

CSE 127 Lecture Notes: Proving Code Correct

Bennet Yee

Feb 24 and 26, 2003

1 Basic Concepts

There are three main notions used in proving code correct:

- the *precondition*, or what is assumed to hold true before a statement executes,
- the *postcondition*, or what is proven to hold after a statement executes, and
- the *loop invariant*, a tool for proving loops to have some desired property.

Generally, we write these statements in clear mathematical notation, using predicate calculus. Often we will see proof fragments that look like

$$\{C_{\text{pre}}\}S\{C_{\text{post}}\},$$

where S is a statement or group of statements in our computer programming language, and C_{pre} and C_{post} are the pre- and postconditions¹. In this notation, C_{pre} and C_{post} are subsets of the space of all possible correspondence of values to variables (the variables' *bindings*), so these subsets are defined by boolean expressions that characterise these subsets. You can think of the expression for C_{pre} as a way of describing the set of facts that we're willing to assume prior to the execution of the statement S , and the expression for C_{post} as describing the result of executing S . Do not be confused, however: the sets contain program states, not boolean expressions. In particular, the subset of the state space being described are *smaller* the more statements you have, i.e., $\{i = 2\} \supseteq \{i = 2 \wedge j = 3\}$. When there are many expressions that must be satisfied, the number of execution states that satisfy all the requirements will necessarily be smaller.

A simple example is

$$\{i = 2 \wedge j = 3\}k=i*j; \{i = 2 \wedge j = 3 \wedge k = 6\},$$

where we simply record in the post-condition the effect of running the statement $S := k=i*j$; on what is assumed to be true in the pre-condition. The execution of S adds the restriction $k = 6$ to the precondition, i.e., $C_{\text{post}} = C_{\text{pre}} \cap \{k = 6\} = \{i = 2 \wedge j = 3 \wedge k = 6\}$.

The statement S may be a simple statement such as the assignment statement above, or it may be a complex statement such as a sequence of statements, a compound statement such as a loop, or an expression involving function calls, etc.

The loop invariant is a logic statement \mathcal{I} that helps us prove some property \mathcal{P} holds when the loop terminates. The way we use it is as follows: (1) We first show that the invariant \mathcal{I} is true prior to entering the loop, i.e., that it can be inferred from the preconditions for the loop. (2) Next, we assume that the invariant is true at the test at some iteration of the loop, and we trace the body through one execution to

¹This notation arose because the pre/postconditions are intended to be annotational comments to the program. Pascal (and Fortran) were the official instructional programming language for the ACM when much of this work was done, and braces—or the parenthesis and asterisk construct—was used to enclose comments in Pascal, so it is my conjecture that this is why this notation is the way it is. Of course, for those of us more familiar with C, C++, and Java where braces serve to group statements (viz, BEGIN/END delimiters in Pascal), this notation can look weird. Just be careful to distinguish statements in logic versus programming language statements being enclosed, and there will be no ambiguity.

Figure 1: Our Favorite Function

```

Integer modexp(Integer x, Bit eA[], Integer n)
{
    Integer y = 1, z = x;
    int i;
    for (i = 0; i < eA.length; i++) {
        if (0 != eA[i]) {
            y = y * z mod n;
        }
        z = z * z mod n;
    }
    return y;
}

```

show that the invariant remains true after the body executes once. N.B.: the invariant may become false during the execution of the body—all we want is that after the entire body completes, the invariant is again true. If you remember your induction proofs in mathematics, you’ll notice that the use of loop invariants is very similar: what we’re doing first is showing that the invariant holds in the base case, and the second step is the induction step. With loops, of course, we are proving a property that holds when the loop exits, so we don’t let the induction go to infinity, but the distinction is minor: the loop body is executed an arbitrary but finite number of times (or the index variable overflows!), and the invariant remains true for all executions. (3) Finally, we combine the loop invariant property \mathcal{I} with the loop termination condition to obtain the desired property \mathcal{P} for the loop termination.

2 An Extended Example

In order to see how it all works, we’ll use our favorite function as an extended example. Figure 1 reprises the `modexp` function that we’ve worked with many times.

