(k:=k+1)`, in order to re-establish `P` it suffices to increment `k` (so that `k` takes on the value that immediately before had been one more than `k`).

The resulting program is as follows:

```
// {precondition: n >= 0}
k = 0;  sum = 0;

// {loop invariant P: sum == 1 + ... + k}
while (k != n)  {
   sum = sum + (k+1); // {sum == 1 + ... + k + (k+1) (i.e., P(k:=k+1))}
   k = k+1;           // {sum == 1 + ... + k  (i.e., P)}
}
// {k = n  &  sum == 1 + 2 + ... + k}
// {postcondition: sum == 1 + 2 + ... + n  (follows from line above)}
```

Note: There is a far better solution that does not use iteration at all, but rather the formula `1+2+...+n = n(n+1)/2` (which, the story goes, was (re-)discovered by Gauss when he was about 7 years old).

This illustrates the fact that the best solution is often not the most obvious one! **End of note.**

Finding a Maximum Element in an Array

Suppose we have an array `a[0..n-1]` (where `n>0`) of, say, integers, and we want to determine the maximum among the values in `a[]`. Using `maxVal` as the output variable, the desired postcondition is

```
maxVal == Max({a[0], a[1], ... , a[n-1]})
```

For brevity, henceforth we shall abbreviate the array segment `<a[0], a[1], ..., a[k]>` by `a[0..k]`.) Using the approach described above, we replace "constant" `n` in the postcondition by variable `k` in order to obtain `P: maxVal == Max(a[0..k-1])` as a candidate for a loop invariant. This leads us to propose the following program skeleton:

```
// {precondition: a.length > 0}
k = 1;
< code to establish P(k:=1): maxVal == Max(a[0..0]) >

// {loop invariant P: maxVal == Max(a[0..k-1])}
while ( k != n )  {
   < code to establish P(k:=k+1): maxVal == Max(a[0..k]) >
   k = k+1;   // re-establishes P
}
// { maxVal == Max(a[0..k-1]) & k == n }
// {postcondition R: maxVal == Max(a[0..n-1]) follows from above}
```

Note: Rather than initializing `k` to `0`, we used `1`. The main reason is that establishing `P(k:=0)` is problematic, because it is not clear how to set maxVal to the maximum value of `a[0..-1]`, which is an empty segment. One idea is to set it to negative infinity, which works in the abstract, but which (at least in many circumstances) has no obvious counterpart in a computer program. Also, due to the fact that the precondition guarantees `n>0`, we are justified in starting `k` at `1`. **End of note.**

Having set `k` to `1`, to establish `P(k:=1)` requires simply that `maxVal` be initialized to the maximum of `a[0..0]`, which is obviously `a[0]` itself.

As for establishing `P(k:=k+1)` inside the loop before incrementing `k`, consider that, at this point, we have `P` (i.e., `maxVal == Max(a[0..k-1])`) and that we want to change `maxVal` to establish `maxVal == Max(a[0..k])`.

How are `Max(a[0..k-1])` and `Max(a[0..k])` related? Given that `a[0..k-1]` and `a[0..k]` are the same, except for the latter containing one extra element, ---namely, `a[k]`--- clearly the only difference is this: if `a[k]` is bigger than the maximum element in `a[0..k-1]`, it is the maximum of `a[0..k]`; otherwise, the maximum of the two array segments is the same. So we refine our program to get

```
k = 1;
maxVal = a[0];    // {P(k:=1): maxVal == Max(a[0..0])}

// {loop invariant P: maxVal == Max(a[0..k-1])}
while (k != n)  {

   if ( a[k] > Max(a[0..k-1]) )
      { maxVal = a[k]; }
   else
      { maxVal = Max(a[0..k-1]); }

   // {P(k:=k+1): maxVal == Max(a[0..k])}

   k = k+1;   // {P: maxVal == Max(a[0..k-1])}
}
// { maxVal == Max(a[0..k-1]) & k == n }
// {postcondition: maxVal == Max(a[0..n-1])}
```

By the fact that `maxVal == Max(a[0..k-1])` holds at the time that the "if" statement is executed, the two instances of `Max(a[0..k-1])` appearing there can be replaced by `maxVal`. We get

```
k = 1;
maxVal = a[0];

// {loop invariant P: maxVal == Max(a[0..k-1]) }
while (k != n)  {
   if ( a[k] > maxVal )
      { maxVal = a[k]; }
   else
      { maxVal = maxVal; }

   // {P(k:=k+1): maxVal == Max(a[0..k])}

   k = k+1;   // re-establishes P
}
// {maxVal == Max(a[0..k-1])  &  k == n}
// {postcondition: maxVal == Max(a[0..n-1])  (follows from above)}
```

Because the body of the else part has no net effect, we can omit it (or replace it by a statement that has no effect, such as `{ }` in Java), giving us as the final program:

```
   k = 1;
   maxVal = a[0];

   // {loop invariant P: maxVal == Max(a[0..k-1])}
   while (k != n)  {

      if (a[k] > maxVal)
         { maxVal = a[k]; }

      // {P(k:=k+1): maxVal == Max(a[0..k])}

      k = k+1;   // re-establishes P

   }
   // { maxVal == Max(a[0..k-1])  &  k == n }
   // {postcondition: maxVal == Max(a[0..n-1]) }
```

2-Color Dutch National Flag Problem

Given an array `a[0..N-1]` each of whose elements can be classified as either RED or BLUE, rearrange `a[]`'s elements (via swapping, but no other operation) so that all occurrences of RED come before all occurrences of BLUE and so that the variable `k` indicates the boundary between the RED and BLUE regions.

