CS 2 2002: Notes on Loops

Mika Nyström
Department of Computer Science
California Institute of Technology
January 20, 2002

1. Introduction

So far in CS 2, the only programs we have dealt with formally have been simple programs without loops. The most difficult construct has been the IF statement. Such programs can always be thought of as running from start to finish, passing over each statement exactly zero times or exactly one time, depending on the conditions in the IF statements. Such programs can always be analyzed through “case analysis”; simply consider every possible sequence of statements and write down the appropriate Hoare triples (or weakest preconditions). It is not necessarily trivial to do so, but it should at least be clear that short programs admit of short analyses, while long analyses are only necessary for long programs.

Programs that have this straightforward behavior cannot be powerful enough to solve all the problems we are interested in. This is because we know that sometimes we want to solve problems that are very large. Ideally, we should like to be able to solve unboundedly* large problems: we should like to be able to write programs that can deal with problems that are larger than any we can fit in our minds when we are programming. This is indeed what makes programming such a rewarding activity: we can write small programs that can deal with problems of any size.

The most common way of generalizing a program from solving problems of only specific finite sizes to solving problems of any finite size is by making the program “loop,” or pass over its program statements a number of times that depends on the problem that is being solved. (Loops are called loops because on the machines that were popular from the 1940s to the 1960s, program input was often in the form of paper tape with holes punched in it, and making a machine “loop” was a matter of gluing together the ends of the paper tape—into a loop—after it had been loaded into the machine.) In CS 1 you saw another way of handling large problems: recursion. Recursion is to functional programming what looping is to imperative programming; the two are equally powerful constructs.

It is a fact of life that adding power to a mechanism in computing (or in mathematics, or indeed in most human endeavors) brings with it increased responsibilities on the part of the user of the mechanism to ensure that the mechanism does not run amok and cause damage. When it comes to loops, we have to introduce new ways of thinking about programs if we want to keep being certain that they are correct. Think of it this way: adding a loop (or a recursion) around a sequence of statements ultimately means that the sequence of statements may be executed infinitely many times; the program text remains finite, but the program “trace” (i.e., the sequence of statements actually executed at runtime) is now potentially infinite. “Case analysis” is obviously inadequate for loops.

In class, we have seen the traditional way of dealing with loops. This consists of two steps:

1. Proving that if the loop terminates, then it terminates in the right state. (Using the loop invariant.)
2. Proving that the loop does indeed terminate (that the “loop guard” will eventually evaluate to false.) (Using the variant function.)

(Remember that “proving” just means “convincing ourselves of.”) The reason that the proof is broken up in these two steps is that it is usually convenient to use different approaches for the steps. The reason that the invariant step is stated first is that the invariant often comes in handy when working on the variant function.

* “Unbounded” does not mean infinite; it means arbitrarily large but still finite.
2. The Loop Invariant

Let us first look at the problem of showing that if a loop terminates, then it terminates in the right state (or more accurately, in some state satisfying the specification).

Given the Hoare-triple specification for some program $S$,

$$ \{P\} S \{Q\} $$

let us say we try to implement $S$ with a loop (in Modula-3). Then we can rewrite the Hoare triple as

$$ \{P\} \text{while } G \text{ do } L \text{ end} \{Q\}, $$

where $L$ is some program (fragment) and we call $G$ the loop guard.

What can we say right away about the loop? Only that when the loop exits, $\neg G$ must hold; therefore

$$ Q \Rightarrow \neg G. $$

Now we introduce the invariant. The invariant should have the property that it is true before every $L$ and after every $L$ (although not necessarily during $L$). If the invariant is true before every $L$ and after every $L$, then it is true before the first $L$ and true after the last $L$, i.e., when the loop is first entered and when the loop is at last exited. From this we already know that we must have

$$ P \Rightarrow I, $$

or the program will not be correct; likewise, we must have

$$ Q \Rightarrow I $$

or else our invariant cannot possibly be right.