Here, the `for` loop is a little awkward to work with, and often the analyst will convert the `for` loop into a simpler form—a `while` loop—as seen in Figure 2, isolating the test so that it has no side-effects (e.g., `++` operator). In this example, we will check that the invariant \mathcal{I} is true at position A. In the next step for the proof of correctness for the loop (a lemma for the overall proof), we will *assume* that \mathcal{I} is true at B, and by tracing the execution symbolically and keeping track of pre-/post-conditions for the statements in the body of the loop, we will prove that \mathcal{I} holds at position C.

2.1 Keeping Track

Suppose we wish to prove that `modexp` is correct, i.e.,

$$\{C_{\text{pre}}\} \text{w=modexp}(x, e, n); \{C_{\text{post}}\}.$$

First, we should determine what C_{pre} and C_{post} are. Clearly, since `modexp` computes the modular exponentiation of its arguments, C_{post} should contain a statement to that effect, e.g., $C_{\text{post}} \subseteq w = x^e \bmod n$, where to be precise we should define

$$e = \sum_{i=0}^{\text{eA.length}-1} \text{eA}[i] \cdot 2^i$$

to relate the integer e with the contents of the array `eA` (how the contents of the bit array should be interpreted). The list of preconditions should be minimal—the only assumptions needed for the execution

Figure 2: Simplified Function

```

Integer modexp(Integer x, Bit eA[], Integer n)
{
    Integer y = 1, z = x; // a
    int i;
    i = 0;
    // A
    while (i < eA.length) { // B
        {
            if (0 != eA[i]) {
                y = y * z mod n;
            }
            z = z * z mod n;
        }
        i++;
        // C
    }
    return y;
}

```

of S to result in C_{post} being true. Clearly, $C_{\text{pre}} \subseteq \{0 \leq x < n \wedge 0 \leq e\}$ should hold, and perhaps more are needed.

At point **a** in the code, we carry the preconditions of `modexp` through the initialization, and what we know to hold is $\{0 < n \wedge 0 \leq x < n \wedge 0 \leq e \wedge y = 1 \wedge z = x\}$. After we reach **A**, we would add in $\{i = 0\}$.

2.2 The Invariant

The invariant \mathcal{I} here is

$$\mathcal{I} = \left\{ z = x^{2^i} \bmod n \wedge y = x^{\sum_{j=0}^{i-1} eA[j] \cdot 2^j} \bmod n \right\}$$

capturing the relationships between the variables.²To see how this is arrived at, we have to understand how the algorithm works. The key observation is that the `modexp` algorithm uses repeated squaring of x to efficiently compute $x^e \bmod n$ using the binary expansion of e . This involves $O(\log e)$ squaring operations, rather than $O(e)$ operations. This is critical for performance when e is a 1024-bit integer—the largest possible value of e is $2^{1024} - 1$, which means that an $O(e)$ algorithm would not terminate in your lifetime, and probably not the lifetime of this universe. We can prove the different clauses in the invariant separately. The easiest is $z = x^{2^i} \bmod n$, so we'll start there.

2.2.1 Invariance of $z = x^{2^i} \bmod n$

To show that $z = x^{2^i} \bmod n$ is a loop invariant property, we must check that it holds before entry to the loop (at **A**) and that it is preserved by an execution of the loop body (assume true at **B**, then prove that it is again true at **C**). To check at **A**, we just plug in the values that we know: $z = x^{2^i} = x^{2^0} = x^1 = x$. To check that this property is preserved by an execution of the loop body is slightly harder.³ We first note that the only statements which affect the values of the variables in the expression need to be considered,

²Note the usual mathematical convention where $\sum_{j=m}^n x_j \doteq 0$ when $m > n$.

³Note that normally we would name the initial values in the variables separately to keep track of how the current values relate to the original input as the program executes, e.g., x' would be the value passed in to the function `modexp` in the variable x , etc, but in this case the inputs are never modified, so this is unnecessary.

i.e., $\{z = x^{2^i} \bmod n\}$ only uses z , x , and i , and x is invariant (unchanged) throughout the execution of the function, so we can ignore the `if (0 != eA[i]) { ... }` block. Next, we make sure to distinguish the variables and the values that they held before the body is executed, e.g., we refer to the values of the variables z and i at B as z' and i' and use plain z and i to denote the (varying) contents of the variables as the loop body is executed. We assume that the property is true at the beginning of the current loop execution, i.e., $z' = x^{2^{i'}} \bmod n$. After `z=z*z mod n`; is executed, we know that $z = z' \cdot z' \bmod n$, so

$$\begin{aligned} z &= x^{2^{i'}} \cdot x^{2^{i'}} \bmod n \\ &= x^{2^{i'}+2^{i'}} \bmod n \\ &= x^{2 \cdot 2^{i'}} \bmod n \\ &= x^{2^{i'+1}} \bmod n \end{aligned}$$

The remaining statement, `i++`;, results in $i = i' + 1$, so we can plug this into the previous equation, obtaining $z = x^{2^i} \bmod n$, showing that the property holds after a single execution of the loop body, at C.

