UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS263
Spring 1996 **Author: K. Yelick**

Program Verification

1 Specification

Before we can talk about testing or verifying the correctness of program modules, we need to know what it means for a program to be correct. The first step, which is not always obvious, is writing a specification for the behavior of the module. For functions, whether they are member functions or standalone functions, we have already discussed a particular style for writing these specifications. Each function has a *precondition* that the caller must ensure before invoking the function, and a *postcondition* that will hold after the function is called, assuming the precondition was true. We often omit the precondition if there is nothing interesting to say. In this case the precondition is simple the predicate *true*, since in this case the precondition will always be satisfied.

A common pitfall in writing specifications is to talk about the implementation of the function or class, rather than its behavior. This tells the user, or *client*, of the module more than they need to know and it can overly-constraint the module implementor's choice of algorithms or representations. For the purpose of program verification, specifications have to be precise, so the meaning is clear even if it is somewhat complicated. Although most programmers do not prove the correctness of each module before or after they implement it, understanding how proofs are done will help you write better informal as well as formal specifications.

2 Testing vs. Verification

For now, we will consider primarily the verification and testing of functions, since the primary problem in proving that a class implementation is correct is proving that each of its member functions is correct. Give a specification of a function and an implementation, there are two approaches one might use to make sure the implementation correctly implements the specification: testing and verification. *Testing* shows that the program is correct on some examples inputs by running the function on several possible inputs and checking that the output of the function along with variables that could be modified by the function. If the test cases are chosen to cover the possible correct as well as exceptional inputs and the boundary cases between them, we can be fairly confident that the function is correct. (Automatically choosing test cases is a matter of ongoing research, and a big problem with testing is adequate coverage of the kinds of inputs.) In contrast to testing, *verification* is done by proving that a program works correctly for all inputs. Verification is done by examing the program text and the specification; it does not involve running the program.

3 Correctness

What does is mean for a procedure to be a correct implementation of a specification? There are two standard notions of correctness:

- Partial correctness: if the precondition is true when the procedure is invoked, then if the procedure terminates, the postcondition must be true when the procedure returns.
- Total correctness: if the precondition is true when the procedure is invoked, then the procedure must terminate, and the postcondition must be true when the procedure returns.

We generally want to prove the total correctness property, which is partial correctness plus a guarantee of termination. The two notions are separated because they require different proof techniques, so we will have to think of each on separately and sometime even give two independent proofs.

There is some standard notation in program verification, which is:

```
{ P }
program text
{ Q }
```

This says that if `P` is true before executing the given program text, then `Q` will be true afterwards. If the program text were a procedure body, then `P` and `Q` would be the precondition and postcondition, respectively. This syntactic convention came about before C or C++ did, and is confusing in the context programs with curly braces. For C or C++ program, we often use the `assert` notation instead:

```
/* assert (P); */
program text
/* assert (Q); */
```

Neither partial nor total correctness says anything about what happens if the precondition is false when the procedure is called. If P is false before executing the program text, then the correctness condition is, "if false is true before executing the program text, then Q will be true afterwards." This is vacuously true, regardless of what the program does.

In some cases we could remove the comments around P and Q, and test that are assertions are true (in some example input) by running the program. This is a good idea if P and Q are valid program expressions and if they are relatively inexpensive, but sometimes we will not be writing code for P and Q. (A common example is a conditional on the size of a C-style array, which is not something we can test explicitly from within C or C++.) Even for assertions that are executable, we will not want to execute them between every pair of statements in the program.

