

Designing Loops with Predicates

1 Proving Programs Correct

CMSC 510 is a course that will survey a large number of algorithms. For each of these algorithms we will want to describe how the algorithm works, maybe look at an implementation in code, try to assess the run time efficiency of the algorithm, and also present a formal proof of correctness for the algorithm. When I say proof, I mean something very much like a formal mathematical proof of the sort you would see in a math course. The purpose of the proof is to establish by means of formal logic that the algorithm does what it claims to do.

In this handout, we are going to study one aspect of proof of correctness for algorithms, a method for constructing loops that makes use of some of these ideas. The method demonstrated here and the examples used come mostly from the book *The Science of Programming* by David Gries.

Predicates and States

Most program proof schemes are based on applications of the Predicate Calculus, which is a set of rules and techniques for manipulating logical expressions known as *predicates*. For our present purposes, a predicate is a statement about the current state of a collection of variables found in a program. The statement can be formed from one or more simple assertions about variables, such as $i > 10$ or $a[k] \neq x$, connected by logical operators, such as **and**, **or**, and **not**. The concept of predicate works hand in hand with the concept of a program's state. The state is essentially a collection of values for the variables in the program. The set of states associated with a given predicate is that collection of values of the variables found in the predicate that will make the predicate be true. The association works both ways: a predicate can be used as a guard at some point in a program to ensure that the variables are in some desired state, or knowledge of the desired state of the program at some point in its execution can be used to construct a goal predicate to describe that state.

Another term sometimes used for predicates in a program is *assertions*. One can think of assertions as tests or claims about the current state of the program that can be applied at strategic locations in the program. Most often these are the error checks that programmers write into their programs in an effort to catch bugs. This traditional use of predicates is not quite what we have in mind here, because assertions are most often written in as an afterthought and serve only to terminate the program with an error message when they fail. What we have in mind is a use of predicates throughout the design process from the very start. These methods will provide us with careful, structured techniques for designing loops that will simultaneously make them easier to construct and increase our confidence in their correctness.

A Bit of Notation

All of the examples below use arrays, and very often we will need to refer to parts of an array, so we will adopt the notation $\mathbf{b}[i..j]$ to mean 'that part of the array \mathbf{b} between index i and index j '.

A Common Structure

The method we will outline here works from the following structure for a loop:

```
< Initialization >
while(! <Termination Condition>)
    <Loop Body>
<Goal State>
```

The general idea is that we do some sort of initialization to prepare for entry into the loop and execute the loop body until the termination condition is true. When execution of the loop is complete, we would then like to have some sort of guarantee that we are in the goal state. Running through this structure is a key predicate called the loop invariant. This invariant ties the whole structure together: the loop invariant must be true after initialization and it must be true after every pass through the loop body. Furthermore, the truth of the loop invariant combined with the truth of the termination condition must be strong enough to satisfy the goal predicate.

The design method for building a loop based on a goal predicate and an invariant starts with the goal predicate. The initial state will not satisfy the goal predicate (if it did, we wouldn't need to write a loop in the first place!), so we weaken the goal predicate enough to make the state it describes broad enough to also include the initial state. This weakened goal predicate becomes the loop invariant. Next, we design a termination condition so that **<Termination>** and **<Invariant>** implies the **<Goal>**. The final task is to design a loop body that brings us closer to termination each time we apply it and does not violate the loop invariant.

2 A Simple Example

The first example we are going to look at is quite simple. In fact, it is so simple that you could probably write down the solution yourself without using any of these methods. Nonetheless, it makes a very good first example of how to apply the method.

Suppose that **B** is an array of **n** integers, and we know that the value **x** appears at least once in the array. Write a loop that finds the index of the location where **x** first appears in the array.

The goal predicate is $\{ (1 \leq i \leq n) \text{ and } (b[i] = x) \text{ and } (b[k] \neq x \text{ for all } 1 \leq k < i) \}$. The best way to weaken this predicate is to drop one of the three requirements: the most obvious one is the requirement that $(b[i] = x)$, for that seems to be the one that would signal success. The weakened predicate, $\{ (1 \leq i \leq n) \text{ and } (b[k] \neq x \text{ for all } 1 \leq k < i) \}$ becomes the invariant. The condition we dropped becomes the termination condition because **<Termination>** and **<Invariant>** should be the **<Goal>**. Next, we write down an obvious initialization condition, $\{ i = 1 \}$, and check that it satisfies the invariant. (It satisfies it trivially because there are no **k**'s such that $1 \leq k < i$ when $i = 1$.) Finally, we write a trivial loop body. About the only thing the loop body has to do is to get us closer to termination each time it is applied. The statement `i++`; will do just fine. Here then is the final form of the loop with the predicates attached:

```
i = 1;
// i = 1
while(b[i] != x )
    i++;
// (1 <= i <= n) and (b[i] = x) and (b[k] != x for all 1 <= k < i)
```

