REFERENCES

[1] P. Brinch Hansen, *Operating System Principles*. Englewood Cliffs, NJ: Prentice-Hall, July 1973.

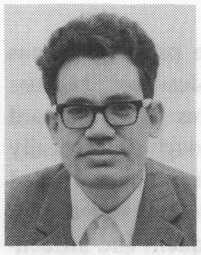
[2] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 549–557, Oct. 1974.

[3] P. Brinch Hansen, "The programming language Concurrent Pascal," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 199–207, June 1975.

[4] —, "The Solo operating system," *Software–Practice and Experience*, vol. 6, pp. 141–205, Apr.–June 1976.

[5] —, "Experience with modular concurrent programming," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 156–159, Mar. 1977.

[6] —, *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ: Prentice-Hall, July 1977.



Per Brinch Hansen was born in Copenhagen, Denmark, on November 13, 1938. He received the M.S. degree in electronic engineering in 1963 and the Doctor Technices degree in 1978, both from the Technical University of Denmark, Copenhagen. He is the first Computer Scientist to receive the Doctor Technices degree which is awarded about once a year to a researcher who has moved Engineering and Applied Science a significant step forward.

From 1963 to 1970, he was a Systems Programmer and Computer Designer for the Danish computer manufacturer, Regnecentralen. Later he became a Research Associate at Carnegie-Mellon University (1970–1972) and Associate Professor of Computer Science at the California Institute of Technology (1972–1976). Since 1976 he has been a Professor in the Department of Computer Science, University of Southern California, Los Angeles, CA. He is the architect of the RC 4000 computer and its multiprogramming system. He is a coinventor of the monitor concept and is the designer of the programming language Concurrent Pascal and three model operating systems written in this language. He is also the author of the books *Operating System Principles* (1973) and *The Architecture of Concurrent Programs* (1977). He has been a consultant to several computer manufacturers. His current research interest is microprocessor networks.

Dr. Brinch Hansen is a member of the Working Group on Programming Methodology sponsored by the International Federation for Information Processing.

End of Special Section

The Logic of Computer Programming

ZOHAR MANNA AND RICHARD WALDINGER

***Abstract*—Techniques derived from mathematical logic promise to provide an alternative to the conventional methodology for constructing, debugging, and optimizing computer programs. Ultimately, these techniques are intended to lead to the automation of many of the facets of the programming process.**

This paper provides a unified tutorial exposition of the logical techniques, illustrating each with examples. The strengths and limitations of each technique as a practical programming aid are assessed and attempts to implement these methods in experimental systems are discussed.

***Index Terms*—Correctness of programs, derivation of programs, program extension, program modification, program synthesis, program transformation, program verification, structured programming, systematic program development, termination of programs.**

> "It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance."
>
> *John McCarthy* [*63*]

Manuscript received October 6, 1977; revised December 21, 1977. This research was supported in part by the Office of Naval Research under Contracts N00014-76-C-0687 and N00014-75-C-0816, by the National Science Foundation under Grant DCR72-03737 A01, by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, and by a grant from the United States-Israel Binational Science Foundation.

Z. Manna is with the Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, on leave at the Stanford Artificial Intelligence Laboratory, Stanford University, Stanford, CA 94305.

R. Waldinger is with the Artificial Intelligence Center, SRI International, Menlo Park, CA 94025.

0098-5589/78/0500-0199$00.75

I. Introduction

IN JUNE 1962, the first American space probe to Venus (Mariner I) went off course and had to be destroyed because of an error in one of the guidance programs in its onboard computer. One statement of the program, though syntactically correct, had a meaning altogether different from that intended by the programmer. Although few bugs have such spectacular effects, errors in computer programs are frequent and influential. There has been substantial effort recently to apply mathematical rigor to the programming process and to enable the accuracy of the machine to compensate for the error-prone human mind.

In the late nineteenth and early twentieth century, mathematics underwent a process of formalization and axiomatization, partially in an effort to escape from paradoxes and logical errors encountered by previous generations of mathematicians. A similar process is underway in the development of a logical theory of programs. This theory has already made our understanding of programs more precise and may soon facilitate our construction of computer programs as well. Logical techniques are being developed to prove programs correct, to detect programming errors, to improve the efficiency of program operation, to extend or modify existing programs, and even to construct new programs satisfying a given specification; many of these techniques have been implemented in experimental programming systems. In the last decade, this field of research has been extremely active; it now has the potential to exert a deep influence on the way computer programs are produced.

The available techniques are already described in the literature, but the relevant papers are scattered through many technical journals and reports, are written in a variety of incompatible notations, and are often unreadable without some background in mathematical logic. In this paper, we attempt to present the principal methods within a unified framework, conveying the intuition behind the methods by examples, and avoiding the formal apparatus of the logicians.

To facilitate a comparison between the various techniques, we use a number of different algorithms for performing the same task: to compute the greatest common divisor of two integers. These algorithms are simple enough to be readily understood, but subtle enough to demonstrate typical difficulties.

The greatest common divisor of two nonnegative integers x and y, abbreviated as $gcd(x\ y)$, is the largest integer that divides both x and y. For instance: $gcd(9\ 12) = 3$, $gcd(12\ 25) = 1$, and $gcd(0\ 14) = 14$. When x and y are both zero there is no greatest common divisor, because every integer divides zero; on the other hand, when x or y is not zero, a greatest common divisor must exist.

A naive algorithm to compute the *gcd* of x and y might behave as follows: make lists of all the divisors of x and of all the divisors of y; then make a third list of all the numbers that appear in both lists (these are the common divisors of x and y); finally, find the largest number in the third list (this is the greatest common divisor of x and y). The cases in which x or y is zero must be handled separately. This algorithm is straightforward but inefficient because it requires an expensive operation, computing all the divisors of a given number, and because it must remember three lists of intermediate numbers to compute a single number.

A more subtle but more efficient algorithm to compute the *gcd* of two numbers can be devised. Until the first number is zero, repeat the following process: if the second number is greater than or equal to the first, replace it by their difference—otherwise interchange the two numbers—and continue. When the first number becomes zero, the answer is the second number. This answer turns out to be the *gcd* of the two original numbers. The new algorithm is more efficient than the naive one, because it only needs to remember two numbers at any one time and to perform the simple *minus* operation.

The above algorithm can be expressed as a stylized program:

Program A (the subtractive algorithm):
input$(x_0\ y_0)$
$(x\ y) \leftarrow (x_0\ y_0)$
more: **if** $x = 0$ **then goto** *enough*
if $y \geq x$ **then** $y \leftarrow y - x$ **else** $(x\ y) \leftarrow (y\ x)$
goto *more*
enough: **output**(y).

The notation $(x\ y) \leftarrow (x_0\ y_0)$ means that the values of x and y are simultaneously set to the input values x_0 and y_0, respectively. Thus, the statement $(x\ y) \leftarrow (y\ x)$ has the effect of interchanging the values of x and y. This program causes the following sequence of values of x and y to be generated in computing the *gcd* of the input values $x_0 = 6$ and $y_0 = 3$:

$x = 6$ and $y = 3$,
$x = 3$ and $y = 6$,
$x = 3$ and $y = 3$,
$x = 3$ and $y = 0$,
$x = 0$ and $y = 3$.

Thus, the output of the program is 3.

Although the earlier naive algorithm was obviously correct, because it closely followed the definition of *gcd*, it is by no means evident that Program A computes the *gcd* function. First of all, it is not clear that when x becomes zero, the value of y will be the *gcd* of the inputs; that this is so depends on properties of the *gcd* function. Furthermore, it is not obvious that x will ever become zero; we might repeatedly execute the **if-then-else** statement forever.

For instance, consider the program A′ obtained from A by replacing the conditional

if $y \geq x$ **then** $y \leftarrow y - x$ **else** $(x\ y) \leftarrow (y\ x)$

by

if $y \geq x$ **then** $y \leftarrow y - x$ **else** $x \leftarrow x - y$.

This program closely resembles Program A, and it actually does compute the *gcd* of its inputs when it happens to produce an output. However, it will run forever and never produce an output for many possible input values; for instance, if $x_0 \neq 0$ and $y_0 = 0$, or if $x_0 \neq 0$ and $y_0 = x_0$. Thus, if $x_0 = y_0 = 3$, the following sequence of successive values of x and y emerges:

$x = 3$ and $y = 3$,
$x = 3$ and $y = 0$,

$x = 3$ and $y = 0$,
$x = 3$ and $y = 0, \ldots$.

These programs are as simple as any we are likely to encounter, and yet their correctness is not immediately clear. It is not surprising, therefore, that bugs occur in large software systems. Although programs may be subjected to extensive testing, subtle bugs frequently survive the testing process. An alternative approach is to prove mathematically that bugs cannot possibly occur in the program. Although more difficult to apply than testing, such mathematical proofs attempt to impart absolute certainty that the program is, indeed, correct.

We will argue that these methods will always fall short of this goal: we can never be absolutely certain that a program is correct and mathematical proofs are unlikely to supersede the testing process entirely. But these methods can be effective in helping us to detect bugs in programs and to impart greater confidence in their correctness.