In picture form, the desired postcondition `R` is

```
     0                k              N
    +-----------------+----------------+
  a |     all RED     |    all BLUE    |
    +-----------------+----------------+
```

That is, `R: every element in a[0..k-1] is RED & every element in a[k..N-1] is BLUE`

It is fairly obvious that any solution will involve a loop. In order to obtain candidates for the loop's invariant and guard, we first rewrite the postcondition `R` as the slightly stronger

```
   R' : every element in a[0..k-1] is RED & every element in a[k..m-1] is BLUE & m = N
```

Note that `R'` is obtained from `R` simply by replacing `N` by `m` in the second conjunct and adding `m = N` as a new third conjunct. It should be clear to the reader that any state satisfying `R'` also satisfies `R`.

As a loop guard `B`, we propose using `m != N`, which is simply the negation of the third conjunct of `R'`. As a loop invariant `P`, we propose using the rest of `R'` (i.e., its first two conjuncts). That gives us

```
        P: every element in a[0..k-1] is RED & every element in a[k..m-1] is BLUE
```

which, as a picture, looks like this:

```
    0           k            m          n
   +-----------+------------+----------+
 a |  all RED  |  all BLUE  |    ?     |
   +-----------+------------+----------+
```

Notice how these choices for `P` (the loop invariant) and `B` (the loop guard) automatically give us

```
P & !B ==> R'
```

which is vital in showing that, when the loop terminates, the postcondition holds.

Notice also that `P` says nothing about the contents of array segment `a[m..n-1]`, which is why we label it in the picture by **?** (which is supposed to suggest that we don't know what's there).

Given the above, we sketch our solution as follows:

```
k = ?;  m = ?;  // initialize k and m to truthify P
// { invariant P : a[0..k-1] is all RED & a[k..m-1] is all BLUE }
while ( m != N )  {
   < code to preserve P while also making progress toward termination >
}
// { P & m == n, from which R' follows, from which R follows }
```

As is true in our example, almost every loop is immediately preceded by a small segment of code in which one or more variables must be initialized. The aim of such a code segment is to ensure that, after it has executed, the loop invariant will have been truthified (i.e., been made true).

So let us consider to which values our variables `k` and `m` should be initialized. Recall that the loop invariant requires that all elements in the segment `a[0..k-1]` be RED and that all elements in `a[k..m-1]` be BLUE. As we have no way of knowing ---before examining any of the array's elements--- which colors they might have, the only way that we can ensure that every element in these two segments is of the right color is to give values to `k` and `m` making those segments empty!!

To make `a[0..k-1]` empty, we set `k` to zero. Given that choice, in order to make `a[k..m-1]` empty we have little choice but to set `m` to zero as well. (The result, of course, is that the RED and BLUE segments are empty and the ? segment occupies the entire array!)

Our still rather sketchy solution has advanced to:

```
k = 0;  m = 0;  // initialize k and m to truthify P
// { invariant P : a[0..k-1] is all RED & a[k..m-1] is all BLUE }
while ( m != N )  {
   < code to preserve P while also making progress toward termination >
}
// { P & m == n, from which R' follows, from which R follows }
```

At this point we observe that, since `m` is initialized to zero and must finish with value `N` (in order to falsify the loop guard, thereby causing it to terminate), it would seem that a reasonable way to achieve the goal of "making progress toward termination" on each iteration would be for the body of the loop to cause `m`'s value to increase. The most obvious way to increase `m`'s value is to increment it. Assuming (as is often the case) that this is done as the last step in the body of the loop, we get:

```
k = 0;  m = 0;  // initialize k and m to truthify P
// { invariant P : a[0..k-1] is all RED & a[k..m-1] is all BLUE }
while ( m != N )  {
   < code to establish P(m:=m+1) : a[0..k-1] is all RED & a[k..m] is all BLUE >
   m = m+1;
```

```
      }
      // { P & m == n, from which R' follows, from which R follows }
```

Question: How to establish `P(m:=m+1)` before incrementing `m`?

Answer: Examine `a[m]`. If it is BLUE, we conclude that `a[k..m]` is all BLUE. (Recall that the invariant tells us that `a[k..m-1]` is all BLUE.) The loop invariant also tells us that `a[0..k-1]` is all RED. But these two facts together are simply `P(m:=m+1)`, which is to say that `P(m:=m+1)` already holds and hence in this case we need do nothing.

If, on the other hand, `a[m]` is RED, swap `a[k]` with `a[m]` (so that `a[m]` gets put alongside other RED elements), the effect of which is to extend the RED region to the right one position and to shift the BLUE region one position to the right. That is, this leaves us with `a[0..k]` being RED and `a[k+1..m]` being BLUE, which corresponds to `P(k,m := k+1,m+1)`. By incrementing `k`, we establish `P(m:=m+1)`, as desired.

Assuming the existence of a static method called `swap` whereby the call `swap(b,i,j)` swaps the values of `b[i]` and `b[j]`, we get this final solution:

```
m = 0;  k = 0;   // initialize k to establish P(m:=0)
// { invariant P: a[0..k-1] is all RED & a[k..m-1] is all BLUE }
while (m != N)  {
   if (a[m] is BLUE)
      { }
   else {  // a[m] is RED
      swap(a,k,m);
      k = k+1;
   }
   // { P(m:=m+1) }
   m = m+1;   // re-establishes P
}
// { P & m == n, from which R' follows, from which R follows }
```