4 Proof Rules

With this notation, we can write inference rules such as the following.

$$\frac{P \supset P', \quad \{P'\}\ S\ \{Q'\}, \quad Q' \supset Q}{\{P\}\ S\ \{Q\}} \qquad \textit{Consequence}$$

$$\frac{\{P\}\ S_1\ \{R\}, \quad \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1; S_2\ \{Q\}} \qquad \textit{Sequence}$$

$$\frac{\{P\ \wedge\ C\}\ S_1\ \{Q\}, \quad \{P\ \wedge\ \neg C\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if C then S}_1\ \texttt{else S}_2\ \texttt{fi}\ \{\texttt{Q}\}} \qquad \textit{Conditional}$$

There are some hidden assumptions in these rules – can you find them? When rules are given for all possible language constructs, we get both a semantics for the programming language and a way to reason about programs in it. These rules define the *axiomatic semantics* for the language, with this particular style due to Tony Hoare and therefore called *Hoare-style proof rules*. An important note about the rule of consequence is that we will use any valid implications when applying this rule, rather than requiring formal proofs of the implications themselves; this is important for pragmatic reasons as it will keep of from digressing into proofs of "known facts" in the middle of our program proofs.

In imperative languages, an interesting construct is the assignment statement. In fact, finding a valid proof rule for assignment was one of the early problems in the field of axiomatic semantics during the 1970's. A plausible but invalid axiom is:

$$\{P\}\ \texttt{x := E}\ \{[\texttt{x}/\texttt{E}]P\}$$

Assume that P is $y \neq z$, so the assertion after executing `x := y` would be $x \neq z$. However, if x is actually an alias for z, meaning both refer to the same location, then `x := y` would assign y to z as well as x and after assignment $y \neq z$ would not be true. The general problem is that substitution does not preserve validity.

The valid proof rule for assignment will instead result in assertions being pushed backwards through the program.

$$\{\texttt{E}/\texttt{x}]P\}\ \texttt{x := E}\ \{P\}$$

5 Verifying Straight-Line Code

We'll start with a trivial example, just to see the general setup for verification.

```
// Postcondition: returns x^3
int cube(int x) {
  int res;
  res = x*x;
  res = res*x;
  return res;
}
```

The first and last lines are a written to make the job of reasoning about the middle of the program simpler – it is often easier to make up names when we are doing formal reasoning, although in this very simple procedure it looks a bit strange. The two statements in the middle are the ones of interest, since they are doing the actual work. The post condition before the return statement is simply the position condition for the function with the "returns" replaced by "res =". The precondition after the declaration is simply "true" since there is no precondition on the procedure invocation and res has not been set to any particular value.

Here is the program again with annotations for the pre and post conditions around the inner two statements.

```
/* assert(true) */
res = x*x;
/* assert ?? */
res = res*x;
/* assert(res == x^3) */
```

The idea is to walk backwards through the program, adjusting the assertion for each assignment statement. We keep doing this until we run into a precondition, which will hopefully be the same or stronger than what we need. The general rule for an assignment statements of the form `v = expr` is, given an assertion P after the assignment, it will be true if P' is true before the assignment, where P' is P with any occurrence of `v` replaced by `expr`. Here is the general rule for assignment:

```
/* assert(P'), where P' is P with expr for v */
v = expr;
/* assert(P), where P contains v */
```

This rule assumes that `expr` does not have any side effects. In the `cube` example, we can use this on the postcondition `res == x`^{3} to obtain `res*x == x`^{3}.

The missing proof step is between the two assignment statements, and is:

```
/* res*x == x^3 */
res = res*x;
/* res == x^3 */
```

This is the *proof obligation*: to prove the final condition `res == x`^{3} we must prove this intermediate result, `res*x == x`^{3} before the assignment. (Note that two assertions refer to two different values of `res`. People often add subscripts or primes to their program variables to keep them straight.) To prove `res*x == x`^{3}, we apply the assignment rule again

If we then apply the assignment rule one more time and get:

```
/* x*x*x == x^3 */
res = x*x;
/* res*x == x^3 */
```

Since we have now run into a precondition, `true`, we check that it implies the current proof obligation, namely, `x*x*x == x`^{3}. Fortunately, this is the definition of a power, so it is trivially true.