Techniques derived from mathematical logic have been applied to many aspects of the programming process, including the following.

1) *Correctness:* Proving that a given program produces the intended results.

2) *Termination:* Proving that a given program will eventually stop.

3) *Transformation:* Changing a given program into an equivalent one, often to improve its efficiency (*optimization*).

4) *Development:* Constructing a program to meet a given specification.

These techniques are intended to be applied by the programmer, usually with some degree of computer assistance. Some of the techniques are fairly well understood and are already being incorporated into experimental programming systems. Others are just beginning to be formulated and are unlikely to be of practical value for some time.

> Our exposition is divided between a basic text, given in an ordinary type font, and secondary notes, like this one, interspersed throughout the text in a smaller font. The basic text presents the principal logical techniques as they would be applied by hand; the secondary notes discuss subsidiary topics, report on implementation efforts, and include bibliographical remarks. Only a few references are given for each topic, even though we are likely to lose some good friends in this way. The hasty reader may skip all the secondary notes without loss of continuity.

In the following pages, we will touch on each of these topics; we begin with correctness, the most investigated and best understood of them all.

II. Partial Correctness

To determine whether a program is correct, we must have some way of specifying what it is intended to do; we cannot speak of the correctness of a program in isolation, but only of its correctness with respect to some specifications. After all, even an incorrect program performs *some* computation correctly, but not the same computation that the programmer had in mind.

For instance, for the *gcd* program we can specify that when the program halts, the variable y is intended to equal the greatest integer that divides both inputs x_0 and y_0; in symbolic notation

$$y = max\{u : u|x_0 \text{ and } u|y_0\}.$$

(Here, the expression $\{u : p(u)\}$ stands for the set of all elements u such that $p(u)$ holds, and the expression $u|v$ stands for "u divides v," i.e., $v = k \cdot u$ for some integer k.) We call such a statement an *output assertion*, because it is expected to be true only when the program halts. Output assertions are generally not sufficient to state the purpose of a program; for example, in the case of the *gcd*, we do not expect the program to work for any x_0 and y_0, but only for a restricted class. We express the class of "legal inputs" of a program by an *input assertion*. For the subtractive *gcd* algorithm (Program A), the input assertion is

$$x_0 \geqslant 0 \text{ and } y_0 \geqslant 0 \text{ and } (x_0 \neq 0 \text{ or } y_0 \neq 0).$$

We require that at least one of the inputs be nonzero, because otherwise the *gcd* does not exist. We do not state explicitly that the inputs are integers, but we will assume throughout this paper that variables always assume integer values.

We have expressed the specifications for Program A as a pair of *input-output assertions*. Our task now is to show that if we execute Program A on any input satisfying the input assertion, the program will halt with output satisfying the output assertion. If so, we say that Program A is *totally correct*. It is sometimes convenient, however, to split the task of proving total correctness of a program into two separate subtasks: showing *partial correctness*, that the output assertion is satisfied for any legal input *if* the program halts; and showing *termination*, that the program does indeed halt for all legal inputs. It will be convenient for us to ignore the problem of termination for a while and deal only with partial correctness.

The language in which we write the assertions is different from the programming language itself. Because the statements of this *assertion language* are never executed, it may contain much higher level constructs than the programming language. For instance, we have found the set constructor $\{u : \ldots\}$ useful in describing the purpose of Program A, even though this notation is not a construct of conventional programming languages. Written in such a high-level language, the assertions are far more concise and naturally expressed than the program itself.

In proving partial correctness, it helps to know more about the program than just the input-output assertions. After all, these assertions only tell us what the program is expected to achieve and give us no information on how it is to reach these goals. For instance, in understanding Program A, it is helpful to know that whenever control passes through the label *more*, the greatest common divisor of x and y is intended to be the same as the greatest common divisor of the inputs x_0 and y_0, even though x and y themselves may have changed. Because this relationship is not stated explicitly in either the input-output assertions or the program itself, we include it in the program as an *intermediate assertion*, expressed in the assertion language:

$max\{u : u|x \text{ and } u|y\} = max\{u : u|x_0 \text{ and } u|y_0\}$.

Another intermediate assertion states that whenever we pass through *more*, the program variables, x and y, obey the same restrictions as the input values x_0 and y_0, i.e.,

$x \geq 0$ and $y \geq 0$ and $(x \neq 0$ or $y \neq 0)$.

We rewrite Program A below, annotated with its assertions (within braces, "{ . . . }"). Note that the assertions are not intended to be executed, but are merely comments expressing relationships that we expect to hold whenever control passes through the corresponding points.

Program A (annotated):

input$(x_0\ y_0)$
$\{x_0 \geq 0$ and $y_0 \geq 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)\}$
$(x\ y) \leftarrow (x_0\ y_0)$
more: $\{x \geq 0$ and $y \geq 0$ and $(x \neq 0$ or $y \neq 0)$
and $max\{u : u|x$ and $u|y\} =$
$max\{u|x_0$ and $u|y_0\}\}$
if $x = 0$ **then goto** *enough*
if $y \geq x$ **then** $y \leftarrow y - x$ **else** $(x\ y) \leftarrow (y\ x)$
goto *more*
enough: $\{y = max\{u : u|x_0$ and $u|y_0\}\}$
output(y).

Our goal is to prove that if the program is executed with input satisfying the input assertion, and if the program halts, then the output assertion will hold when the program reaches *enough*.

For this purpose, we will show that the intermediate assertion is true whenever control passes through *more;* in other words, it is *invariant* at *more*. The proof is by *mathematical induction* on the number of times we reach *more*. That is, we will start by showing that if the input assertion is true when we begin execution, the intermediate assertion will be true the first time we reach *more;* we will then show that if the intermediate assertion holds when we pass through *more*, then it will be true again if we travel around the loop and return to *more;* therefore, it must be true every time we pass through *more*.

Finally, we will show that if the intermediate assertion holds at *more*, and if control happens to pass to *enough*, then the output assertion will be true. This will establish the partial correctness of the program with respect to the given input and output assertions.

Let us first assume that the input assertion is true when we begin execution, and show that the intermediate assertion holds the first time we reach *more*. In other words, if

$x_0 \geq 0$ and $y_0 \geq 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)$,

and we execute the assignment

$(x\ y) \leftarrow (x_0\ y_0)$,

then

$x \geq 0$ and $y \geq 0$ and $(x \neq 0$ or $y \neq 0)$
and $max\{u : u|x$ and $u|y\} = max\{u : u|x_0$ and $u|y_0\}$,

for the new values of x and y.

Because the assignment statement sets x to x_0 and y to y_0, we are led to prove the *verification condition*

1) $x_0 \geq 0$ and $y_0 \geq 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)$
$\Rightarrow x_0 \geq 0$ and $y_0 \geq 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)$
and $max\{u : u|x_0$ and $u|y_0\} =$
$max\{u : u|x_0$ and $u|y_0\}$.

(Here the notation $A \Rightarrow B$ means that the antecedent A implies the consequent B.) The consequent was formed from the intermediate assertion by replacing x by x_0 and y by y_0.

Next, assuming that the intermediate assertion is true at *more* and control passes around the loop, we need to show that the assertion will still be true for the new values of x and y when we return to *more*. In other words, if the intermediate assertion

$x \geq 0$ and $y \geq 0$ and $(x \neq 0$ or $y \neq 0)$
and $max\{u : u|x$ and $u|y\} = max\{u : u|x_0$ and $u|y_0\}$

holds, if the exit text $x = 0$ is false (i.e., $x \neq 0$), and if the conditional statement

if $y \geq x$ **then** $y \leftarrow y - x$ **else** $(x\ y) \leftarrow (y\ x)$

is executed, then the intermediate assertion will again be true. To establish this, we distinguish between two cases. If $y \geq x$, the assignment statement $y \leftarrow y - x$ is executed, and we therefore must prove the verification condition

2) $x \geq 0$ and $y \geq 0$ and $(x \neq 0$ or $y \neq 0)$
and $max\{u : u|x$ and $u|y\} = max\{u : u|x_0$ and $u|y_0\}$
and $x \neq 0$
and $y \geq x$
$\Rightarrow x \geq 0$ and $y - x \geq 0$ and $(x \neq 0$ or $y - x \neq 0)$
and $max\{u : u|x$ and $u|y - x\} =$
$max\{u : u|x_0$ and $u|y_0\}$.

The antecedent is composed of the intermediate assertion and the tests for traversing this path around the loop. The consequent was formed from the intermediate assertion by replacing y by $y - x$.

In the alternate case, in which $y < x$, the consequent is formed by interchanging the values of x and y. The corresponding verification condition is

3) $x \geq 0$ and $y \geq 0$ and $(x \neq 0$ or $y \neq 0)$
and $max\{u : u|x$ and $u|y\} = max\{u : u|x_0$ and $u|y_0\}$
and $x \neq 0$
and $y < x$
$\Rightarrow y \geq 0$ and $x \geq 0$ and $(y \neq 0$ or $x \neq 0)$
and $max\{u : u|y$ and $u|x\} =$
$max\{u : u|x_0$ and $u|y_0\}$.