Obviously, many statements that would not help us much with understanding our program would fit that bill (e.g., “2+2=4”). So we do not just want any old true statement. What we should pick is a predicate that is “relevant” to the program. What does that mean? Well, for the invariant to be useful, we must be able to use it to show that we do indeed establish $Q$. Since $Q$ is not in general true in every state where $I$ is true (if it were, then we should have $P \Rightarrow Q$ and no need for executing any program at all!), $Q$ must represent a strengthening of $I$; the only thing that comes to mind is $\neg G$, which also must hold after the loop. In other words, we should have

$$ I \wedge \neg G \Rightarrow Q. $$

Together with our earlier observation that $P \Rightarrow I$, this already limits the things that are useful as invariants.

Example. Let us consider the search program from lecture. We are given an array $a$ and we are to write a program that will find the first element of $a$ that matches some given value $e$; we assume that we know that $e$ exists in $a$; our task is to leave the index of the matching element in the variable $i$. In other words, what we want to do is implement the following specification—we have left out the lower bound on the index of $a$; let us assume that is zero:

$$ \{ \exists j : a[j] = e \} S \{ a[i] = e \wedge \forall k : k < i : a[k] \neq e \} $$

(Note the careful differentiation between the program variable $i$ and the bound variables $j$ and $k$; you could call $j$ and $k$ “ghost variables” since they themselves do not appear in the program.)

What do we do now? We have the specification, but as yet no program and no invariant. Well, it is obvious from the two formulas $Q \Rightarrow I$ and $I \wedge \neg G \Rightarrow Q$ that the invariant should be closely related to the postcondition. From $Q \Rightarrow I$, we know it must be weaker than the postcondition, and from $I \wedge \neg G \Rightarrow Q$ we know that it cannot be too much weaker, or we should never be able to strengthen it with just $\neg G$ and from that derive the postcondition.

2
When you get to this point in designing your program, you should probably take a step back and think
informally about the operation of some possible programs that might satisfy the specification. Programming
is not a simple, mechanical activity where there is always only one right answer. On the contrary: there are
always infinitely many programs that satisfy a given implementable specification!

Ok, so let us guess that since what we are looking for is the lowest-indexed element of a[i] such that
a[i] = e, we should search the array from bottom to top. This suggests that all the elements our program
will inspect are at lower indices than the desired one (for inspecting a higher-indexed element gives no
information about the location of the desired one). So we accordingly might guess the invariant

\[ I = (i < a[i] = e \land \forall k : k < i : a[k] \neq e), \]

where we have introduced the new variable \( i \) as shorthand for “the desired final value of \( i \)”; note that
this invariant simply says the same thing about \( i \) that the specification said about \( i \); in other words, the
invariant is a long-winded (but precise) way of saying that the program variable is always less than the value
it will have at the end of the loop. (Actually, to be completely precise, the invariant should probably be written

\[ I = \exists i : (i < a[i] = e \land \forall k : k < i : a[k] \neq e). \]

Why? Because that relieves us of having to define \( i \) in the English prose and instead binds it directly to
the formula.

Oops! That is not right. If \( i \) is always less than its final value, then it cannot ever have its final value.
That is a contradiction (at least as long as we optimistically assume that we shall finally be able to write a
properly functioning program). We try weakening \( I \), as little as possible:

\[ I = (i \leq a[i] = e \land \forall k : k < i : a[k] \neq e) \]

The new \( I \) does not seem to run afoul of anything. It simply says that \( i \) is less than or equal to its final
value, at all times. That seems reasonable, since we are scanning the array from lowest index to highest.

The program that accomplishes the scanning is written:

```plaintext
i := 0;
WHILE a[i] # e DO
  i := i + 1
END
```

You probably already knew that, because this is such an obvious program. But let us anyhow observe
that we have actually skipped a step. The program should really be written

```plaintext
i := 0; WHILE i # i END i := i + 1 END
```

because what we want to establish is actually \( i = \text{if} \), not the weaker \( a[i] = e \). But we cannot use \( i \)
directly in our program since we do not know it, so we settle for the weaker \( a[i] = e \). This leaves us with
the obligation to prove that, under the very specific conditions that obtain in this program, \( a[i] = e \) \( \Rightarrow \)
(\( i = \text{if} \)). This last statement is not true in general since \( a[i] = e \) is true if, for instance, \( a[i] \) should
happen to be the second occurrence of \( e \) in the array. “The very specific conditions that obtain in this
program” are exactly the conditions expressed by the loop invariant.