2.2.2 Invariance of $y = x^{\sum_{j=0}^{i-1} eA[j] \cdot 2^j} \bmod n$

Next, we do the same thing for

$$y = x^{\sum_{j=0}^{i-1} eA[j] \cdot 2^j} \bmod n.$$

Again, we check that this property holds at A: $x^{\sum_{j=0}^{i-1} eA[j] \cdot 2^j} = x^0 = 1 = y$, so we're fine. Then, we assume it holds at B, and denote the values at that point by y' and i' . We know then that $y' = x^{\sum_{j=0}^{i'-1} eA[j] \cdot 2^j}$. Next, we trace the execution of the loop body and see how it changes the variables. As we enter the loop body, we are immediately faced with the conditional statement `if (0 != eA[i]) ...`, so we have to consider the two cases separately.

Case: $eA[i] = 0$ If $eA[i'] = 0$, then the body of the `if` statement is not executed, and the only effect is the modifications to z and i :

$$z = z'^2 \bmod n \wedge i = i' + 1.$$

Since y is unchanged, we have

$$\begin{aligned} y &= y' \\ &= x^{\sum_{j=0}^{i'-1} eA[j] \cdot 2^j} \bmod n \\ &= x^{\sum_{j=0}^{i'-1} eA[j] \cdot 2^j} \cdot 1 \bmod n \\ &= x^{\sum_{j=0}^{i'-1} eA[j] \cdot 2^j} \cdot x^{eA[i'] \cdot 2^j} \bmod n \\ &= x^{(\sum_{j=0}^{i'-1} eA[j] \cdot 2^j) + eA[i'] \cdot 2^j} \bmod n \\ &= x^{\sum_{j=0}^{i'} eA[j] \cdot 2^j} \bmod n \\ &= x^{\sum_{j=0}^{i-1} eA[j] \cdot 2^j} \bmod n \end{aligned}$$

this property holds with respect to the loop when $eA[i'] = 0$.

Case: $eA[i'] = 1$ If $eA[i'] = 1$, then the body of the `if` statement is executed. So, we have

$$y = y' \cdot z' \bmod n \wedge z = z'^2 \bmod n \wedge i = i' + 1.$$

In class I include $x^e = y \cdot z^{\sum_{j=i}^{eA.length-1} eA[j] \cdot 2^j} \bmod n$ as an invariant property. While it is invariant, we don't really need it in the proof since we could just use the y invariant directly (see 2.3).

Simplifying as before, we have

$$\begin{aligned}
y &= y' \cdot z' \bmod n \\
&= x^{\sum_{j=0}^{i'-1} eA[j] \cdot 2^j} \cdot x^{2^{i'}} \bmod n \\
&= x^{\left(\sum_{j=0}^{i'-1} eA[j] \cdot 2^j\right) + 2^{i'}} \bmod n \\
&= x^{\sum_{j=0}^{i'} eA[j] \cdot 2^j} \bmod n \\
&= x^{\sum_{j=0}^{i-1} eA[j] \cdot 2^j} \bmod n
\end{aligned}$$

so the property holds when $eA[i'] = 1$. Since the property holds for all possible values of $eA[i']$, the property is invariant with respect to the the loop.