To complete the proof we must also show that if the intermediate assertion holds at *more* and control passes to *enough*, then the output assertion will hold. For this path, we need to establish the verification condition

4) $x \geq 0$ and $y \geq 0$ and $(x \neq 0$ or $y \neq 0)$
and $max\{u : u|x$ and $u|y\} = max\{u : u|x_0$ and $u|y_0\}$
and $x = 0$
$\Rightarrow y = max\{u : u|x_0$ and $u|y_0\}$.

These verification conditions are lengthy formulas, but it is not difficult to prove that they are all true. Conditions 1) and 3) are logical identities, which can be proved without any knowledge of the integers. The proofs of Conditions 2) and 4) depend on three properties of the integers:

a) $u|x$ and $u|y \Leftrightarrow u|x$ and $u|y - x$

(the common divisors of x and $y - x$ are the same as those of x and y),

b) $u|0$

(any integer divides zero), and

c) $max\{u : u|y\} = y$ if $y > 0$

(any positive integer is its own greatest divisor).

To prove Property a), assume $u|x$ and $u|y$. Then we must show that $u|y - x$ as well. We know that $x = k \cdot u$ and $y = l \cdot u$, for some integers k and l. But then $y - x = (l - k) \cdot u$, and hence $u|y - x$, as we wanted to show. Similarly, if $u|x$ and $u|y - x$, then $x = m \cdot u$ and $y - x = n \cdot u$ for some integers m and n. But then $y = x + (y - x) = (m + n) \cdot u$, and hence $u|y$.

To prove Condition 2), let us consider the consequents one by one. That $x \geq 0$, $y - x \geq 0$, and ($x \neq 0$ or $y - x \neq 0$) are true follows directly from the antecedents $x \geq 0$, $y \geq x$, and $x \neq 0$, respectively. That

$$max\{u : u|x \text{ and } u|y - x\} = max\{u : u|x_0 \text{ and } u|y_0\}$$

follows from the antecedent

$$max\{u : u|x \text{ and } u|y\} = max\{u : u|x_0 \text{ and } u|y_0\}$$

and Property a).

To prove Condition 4), first observe that the antecedents imply

$$y > 0,$$

because $x = 0$ and ($x \neq 0$ or $y \neq 0$) imply $y \neq 0$, but $y \neq 0$ and $y \geq 0$ imply $y > 0$. Now, since $x = 0$, applying Property b) to

$$max\{u : u|x \text{ and } u|y\} = max\{u : u|x_0 \text{ and } u|y_0\}$$

yields

$$max\{u : u|y\} = max\{u : u|x_0 \text{ and } u|y_0\}.$$

Because $y > 0$, applying Property c) yields

$$y = max\{u : u|x_0 \text{ and } u|y_0\},$$

the consequent of Condition 4).

This concludes the proof of the partial correctness of Program A. Note again that we have not proved the termination of the program: we have proved merely that if it does terminate then the output assertion is satisfied. A similar proof can be applied to Program A′ (the program formed from Program A by replacing the statement $(x\ y) \leftarrow (y\ x)$ by $x \leftarrow x - y$), even though that program may loop indefinitely for some legal inputs. Program A′ is partially correct, though not totally correct, because it does compute the *gcd* of those inputs for which it happens to halt.

The proof of the partial correctness of Program A involved reasoning about four loop-free program paths: one path from the input assertion to the intermediate assertion, two paths from the intermediate assertion around the loop and back to the intermediate assertion, and one path from the intermediate assertion to the output assertion. Had we not introduced the intermediate assertion, we would have had to reason about an infinite number of possible program paths between the input assertion and the output assertion corresponding to the indefinite number of times the loop might be executed. Thus, the intermediate assertion is essential for this proof method to succeed.

Although a program's assertions may become true or false depending on the location of control in the program, the verification conditions are mathematical statements whose truth is independent of the execution of the program. Given the appropriate assertions, if the program is partially correct, then all the verification conditions will be true; inversely, if the program is not partially correct, at least one of the verification conditions will be false. We have thus transformed the problem of proving the partial correctness of programs to the problem of proving the truth of several mathematical theorems.

The verification of a program with respect to given input-output assertions consists of three phases: finding appropriate intermediate assertions, generating the corresponding verification conditions, and proving that the verification conditions are true. Although generating the verification conditions is a simple mechanical task, finding the intermediate assertions requires a deep understanding of the principles behind the programs, and proving the verification conditions may demand ingenuity and mathematical facility. Also, a knowledge of the subject domain of the program (e.g., the properties of integers or the laws of physics) is required both for finding the intermediate assertions and proving the verification conditions.

One way to apply the above technique is to generate and prove verification conditions by hand. However, in performing such a process we are subject to the same kinds of errors that programmers commit when they construct a program in the first place. An alternate possibility is to generate and prove the verification conditions automatically, by means of a *verification system.* Typically, such a system consists of a *verification condition generator*, which produces the verification conditions, and a *theorem prover*, which attempts to prove them.

Invariant assertions were introduced by Floyd [32] to prove partial correctness of programs, although some traces of the idea appear earlier in the literature. King [48] implemented the first system that used invariant assertions to prove the partial correctness of programs. Given a program, its input-output assertions, and a set of proposed intermediate assertions, King's system generated the verification conditions and attempted to prove them. Some later systems (such as those of Deutsch [26], Elspas *et al.* [31], Good *et al.* [37], Igarashi *et al.* [46], and Suzuki [79]) adopted the same basic approach but employed more powerful theorem provers to prove the verification conditions. Therefore, they were able to prove the partial correctness of a wider class of programs.

Although the above systems have advanced somewhat

beyond King's original effort, they have two principal shortcomings. They require that the user supply an appropriate set of intermediate assertions, and their theorem provers are not powerful enough to prove the verification conditions for most of the programs that arise in practice. Let us consider each of these difficulties separately.

Finding Invariant Assertions: Although the invariant assertions required to perform the verification are guaranteed to exist, to find them one must understand the program thoroughly. Furthermore, even if we can discover the program's principal invariants (e.g., $max\{u : u|x \text{ and } u|y\} = max\{u : u|x_0 \text{ and } u|y_0\}$ above) we are likely to omit some subsidiary invariants (e.g., $y \geqslant 0$ above) that are still necessary to complete the proof. Of course, it would be ideal for the programmer to supply only the program and its input–output assertions and to rely on the verification system to construct all the required intermediate assertions automatically. Much research in this direction has already been done (see, for example, German and Wegbreit [36] and Katz and Manna [47]). However, it is more difficult for a computer system to find the appropriate assertions than for the programmer to provide them, because the principles behind a program may not be readily revealed by the program's instructions. A less ambitious goal is to require the programmer to supply the principal invariants and expect the system to fill in the remaining subsidiary assertions.

Proving Verification Conditions: Verification conditions may be complex formulas, but they are rarely subtle mathematical theorems. Current verification systems can be quite effective if they are given strategies specifically tailored to the subject domain of the program. However, the programs we use in everyday life rely on a large and varied body of subject knowledge, and it is unusual that a system can verify a program in a new subject domain without needing to be extended or adapted in some way (cf. Waldinger and Levitt [83]). Of course, some of this difficulty may be remedied by future theorem-proving research and by the development of interactive verification systems.

The invariant assertions that we attach to intermediate points to prove partial correctness relate the values of the program variables at the intermediate points to their initial values. For instance, in Program A we asserted that

$$x \geqslant 0 \text{ and } y \geqslant 0 \text{ and } (x \neq 0 \text{ or } y \neq 0)$$
$$\text{and } max\{u : u|x \text{ and } u|y\} = max\{u : u|x_0 \text{ and } u|y_0\}$$

at the label *more*. A more recent method, the *subgoal-assertion* method, employs *subgoal assertions* that relate the intermediate values of the program variables with their ultimate values when the program halts. For Program A the subgoal assertion at *more* would be

$$x \geqslant 0 \text{ and } y \geqslant 0 \text{ and } (x \neq 0 \text{ or } y \neq 0)$$
$$\Rightarrow y_f = max\{u : u|x \text{ and } u|y\},$$

where y_f denotes the final value of y at termination. This assertion expresses that whenever control passes through *more* with acceptable values for x and y, the *gcd* of the current values of x and y will be the ultimate value of y.

We prove this relationship by induction on the number of times we have yet to traverse the loop before the program terminates. Whereas the induction for the invariant-assertion method follows the direction of the computation, the induction for the subgoal-assertion method proceeds in the opposite direction. Thus, we first show that the subgoal assertion holds the last time control passes through *more*, when we are about to leave the loop. We then show that if the subgoal assertion holds at *more* after traversing the loop, then it also holds before traversing the loop. This implies that the subgoal assertion holds every time control passes through *more*. Finally, we show that if the subgoal assertion is true the first time control passes through *more*, the desired output assertion holds.

To apply this method to prove the partial correctness of Program A, we need to prove the following verification conditions:

1) $x = 0$
$\Rightarrow [x \geqslant 0 \text{ and } y \geqslant 0 \text{ and } (x \neq 0 \text{ or } y \neq 0)$
$\Rightarrow y = max\{u : u|x \text{ and } u|y\}]$

(the subgoal assertion holds when we are about to leave the loop).

