TEXTS AND MONOGRAPHS IN COMPUTER SCIENCE

THE SCIENCE OF PROGRAMMING

David Gries

Springer-Verlag
New York Heidelberg Berlin

Part 0
Why Use Logic?
Why Prove Programs Correct?

A story

We have just finished writing a large program (3000 lines). Among other things, the program computes as intermediate results the quotient q and remainder r arising from dividing a non-negative integer x by a positive integer y. For example, with $x = 7$ and $y = 2$, the program calculates $q = 3$ (since $7 \div 2 = 3$) and $r = 1$ (since the remainder when 7 is divided by 2 is 1).

Our program appears below, with dots "..." representing the parts of the program that precede and follow the remainder-quotient calculation. The calculation is performed as given because the program will sometimes be executed on a micro-computer that has no integer division, and portability must be maintained at all costs! The remainder-quotient calculation actually seems quite simple; since $\div$ cannot be used, we have elected to repeatedly subtract divisor y from a copy of x, keeping track of how many subtractions are made, until another subtraction would yield a negative integer.

```
...
r:= x; q:= 0;
while r > y do
  begin r:= r-y; q:= q+1 end;
...
```

We're ready to debug the program. With respect to the remainder-quotient calculation, we're smart enough to realize that the divisor should initially be greater than 0 and that upon its termination the variables should satisfy the formula

$$x = y*q + r,$$

so we add some output statements to check the calculations:

```
. . .
write('dividend x =', x, 'divisor y =', y);
r := x; q := 0;
while r > y do
   begin r := r-y; q := q+1 end;
write('y*q + r =', y*q + r);
. . .
```

Unfortunately, we get voluminous output because the program segment occurs in a loop, so our first test run is wasted. We try to be more selective about what we print. Actually, we need to know values only when an error is detected. Having heard of a new feature just inserted into the compiler, we decide to try it. If a Boolean expression appears within braces { and } at a point in the program, then, whenever "flow of control" reaches that point during execution, it is checked: if false, a message and a dump of the program variables are printed; if true, execution continues normally. These Boolean expressions are called *assertions*, since in effect we are asserting that they should be true when flow of control reaches them. The systems people encourage leaving assertions in the program, because they help document it.

Protests about inefficiency during production runs are swept aside by the statement that there is a switch in the compiler to turn off assertion checking. Also, after some thought, we decide it may be better to always check assertions —detection of an error during production would be well worth the extra cost.

So we add assertions to the program:

```
    . . .
    {y > 0}
    r := x; q := 0;
(1) while r > y do
       begin r := r-y; q := q+1 end;
    {x = y*q + r}
    . . .
```

Testing now results in far less output, and we make progress. Assertion checking detects an error during a test run because y is 0 just before a remainder-quotient calculation, and it takes only four hours to find the error in the calculation of y and fix it.

But then we spend a day tracking down an error for which we received no nice false-assertion message. We finally determine that the remainder-quotient calculation resulted in

$x = 6, y = 3, q = 1, r = 3.$

Sure enough, both assertions in (1) are true with these values; the problem is that the remainder should be less than the divisor, and it isn't. We determine that the loop condition should be $r \geqslant y$ instead of $r > y$. If only the result assertion were strong enough —if only we had used the assertion $x = y*q + r$ **and** $r < y$— we would have saved a day of work! Why didn't we think of it?

We fix the error and insert the stronger assertion:

```
. . .
{$y > 0$}
$r := x$;  $q := 0$;
while $r \geqslant y$ do
   begin $r := r - y$;  $q := q + 1$ end;
{$x = y*q + r$ and $r < y$}
. . .
```