To make this last point absolutely clear, let us consider a slightly different specification; let us consider
designing a program that is to leave \( i \) such that

\[ \{ i : a[i] = e \} S\{ a[i] = e \}; \]

in other words, we are asking for a program that will leave \( i \) the index of any element of \( a \) that happens
to be equal to \( e \). This “find-any-e” specification is weaker than the “find-first-e” specification since the
postcondition is weaker than for the find-first specification; in other words, any program that is a find-first
is also a find-any, but not the other way around.

3
So what does it take to show that the weaker find-any specification is satisfied by our old program?

\[ i := 0; \]
\[ \text{WHILE } a[i] \neq e \text{ DO} \]
\[ i := i + 1 \]
\[ \text{END} \]

Well, we know that when the \texttt{WHILE} loop exits, \( I \land \neg G \) must hold. But \( \neg G \) is exactly \( a[i] = e \), which is the whole postcondition! So to prove that the program satisfies the find-any specification, the invariant can simply be \texttt{true}, the weakest of all invariants.

Getting back to the find-first program, let us proceed with our proof obligation. We have to show that \( (a[i] = e) \Rightarrow (i = if) \). This maybe seems silly, because it is obvious that since we are scanning the array, and we stop at the first element matching \( e \), we get the first element matching \( e \). Well, consider the program:

\[ i := 0; \]
\[ \text{WHILE } a[i] \neq e \text{ DO} \]
\[ i := i + 2 \]
\[ \text{END} \]

This program satisfies the find-any specification as far as the invariant goes, but it does not satisfy the find-first specification. Why? Because it will not find the first \( e \) if it should happen to be at an odd index. \textit{But note carefully} that as far as the loop invariant goes, this program is correct for the find-any specification, because \( if \) it terminates, \( then \) it terminates in a state where \( a[i] = e \), which is all we are asking for. Showing that it \textit{does} terminate is a matter for the variant function, not for the invariant. So let us get back to:

\[ i := 0; \]
\[ \text{WHILE } a[i] \neq e \text{ DO} \]
\[ i := i + 1 \]
\[ \text{END} \]

We will show that \( (a[i] = e) \Rightarrow (i = if) \). Let us do it by contradiction. So:

1. Assume that
   \[ (a[i] = e) \land (i \neq if). \]

2. By the definition of \( if \), that means that \( i > if \).
3. How could \( i \) have arrived in such a state? Initially, \( i \leq if \), so some program statement must have taken \( i \) from \( i \leq if \) to \( i > if \).
4. The only program statement that modifies \( i \) is \( i := i + 1 \).
5. Now the weakest precondition comes in handy. Observe that
   \[ \text{wp}(i := i + 1, i > if) = (i \geq if). \]

But we started out in a state where we had that \( i \leq if \); therefore, to get from \( i \leq if \) to \( i > if \), we must have executed the increment at least once in a state where \textit{both} \( i \leq if \) \textit{and} \( i \geq if \); that is to say, in a state where

\[ i = if. \]

In other words, if we have "missed" \( if \), then the program \textit{must} at one point have executed \( i := i + 1 \) when the prior value of \( i \) was \( if \); given the program statements, it could not possibly have had any other value.

6. So we have concluded that we reached the increment statement in a state where \( i = if \). But this is impossible, because if \( i = if \), then \( a[i] = e \), and the \texttt{WHILE} statement would have exited.
7. Therefore the conjecture \((a[i] = e) \land (i \neq if)\) must be false, and we have
\[(a[i] = e) \Rightarrow (i = if).\] Q.E.D.

I apologize for the excruciating detail of this proof; especially because the program is so “obviously right.” This level of detail is almost never necessary in practice, because it is enough to argue informally that, for instance, “we don’t miss any values.” Just the same, the proof structure is extremely important: there are many problems where the execution pattern is far less obvious than in this one, but the same proof structure still applies. (Instructive exercise for the reader: can you figure out where the proof will go wrong if we replace \(i := i + 1\) by \(i := i + 2\)?)