2) $[x \geqslant 0 \text{ and } y - x \geqslant 0 \text{ and } (x \neq 0 \text{ or } y - x \neq 0)$
$\Rightarrow y_f = max\{u|x \text{ and } u|y - x\}]$
and $x \neq 0$
and $y \geqslant x$
$\Rightarrow [x \geqslant 0 \text{ and } y \geqslant 0 \text{ and } (x \neq 0 \text{ or } y \neq 0)$
$\Rightarrow y_f = max\{u : u|x \text{ and } u|y\}]$

(the subgoal assertion after traversing the **then** path of the loop implies the subgoal assertion before traversing the path).

3) $[y \geqslant 0 \text{ and } x \geqslant 0 \text{ and } (y \neq 0 \text{ or } x \neq 0)$
$\Rightarrow y_f = max\{u : u|y \text{ and } u|x\}]$
and $x \neq 0$
and $y < x$
$\Rightarrow [x \geqslant 0 \text{ and } y \geqslant 0 \text{ and } (x \neq 0 \text{ or } y \neq 0)$
$\Rightarrow y_f = max\{u : u|x \text{ and } u|y\}]$

(the subgoal assertion after traversing the **else** path of the loop implies the subgoal assertion before traversing the path).

4) $x_0 \geqslant 0 \text{ and } y_0 \geqslant 0 \text{ and } (x_0 \neq 0 \text{ or } y_0 \neq 0)$
and $[x_0 \geqslant 0 \text{ and } y_0 \geqslant 0 \text{ and } (x_0 \neq 0 \text{ or } y_0 \neq 0)$
$\Rightarrow y_f = max\{u|x_0 \text{ and } u|y_0\}]$
$\Rightarrow y_f = max\{u : u|x_0 \text{ and } u|y_0\}$

(the input assertion and the subgoal assertion the first time we enter the loop imply the output assertion).

Each of these conditions can be easily proved. Conditions (1)–(3) establish that our intermediate assertion is indeed a subgoal assertion. Thus, whenever control reaches *more* the assertion holds for the current values of the program variables

x and y and the ultimate value y_f of y. Condition (4) then ensures that the truth of the subgoal assertion the first time we reach *more* is enough to establish the desired output assertion. Together, these conditions prove the partial correctness of Program A.

From a theoretical point of view, the invariant-assertion method and the subgoal-assertion method are equivalent in power, in that a proof of partial correctness by either of the methods can immediately be rephrased as an equivalent proof by the other method. In practice, however, for a given program the subgoal assertion may be simpler than the invariant assertion, or vice versa. It is also quite possible to apply both methods together in verifying a single program. Thus, the two methods may be regarded as complementary.

The subgoal-assertion method was suggested by Manna [57] and developed by Morris and Wegbreit [65].

In demonstrating the partial correctness of Program A, we employed rigorous but informal mathematical arguments. It is possible to formalize these arguments in a deductive system, much in the same way that logicians formalize ordinary mathematical reasoning. To introduce an *invariant deductive system* for the invariant-assertion approach, we use the notation

$$\{P\}\ F\ \{Q\},$$

where P and Q are logical statements and F is a program segment (a sequence of program instructions), to mean that if P holds before executing F, and if the execution terminates, then Q will hold afterwards. We call an expression of this form an *invariant statement.* For instance,

$$\{x<y\}\ (x\ y)\leftarrow(y\ x)\ \{y<x\}$$

is a true invariant statement, because if the value of x is less than the value of y before interchanging those values, the value of y will be less than the value of x afterwards.

Using this notation, we can express the partial correctness of a program with respect to its input and output assertions by the invariant statement

$$\{\text{input assertion}\}\ \text{program}\ \{\text{output assertion}\}.$$

This statement means that if the input assertion holds, and if the program terminates, then the output assertion will hold; therefore, it adequately states the partial correctness of our program.

To prove such invariant statements we have a number of *rules of inference*, which state that to infer a given invariant statement it suffices to prove several subgoals. These rules are usually presented in the form

$$\frac{A_1, A_2, \cdots, A_n}{B}$$

meaning that to infer the consequent B, it suffices to prove the antecedents $A_1, A_2, \cdots, A_n$. Here B is an invariant statement, and each of $A_1, A_2, \cdots, A_n$ is either a logical statement or another invariant statement. We have one rule corresponding to each statement in our language.

Assignment Rule: Corresponding to the assignment statement

$$(x_1\ x_2 \cdots x_n)\leftarrow(t_1\ t_2 \cdots t_n),$$

which assigns the value of each term t_i to its respective variable x_i simultaneously, is

$$\frac{P(x_1\ x_2 \cdots x_n) \Longrightarrow Q(t_1\ t_2 \cdots t_n)}{\begin{array}{l}\{P(x_1\ x_2 \cdots x_n)\}\\ (x_1\ x_2 \cdots x_n)\leftarrow(t_1\ t_2 \cdots t_n)\\ \{Q(x_1\ x_2 \cdots x_n)\}\end{array}}$$

where $P(x_1\ x_2 \cdots x_n)$ and $Q(x_1\ x_2 \cdots x_n)$ are arbitrary logical statements, and $Q(t_1\ t_2 \cdots t_n)$ is the result of simultaneously substituting t_i for x_i wherever it appears in $Q(x_1\ x_2 \cdots x_n)$. In other words, to infer the invariant statement

$$\begin{array}{l}\{P(x_1\ x_2 \cdots x_n)\}\\ (x_1\ x_2 \cdots x_n)\leftarrow(t_1\ t_2 \cdots t_n)\\ \{Q(x_1\ x_2 \cdots x_n)\},\end{array}$$

it suffices to prove the logical statement

$$P(x_1\ x_2 \cdots x_n) \Longrightarrow Q(t_1\ t_2 \cdots t_n).$$

For example, to prove the invariant statement

$$\{x<y\}\ (x\ y)\leftarrow(y\ x)\ \{y\leqslant x\}$$

it is enough to prove $x<y \Longrightarrow x\leqslant y$.

This rule is valid because each x_i has been assigned the value t_i by the assignment statement. Thus, $Q(x_1\ x_2 \cdots x_n)$ will hold after the assignment if $Q(t_1\ t_2 \cdots t_n)$ held before. Because we are assuming $P(x_1\ x_2 \cdots x_n)$ held before the assignment, it is enough to show $P(x_1\ x_2 \cdots x_n) \Longrightarrow Q(t_1\ t_2 \cdots t_n)$.

Conditional Rule: The rule for the statement "**if R then F_1 else F_2**" is

$$\frac{\{P \text{ and } R\}\ F_1\ \{Q\},\quad \{P \text{ and } \neg R\}\ F_2\ \{Q\}}{\{P\}\ \textbf{if R then } F_1 \textbf{ else } F_2\ \{Q\}}$$

That is, to establish the consequent it suffices to prove the two antecedents $\{P \text{ and } R\}\ F_1\ \{Q\}$, corresponding to the case that R is true, and $\{P \text{ and } \neg R\}\ F_2\ \{Q\}$, corresponding to the case that R is false.

To treat loops in this notation it is convenient to use the **while** statement instead of the **goto**. The statement

while R **do** F

means that the program segment F is to be executed repeatedly as long as the logical statement R is true. In other words, this statement is equivalent to the program segment

more: **if** *not* R **then goto** *enough*
F
goto *more*
enough:

The more concise structure of the **while** statement simplifies the formulation of its rule.

While Rule: Corresponding to the **while** statement we have the rule

$$\frac{P \Rightarrow I,\ \{I \text{ and } R\}\ F\ \{I\},\ \ I \text{ and } \neg R \Rightarrow Q}{\{P\}\ \textbf{while } R \textbf{ do } F\ \{Q\}}$$

for any I. Here, I plays the same role as the invariant assertion in our informal proof; the condition "$P \Rightarrow I$" states that the invariant I is true when we enter the loop; the condition "{I and R} F {I}" conveys that if I is true before executing the loop body F, and if the execution of F terminates, I will be true afterwards; then the condition "I and $\neg R \Rightarrow Q$" ensures that if control ever exits from the loop, then Q will be true.

To apply the **while** rule to infer the desired consequent, we need to find a logical statement I satisfying the three antecedents.

Concatenation Rule: This rule enables us to make inferences about the concatenation $F_1\ F_2$ of two program segments, F_1 and F_2:

$$\frac{\{P\}\ F_1\ \{R\},\ \ \{R\}\ F_2\ \{Q\}}{\{P\}\ F_1\ F_2\ \{Q\}},$$

for any R. The consequent follows from the antecedents. For suppose that P holds before executing $F_1\ F_2$, and that the execution terminates. Then R holds after executing F_1 (by the first antecedent), and therefore Q holds after executing F_2 (by the second antecedent).

These are all the rules in our deductive system. Additional rules are necessary if we wish to add new statements to our programming language.

To prove an invariant statement {P} F {Q}, we apply the appropriate inference rule, of the form

$$\frac{A_1, A_2, \cdots, A_n}{\{P\}\ F\ \{Q\}}.$$

If A_i is an invariant statement, then it is of form {P'} F' {Q'}, where F' is a subsegment of F. In this case, we repeat the process for this antecedent. On the other hand, if A_i is a logical statement, we prove it directly without using any of the rules of the invariant deductive system. Eventually, all the subgoals are reduced to logical statements, which are proved to be true.