Recall that we said that the invariant has to be true after each pass through the loop body. Let's check that this is actually the case. First of all, note that for the loop body to get a chance to execute, we have

to make it past the termination condition. Also, the invariant will still be true from the initialization or the last pass through the loop. That means that before the execution of the statement, we have $\{ (1 \leq i \leq n) \text{ and } (b[k] \neq x \text{ for all } 1 \leq k < i) \}$ and $b[i] \neq x$. This implies that $\{ (1 \leq i+1 \leq n) \text{ and } (b[k] \neq x \text{ for all } 1 \leq k < i+1) \}$. (Why is $i+1 \leq n$?) Notice that we can safely replace $i + 1$ with i in this predicate and have it still be true. That is exactly what the `i++` statement does; in fact, replacing $i + 1$ with i gives us the invariant back again and guarantees its truth. At the same time, doing this brings us closer to termination.

3 The Plateau Problem

Suppose `b` is an array of `n` integers sorted in ascending order. We want to write a loop to find the length of the longest plateau, or string of identical numbers, in the array. Our goal is to find a number `p` with the property that (`b[1..n]` contains a plateau of length `p`) and (`b[1..n]` does not contain a plateau of length `p + 1`). A more precise way to write the goal predicate is to say (*There is a `k`, $1 \leq k \leq n - 1 - p$, such that $b[k] = b[k+p-1]$*) and (*For all `k`, $1 \leq k \leq n - p$, $b[k] \neq b[k+p]$*) This predicate comes from the observation that since the elements in the array are ordered, to check for a plateau of length `p` one just has to find two elements `p` steps apart that are the same.

Note that the goal will be satisfied after we have had a chance to search all of `b`. An obvious way to weaken the goal into an invariant is to have the invariant talk about our having just searched part of the array. The invariant becomes $\{ (b[1..i] \text{ contains a plateau of length } p) \text{ and } (b[1..i] \text{ does not contain a plateau of length } p + 1) \text{ and } (1 \leq i \leq n) \}$ The termination condition is obvious: the loop stops when $i = n$. An obvious initialization condition satisfying the invariant is $\{ (i = 1) \text{ and } (p = 1) \}$.

The only thing left to do now is to write a loop body. In order to figure out what the loop body should be, let's start with a stupid guess, `i++`, and see that the guess would violate the invariant. Before application of the loop statement, the invariant has to be true and the termination condition false. That is, $\{ (b[1..i] \text{ contains a plateau of length } p) \text{ and } (b[1..i] \text{ does not contain a plateau of length } p + 1) \text{ and } (1 \leq i \leq n) \}$ and $i < n$. The problem with just blindly increasing `i` is that we could already be sitting at the end of a plateau of length `p`, and bumping up `i` by one could cause us to have a plateau of length `p+1` and hence violate the invariant. The way out of this problem is to check whether or not bumping up `i` would cause us to have a plateau of length `p+1` ending at `i+1`: if it does, we bump up `p` by one at the same time that we bump up `i` by one. The invariant would still be true then because we will be certain of the existence of a plateau of length `p` for the new larger `p`. Here is the final version of the code with the appropriate documentation:

```

i = 1;
p = 1;
// Loop invariant :
// ( b[1..i] contains a plateau of length p ) and
// ( b[1..i] does not contain a plateau of length p + 1 )
// and ( 1 <= i <= n )
while(i < n) {
    if (b[i+1] == b[i+1-p])
        p++;
    i++;
}
// Goal :
// ( b[1..n] contains a plateau of length p ) and
// ( b[1..n] does not contain a plateau of length p + 1 )

```

4 A Sorting Algorithm

Now that we have seen a couple of examples, it is time to turn our attention to a problem that more closely approaches the kind of problems real programmers have to contend with. In this next example we are going to look at a sorting algorithm called bubble sort. This is an example of a relatively inefficient sorting algorithm, but it has enough complexity to make it worth our time to study it.

In the discussion below, the notation $A[1..i] \leq A[i+1..n]$ stands for "every element in the range $A[1..i]$ is less than or equal to every element in the range $A[i+1..n]$." Further, if one or both of the two ranges is empty, the statement is understood to be trivially true.