3. Choosing an Invariant

We have seen one example of using the invariant; we have noted that the invariant is closely related to the postcondition; and we have noted that the invariant and the negation of the loop guard together should imply the postcondition. How do we choose such an invariant? Is there any method to it? Yes, there is. From \(Q \Rightarrow I\) and \(P \Rightarrow I\), we know that the invariant must be a weakening of the postcondition. (For instance, in the search example, the postcondition implies the invariant \((i = if) \Rightarrow (i \leq if)\) just the same as the precondition does \((i = 0) \Rightarrow (i \leq if)\).) Three useful ways of weakening the postcondition are:

1. Deleting a conjunct.
2. Replacing a constant by a variable.
3. Enlarging the range of a variable.

In our search example, we enlarged the range of \(i\) from \(i \in [if, if]\) to \(i \in [0, if]\). Then we used the negation of the loop guard to terminate the program exactly when the postcondition holds, i.e., when \(i \in [if, if]\).

The technique is general: if you want to establish postcondition \(Q\), then come up with an invariant that is similar to \(Q\) but weak enough that it contains some reasonable precondition \(P\); we should be able to establish \(P\) without really thinking about it. For instance, in the loop example, we initialize \(i\) to zero; this is required, or the invariant would not hold at the start of the loop. Then, when you have the invariant, come up with a loop guard that guarantees that the postcondition holds when the loop exits; next, write down the commands necessary for establishing the “reasonable” precondition you have chosen (e.g., \(i := 0\)). If you are developing the loop from scratch yourself, only then write the commands that go inside the loop.

Now for a few short examples of the other two approaches to weakening the postcondition.

**Example: deleting a conjunct.** Let us say we wanted to establish, given some fixed \(n\),
\[0 \leq a^2 \leq n < (a + 1)^2;\]
in other words, \(a\) approximates \(\sqrt{n}\). Using our experience from the linear search problem, we might consider some sort of searching from below. So we might want an invariant that is true for \(a = 0\), for instance. Rewriting the invariant as a conjunction,
\[0 \leq a^2 \land a^2 \leq n \land n < (a + 1)^2,\]
and deleting the second conjunct (which is not true for \(a = 0\)), we obtain
\[0 \leq a^2 \land a^2 \leq n.\]
(If we somehow knew we wanted to approach the final value from above, we should delete the first two conjuncts instead of the last one.) Since we know that we want the loop to exit when we have reached an \(a\) such that \(n < (a + 1)^2\), we start writing the loop as:

```
WHILE (a + 1)*(a + 1) <= n DO ... END
```
Re-introducing the deleted conjunct thus is a general approach: it is not a coincidence that the loop guard is exactly the inverse of the conjunct we deleted! Completing the program is left as an exercise for the reader.

*Example: replacing a constant by a variable.* Let us now investigate the postcondition

\[ s = \sum_{0}^{n-1} b[j], \]

where \( b \) is some given array and \( n \) some given number less than or equal to the size of \( b \). If we now replace the constant expression \( n - 1 \) by a program variable \( k \), we can write an invariant

\[ s = \sum_{0}^{k} b[j]. \]

This is a bit tricky, because the invariant must hold upon entry to the loop. How can we make it hold? One way is to take one statement out of the loop, viz.:

\[ s := b[0]; \ k := 0; \ \{s = \sum_{0}^{0} b[j]\} \ \text{WHILE} \ldots \]

Note that this only works if we are guaranteed that the array \( b \) actually has at least one element! A possibly more elegant way of doing it is to make the sum initially run over an empty range, viz.:

\[ s := 0; \ k := -1; \ \{s = \sum_{0}^{-1} b[j]\} \ \text{WHILE} \ldots \]

Yet another way is changing the invariant slightly. Try the invariant

\[ s = \sum_{0}^{k-1} b[j]; \]

then we should have the following:

\[ s := 0; \ k := 0; \ \{s = \sum_{0}^{-1} b[j]\} \ \text{WHILE} \ldots \]

Completing the programs is, again, left as exercises for the reader.

4. **The Variant Function**

The variant function (or bound function) is what you use to show that a loop actually does eventually terminate. It is usually best to develop the variant function after you have developed the invariant, because by then you already know that the invariant will remain true during the loop’s execution, and you can take advantage of this information when you derive the variant function.