Recall that to establish the partial correctness of a program with respect to given input-output assertions, we prove the invariant statement

{input assertion} program {output assertion}.

In this case, the logical statements produced in applying the above procedures are the program's *verification conditions.*

To show how this formalism applies to the partial correctness of the subtractive *gcd* algorithm (Program A), we rewrite this program using a **while** statement instead of a **goto**:

Program A (with while statement):

input$(x_0\ y_0)$
$\{x_0 \geqslant 0$ and $y_0 \geqslant 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)\}$
$(x\ y) \leftarrow (x_0\ y_0)$
while $x \neq 0$ **do**
  $\{invariant(x\ y)\}$
  if $y \geqslant x$ **then** $y \leftarrow y - x$ **else** $(x\ y) \leftarrow (y\ x)$
$\{y = max\{u : u|x_0$ and $u|y_0\}\}$
output(y),

where $invariant(x\ y)$ is taken to be the same invariant we used in our informal invariant-assertion proof, i.e.,

$x \geqslant 0$ and $y \geqslant 0$ and $(x \neq 0$ or $y \neq 0)$
and $max\{u : u|x$ and $u|y\} = max\{u : u|x_0$ and $u|y_0\}$.

This program has the form

input$(x_0\ y_0)$
$\{x_0 \geqslant 0$ and $y_0 \geqslant 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)\}$
Body A
$\{y = max\{u : u|x_0$ and $u|y_0\}\}$
output(y),

and the invariant statement to be proved is

Goal 1. $\{x_0 \geqslant 0$ and $y_0 \geqslant 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)\}$
Body A
$\{y = max\{u : u|x_0$ and $u|y_0\}\}$.

Note that **Body A** is a concatenation of an assignment statement and a **while** statement; thus, the concatenation rule tells us that to establish Goal 1, it suffices to prove

Goal 2. $\{x_0 \geqslant 0$ and $y_0 \geqslant 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)\}$
$(x\ y) \leftarrow (x_0\ y_0)$
$\{R(x\ y)\}$

and

Goal 3. $\{R(x\ y)\}$
while $x \neq 0$ **do** . . .
$\{y = max\{u : u|x_0$ and $u|y_0\}\}$

for some assertion $R(x\ y)$. Here, $R(x\ y)$ can be taken to be $invariant(x\ y)$ itself. (If we make an inappropriate choice for $R(x\ y)$, we may be unable to complete the proof.)

To infer Goal 2, it suffices by the assignment rule to prove the logical statement

Goal 4. $x_0 \geqslant 0$ and $y_0 \geqslant 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)$
$\Rightarrow invariant(x_0\ y_0)$,

which is easily established, because $invariant(x_0\ y_0)$ is simply

$x_0 \geqslant 0$ and $y_0 \geqslant 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)$
and $max\{u : u|x_0$ and $u|y_0\} = max\{u : u|x_0$ and $u|y_0\}$.

The **while** rule reduces Goal 3 to the trivial logical statement

$invariant(x\ y) \Rightarrow invariant(x\ y)$,

and the two new subgoals

Goal 5. $\{invariant(x\ y)$ and $x \neq 0\}$
if $y \geqslant x$ **then** . . . **else** . . .
$\{invariant(x\ y)\}$

and

Goal 6. $invariant(x\ y)$ and $x = 0$
$\Rightarrow y = max\{u : u|x_0$ and $u|y_0\}$.

The **if-then-else** rule reduces Goal 5 to

Goal 7. $\{invariant(x\ y)$ and $x \neq 0$ and $y \geqslant x\}$
$y \leftarrow y - x$
$\{invariant(x\ y)\}$

and

Goal 8. $\{invariant(x\ y)$ and $x \neq 0$ and $y < x\}$
$(x\ y) \leftarrow (y\ x)$
$\{invariant(x\ y)\}$.

Applying the assignment rule to each of these goals yields

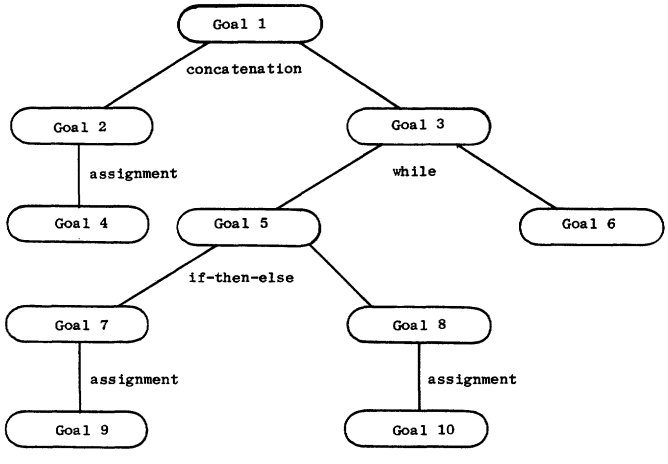
Goal 9. $invariant(x\ y)$ and $x \neq 0$ and $y \geqslant x$
$\Rightarrow invariant(x\ y - x)$

and

Goal 10. $invariant(x\ y)$ and $x \neq 0$ and $y < x$
$\Rightarrow invariant(x\ y - x)$

Now the remaining Goals 6, 9, and 10, like Goal 4, are all logical statements; these are the four verification conditions of Program A. Each of these statements can be shown to be true, and the partial correctness of Program A is thus established.

The above deduction can be summarized in the following "deduction tree":



The above invariant deductive system is essentially the same as the one introduced by Hoare [42].

Whenever a new deductive system is developed, it is natural to ask whether it possesses certain desirable logical properties. The deductive system we have presented has been proved (Cook [18]) to have the following properties.

Soundness: If the verification conditions of a program are true, the program is indeed partially correct.

Completeness: If the program is partially correct, it has true verification conditions.

We have presented the inference rules for only a very simple programming language. Such rules have also been formulated for *goto's*, procedures, and other common programming features (e.g., see Clint and Hoare [17] and Ashcroft *et al.* [3]). However, when more complex features are introduced, finding sound and complete rules to describe them becomes a serious challenge. It has actually been proven impossible to formulate complete rules of inference for certain programming constructs (Clarke [16]).

Part of the difficulty in formulating rules of inference for certain constructs arises because, traditionally, programming languages have been designed without considering how programs using their constructs are to be verified. It has been argued that programming languages designed to allow easier verification will also facilitate the construction of more comprehensible programs. Some recent programming languages designed with such considerations in mind are LUCID (Ashcroft and Wadge [4]), EUCLID (Lampson *et al.* [53]), CLU (Liskov [54]), and ALPHARD (Wulf *et al.* [87]).

Our treatment of partial correctness has been rather idealized: our programming language includes only the simplest of features, and the program we considered was quite straightforward. We have not discussed the more complex problems that occur in verifying the kinds of programs that actually arise in practice.

Let us briefly mention a few of the trouble spots in proving the correctness of practical programs.

Computer Arithmetic: We have assumed that the arithmetic operations performed by the computer correspond precisely with the ideal operations of the mathematician; in fact, the computer is limited in the precision to which a real number can be represented. Consequently, our notion of correctness should be modified to take into account that a computer program only computes an approximation of the mathematical function it is intended to compute (see, e.g., Hull *et al.* [45]).

Cleanness: A computer program may be incorrect not only because it fails to satisfy its output specification, but also because of mishaps that occur during the computation: it may generate a number larger or smaller than the computer system can store (*overflow* or *underflow*), for instance, or it may attempt to divide a number by zero or to find the square root of a negative number. It is possible to prove that a program is *clean* (i.e., that no such accident can occur) by establishing an appropriate invariant before each program statement that might cause offense (Sites [74]). For example, before a statement $z \leftarrow x/y$ we can introduce the assertions that $y \neq 0$ and that $\epsilon \leqslant |x/y| \leqslant E$, where ϵ and E are the smallest and largest positive real numbers, respectively, that the computer system can store.

Side Effects: Many programming constructs have indirect side effects: their execution can alter the properties of entities not explicitly mentioned by the instructions themselves. For instance, suppose our programming language allows assignment to the elements of an array. Then the instruction $A[i] \leftarrow t$, which assigns t to the ith element of an array A, can alter the value of $A[j]$ if it happens that $i = j$, even though $A[j]$ itself is not explicitly mentioned in the instruction. To prove the correctness of programs employing such constructs requires an alteration of the principles outlined here. For example, one consequence of the assignment rule is the invariant statement

$$\{P\}\ x \leftarrow t\ \{P\},$$

where the variable x does not occur in P. If array assign-

ments are admitted, however, one instance of this statement is

$$\{A[j] = 5\}\ A[i] \leftarrow 4\ \{A[j] = 5\}.$$

This statement is false if i can equal j. (For a discussion of such problems, see Oppen and Cook [66].)

