CS221 and CS321: Programming Languages
Autumn Quarter, 1997

Loop Invariants and their Relevance to Designing and Reasoning about Programs

There are some important lessons to be learned from our discussions in class pertaining to the correctness of programs. These lessons are the following:

- Identifying loop invariants and ways of progressing towards the goal while preserving these invariants is a central part of programming.
- Once such loop invariants are identified, the proof rules (or axiomatic semantics) associated with each statement provide a natural encoding of the way we would try to argue for the correctness of programs.
- Actually, loop invariants should be used directly in the *design* of programs. They are critical in determining their structure anyway and making their role explicit allows us to develop programs *and* show their correctness at the same time.

There are auxiliary remarks to be made as well. First, in the whole process we have presumed that the control flow statements in our language have a well defined axiomatic semantics. Of course, this need not always be true and what our exercises bring out is the importance of this being true. Second, the loop invariants turn out to be the heart of programming and, so, another point that is stressed is that of choosing invariants well; this can have a significant impact on the facets such as efficiency, simplicity and clarity of programs.

These points are all very simple to state. However, the *only* way to appreciate them is to go through a few detailed exercises. This can get a little boring and trying to assimilate details in a short period can get confusing. However, it would be unfortunate if the important points are missed as a result. So, what I have done below is sketched out the details for you so that you can go through them slowly. Take a break when you start feeling dizzy from details and recall the big picture at these points. And remember: this is really the *only* time in this course that you will have to work through the details of the proof rules for assignment and conditional statements and the consequence rule. Henceforth, we will assume that we are dealing with intelligent humans (rather than dumb computers) who can agree on the truth or falsity of simple Hoare-triples and the only time explanation is needed is when the truth of the triple is not readily apparent.

The Proof Rules

First, let us recall the rules pertaining to the various statements in our language. These rules are reproduced in Figure 1. We have spent some time talking about these rules in class and their content is, hopefully clear by this point. The one rule that is perhaps a little difficult to understand is the *While* rule. To reiterate, in thinking about a *while* statement in a program, the key to success is to find some condition that is true *every* time around the loop. The premise of this rule (*i.e.* the portion of the rule that appears above the line) involves showing that this condition is, in fact, an invariant: assuming that it is true *before* the body of the loop is executed *and* the body is actually executed, which happens only if the test is true, then it is true *after* the body finishes execution. If we succeed in showing this, then we can conclude that if the condition is true before the *while*

$$\frac{}{\{R[E/x]\}\ x := E\ \{R\}}\ \textit{Assignment}$$

$$\frac{\{R\}\ S_1\ \{Q'\} \qquad \{Q'\}\ S_2\ \{Q\}}{\{R\}\ S_1; S_2\ \{Q\}}\ \textit{Composition}$$

$$\frac{\{R\ \textit{and}\ E\}\ S_1\ \{Q\} \qquad \{R\ \textit{and}\ (\textit{not}\ E)\}\ S_2\ \{Q\}}{\{R\}\ \textit{if}\ E\ \textit{then}\ S_1\ \textit{else}\ S_2\ \{Q\}}\ \textit{Conditional}$$

$$\frac{\{R\ \textit{and}\ E\}\ S\ \{R\}}{\{R\}\ \textit{while}\ E\ \textit{do}\ S\ \{R\ \textit{and}\ (\textit{not}\ E)\}}\ \textit{While}$$

$$\frac{R\ \textit{implies}\ R' \qquad \{R'\}\ S\ \{Q'\} \qquad Q'\ \textit{implies}\ Q}{\{R\}\ S\ \{Q\}}\ \textit{Consequence}$$

Figure 1: Proof Rules for Control Constructs

statement starts executing, then it is true after it finishes, if it does, in fact, finish. Of course, at this stage you can additionally assume that the test of the loop is false.