The Code

Suppose $A[1..n]$ is an initially unsorted array of integers. Here is some code to sort the array using the bubble sort algorithm.

```
i = 1;
while(i <= n)
{
  j = n;
  while(j > i)
  {
    if(A[j] < A[j-1])
    {
      temp = A[j-1];
      A[j-1] = A[j];
      A[j] = temp;
    }
    j--;
  }
  i++;
}
```

Analysis of the inner loop

Since this code is structured as a pair of nested loops, we will apply the proof technique to both loops in turn. Since the outer loop will rely on the inner loop, we start our analysis with the inner loop.

$\langle goal \rangle = "A[i] \leq A[i+1..n]"$

$\langle invariant \rangle = "A[j] \leq A[j+1..n]"$

$\langle initialization \rangle = "j = n"$

$\langle termination \rangle = "j = i"$

The invariant is trivially true at initialization time because the range $A[j+1..n]$ is empty. It is also easy to see that $\langle invariant \rangle + \langle termination \rangle = \langle goal \rangle$. The only thing to prove is that the loop body maintains the invariant.

At the beginning of the loop body we have that $A[j]$ is the smallest element in $A[j..n]$. If $A[j]$ is also smaller than $A[j-1]$, we make the two trade places so that now $A[j-1] \leq A[j..n]$. Otherwise, we leave

$A[j-1]$ in place so that once again $A[j-1] \leq A[j..n]$. After doing the decrement of j at the end of the loop body the inequality looks like the invariant again because the decrement effectively replaces $j-1$ with j .

Analysis of the outer loop

$\langle goal \rangle = "A[1..n]$ is sorted"

$\langle invariant \rangle = "(A[1..i-1]$ is sorted) and $(A[1..i-1] \leq A[i..n])"$

$\langle initialization \rangle = "i = 1"$

$\langle termination \rangle = "i = n + 1"$

The invariant is trivially true at initialization time because the range $A[1..i-1]$ is empty. $\langle invariant \rangle + \langle termination \rangle = \langle goal \rangle$ because at termination time $A[1..i-1] = A[1..n]$ and the range $A[i..n]$ is empty at termination. That means that at termination the invariant reduces to $"(A[1..n]$ is sorted) and (true)", which is the same as $\langle goal \rangle$.

At the beginning of the outer loop body we have $"(A[1..i-1]$ is sorted) and $(A[1..i-1] \leq A[i..n])"$. After execution of the inner loop we also have $"A[i] \leq A[i+1..n]"$. The condition $(A[1..i-1] \leq A[i..n])$ implies that $A[i-1] \leq A[i]$, because $A[i]$ is just one element of the larger range $A[i..n]$ and $A[i-1]$ is a representative of the range $A[1..i-1]$. Combining $(A[1..i-1]$ is sorted) with $(A[i-1] \leq A[i])$ gives $(A[1..i]$ is sorted). This maintains the first half of the invariant. Coupling the inner loop goal $"A[i] \leq A[i+1..n]"$ with $(A[1..i-1] \leq A[i..n])$ leads to $(A[1..i] \leq A[i+1..n])$, maintaining the second half of the invariant.

After execution of the inner loop we have $"(A[1..i]$ is sorted) and $(A[1..i] \leq A[i+1..n])"$. Incrementing i at the end of the outer loop body restores the invariant back to the original $"(A[1..i-1]$ is sorted) and $(A[1..i-1] \leq A[i..n])"$.

Commentary

Here are some things to think about when designing invariants.

- Make your invariant as strong as possible. Generally speaking, the strongest possible invariant that is still true at initialization is best. That is why the outer loop invariant used above is better than the weaker $"A[1..i-1]$ is sorted".
- In a nested loop situation, the best outer loop invariant is one that has something to do conceptually with what the inner loop is trying to accomplish. That is why the part of the outer invariant that says $"A[1..i-1] \leq A[i..n]"$ is a good choice.
- To prove that an invariant is maintained by a loop body, the only things you have to go on are the truth of the invariant at the start of the loop body and the code in the loop body itself. This is another reason to make the invariant as strong as possible, because a stronger, more detailed invariant carries more useful information.

5 The GCD problem

The greatest common divisor function takes two positive integers, X and Y , as input and returns the largest positive integer, y , that evenly divides both X and Y . Let us see how to write a loop to compute a greatest common divisor.