Intermediate Behavior of Programs: We have formulated the correctness of a program by providing an output assertion that is intended to be satisfied when the program terminates. However, there are many programs that are not expected to terminate, such as airline reservation systems, operating systems, and conversational language processors. The correctness of these programs cannot be characterized by an output assertion (e.g., see Francez and Pnueli [33]). Moreover, certain properties of such programs are more naturally expressed as a relation between events that occur while the program is running. For instance, in specifying an operating system, we might want to state that if a job is submitted it will ultimately be executed. Even if the operating system does terminate, this property cannot be expressed conveniently as an output assertion. Similarly, in specifying the *security property* of a data base system, to ensure that a user cannot access or alter any file without the proper authorization, we are concerned with the intermediate behavior of the system during execution, and not with any final outcome.

Indeterminacy: Some programming languages have introduced control features that allow the system to choose arbitrarily between several alternate sources of action during execution. For example, the *guarded command* construct (see Dijkstra [27]) allows one to express a program that computes the *gcd* of two strictly positive integers as follows:

input$(x_0\ y_0)$
$(x\ y) \leftarrow (x_0\ y_0)$
do $x > y \Rightarrow x \leftarrow x - y$
$\square\ x > y \Rightarrow (x\ y) \leftarrow (y\ x)$
$\square\ y > x \Rightarrow y \leftarrow y - x$
od
output(x)

This denotes that if $x > y$, we can execute either $x \leftarrow x - y$ or $(x\ y) \leftarrow (y\ x)$, while if $y > x$ we must execute $y \leftarrow y - x$. The statements within the **do** . . . **od** construct are executed repeatedly until neither condition $x > y$ or $y > x$ applies, i.e., until $x = y$. (The terminator "**od**" of the construct is merely "**do**" backwards.) Although for a given input there are many ways of executing the program, the ultimate output is always the *gcd* of the inputs. Extensions of our proof methodology exist to prove the correctness of such programs.

Parallelism: We have only considered programs that are executed sequentially by a single computer processor, but some programs are intended to be executed by several processors at the same time. Many different parts of such a program might be running simultaneously, and the various processors may cooperate in producing the ultimate output. Because the processors may interact with each other during the computation, new obstacles arise in proving the correctness of a parallel program. For example, it becomes desirable to show the absence of *deadlock*, a situation in which two processors each halt and wait for the other to conclude some portion of the task, thus preventing the completion of the program's execution. To prove the correctness of parallel programs requires special techniques; this is currently an active research area (cf., Ashcroft [2], Hoare [44], Owicki and Gries [67]).

Very Large Programs: For the sake of clarity we have discussed only the verification of small programs, but in practice it is the large and complex systems that really require verification. As one would expect, the verification of such programs is obstructed by the larger number and greater complexity of the intermediate assertions and verification conditions. Furthermore, the specifications of a large system are likely to be more difficult even to formulate: one must detail all the situations a spacecraft guidance system is expected to handle, for instance, or all the error messages a compiler is expected to produce. Finally, in a larger system the specifications are likely to be higher level and more abstract, the discrepancy between the specifications and the implementation will be greater, and the verification conditions will be correspondingly more difficult to prove than we have found so far.

It has been argued that such large programs cannot be verified unless they are given a *hierarchical structure* that reduces their apparent complexity. A hierarchically structured program will be decomposed into a few top-level modules, each of which in turn will be decomposed into a few more detailed modules at a lower level. The verification of a module at a given level thus involves only a few lower level modules, each of which may be regarded as a primitive instruction. Therefore, the program becomes understandable, and its verification manageable. (Examples of hierarchical decomposition are given, e.g., in Parnas [68] and Spitzen *et al.* [75].)

One might hope that the above methods for proving the correctness of programs, suitably extended and incorporated into verification systems, would enable us to guarantee that programs are correct with absolute certainty. In the balance of this section we will discuss certain theoretical and philosophical limitations that will prevent this goal from ever being reached. These limitations are inherent in the program verification process, and cannot be surmounted by any technical innovations.

1) We can never be sure that the specifications are correct.

In verifying a program the system assures us that the program satisfies the specifications we have provided. It cannot determine, however, whether those specifications accurately reflect the intentions of the programmer. The intentions, after all, exist only in the mind of the programmer and are inaccessible to a program verification system. If he has made an error in expressing them, the system has no way of detecting the discrepancy.

For example, in specifying a *sort* program one is likely to assert that the elements of the array are to be in order when the program halts, but to neglect to assert that the array's final contents are some permutation of its original contents. In this event, a program that merely resets the first element to 1, the second to 2, and so on, may be verified as a correct

sort program. However, no system will ever be able to detect the missing portion of the specification, because it cannot read the mind of the programmer.

To some extent, these difficulties can be remedied by the use of a well-designed, high-level assertion language. The programmer can express his intentions in such a language quite naturally, and with little chance of error, presumably because he thinks about his problem in the same terms as he expresses it.

2) No verification system can verify every correct program.

For a system to verify a program, it must prove the appropriate verification conditions. Typically, these conditions are logical statements about the numbers or other data structures. Any system that attempts to prove such statements is subject to certain theoretical limitations, no matter how powerful it may be. In particular, it is known to be impossible (as a consequence of Godel's Incompleteness Theorem) to construct a system capable of proving every true statement about the numbers. Consequently, for any verification system there will be some correct program that it cannot verify, even though its specifications are correct and complete.

This theoretical limitation does not preclude the construction of theorem provers useful for program verification. After all, verification conditions are usually not deep mathematical theorems, and it is entirely possible that a computer system will be developed that will be able to verify all the programs that arise in practice. But no matter how powerful a verification system may be, when it fails to verify a program we can never rule out the possibility that the failure is attributable to the weakness of its theorem prover, and not to an error in the program.

3) We can never be certain that a verification system is correct.

When a program has been verified, we must have confidence in the verification system before we believe that the program is really correct. However, a program verifier, like any large system, is subject to bugs, which may enable it to verify incorrect programs. One might suppose that bugs in a verification system could be avoided by allowing the verifier to verify itself. Do not be fooled: if the system does contain bugs, the bugs themselves may cause the program to be verified as correct. As an extreme case, a verifier with a bug that allowed it to verify any program, correct or incorrect, would certainly be able to verify itself.

This philosophical limitation does not imply that there is no use in developing verification systems. Even if the system has bugs itself, it may be useful in finding other bugs in computer programs. A large system (which presumably had *some* bug), written by a graduate student to check mathematical proofs, was able to discover several errors in the *Principia Mathematica* of Whitehead and Russell, a classical source in mathematical logic; a slightly incorrect program verification system could be of comparable value. Moreover, once we have developed a verification system, we make it the focus of all our debugging efforts, instead of spreading our attention over every program that we construct. In this way, although we can never hope to achieve utter certainty that the system is correct, we can establish its correctness "beyond reasonable doubt."

Gerhart and Yelowitz [35] have presented a collection of programs whose verifications were published in the literature but which contained bugs. DeMillo *et al.* [23] advance a philosophical and "sociological" argument against the utility of verifying programs. Dijkstra [29] expresses pessimism about constructing a useful automatic verification system.

Critics of logical techniques for ensuring program correctness often recommend the traditional approach to detecting bugs by *program testing.* In this approach, the program is actually executed on various inputs, and the resulting outputs are examined for some evidence of error. The sample inputs are chosen with the intention of exercising all the program's components, so that any bug in the code will be revealed; however, subtle bugs often escape the most thorough testing process. Some bugs may escape because they occur only upon some legal input configuration that was not anticipated, and therefore not tried, by the programmer. Other bugs may actually occur during a test execution but escape observation because of human carelessness. These problems are discussed in a special issue of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, September 1976, devoted to testing.

Some efforts have been made to apply logical techniques to systematize the testing process. For instance, the SELECT system (Boyer *et al.* [9]) attempts to construct a sample input that will force a given path of the program to be executed. The EFFIGY system (King [49]) executes the program on symbolic inputs rather than concrete numerical quantities, thereby testing the program for an entire class of concrete inputs at once.

It is unlikely that program-verification systems will ever completely eliminate the need for testing. Executing a program is the simplest way to detect obvious bugs. Furthermore, testing a program that has been proved "correct" can indicate some aspect of the programmer's intentions that is not reflected in the specifications.

The techniques we have given in this section establish the partial correctness of a computer program but not its termination. We now turn our attention to techniques for proving the termination of programs.

III. TERMINATION

Proving the termination of programs can be as difficult as proving partial correctness. For instance, consider the following program:

input(x)
while $x \neq 1$ **do**
    if $even(x)$ **then** $x \leftarrow x/2$ **else** $x \leftarrow 3x + 1$
output(x).

This program is known to terminate for every positive integer less than $3 \cdot 10^8$. However, for over a decade no researcher has succeeded in proving its termination for every positive integer, nor in producing a positive integer for which it fails to terminate. Resolution of this question could depend on some deep unknown property of the integers.