Reasoning about the Naive GCD Program

Now let us consider the simple program for finding the gcd of two numbers that is written out below:

```
{ (n0 > 0) and (m0 > 0) }

   n := n0;
   m := m0;
   if n > m then i := m
   else i := n;
   while ((n mod i) <> 0) or ((m mod i) <> 0)
   do i := i - 1;

{ i = gcd(m0,n0) }
```

I have included a pre-condition and a post-condition in this program that corresponds to showing that the program does what we want it to do if it terminates: given two positive, non-zero integers, it finds their greatest common denominator. (One may ask that the value of `i` be printed out at this point if desired.) Saying that this is what is encoded in the pre-condition and post-condition actually requires noting that the values of `n0` and `m0` themselves are unchanged in the program. This is not difficult to see, especially if we assume that these are only input values and not variables.

What we have actually written down above is an assertion of the kind we looked at in class: in particular, a triple consisting of a pre-condition, a program and a post-condition. How do we show that this triple represents a true assertion? Clearly we have to do this by using the proof rules for the various statements that form the program. However, it is useful to think first of a principle for organizing our use of these rules.

The key here, as in every other interesting case, is to think of *loop invariants*: these will form the intermediate properties in the program that we reason towards both from the beginning of the program and the end. And, as we have stressed a few times before, thinking of the loop invariants involves thinking about the main *insight* in programs.

So what is the main insight in the above (naive) *gcd* program? This insight, if one might indeed call it that given its simplicity, is based on the definition, so to speak, of the *gcd* function:

> i is the *gcd* of m and n just in case it is the largest factor of m and n.

Using this definition, we can rewrite the post-condition of the program as follows:

```
((n0 mod i) = 0) and ((m0 mod i) = 0)   and
for all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)
```

Notice that this is really a statement about the value of a *particular* program variable, namely `i` and thus constitutes a description of instantaneous states. Now, if you look at the program again, you will see that what it does is the following: it picks an initial value for `i` that meets the part of this condition that is on the second line, and keeps reducing this value till the part of the condition on the first line is met.

This then is the main insight in the program. If you think about this differently you will see that we have described the loop invariant for the above program *and* the way in which the program tries to get closer to its goal *while* maintaining the invariant. These are the two crucial ingredients of any well designed program that uses loops.

Let us put in the loop invariant at the right place in the program and see what we get:

```
 { (n0 > 0) and  (m0 > 0) }

    n := n0;
    m := m0;
    if n > m then i := m
    else i := n;

 {For all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)}

    while ((n mod i) <> 0) or ((m mod i) <> 0)
    do i := i - 1;

 {((n0 mod i) = 0) and ((m0 mod i) = 0)   and
  for all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)}
```

Before going any further, think about the program above with the properties attached at the relevant places. Two things should strike you already. First, you should be able to see at least intuitively that the properties do hold whenever control reaches the points where they appear. Second, putting in the loop invariant really does capture some of the essence of the program. We will go through the details of showing that the properties inserted into the text of the program hold as they are supposed to, and in this process we will understand the specific rules for statements. However, these are, in a sense, just details and before you immerse yourself in them you must understand the "big picture" clearly.

Now we can get to the details. We can split these up into two tasks: showing that what we are thinking of as the loop invariant is true the *first* time control arrives at beginning of the loop, and that, given this condition, the post-condition for the program is true *after* the loop finishes execution. The latter part will involve showing that our "loop invariant" in in fact really one.

At this stage we make some cosmetic changes to the loop invariant. These changes are needed for the following reason: we are trying to show that `i` is the gcd of `m0` and `n0`, but the statements in the program (*e.g.* the tests in the *if-then-else* and the *while*) refer to the variables `m` and `n`. For our reasoning processes to go through smoothly, we have to note that the values of `m` and `n` are always identical to `m0` and `n0` respectively. We do this by including this property in the loop invariant.