In general, you will probably not want to do proofs at this level of detail, but the basic idea of working backwards from the postcondition is generally useful both in writing and reasoning about programs. Given the simple nature of each of our proof steps, you might imagine writing a program to do the proof automatically – this is the still active and well-studied research area of automatic program verification. There are two problems, though, the first being that it is provably impossible to write such a program that will work on all programs. Nevertheless, people have built systems to do program verification based on the assumption that most programs actually require fairly simple but tedious reasoning, and while we can't verify all programs, we might be able to verify the ones that we care to write. These systems have suffered from another problem, which is that the proof obligations grow unmanageable large for real programs, and in addition, writing formal specifications, which is the first step in verification, is not something that most programmers are willing to do.

6 Proving Properties of Loops

We have been a bit loose in the above example and proof rule for assignment by not distinguishing the l-values (locations associated with a variable occurring on the left-hand-side of an assignment statement) from the r-values (contents of the locations). We could be even more formal about the above proof, but instead, will consider a programming construct that is often tricky to get right in real programs, loops, and see how verification techniques can help in programming and documentation. This should help you write programs like binary search, quicksort, and similar things more quickly (with less debugging time).

There are two techniques for proving things about loops, one for partial correctness and one for termination.

- *loop invariant*: a predicate that is true before and after each iteration of the loop body.
- *variant function*: an integer valued function that gets closer to zero on every iteration of the loop, and once the function is less than or equal to zero, the loop will terminate. Note: this is a special case – in general, any values with a Noetherian ordering can be used.

6.1 Partial Correctness

We will start by concentrating on partial correctness using loop invariants. A loop invariant must be invariant over the body of the loop: if it is true before the loop body executes, it must be true aftwards. A loop invariant does not have to be true after each statement in the loop, and rarely will be, since the statements in the loop will affect variables in the invariant. In addition, the loop invariant must be true the first time we execute the loop and must be strong enough to capture the correctness of the algorithm. What do we want it to do for an arbitrary `while` loop with condition B and loop body S?

```
/* assert(P) */
while (B) { S; }
/* assert(Q) */
```

Informally, the invariant I must satisfy the following properties:

1. I must be true before we execute the while loop. I.e., $P \Rightarrow I$.
2. If I is true before evaluating B, it is true after evaluating B. (This is easy to gaurantee if B does not contain any assignment statements, which is generally good programming style, anyway.)
3. If I and B are true at the beginning of the loop body, then I must be true at the end of the loop
4. I plus the negation of the condition (i.e., $I \wedge \neg B$) must imply the postcondition Q.

The proof rule for while loops makes use of such an invariant property. Although it is not obvious, this rule is both sound and complete for proving properties of loop constructs.

$$\frac{\{I \wedge B\}\ \mathtt{S}\ \ \{I\}}{\{I\}\ \mathtt{while\ B\ do\ S\ od}\ \{\mathtt{I} \wedge \neg\mathtt{B}\}}\ \ loops$$

We are ignoring the possibility of return statements inside the loop body S. Such returns complicate both formal and informal reasoning about the function, although we could extend this framework to handle them. We will also consider only `while` loops and not `for` loops, but if we wanted to prove something about a `for` loop we could first transform:

```
for (S1; B; S2) {      into        S1; while (B) {
   S;                                             S;  S2;
}                                                 }
```

and then use the above technique for `while` loops.

6.2 Example: Linear Search

Given below is a slightly more complicated example than the `cube` function, a linear search algorithm.

```
// Precondition:  n > 0 and the size of a == n.
// Postcondition: Returns res.  If key is in a, a[res] = key,
 //                 otherwise (if key is not in a), res = -1.
 int search (int a[], int n, int key) {
   int res = -1;
   int i = 0;
   while (i < n) {
     if (a[i] == key) {
          res = i;
        }
     else i++;
   }
   return res;
 }
```