Things go fine for a while, but one day we get incomprehensible output. It turns out that the quotient-remainder algorithm resulted in a negative remainder $r = -2$. But the remainder shouldn't be negative! And we find out that r was negative because initially x was -2. Ahhh, another error in calculating the input to the quotient-remainder algorithm —x isn't *supposed* to be negative! But we could have caught the error earlier and saved two days searching, in fact we *should* have caught it earlier; all we had to do was make the initial and final assertions for the program segment strong enough. Once more we fix an error and strengthen an assertion:

```
. . .
{$0 \leqslant x$ and $0 < y$}
$r := x$;  $q := 0$;
while $r \geqslant y$ do
   begin $r := r - y$;  $q := q + 1$ end;
{$x = y*q + r$ and $0 \leqslant r < y$}
. . .
```

It sure would be nice to be able to invent the right assertions to use in a less *ad hoc* fashion. Why can't we think of them? Does it have to be a trial-and-error process? Part of our problem here was carelessness in specifying what the program segment was to do —we should have written the initial assertion ($0 \leqslant x$ **and** $0 < y$) and the final assertion ($x = y*q + r$ **and** $0 \leqslant r < y$) *before* writing the program segment, for they form the definition of quotient and remainder.

But what about the error we made in the condition of the while loop? Could we have prevented that from the beginning? Is there is a way to prove, just from the program and assertions, that the assertions are true when flow of control reaches them? Let's see what we can do.

Just before the loop it seems that part of our result,

(2) $x = y*q + r$

holds, since $x = r$ and $q = 0$. And from the assignments in the loop body we conclude that if (2) is true before execution of the loop body then it is true after its execution, so it will be true just before and after *every* iteration of the loop. Let's insert it as an assertion in the obvious places, and let's also make all assertions as *strong* as possible:

```
. . .
{0≤x and 0<y}
r:= x; q:= 0;
{0≤r and 0<y and x =y*q +r}
while r ≥y do
  begin {0≤r and 0<y ≤r and x =y*q +r}
          r:= r−y; q:= q+1
          {0≤r and 0<y and x =y*q +r}
end;
{0≤r <y and x = y*q +r}
. . .
```

Now, how can we easily determine a correct loop condition, or, given the condition, how can we prove it is correct? When the loop terminates the condition is false. Upon termination we want $r < y$, so that the complement, $r \geqslant y$ must be the correct loop condition. How easy that was!

It seems that if we knew how to make all assertions as strong as possible and if we learned how to reason carefully about assertions and programs, then we wouldn't make so many mistakes, we would *know* our program was correct, and we wouldn't need to debug programs at all! Hence, the days spent running test cases, looking through output and searching for errors could be spent in other ways.

Discussion

The story suggests that assertions, or simply Boolean expressions, are really needed in programming. But it is not enough to know how to write Boolean expressions; one needs to know how to *reason* with them: to simplify them, to prove that one follows from another, to prove that one is not true in some state, and so forth. And, later on, we will see that it is necessary to use a kind of assertion that is not part of the usual Boolean expression language of Pascal, PL/I or FORTRAN, the "quantified" assertion.

Knowing how to reason about assertions is one thing; knowing how to reason about *programs* is another. In the past 10 years, computer science has come a long way in the study of proving programs correct. We are reaching the point where the subject can be taught to undergraduates, or to anyone with some training in programming and the will to become more proficient. More importantly, the study of program correctness proofs has led to the discovery and elucidation of methods for *developing* programs. Basically, one attempts to develop a program and its proof hand-in-hand, with the proof ideas leading the way! If the methods are practiced with care, they can lead to programs that are free of errors, that take much less time to develop and debug, and that are much more easily understood (by those who have studied the subject).

Above, I mentioned that programs could be free of errors and, in a way, I implied that debugging would be unnecessary. This point needs some clarification. Even though we can become more proficient in programming, we will still make errors, even if only of a syntactic nature (typos). We are only human. Hence, some testing will always be necessary. But it should not be called debugging, for the word debugging implies the existence of bugs, which are terribly difficult to eliminate. No matter how many flies we swat, there will always be more. A disciplined method of programming should give more confidence than that! We should run test cases not to look for bugs, but to increase our confidence in a program we are quite sure is correct; finding an error should be the exception rather than the rule.

With this motivation, let us turn to our first subject, the study of logic.