Programming Languages
J.J. Horning Editor

On the Proof of Correctness of a Calendar Program

Leslie Lamport
SRI International

A formal specification is given for a simple calendar program, and the derivation and proof of correctness of the program are sketched. The specification is easy to understand, and its correctness is manifest to humans.

Key Words and Phrases: program specification, program verification, inductive assertions
CR Category: 5.24

Introduction

In [1], Geller introduced a method for proving the correctness of a program based upon the concept of test data, and illustrated it with a simple calendar program. The program accepts as input a pair of dates in the same year, and computes the number of days between those dates. Geller presented two proofs for this program: one using his method, the other using the customary inductive assertion method. He made the following valid criticism of the assertional proof:

> We also have less confidence in the proof that our output assertion actually ensures that the program computes values that correspond with the way real calendars behave. The formal specification . . . could easily be off by a constant of 1 or 2.

This criticism echoes the following objection, which is often raised against the whole idea of proving the correctness of programs:

> Real programs are difficult to specify, and formal specifications for them are very complicated. A complicated specification is just as likely to be incorrect as the program itself, so why bother proving that a program satisfies an unreliable specification?

Author's address: L. Lamport, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.

The example of the calendar program seems to corroborate this view. The calendar program has a precise *informal* specification, based upon our knowledge of the calendar. However, one cannot formally prove that a program meets an informal specification. It is easy to test the program to "see if it works", but it would appear that a formal specification must be quite tedious, and just as subject to error as the program itself. A specification of such a simple program is worthless if its correctness is not easily verified by human inspection.

Geller's method provides no answer to this objection, since his proof is not based upon any formal statement of the problem. In fact, since there is no precise specification of what constitutes valid input, the program could be considered incorrect because it may give wrong answers for dates earlier than 15 October 1582—the "first day" of the Gregorian calendar. One may object that this is a quibble, but of what value is a methodology that cannot produce an unequivocally correct solution to such a simple problem?

We believe that if one really understands what a program should do, then he can specify it precisely in an understandable manner. To demonstrate this, we will give a specification for the calendar program that is simple, natural, and quite easy for a human to understand and verify. We will do this by carefully formalizing our ordinary concepts of dates and calendars, and then turning our informal idea of what the calendar program is supposed to do into a formal specification. We will show that the inductive assertion method is well-suited to proving that Geller's program satisfies this formal specification.

The Solution

In general, constructing a correct program requires the following three steps:

1. *Write a correct specification of the problem.* A correct specification is one that expresses what the user wants the program to do. Only the (human) user knows what he wants, so only he can decide if the specification is correct. In order to decide if the specification is correct, the user must be able to understand it. Programs as simple as the calendar program should have simple specifications. As Geller observed, this is not the case for specifications in terms of complex input and output assertions. Incorrect specifications are a significant source of program errors [2].

2. *Devise a correct method for solving the problem.* Unless the problem is completely trivial, one cannot write a program directly from the specification. One must first derive some method for computing the result, and prove that it computes a correct result—a result which satisfies the specification. Formally, this means proving theorems about the data, which can be used to construct a correct program. Such theorems usually already exist. A programmer writing a GCD subroutine

does not have to invent and prove the correctness of Euclid's algorithm—Euclid already did that. Advocates of programming methodologies have tended to talk as if their methodologies automatically generate good programs. A programming methodology is no substitute for intelligent reasoning about algorithms and their complexity, and cannot by itself lead one to a good method of solution. (For example, see [4].) "Structured programming" would not have helped Euclid discover his algorithm.

3. *Write a program that correctly implements the method of solution.* Finally, one must construct a program to implement the method chosen in Step 2, and must prove that the resulting program meets its specification. (The theorems from Step 2 will be used in that proof.) This final step has been the focus of most of the literature on programming methodology and correctness proofs, and further discussion by us would be superfluous.

We now indicate how the calendar problem can be solved in these three steps.

Step 1: Specification

A *date* is a triple of integers such as (7, February, 1978), where we let "January", . . . , "December" be alternate names for the integers 1, . . . , 12. A calendar is a method of assigning dates to days. More precisely, let us define an *era* to be an infinite sequence of days. A calendar for that era is an assignment of a date to each day in the era. For simplicity, we consider the specific era beginning on 15 October 1582, and we let the sequence of days be represented by the sequence of positive integers. A calendar is then a mapping from integers to dates (triples of integers). We define the Gregorian calendar to be the mapping *gregorian* from positive integers to dates such that

$$gregorian[n] \equiv (day[n], month[n], year[n]),$$

where the integer-valued functions *day*, *month*, and *year* are defined inductively as follows:

```
(day[1], month[1], year[1]) ≡ (15, October, 1582)
(day[n + 1], month[n + 1], year[n + 1]) ≡
      if (day[n], month[n]) = (31, December)
        then (1, January, 1 + year[n])
        else if day[n] = daysin[month[n], year[n]]
                then (1, 1 + month[n], year[n])
                else (1 + day[n], month[n], year[n]).
```

The function *daysin* used in this definition gives the number of days in the specified month, and is defined (for all months from January to December) by:

```
daysin[month, year] ≡
     if month = February
       then if (year ≡ 0 mod 4) and
               (year ≢ 0 mod 100 or year ≡ 0 mod 400)
                  then 29
                  else 28
       else if month ∈ {September, April, June, November}
                  then 30
                  else 31.
```

We believe that this definition of a calendar is simple and natural, and that a human can understand it and verify its correctness. We will use this definition to specify a calendar program.