The result that we want is a proof of

$$\{ y = \text{GCD}(X,Y) \}$$

The key to this problem is a trio of properties of the GCD function that are easily verified:

$$\text{GCD}(y,0) = y$$

$$\text{if } x \leq y, \text{GCD}(x,y) = \text{GCD}(x,y-x)$$

$$\text{GCD}(x,y) = \text{GCD}(y,x)$$

The first property allows us to recast the goal as

$$\{ y = \text{GCD}(0,y) = \text{GCD}(X,Y) \}$$

Given this set-up, a strategy suggests itself: start by setting x to X and y to Y . Try to drive the value of x down until it hits zero, all the while making sure that $\text{GCD}(x,y)$ stays equal to $\text{GCD}(X,Y)$ as we change x and y .

We have everything we need now: we have a start condition:

$$\{ x = X, y = Y \}$$

an invariant,

$$\{ \text{GCD}(x,y) = \text{GCD}(X,Y) \}$$

and a goal

$$\{ x = 0 \text{ and } \text{GCD}(x,y) = \text{GCD}(X,Y) \}$$

Clearly the invariant is true at start and at the goal, so all that we need now is some way to make a loop that maintains the invariant and makes progress toward the goal. The key to making progress toward the goal is the second of the three properties mentioned above. Here is a loop that uses this property to make progress to termination:

```
x = X;
y = Y;
// Loop invariant
// x >= 0 and y >= 0 and GCD(x,y) = GCD(X,Y)
while(x > 0)
  if(x <= y)
    y = y-x
    // This branch maintains the loop invariant
    // by property (2) of the GCD
  else
    {
      temp = x;
      x = y;
      y = temp;
      // This branch maintains the loop invariant
      // by property (3) of the GCD
    }
// After termination, x = 0 and GCD(x,y) = GCD(X,Y)
// By property (1) of GCD we have y = GCD(X,Y)
```

The loop body makes progress toward this goal by reducing y until it is smaller than x . When y is smaller than x , the two are switched and the process continues. Notice that this really does make progress

toward termination because except for the occasional swap, y continues to decrease in size.

Finally, let us make two small adjustments to increase the efficiency of the loop. Instead of subtracting just one x from y each time we do a subtraction, let's subtract as many copies of x as we can get away with. What do you get when you subtract x from y as many times as possible? You get $y \bmod x$! This simple observation speeds the code up. Also, when we do $y = y \% x$, the result will automatically make y smaller than x . The next time around the loop we will then definitely have to do a swap. We can make a short-cut by doing the swap at the same time as the mod:

```
x = X;
y = Y;
if(y < x)
{
    temp = x;
    x = y;
    y = temp;
}
// Loop invariant
// y ≥ x and x ≥ 0 and GCD(x,y) = GCD(X,Y)
while(x > 0)
{
    temp = y % x;
    y = x;
    x = temp;
    // Maintains the invariant by properties (2) and (3) of GCD
}
// After termination, x = 0 and GCD(x,y) = GCD(X,Y)
// By property (1) of GCD we have y = GCD(X,Y)
```

6 A Slick Exponentiation Algorithm

Computing x^y for positive integers X and Y is a trivial exercise in loop writing:

```
z = x;
t = y;
while(t > 0)
{
    z = z * x;
    t = t - 1;
}
```

Is this solution correct? At termination, we have $t = 0$ and z should be x^y . At start, we have $t = y$ and $z = x$. What sort of invariant would cover both of these conditions? The correct invariant for this problem is $x^y = z * x^t$. This is appropriate because it captures the central idea: as t decreases, z should increase to keep the equality true. We can also see that the initialization is wrong. $\{ z = x \text{ and } t = y \}$ does not satisfy the invariant. To fix it, we should say $\{ z = x \text{ and } t = y - 1 \}$.

We measure progress toward termination in this example by reducing t . The rate at which we reduce t also determines how long the loop takes to run. Can we think of a faster way to decrease t and still have the loop invariant be true after each iteration?

A faster way to reduce t is to do $t = t/2$ each time through (provided t is even). We can do this and still preserve the invariant by letting the x above be a variable instead of being fixed at x . The new invariant is $x^y = z * s^t$. Notice that we can maintain the invariant by saying $s = s*s$ at the same time we do $t = t/2$. Here is the loop in the faster version:

```
z = x;
s = x;
t = y - 1;
while(t > 0)
    if(t % 2 == 0)
        {
            t = t / 2;
            s = s * s;
        }
    else
        {
            z = z * s;
            t = t -1;
        }
```