The variant function “measures” the progress that the loop makes towards termination. Conventionally, we pick a variant function \( V \) with the following four properties:

1. \( V \) is an integer function.
2. \( V \) starts out finite and is bounded below by zero.
3. Every iteration of the loop reduces \( V \).

Clearly, if we can find a \( V \) that has these properties, then we can immediately conclude that our loop terminates in finite time. Note however that even if we cannot find a \( V \) satisfying the three conditions, the loop might still terminate in finite time; that this is true no matter how “smart” we are is a consequence of the Halting Theorem (CS1 Lecture, Dec. 5). This brings us back to the original question: why bother
learning proof techniques? The answer is not that proof techniques are to be used to check whether or not a given program is correct, but rather during program construction. You should always write a program in such a way that you could, at least if your life depended on it, give a convincing argument that it is correct. *(Some people say that you should prove every program you write correct, but we will not go that far...)* A program constructed so that it is amenable to our proof techniques will be one that is, most likely, simple and easy to understand; this in turn makes it more likely to be correct than a program that is complicated and difficult to understand. That is a noble goal to strive for.

Let us be a bit more concrete. How do we verify V’s properties? The first property, V’s being an integer function, should be obvious from the form of V, so we need not worry more about that. The second and third properties, the facts that each iteration of the loop reduces V and that V ≥ 0 remains true, should be relatively straightforward to prove using the (already developed) invariant and the program text. What we need to prove is that the following assertions hold:

\[
\text{WHILE } G \text{ DO } \{V = k \land V \geq 0\} S\{V < k \land V \geq 0\} \text{ END}
\]

**First example. The search problem.** Using the variable \( i \) again for the index of the first element whose value is \( e \), we write:

\[
i := 0; \text{ WHILE } a[i] \neq e \text{ DO } i := i + 1 \text{ END } \{i = if\}
\]

Ok, fair enough. What should the variant function be? Well, it should represent “getting closer to the solution.” The obvious choice is \( V = if - i \). Let us verify the properties 1.–3.:

1. \( V \) is obviously an integer function.
2. Initially, \( V = if - 0 = if; \) this is a finite number. Furthermore, we have the invariant that \( i \leq if; \) we already proved that this is true irrespective of the loop’s termination. But that means that \( V \geq 0 \), which is what we wanted.
3. Every iteration of the loop entails the execution of \( i := i + 1 \); that this reduces \( V \) is obvious, but let us investigate the Hoare triple anyway. The Hoare triple for assignment is \( \{R\}x := a\{R_{a \rightarrow x}\} \). Therefore,

\[
\{V = k\}i := i + 1\{(V = k)_{i+1 \rightarrow i}\}
\]

What does \( (V = k)_{i+1 \rightarrow i} \) mean? First of all, substituting \( i \) for \( i+1 \) is the same thing as substituting \( i-1 \) for \( i \), except that the latter is easier to handle. Expand \( V \); we get

\[
(if \ - i = k)_{i \rightarrow i-1};
\]

in other words,

\[
if \ - i + 1 = k,
\]

or indeed

\[
V = if \ - i = k - 1,
\]

the desired decrease in \( V \). Again, apologies for the excruciating detail.

**Second example. Euclid’s algorithm for the greatest common divisor.** Recall from lecture the GCD-finding loop:

\[
\text{WHILE } x \neq y \text { DO IF } x > y \text{ THEN } x := x - y \text{ ELSE } y := y - x \text{ END END}
\]

* All right, I suppose this statement should be qualified a little bit. There are actually nontrivial programs that people find useful and don’t know how to prove correct. One example is the recently invented “turbo codes” that are used for error correction in communication systems that have to communicate over very noisy channels. The turbo codes are the best error-correcting codes known for many applications, but no one has yet understood why. It is not even known that the turbo encoder and decoder programs always terminate. But these kinds of programs (useful, yet unprovable) are the exception and not the rule.
The loop invariant is that \( \text{GCD}(x, y) = \text{GCD}(X, Y) \), where \( X \) and \( Y \) are the initial values of \( x \) and \( y \). Showing this is easy once we have the lemma \( \text{GCD}(a, b) = \text{GCD}(a - b, b) = \text{GCD}(a, b - a) \), because it is obviously true that

\[
\{ x = a \land y = b \} \text{IF } x > y \text{ THEN } x := x - y \text{ ELSE } y := y - x \text{ END}\{ x = a - b \land y = b \lor x = a \land y = b - a \};
\]

the postcondition implies that \( \text{GCD}(x, y) \) equals \( \text{GCD}(a, b) \). Together with the second lemma, \( \text{GCD}(a, a) = a \), it is easy to prove that if the loop terminates, then it terminates in a state where \( x = y = \text{GCD}(X, Y) \).