Let us examine the subtractive *gcd* algorithm (Program A) again to see informally why we believe it terminates for every input satisfying the input assertion.

input$(x_0\ y_0)$
$\{x_0 \geq 0$ and $y_0 \geq 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)\}$
$(x\ y) \leftarrow (x_0\ y_0)$
more: $\{x \geq 0$ and $y \geq 0$ and $(x \neq 0$ or $y \neq 0)$
and $max\{u : u|x$ and $u|y\} =$
$max\{u : u|x_0$ and $u|y_0\}\}$
if $x = 0$ **then goto** *enough*
if $y \geq x$ **then** $y \leftarrow y - x$ **else** $(x\ y) \leftarrow (y\ x)$
goto *more*
enough: $\{y = max\{u : u|x_0$ and $u|y_0\}\}$
output(y).

Note that in showing the partial correctness of this program we have established as invariant that x and y will always be nonnegative at *more*. Now, observe that every time we go around the loop, either x is reduced, or x is held fixed and y is reduced. First, x is reduced if x and y are interchanged, because y is less than x in this case. On the other hand, if y is set to $y - x$, then x is held fixed and y is reduced, because x is positive when this assignment is executed. The crux of the argument lies in observing that we cannot forever continue reducing x, or holding x fixed and reducing y, without eventually making one of them negative, contradicting the invariant.

To make this argument more rigorous, we introduce the notion of the *lexicographic ordering* $\succ$ on pairs of nonnegative integers. We will say that

$$(x_1\ y_1) \succ (x_2\ y_2),$$

i.e., $(x_1\ y_1)$ is greater than $(x_2\ y_2)$ under the lexicographic ordering, if

$$x_1 > x_2$$
or $x_1 = x_2$ and $y_1 > y_2$.

[Thus $(2\ 2) \succ (1\ 100)$ and $(1\ 4) \succ (1\ 3)$.] The set of pairs of nonnegative integers has the special property that there exist no infinite decreasing sequences under this ordering; i.e., there are no sequences such that

$$(x_1\ y_1) \succ (x_2\ y_2) \succ (x_3\ y_3) \succ \cdots.$$

Proof: Suppose that $(x_1\ y_1), (x_2\ y_2), (x_3\ y_3), \cdots$ is an infinite decreasing sequence of pairs of nonnegative integers. The definition of the lexicographic ordering then requires that $x_1 \geq x_2 \geq x_3 \geq \cdots$, but because the nonnegative integers themselves admit no infinite decreasing sequences, there must exist some n such that $x_n = x_{n+1} = x_{n+2} = \cdots$. (Otherwise we could extract an infinite decreasing subsequence from $x_1, x_2, x_3, \cdots$.) The definition of lexicographic ordering, again, implies that then $y_n > y_{n+1} > y_{n+2} > \cdots$, which violates the same property of the nonnegative integers.

In general, if a set is ordered in such a way that there exist no infinite decreasing sequences, we say that the set is a *well-founded set*, and the ordering a *well-founded ordering*. Thus, the lexicographic ordering is a well-founded ordering on the set of pairs of nonnegative integers, as we showed above.

The nonnegative integers themselves are well-founded under the usual $>$ ordering. However, there exist other well-founded orderings over the nonnegative integers. For example, the ordering defined so that $x \succ y$ if y *properly divides* x, i.e.,

$$y|x \text{ and } y \neq x,$$

is a well-founded ordering.

The well-founded set concept allows us to formulate a more rigorous proof of the termination of Program A. To construct such a proof, we must find a set W with a well-founded ordering $\succ$, and a *termination expression* $E(x\ y)$, such that whenever control passes through the label *more*, the value of $E(x\ y)$ belongs to W, and such that every time control passes around the loop, the value of $E(x\ y)$ is reduced under the ordering $\succ$. This will establish the termination of the program, because if there were an infinite computation, control would pass through *more* an infinite number of times; the corresponding sequence of values of $E(x\ y)$ would constitute an infinite decreasing sequence of elements of W, contradicting the well-foundedness of the set.

To formulate such a termination proof for Program A, we must prove the following three termination conditions for some invariant assertion *invariant*$(x\ y)$ at *more:*

1) *invariant*$(x\ y) \Longrightarrow E(x\ y) \in W$,

   (the value of the termination expression belongs to W when control passes through *more*),

2) *invariant*$(x\ y)$ and $x \neq 0$ and $y \geq x$
   $\Longrightarrow E(x\ y) \succ E(x\ y - x)$

   (the value of the termination expression is reduced if control passes through the **then** branch of the loop), and

3) *invariant*$(x\ y)$ and $x \neq 0$ and $y < x$
   $\Longrightarrow E(x\ y) \succ E(y\ x)$

   (the value of the termination expression is reduced if control passes through the **else** branch of the loop).

Because the invariant will be true every time control passes through *more*, the above conditions suffice to establish termination.

Perhaps the most straightforward way to construct such a termination proof for Program A is to follow our informal demonstration and to take W to be the set of pairs of nonnegative integers, $\succ$ to be the lexicographic ordering, and $E(x\ y)$ to be the pair $(x\ y)$ itself. The invariant assertion *invariant*$(x\ y)$ can simply be taken to be $x \geq 0$ and $y \geq 0$. The termination conditions are then

1) $x \geq 0$ and $y \geq 0$
   $\Longrightarrow (x\ y) \in \{$pairs of nonnegative integers$\}$,

2) $x \geq 0$ and $y \geq 0$ and $x \neq 0$ and $y \geq x$
   $\Longrightarrow (x\ y) \succ (x\ y - x)$, and

3) $x \geq 0$ and $y \geq 0$ and $x \neq 0$ and $y < x$
   $\Longrightarrow (x\ y) \succ (y\ x)$.

We have already indicated in our informal argument the justification for these conditions.

A trickier termination proof may be constructed by taking W to be the nonnegative integers, $\succ$ to be the usual $>$ ordering, and $E(x\ y)$ to be the expression $2x + y$. The termination conditions are then

1) $x \geqslant 0$ and $y \geqslant 0$
   $\Longrightarrow 2x + y \in$ {the nonnegative integers},
2) $x \geqslant 0$ and $y \geqslant 0$ and $x \neq 0$ and $y \geqslant x$
   $\Longrightarrow 2x + y > 2x + (y - x)$, and
3) $x \geqslant 0$ and $y \geqslant 0$ and $x \neq 0$ and $y < x$
   $\Longrightarrow 2x + y > 2y + x$.

These conditions can also be easily established.

The above description illustrates how to prove the termination of a program with only a single loop. If we want to apply the well-founded ordering method to show the termination of a program with several loops, we must designate a particular loop label within each of the program's loops. We choose a single well-founded set and with each designated loop label we associate a termination expression whose value belongs to the well-founded set. These expressions must be chosen so that each time control passes from one designated loop label to another, the value of the expression corresponding to the second label is smaller than the value of the expression corresponding to the first label. Here, "smaller" means with respect to the ordering of the chosen well-founded set. This method establishes the termination of the program, because if there were an infinite computation of the program, control would pass through an infinite sequence of designated labels; the corresponding sequence of values of the termination expressions would constitute an infinite decreasing sequence of elements of the well-founded set, contradicting the well-foundedness of the set, as in the one-loop case.

The well-founded set approach introduces machinery to prove termination completely different from that required to prove partial correctness. There is an alternate approach which extends the invariant-assertion method to prove termination as well as partial correctness. In this approach we alter the program, associating with each loop a new variable called a *counter*. The counter is initialized to 0 before entering the loop and incremented by 1 within the loop body. We must also supply a new intermediate assertion at a point inside the loop, expressing that the corresponding counter does not exceed some fixed bound. In proving that the new assertion is invariant, we show that the number of times the loop can be executed is bounded. (If for some reason control never passes through the assertion, the number of times the loop can be executed is certainly bounded—by zero.) Once we have proved that each loop of the program can only be executed a finite number of times, the program's termination is established.

For instance, to prove that our subtractive *gcd* algorithm (Program A) terminates, we introduce a counter i, and establish that the assertion

$$i \leqslant 2x_0 + y_0$$

is invariant at *more*. To show this, it is actually necessary to prove the stronger assertion

$$x \geqslant 0 \text{ and } y \geqslant 0 \text{ and } 2x + y + i \leqslant 2x_0 + y_0$$

is invariant at *more*. (The stronger assertion implies the weaker because if $x \geqslant 0$ and $y \geqslant 0$ then $2x + y \geqslant 0$.)

Augmented with the counter i and the new intermediate assertion, Program A appears as follows:

Program A (with counter):
input$(x_0\ y_0)$
$\{x_0 \geqslant 0$ and $y_0 \geqslant 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)\}$
$(x\ y) \leftarrow (x_0\ y_0)$
$i \leftarrow 0$
more: $\{x \geqslant 0$ and $y \geqslant 0$ and $2x + y + i \leqslant 2x_0 + y_0\}$
if $x = 0$ **then goto** *enough*
if $y \geqslant x$ **then** $y \leftarrow y - x$ **else** $(x\ y) \leftarrow (y\ x)$
$i \leftarrow i + 1$
goto *more*
enough: **output**(y).

The new assertion is clearly true at *more* initially; it remains true after each execution of the loop body, because each execution reduces the quantity $2x + y$ by at least 1, and i is increased by only 1.

The counter method yields more information than the well-founded set method, because it enables us to establish a bound on the number of times each loop is executed and, hence, on the running time of the program, while termination is being proved. By the same token, however, the counter method is more difficult to apply, because it requires that suitable bounds be known, and we often can prove that a program terminates without knowing such bounds.