The loop invariant for this function is: `(res = -1) => (forall j, 0 <= j < i, a[j] != key)` and (res != -1) =¿ a[res] = key. Let's look at the loop invariant in more detail. We can now annotate the program with the proof obligation at each step. If you start at the end and following the program backwards, you will see how each obligation is derived.

```
{
res = -1;
i = 0;
/* assert (n>0 & size of a == n) */
  while (i < n) {
    /* assert((i < n)
              & (res = -1) => (forall j, 0 <= j < i, a[j] != key));
              & (res != -1) => a[res] = key */
    if (a[i] == key) res = i;
    /* assert((i < n)
              & (res = -1) => (forall j, 0 <= j < i+1, a[j] != key));
              & (res != -1) => a[res] = key */
    else i = i+1;
    /* assert((i < n)
              & (res = -1) => (forall j, 0 <= j < i, a[j] != key));
              & (res != -1) => a[res] = key */
  }
    /* assert((i >= n)
              & (res = -1) => (forall j, 0 <= j < n, a[j] != key));
              & (res != -1) => a[res] = key */
    /* implies */
    /* assert If key is in a, then a[res] == key,
```

```
        otherwise, res = -1. */
return res;
}
```

This proves the loop invariant, and that the postcondition holds if the loop completes. The next step is to show that the loop terminates. We choose a variant function that decreases each time through the loop and will be zero when the loop terminates. The variant function is: `n - i`.

7 Writing Correct Loops

In general, the best way to write provably correct programs is to develop the program and the proof simultaneously – minor changes in the program can make the proof much more difficult. Here are some rules taken from Jan L.A. van de Snepscheut, "Computations, Computers, and Programs," 1989, on how to write provably correct loops. We think of these rules as transformations on a specification to produce a program. Of course, they will not be complete and in some cases more than one rule may apply – otherwise, we would just write the specification and let the compiler apply these rules to write the program.

7.1 Omit a conjunct

The first rule is: Given a conjuction (Q1 & Q2) in the postcondition, make one of the conjucts a loop invariant, and the negation of other as a condition for the loop.

As an example of using this rule, consider the problem of writing a function to compute the integer square root of a non-negative integer, i.e., find the largest integer whose square is less than or equal to the input. Here's a specification of this procedure, called `intsqrt`.

```
int intsqrt (int n)
  /* Precondition: n >= 0. */
  /* Postcondition: returns res, such that (res^2 <= n) & (n < (res+1)^2) */
```

First, we will add a variable to hold the return value:

```
int intsqrt (int n) {
  int res;
  /* assert(n >= 0); */
    S;
  /* assert((res^2 <= n) & (n < (res+1)^2)); */
  return res;
}
```

Applying the above rule, we can either have the invariant $(\texttt{res}^2 <= \texttt{n})$ or $(\texttt{n} < (\texttt{res+1})^2)$. We will try the first condition at the loop invariant and the second condition (or rather, its negation) as the stopping criterion for the loop.

```
// assert(n >= 0).
S0;                                // fill in later
while (n >= (res+1)^2) {
  // assert(res^2 <= n);         // loop invariant
  S1;                              // fill in later
}
// assert((res^2 <= n) & (n < (res+1)^2))
```

The last line is implied by the loop invariant and rule for loops, assuming we can prove the loop invariant for the base case and middle.

We also need a variant function. Since there's only one variable to play around with, `res`, we'll make `res` start at 0, and increase. (At 0, the invariant obviously holds.) How much should it increase by? 1 will work well, since the condition has a term `(res+1)`^{2} and the invariant has an `res`^{2}. Thus, the variant function is `n - (res+1)`^{2}, a value that will start at n, decrease each time through the loop and will cause the loop to terminate when it is less than be less than 0.

```
// assert(n >= 0);
res = 0;
while (n >= (res+1)^2) {
  // assert(res^2 <= n);
  res = res+1;
}
// assert((res^2 <= n) & (n < (res+1)^2));
```

We could have chosen the other conjunct, `(n < (res+1)`^{2}`)` as the invariant with the negation of `(res`^{2} `<= n)` as the loop condition. This would have led to a solution that starts `res` at `n` and decrement it in the loop. The rule of omiting a conjunct could also be used to produce the linear search program similar to the one at the beginning of this handout.