Let us now consider the second of the assertions that we have to show, namely

```
{(n = n0) and (m = m0) and
 for all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)}

   while ((n mod i) <> 0) or ((m mod i) <> 0)
   do i := i - 1;

{((n0 mod i) = 0) and ((m0 mod i) = 0)   and
 for all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)}
```

The program statement in this assertion is a *while* and we will therefore have to use the *While* rule to prove this assertion. One requirement of the *While* rule is that the post-condition be identical to the pre-condition and the negated test, i.e. that

```
((n0 mod i) = 0) and ((m0 mod i) = 0)   and
for all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)
```

be identical to

```
(n = n0) and (m = m0) and
for all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)
and (not (((n mod i) <> 0) or ((m mod i) <> 0))).
```

Let us refer to these conditions as F' and F respectively. Now, of course, F' and F are not identical. However, the *Consequence* rule allows us to tide over the differences. In particular, since F *implies* F', it is enough that we show

```
{(n = n0) and (m = m0) and
 for all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)}

   while ((n mod i) <> 0) or ((m mod i) <> 0)
   do i := i - 1;

{(n = n0) and (m = m0) and
 for all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)
 and (not (((n mod i) <> 0) or ((m mod i) <> 0)))}
```

is true. Recall that we referred in class to the *Consequence* rule as the glue needed for combining the use of the various other rules together. The sense in which this is meant is illustrated here.

Looking at the details of the *While* rule gets us to the heart of the matter: showing that the loop invariant is in fact one. This amounts to showing that if the necessary condition is true before the statement `i := i - 1` gets executed and the statement *does* get executed, then it is true afterwards. In other words, we need to show that the assertion

```
{(n = n0) and (m = m0) and
 for all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0) and
 (n mod i) <> 0) or ((m mod i) <> 0)}

  i := i - 1

{(n = n0) and (m = m0) and
 for all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)}
```

is true. Do this as an exercise to see that you are following the details properly. You will need to use the *Assignment* and *Consequence* rules in this process.

The remaining task is to show the following assertion to be true:

```
{ (n0 > 0) and (m0 > 0) }

   n := n0;
   m := m0;
   if n > m then i := m
   else i := n;

{(n = n0) and (m = m0) and
 for all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)}
```

Notice that what the *if-then-ese* statement does is set `i` to the smaller of `n0` and `m0` based on the fact that nothing larger than this can be a factor of both `m0` and `n0`. This suggests inserting the conjunction of the post-condition with `j` replaced by `m0` and `n0` respectively before the *if-then-else* statement. With some simplifications, we get the following:

```
{ n0 > 0, m0 > 0 }

   n := n0;
   m := m0;

{(n = n0) and (m = m0) and
 for all j greater than n0 either ((n0 mod j) <> 0) or ((m0 mod j) <> 0) and
 for all j greater than m0 either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)}

   if n > m then i := m
   else i := n;

{(n = n0) and (m = m0) and
 for all j greater than i either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)}
```

Once again, our task of showing that the required conditions hold at the relevant points splits in two parts. For the first part, i.e. to get to the intermediate condition from the pre-condition, we proceed as follows. First, we use the *Assignment* rule twice to note the truth of the following triple:

```
{(n0 = n0) and (m0 = m0) and
 for all j greater than n0 either ((n0 mod j) <> 0) or ((m0 mod j) <> 0) and
 for all j greater than m0 either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)}

   n := n0;
   m := m0;

{(n = n0) and (m = m0) and
 for all j greater than n0 either ((n0 mod j) <> 0) or ((m0 mod j) <> 0) and
 for all j greater than m0 either ((n0 mod j) <> 0) or ((m0 mod j) <> 0)}
```

Now we observe that pre-condition in the fragment above is implied by `(n0 > 0) and (m0 > 0)`. Notice that the non-negativeness of `n0` and `m0` get used in an essential fashion in this process and so our program has not been shown to be correct without it. Notice also that certain properties of arithmetic get used in establishing this implication and these would have to be made explicit if we try to get a machine to carry out the proof by itself. Given the implication and the truth of the triple above, we can use the *Consequence* rule to complete the argument for the first part.