Before we can write a specification, we must decide just what we want to specify. We will consider the problem of specifying a function, called DAYS, to be written in some Algol-like programming language. (We adopt the convention of using uppercase letters as the names of objects in the programming language.) This function takes three arguments: a year and two (day, month) pairs, and returns the number of days from the first (day, month) to the second (day, month) of that year. More precisely, the function DAYS is specified as follows:

If $(day1, month1, yr) = gregorian[n1]$
and $(day2, month2, yr) = gregorian[n2]$
and $n1 \leq n2$
then DAYS$(yr, (day1, month1), (day2, month2))$
$= n2 - n1.$

Observe that the first two hypotheses state that the arguments represent valid dates—i.e. that there exist days $n1$ and $n2$ having these dates, thereby excluding such inputs as (32, January). The third hypothesis states that the second date is not earlier than the first. The conclusion is the precise statement that DAYS computes what we want it to: the number of days between the two dates. Note how clear and precise this specification is.

Step 2: The Method

To compute the values of DAYS directly from its specification, we would need to compute the inverse of the function *gregorian*. Since this is impractical, we might reason as follows to obtain a practical method for computing it. We first ask: How many days are there between a given date and the beginning of the year? The answer is provided by the following result, where a sum of the form

$$\sum_{i=1}^{0} \cdots$$

is defined to equal zero.

Theorem: If $(dy, mon, yr) = gregorian[n]$
and $(1, \text{January}, yr) = gregorian[m]$,
then $n - m = dy - 1 + \sum_{i=1}^{mon-1} daysin[i, yr]$.

The proof of this theorem, using our definition of *gregorian*, is a nice exercise in mathematical induction, and is left to the reader. If we write $n2 - n1 = (n2 - m) - (n1 - m)$ and apply the theorem twice, using the easily proved result that if $n1 \leq n2$ and $year[n1] = year[n2]$ then $month[n1] \leq month[n2]$, and do some algebra, we obtain the following corollary to the theorem.

Corollary: If $(day1, month1, yr) = gregorian[n1]$
and $(day2, month2, yr) = gregorian[n2]$
and $n1 \leq n2$
then $n2 - n1 = day2 - day1 + \sum_{i=month1}^{month2-1} daysin[i, yr]$.

Comparing this corollary with the specification, we see that it provides the simple method for computing DAYS that we were looking for.

Step 3: Writing the Program

We are now faced with a simple exercise in constructing and proving the correctness of a program. We define the function DAYS in terms of two program statements: COMPUTE.DAYSIN and COMPUTE.SUM. The statement COMPUTE.DAYSIN sets the ith element of the array DAYSIN equal to *daysin*[i, YEAR], where YEAR is a program variable. Its formal specification in terms of input and output assertions is as follows:

Input Assertion: YEAR = yr

Output Assertion:
For all $i \in$ {January, . . . , December}:
DAYSIN(i) = *daysin*[i, yr].

The program statement COMPUTE.SUM computes the sum over i in the above corollary when *month*1 < *month*2. Its formal specification is as follows:

Input Assertions:
MONTH1 = $m1$
MONTH2 = $m2$
$m1 < m2$
For all $i \in$ {January, . . . , December}:
DAYSIN(i) = $dn(i)$

Output Assertion:
$$\text{SUM} = \sum_{i=m1}^{m2-1} dn(i).$$

It is a simple exercise in the inductive assertion method to write and prove the correctness of COMPUTE.DAYSIN and COMPUTE.SUM. The proof of correctness of COMPUTE.DAYSIN must use the definition of the function *daysin* that we gave in Step 1, since that function appears in the output assertion. After proving that these statements meet the above input/output specifications, it is a simple matter to use the corollary from Step 2 to prove that the following function DAYS satisfies the specification given in Step 1:

```
DAYS(YEAR, (DAY1, MONTH1), (DAY2, MONTH2)) ≡
    if MONTH1 = MONTH2
        then return(DAY2 − DAY1)
        else COMPUTE.DAYSIN;
             COMPUTE.SUM;
             return(DAY2 − DAY1 + SUM)
    fi.
```

This function is essentially the same as the program given by Geller in [1].

Conclusion

Of the three steps in constructing a correct program, the first—specifying the problem—has received the least attention. The inductive assertion method is useful for proving that a program meets a specification, but it does not help with the problem of writing that specification. To be useful, a specification must be both formal and humanly understandable. Most of the work on formal specification seems to have concentrated on the first requirement and neglected the second. We believe that if one understands what the program is supposed to do, then he can specify it both formally and understandably. One cannot expect a more complicated program to have as simple a specification as the calendar program. However, the specification should be much easier to understand than the program. We believe that such specifications are possible, and we cite as evidence our calendar program as well as a more complicated example given in [3]. However, these are just examples, and much work remains to be done in this area.

Received February 1979; revised June 1979

References

1. Geller, M. Test data as an aid in proving program correctness. *Comm. ACM 21*, 5 (May 1978), 368–375.
2. Gerhart, S., and Yelowitz, L. Observations of fallibility in applications of modern programming methodologies. *IEEE Trans. Software Eng. SE-2*, 3 (Sept. 1976), 195–207.
3. Lamport, L. The specification and proof of correctness of interactive programs. Proc. Mathematical Studies of Information Processing, Kyoto, Japan, Aug. 1978, E.K. Blum, M. Paul, and S. Takasu, Eds., *Lecture Notes in Computer Science 75*, Springer-Verlag, Berlin, pp. 474–537.
4. McMaster, C.L. An analysis of algorithms for the Dutch national flag problem. *Comm. ACM 21*, 10 (Oct. 1978), 842–846.