7.2 Replace a constant by a variable

This is another contrived example to demonstrate the use of this rule. Suppose we want compute the square of a non-negative integer without using multiplication.

```
/* Precondition: n >= 0 */
/* Postcondition: returns n^2 */
int square(int n)
{
  int res;
  S;     // fill in later
  return res;
}
```

`S` has the following constraints:

```
/* assert(n >= 0) */
S;       // fill in later
/* assert(res = n^2) */
```

We want to leave the expression `n`^{2} is constant, since it is part of the final postcondition, but we introduce a new variable, `m`, using `res == m`^{2} as part of a loop invariant. This new variable will start at 0 and increase its value to `n`, maintaining the invariant `res == m`^{2}. `S` is now:

```
res = 0;
m = 0;
while (m != n) {
  // assert(res = m^2 & 0 <= m <= n);      // loop invariant
  S1;                                         // fill in later
  m = m + 1;
  }
```

`S1` is a statement that stores `(m+1)`^{2} into `res`. Consider the first term in the loop invariant, `(res = m`^{2}`)`. We want this to be true at the bottom of the loop, so using the rule for assignemnt, we can push this backwards through the increment of `m`, obtaining the following postcondition for `S1`:

```
assert(res == (m+1)^2);
```

This can be rewritten using basic algebra to:

```
assert(res == m^2 + 2*m + 1);
```

The precondition for `S1` (from the invariant) is:

```
res == m^2;
```

If we use an assignment statement of the form `res = res + ?`, then we can replace the $\mathtt{m}^2$ term in the postcondition by the value of `res` in the precondition.

```
res = res + 2*m + 1
```

Since we cannot use multiplication, we convert this expression to:

```
res = res + m + m + 1;
```

The original invariant included the term `0 <= m <= n`, which we have since ignored. Although it was not needed in this particular example, it was true and it often necessary for rewriting expressions in the loop body (e.g., if we had an array, we would want to know the index was in range).

The variant function for this example is obviously `(n - m)`, which starts positive and will terminate the loop when `n` and `m` are equal.

7.3 Enlarge the Range of a Variable

The next example we consider is a two-dimensional search problem called *saddleback search*. Assume we are given a matrix a, of size `n` by `m` and we are looking for a value `x` that occurs at some location `a(i,j)`, `0 <= i < n` and `0 <= j < m`. Furthermore, in `a` each row is sorted in ascending order and each column is sorted in descending order, i.e., `a(i,j) <= a(i,j+1)` and `a(i,j) >= a(i+1,j)` for `i` and `j` in the appropriate range.

We start by making up variables that store some index at which `x` occurs, i.e., `a(ii, jj) == x`, although this may not be the occurence that the algorithm finds if `x` occurs multiple times in `a`. We also add two program variables, `i` and `j` and use the loop invariant:

```
assert ((0 <= i <= ii) & (0 <= j <= jj));
```

and the variant function:

```
jj - j + ii - i
```

By initializeing both `i` and `j` to 0, the invariant will hold. We then assume that either `i` or `j` will be incremented each time through the loop, thereby making the variant function decrease. Under what circumstances will the invariant hold if we increment `i`?

```
        0 <= i+1 <= ii & 0 <= j <= jj
implies
        0 <= i < ii & 0 <= j <= jj
implies
        a(i,j) > a(ii,j) & 0 <= i & 0 <= j <= jj
implies
        a(i,j) > a(ii,jj) & 0 <= i & 0 <= j <= jj
implies
        a(i,j) > x  &  0 <= i  & 0 <= j <= jj
```

Therefore, if we test the condition `a(i,j) > x`, we can increment `i`. Using a symmetric argument, we can increment `j` and maintain the invariant as long as `a(i,j) < x`. Therefore, the body of the search function becomes:

```
void saddle (matrix a, int x, int &i, int &j)
i = 0;
j = 0;
while (a(i,j) != x) {
   // assert(0 <= i <= ii < n & 0 <= j <= jj < m, where a(ii,jj) == x);
   if (a(i,j) > x) {
       i = i + 1;
   }
   else { // (a(i,j) < x)
       i = i + 1;
   }
// assert(a(i,j) == x);
}
```

This program assumes that `x` is in the matrix. If we want it to work even if `x` does not exist, we would need to define what values i and j should have, and use a more complicated loop invariant and loop condition, but the idea would be the same.