For the second part we will need to use the *Conditional* rule and, once again, the *Consequence* rule. You should work through this part just to make sure you have understood all the rules sufficiently well.

After all this effort, we have only shown the program to be partially correct. What is the missing part to total correctness? Well, we have not shown that the program will terminate! And how do we do this? The key is to show that the loops in the program terminate because without loops a program will always terminate whether normally or abnormally.

So how do we show termination in this particular case? The general method is to find something that decreases each time around the loop and that also has a smallest value. In the case of the gcd program above we can use the value of the variable `i` for this purpose. Notice that value of `i` cannot be less than 1. (Why?) Further, this value diminishes by 1 each time around the loop. It is therefore untenable that the loop go on for ever.

This was a lot of work just to show that a simple program is correct! Furthermore, we haven't really written out the proof of partial correctness in a formal enough fashion: for this we would have to write of a detailed derivation of the program with its pre- and post-conditions. If you think about this a little, you will see that writing out such a proof can be extremely tedious. Fortunately, there is a reprieve as we have noted. First, in communication between humans, lots of little details can be omitted. You should do this for your assignment and I would have done so even here except that I wanted, this one time, to illustrate carefully the way in which the proof rules fit in to a correctness argument. The second observation is that the tedious verification of details can be left to a computer and, in some sense, all that we are doing here is seeing that the matter can be reduced to a level where the computer can carry it out.

Designing a less naive GCD program

The tediousness of a detailed verification of a program should not cause the important points to be missed. First, whether we like it or not, the correctness of our programs is critically dependent on a precise understanding of the effects of program statements. Such an understanding can be provided through proof rules or in some other way, but it must be provided in a way that it can be utilized. Notice that, at one end, the *goto* statement has a precise meaning but not one that can be presented in a way that is useful in the overall process of reasoning about programs. At the other end there are statements whose meaning is dependent on the machine on which we run them on and this is, once again, not a happy state of affairs from the perspective of showing the correctness of programs.

The second, perhaps more important, point is the centrality of loop invariants to correctness arguments and to programming itself. We certainly saw the first facet of loop invariants in the earlier discussions. We got a glimpse of the second in the process of arriving at the loop invariant: thus, we first noted the crucial insight underlying the program and this led us on to the loop invariant. However, this point can be brought out in a different and sharper way as well. Here is the critical thought: *the same problem can be solved in different ways and this boils down ultimately to picking different loop invariants.* Thus, the choice of loop invariants (and of ways of maintaining these while proceeding towards a desired goal state) lies at the very heart of creativity in programming.

Let us look at an example towards bringing this point out. Clearly, what we will have to do is think of a different GCD program and link this to a different choice of loop invariant. The method that I will now describe eventually underlies what is known as Euclid's algorithm. Perhaps you know it already and, in any case, it is a simple program. However, I think it will help in making the essential point.

So what is the main mathematical insight underlying this method? It is the following observation: if m and n are two positive, non-zero integers and, in addition $m > n$, then $gcd(m, n) = gcd(m - n, n)$. How do we know this? Well, we observe that, under the conditions described, any factor of m and n must also be a factor of $m - n$ and n and conversely. This is easy to see and so I leave it as an exercise.

Here is how we can use this insight. Suppose we are interested in finding the gcd of the two numbers n_0 and m_0. We will use two integer variables n and m whose values are such that $gcd(n, m) = gcd(n_0, m_0)$. Our goal will be to transform the values of these variables into a form from which their gcd can be easily read off. And when might this be possible? Well, when their values are equal. And how might one transform them? Well, by using the mathematical identity just described: if $n > m$ then we replace the value of n by $n - m$ and if $m > n$ we replace the value of m by $m - n$.