2.3 Using the invariants

The z invariant was useful for proving the y invariant, and the y invariant will help us get the postcondition. The general idea is that we use the termination condition for the loop and the invariant to show that *if the loop terminates, we get the desired result*. We need to also prove that the loop indeed terminates; in this case, it is clear from inspection, since i monotonically increases by 1 each iteration from a starting value of 0, and so the loop will terminate when $i = eA.length$. (We assume that our representation of e in eA implies that $eA.length > 0$ holds, so we won't run into the situation where $eA.length < 0 = i$.) When the loop terminates, we know:

$$z = x^{2^i} \bmod n \wedge y = x^{\sum_{j=0}^{i-1} eA[j] \cdot 2^j} \bmod n \wedge i = eA.length$$

so we can simplify this, obtaining

$$\begin{aligned}
y &= x^{\sum_{j=0}^{eA.length-1} eA[j] \cdot 2^j} \bmod n \\
&= x^e \bmod n
\end{aligned}$$

which gives the return value, so the postcondition $C_{\text{post}} = \{w = x^e \bmod n\} \cap \{0 \leq x < n \wedge 0 \leq e\}$ holds. \square

3 Weakest Precondition

The notion of the *weakest precondition* of a piece of code is important for knowing how to use that code correctly—it is the most general and least restrictive condition that is required for the execution of that code to result in the desired postcondition. For example, the `modexp` function's precondition of

$$\left\{ 0 \leq x < n \wedge 0 \leq e = \sum_{j=0}^{eA.length-1} eA[j] \cdot 2^j \right\}$$

is weaker than

$$\left\{ 0 < x < n \wedge 0 < e = \sum_{j=0}^{eA.length-1} eA[j] \cdot 2^j \right\}$$

, since the first set is larger. It is not, however, the weakest precondition: we do not need to assume that $0 \leq x < n$ holds, since all nontrivial assignments to y involves a `mod n` operation. Thus, the weakest precondition is

$$\left\{ 0 < n \wedge 0 \leq e = \sum_{j=0}^{eA.length-1} eA[j] \cdot 2^j \right\}$$

—as long as the $\text{mod } n$ operation always results in a number that is between 0 and $n - 1$ inclusive, i.e., $\forall x \in \mathbb{Z} : 0 \leq x \text{ mod } n < n$, there will not be any problems with returning a value that is outside of the desired range.

Knowing the weakest precondition C_{wp} for some piece of code means that anytime you have the actual precondition C_{pre} for a given use of the code, you can determine if the postcondition will be guaranteed by testing to see if $C_{\text{wp}} \stackrel{?}{\supseteq} C_{\text{pre}}$ holds.

4 Verifying Preconditions

There is no single right answer to whether you should verify preconditions when writing code. You certainly would not do so for simple statements like our `k=i*j` example above. For larger code blocks like loops or functions, you should usually add in assertions that verify that the preconditions hold, especially for code that is “tricky” or complex. Sometimes the expense of verifying preconditions is very high. For example, if one input to a function is a sorted balanced binary tree, verifying this property requires a complete tree walk—when the operation to be performed should cost $O(\log(n))$, where n is the number of elements in the tree (e.g., insert, delete), doing an $O(n)$ precondition check is unacceptable. Sometimes you would include precondition checks only for “debug” versions of the code, so that the expense is not paid after you are confident that the code is correct.

Some languages like C or C++ provide preprocessor mechanisms for conditional compilation so that debug code can be switched in or out easily. You do *not* want to manually comment out debug code, since that is error prone and you want to make it easy to switch the debug code on or off—new bugs may get introduced later, and having the ability to turn on the checks easily can be invaluable. Some people advocate leaving the debug code in the final build, but have its operation be controlled via a global variable; this way, the only difference between the released binary and the test binary is the value of the global debug flags (different flags can control different groups of debugging code), and this reduces the likelihood that stray pointer bugs etc would behave radically differently in the production code as compared with the test code.

What is the appropriate thing to do after detecting a precondition violation depends on many factors, such as the programming language used and the API design. If the language has exceptions (e.g., Java, C++), it is often appropriate to check the preconditions and throw an exception if they do not actually hold. Sometimes you are faced with the choice of aborting the program (e.g., how the `#include <assert.h>` macros in C works), or of returning a special error value. Generally, if you detected that the application programmer did not ensure that the preconditions held, you have detected a serious bug in the code, though like throwing exceptions it would seem nicer to make the error recoverable (by the calling code) by returning an error indication instead of aborting the program. One consideration: if the programmer who is calling your code had trouble with the API semantics and did not satisfy the precondition, why should you have any confidence that the programmer will correctly check for these error indication return values? Usually it is better to make sure that the program either handles errors correctly or stops, since letting the program continue with bad data (especially when an error indication is treated as a valid data value) can let the errors magnify, though library code API design often requires that the caller be allowed to handle errors and so failures should be “silent”: no printing to standard error, no aborting the program, just a clean error indication through exceptions or a special return value.