7.4 Reduce the Problem Size

This rule involves the familiar divide-and-conquer idea. Assume we are writing the code for a `gcd` non-recursive function using Euclid's algorithm. Here is the main part of the function:

```
  while (a != b) {
    if (a > b) {
       a = a - b;
    }
    else { // (b > a)
       b = b - a;
    }
    return a;
```

How do we convince ourselves that this algorithm is correct? The key idea is that the `gcd` of `a` and `b` remain the same through the loop, even though `a` and `b` are themselves changing. As in the previous example, we make up two auxiliary constants, `aa` and `bb`, for the proof that do not appear in the program. `A` is the initial value of `a` and `B` is the initial value of `b`. The loop invariant is:

```
gcd(aa,bb) == gcd(a,b) and a > 0 and b > 0.
```

and the variant function is:

```
a + b
```

Verification of the loop is straightforward and is done by checking each of the two branches. Consider the branch in which `a` is modified. Pushing the invariant backwards thought the assignment yields:

```
    gcd(aa,bb) == gcd(a-b,b) and a-b > 0 and b > 0.
```

but this is true because:

```
    gcd(a-b,b) == gcd(a,b) if a > b
and
    gcd(aa,bb) == gcd(a,b) by the loop invariant
```

The negation of the loop condition implies the `a == b` after the loop terminates, and since `gcd(aa,bb) == gcd(a,b)` by the invariant, this also means that `gcd(aa,bb) == gcd(a,a)`. We then use a basic fact about `gcd`, namely `gcd(a,a) == a`, and thus get the final postcondition for the function, `gcd(aa,bb) == a`.

8 Verifying Classes

Verifying classes is essentially a set of function verifications, one for each member function. We will not cover this in depth, but will make one point about the way reasoning is done on abstract data types: some functions may depend on a representation invariant for their correctness, for example, that the `size` variable inside the linked list class is equal to the number of nodes in the list. This is incorporated in the the formal setup by adding the addition rules for verifying member functions.

- For each constructor, prove that the representation invariant holds after construction, and for each each non-constructor, prove the representation holds after the function completes.
- For each non-constructor member function, assume the representation invariant holds on each object in the class when the function is called. (This includes the argument passed into the copy constructor.)

This depends on encapsulation of the data type. If there is a friend class, as with iterator objects, then you need to verify that each member function on the friend class also maintains the invariant. The same is true for friend functions, for example, overloaded operators. If the representation exposed to the user, i.e., the client of the abstraction is allowed to directly modify some of the data members, then technically we need to prove that every modification the client makes also leaves the object in a state that satisfies the representation invariant. For example, if the `size` variable is public rather than privat, then to verify an program we need to consider every place that `size` is modified. This means we cannot prove anything about the data structure without the rest of the program, so this shows how the lack of abstraction affects formal reasoning about the program: the loss of modularity in the program affects modularity in reasoning about it either formally or informally.

9 Formal Specifications

10 Exercises

For each of the first four questions, start by writing a more formal pre and post specification of this function and then apply some of the above rules to get the solution. In each case you should identify a loop invariant and variant function.

1. Write a program that finds the largest integer that is a power of 2 and does not exceed a given input, `n`.
2. Given an array `a`, of `n` integers, write a program that returns true if there is some index `i`, such that `0 <= i < n` and `a[i] == i` and returns false if no such `i` exists.
3. You are given three vectors, `f`, `g`, and `h`, that all sorted in ascending order. At least one value is common to all three. Write a function to find the least common element. (You may use the `size()` member function on vectors.)

4. Given a fixed non-negative number `n`, write a function that computes the sum of decimal digits in `n`.

5. (This question is from last year's final exam.) Given the following implementation of radix sort, state a loop invariant that could be used to prove it works. Show that if your loop invariant holds for $i = k$, it also hold for $i = k - 1$.

```
RADIX_SORT(A)
  maxKeyLength = length of longest key in A;
  for (i = maxKeyLength-1; i >= 0; i--)
    stably sort A using digit i of each key
```