Let us introspect on what we have just done. We have, in fact, identified

(1) a (loop) invariant which is $gcd(n, m) = gcd(n_0, m_0)$,

(2) a desired goal that transparently yields the result we desire, this being $n = m$ and $gcd(n, m) = gcd(n_0, m_0)$, and

(3) *and* a way of getting closer to this goal while maintaining the invariant.

As we see below, we can quickly pull out a program from this and be quite certain that it is correct.

So let us write out the skeleton of the program with properties that we want to hold at various points. The skeleton will be very sparse at the first instance in that it will contain no code in it! In fact, it is the following:

```
{ (n0 > 0) and (m0 > 0) }

  some code to be put in soon

{ gcd(n,m) = gcd(n0,m0) }

  more code to be put in

{ n = m and gcd(n,m) = gcd(n0,m0) }
```

To complete the program, we need (merely!) to insert the right code at the right places.

What code can we put in to make `gcd(n,m) = gcd(n0,m0)` in the first instance? This is really quite obvious. We simply need to set `n` and `m` to `n0` and `m0` respectively. And how about the code between the invariant and the "goal" condition? Here we need a loop that uses the rule described above and whose exit condition guarantees the goal state. The last requirement makes the exit condition obvious: it should be `n = m`. Putting all these thoughts together, we get the following program:

```
{ (n0 > 0) and (m0 > 0) }

  n := n0;
  m := m0;

{ gcd(n,m) = gcd(n0,m0) }

  while (n <> m) do
   if (n > m) then n := n - m
   else m := m - n;

{ n = m and gcd(n,m) = gcd(n0,m0) }
```

From the way we have developed the program, its partial correctness is obvious. How about termination? This is simple and only involves making precise our sense that we are in fact getting closer to the goal each time around the loop. Formally, notice that *every* turn around the loop decreases the value of `(n + m)` and that this sum must always be greater than 0.

We had noted at the very outset that, in addition to coming up with a loop invariant, the identification of a way of maintaining the invariant while making progress towards a desired goal is also a critical part of programming. The present context is a good place to bring this point out. Suppose we define $gcd(n, 0)$ to be n. Then, if $n \geq m > 0$, it can be seen that $gcd(n, m) = gcd(n \ mod \ m, m)$. Using this observation and by strengthening our loop invariant slightly to

```
n >= m >= 0 and gcd(n, m) = gcd(n0,m0)
```

we can design a different and more efficient program that the one that we have just finished discussing. If you are not familiar with this program/algorithm, try arriving at it by the process described in this handout. This program is what is known as Euclid's algorithm.

It is now time to revert to a higher level. Notice that the two new programs that we have just arrived at are different from each other and also the earlier one whose verification we had labored over. Notice also that the essence of the difference is in the loop invariant and the way in which we proceed towards a goal condition while maintaining the invariant. Finally, notice that thinking of developing the program through the invariant leads us quickly to a transparently correct program. This is *not* to say that choosing the right invariant is easy. Undoubtedly, this could be very difficult, depending on the problem and the attributes of the solution (such as efficiency, clarity, simplicity) that are desired. However, what is being said is that we should focus first on what is really the hard part (the choice of invariants, the method of progress) and look at the simple details (the precise statements to be used, etc) only after we have complete conceptual control of these difficult issues.

Some final comments. The gcd program may look simple, but the principles discussed using it are very general. Look at the discussions in the book and the binary search program in Jon Bentley's paper for other examples. To really convince yourself of the relevance of the ideas discussed here to programming, take up Bentley's challenge and try to write a binary search program *before* you look at his solution. Read Bentley's paper carefully — you will learn a lot from it and it will be useful in solving the assignment. Finally the assignment itself will give you another opportunity to appreciate the importance of these ideas in program development. Incidentally, for the assignment, a presentation of correctness arguments similar to that used for the second gcd program will be adequate. Make sure though that observations of any non-obvious properties are supported by adequate arguments. These arguments can be informal in that you need not (and really should not) use proof rules in your presentation, but the arguments do need to be precise.