But does the loop terminate? To investigate this, we first need to pick a good variant function. \( x \) and \( y \) suggest themselves as obvious candidates. But they will not quite work, since the third property does not hold for them: it is not true that every iteration reduces \( x \), nor that every iteration reduces \( y \), because the program changes one variable and leaves the other unchanged. So we obviously have to combine them somehow. What I chose in lecture, \( x + y \), is only one of several alternatives. Others that would work are \( \max(x, y) \) and \( xy \).

Well, they would not quite work, since the loop does not terminate without a more stringent precondition. We shall simply observe that enforcing the preconditions \( X > 0 \) and \( Y > 0 \) makes the variant function “work out.” The proof that the loop does not terminate for certain values, a simple inductive proof, is left as an exercise for the reader. (It is simply a matter of proving that the program never gets from a state where \( x < 0 \land x \neq y \) to a state where \( x = y \).)

So let us check the variant function conditions for \( x + y \):

1. That \( x + y \) is an integer function is obvious from its form.
2. Initially, \( x + y = X + Y \), a finite number. Furthermore,

\[
\{ x > 0 \land y > 0 \land x \neq y \} \text{IF } x > y \text{ THEN } x := x - y \text{ ELSE } y := y - x \text{ END}\{ x > 0 \land y > 0 \};
\]

we can check this by case analysis. In the first case, if \( x := x - y \) is executed, then we must—because of the IF—have that \( x > y \), and clearly,

\[
\{ x > 0 \land y > 0 \land x > y \} x := x - y \{ x > 0 \land y > 0 \}.
\]

(He that so desires can go through the exercise of checking this with the Hoare triple for assignment as we did above for the search loop.) In the other case, \( x \leq y \) by the IF, but the WHILE guarantees that \( x \neq y \); therefore, \( x < y \), and we simply repeat the previous argument with \( x \) and \( y \) swapped.

3. Finally, does every iteration of the loop reduce \( V \)? Well, we have shown that given our new, stronger precondition, \( x > 0 \land y > 0 \) throughout the execution of the loop. If we now recall the Hoare triple we used when arguing that the invariant was correct,

\[
\{ x = a \land y = b \} \text{IF } x > y \text{ THEN } x := x - y \text{ ELSE } y := y - x \text{ END}\{ x = a - b \land y = b \lor x = a \land y = b - a \},
\]

and add to it a statement about \( V \), we get

\[
\{ x = a \land y = b \land V = a + b \}
\]

\[
\text{IF } x > y \text{ THEN } x := x - y \text{ ELSE } y := y - x \text{ END}
\]

\[
\{ x = a - b \land y = b \land V = a \lor x = a \land y = b - a \land V = b \},
\]

So, as far as \( V \) goes, we have \( \{ V = a + b \} S \{ V = a \lor V = b \} \); since we have already proved that \( a \) and \( b \) (just other names for the values of \( x \) and \( y \)) are always greater than zero, \( V \) must always decrease.

Where would this proof go wrong without the strong precondition \( X > 0 \land Y > 0 \)? First of all, our chosen \( V \) would not be bounded below by zero, but that is a minor criticism; we could choose to use \( |x| + |y| \) as variant function instead, but the program still would not work! (A broken program is a broken program, and no amount of messing around with the variant function can help that.) Since \( |x| + |y| \) is clearly bounded below by zero and starts out finite, the only thing that can go wrong is proving that it decreases. And you cannot prove that, because what happens is that \( V \) in fact increases if \( x < 0 \) or \( y < 0 \); \( V \) stays constant if \( x > 0 \land y = 0 \) or the other way around. As usual, the working out of the details is left as an exercise for the reader.