Well-founded sets were first used to prove the termination of programs by Floyd [32], in the same paper in which he introduced the invariant-assertion method. The alternate approach, using counters, was suggested by Knuth [50, p. 19]. The program verifier of Luckham and Suzuki [56] proves termination by this method.

IV. Well-Founded Induction

The well-founded sets that we have used to prove termination actually have a much broader domain of application; they can serve as the basis for a proof by mathematical induction using the following *principle of well-founded induction:*

Let W be a set with well-founded ordering $\succ$.
To prove P(w) holds for every element w of W,
  consider an arbitrary element w of W and prove that
    P(w) holds under the assumption that
    P(w') holds for every element w' of W such that
      $w \succ w'$.

In other words, in attempting to prove that every element of a well-founded set has a certain property, we can choose an arbitrary element w of the set, assume as our *induction hypothesis* that every element less than w (in the well-founded ordering) has the property, and prove that w has the property too. (In the special case that no element of W is less than w, the inductive assumption does not tell us anything, and is therefore of no help in proving that w has the property.)

For example, suppose we want to show that every integer greater than or equal to 2 can be expressed as a product of

prime numbers. We can use the principle of well-founded induction, taking W to be the set of integers greater than or equal to 2, and $\succ$ to be the ordinary "greater than" ordering, which is a well-founded ordering of W. Thus, to prove the desired property, we let w be any element of W, and show that w can be expressed as a product of prime numbers using the induction hypothesis that every element of W less than w can be expressed as a product of prime numbers. The proof distinguishes between two cases: if w is a prime, the property holds, because the product of the single prime w is w itself. On the other hand, if w is not a prime, it is the product of two integers w_1 and w_2, each smaller than w and greater than or equal to 2. Because w_1 and w_2 are each members of W less than w under the ordering $\succ$, our induction hypothesis implies that each of them is a product of primes, and hence w is also a product of primes. We then conclude by well-founded induction that every member of W can be expressed as a product of primes. (Alternatively, we could prove the same property taking the well-founded ordering $x \succ y$ to be the properly divides relation defined earlier, i.e., $y|x$ and $y \neq x$. Clearly, if w is the product of w_1 and w_2, then $w \succ w_1$ and $w \succ w_2$ under this ordering.)

> The validity of the principle of well-founded induction is a direct consequence of the definition of a well-founded set. For, suppose we have used the induction hypothesis to prove that P(w) holds for an arbitrary w, but that there actually exists some element w_1 of W such that $\neg$P(w_1). Then for some element w_2 such that $w_1 \succ w_2$, $\neg$P(w_2) holds as well; otherwise, our proof using the induction hypothesis would allow us to conclude P(w_1), contrary to our supposition. The same reasoning applied to w_2 implies the existence of an element w_3 such that $w_2 \succ w_3$ and $\neg$P(w_3), and so on. In this way we can construct an infinite descending sequence of elements $w_1, w_2, w_3, \cdots$ of W, such that $w_1 \succ w_2 \succ w_3 \succ \cdots$, contradicting the well-foundedness of W.

The well-founded ordering method we introduced for proving termination may be regarded as an application of the principle of well-founded induction. For instance, recall that to apply the well-founded set method to prove the termination of Program A, we need to find a well-founded set W ordered by the ordering $\succ$ and a termination expression E(x y) such that whenever control passes through *more*, the value of E(x y) belongs to W, and such that whenever control passes around the loop, the value of E(x y) is reduced under the ordering $\succ$. To phrase this method as a well-founded induction proof, we prove the property that if during a computation control passes through *more*, the computation will terminate. The well-founded set used as a basis for the induction is the set of pairs of nonnegative integers, and the ordering $\succ\succ$ is defined by

$$(w_1\ w_2) \succ\succ (w_1'\ w_2') \quad \text{if} \quad \mathrm{E}(w_1\ w_2) \succ \mathrm{E}(w_1'\ w_2').$$

We show that the property holds for arbitrary values $(w_1\ w_2)$ of the pair $(x\ y)$ at *more*, assuming the induction hypothesis that the program will terminate if control passes through *more* with values $(w_1'\ w_2')$ of $(x\ y)$ such that $(w_1\ w_2) \succ\succ (w_1'\ w_2')$, i.e., such that $\mathrm{E}(w_1\ w_2) \succ \mathrm{E}(w_1'\ w_2')$. Following the two well-founded sets in the termination proofs of the previous section, we can either take $E(x\ y)$ to be $(x\ y)$ itself, and $\succ$ to be the lexicographic ordering between pairs of nonnegative integers, or we can take $E(x\ y)$ to be $2x + y$, and $\succ$ to be the usual greater than ordering between nonnegative integers. The details of the proof then correspond closely to the steps in the well-founded set termination proof.

In proving partial correctness by the invariant-assertion and the subgoal-assertion methods, we employed induction based on the number of steps in the computation; for this reason they are classified as forms of *computational induction*. On the other hand, our proof of termination employed an induction independent of the computation; such proofs are generally referred to as *structural induction* proofs.

In subsequent sections we will encounter the principle of well-founded induction in many guises.

V. Total Correctness

So far we have considered correctness separately from termination; to prove that a program halts and produces the desired result required two separate proofs. In this section we will introduce a technique that establishes the *total correctness* of a program, i.e., its termination and correctness, with a single proof.

In our previous correctness proofs we attached assertions to points in the program, with the intended meaning that the assertion is to be invariant, that is to hold *every* time control passes through the corresponding point. Conceivably, the assertion could be proved to be invariant even though control never passes through the point in question. In particular, we can prove that the output assertion is invariant even though the program never halts; thus, a separate termination proof is required.

In the method we are about to introduce, we will also attach assertions to points in the program, but with the intended meaning that *sometime* control will pass through the point and satisfy the attached assertion. In other words, control may pass through the point many times without satisfying the assertion, but control will pass through the point at least once with the assertion satisfied; therefore, we call these assertions *intermittent assertions*. If we manage to prove that the output assertion is an intermittent assertion at the program's exit, we have simultaneously shown that the program must halt and satisfy the output assertion. This establishes the program's total correctness.

We will use the phrase

sometime Q at L

to denote that Q is an intermittent assertion at the label L, i.e., that sometime control will pass through L with assertion Q satisfied. (Similarly, we could have used the phrase "always Q at L" to indicate that Q is an invariant assertion at L.) If the entrance of a program is labelled *start* and its exit is labelled *enough*, we can express the total correctness of the program with respect to an input assertion P and output assertion R by

if sometime P at *start*
then sometime R at *enough*.

Generally, to prove this statement as a theorem, we must affix intermittent assertions to some of the program's intermediate points, and supply lemmas to relate these assertions. The proof of these lemmas typically employs well-founded induction.

To illustrate this method we introduce a new program to compute the greatest common divisor.

Program B (the symmetric algorithm):
input$(x_0\ y_0)$
start: $(x\ y) \leftarrow (x_0\ y_0)$
more: **if** $x = y$ **then goto** *enough*
reducex: **if** $x > y$ **then** $x \leftarrow x - y$
goto *reducex*
reducey: **if** $y > x$ **then** $y \leftarrow y - x$
goto *reducey*
goto *more*
enough: **output**(y).

This program is only intended to be used for positive x_0 and y_0, whereas the previous Program A can also be used when either $x_0 = 0$ or $y_0 = 0$. The indenting indicates that the two instructions $x \leftarrow x - y$ and **goto** *reducex* are to be treated as a block; both are to be executed only if $x > y$. Similarly, the two instructions $y \leftarrow y - x$ and **goto** *reducey* are executed only if $y > x$.

The intuitive basis for this program rests on the following properties of the integers:

a) $u|x$ and $u|y \Leftrightarrow u|x - y$ and $u|y$
(the common divisors of $x - y$ and y are the same as those of x and y), or, equivalently,
b) $u|x$ and $u|y \Leftrightarrow u|x$ and $u|y - x$
(the common divisors of x and $y - x$ are the same as those of x and y), and
c) $max\{u : u|y\} = y$ if $y > 0$
(any positive integer is its own greatest divisor).

We would like to use the intermittent-assertion method to prove the total correctness of Program B. The total correctness can be expressed as follows:

Theorem: if sometime $x_0 > 0$ and $y_0 > 0$ at *start*
then sometime $y = max\{u : u|x_0$ and $u|y_0\}$
at *enough.*

This theorem states the termination as well as the partial correctness of Program B, because it asserts that control must eventually reach *enough*, the exit of the program, given that it begins execution with positive x_0 and y_0.

To prove this theorem we need a lemma that describes the internal behavior of this program:

Lemma: if sometime $x = a$ and $y = b$ and $a, b > 0$ at *more*
or sometime $x = a$ and $y = b$ and $a, b > 0$ at
reducex
or sometime $x = a$ and $y = b$ and $a, b > 0$ at
reducey
then sometime $y = max\{u : u|a$ and $u|b\}$ at
enough.

To show that the lemma implies the theorem, we assume that

sometime $x_0 > 0$ and $y_0 > 0$ at *start.*

Then control passes to *more*, with x and y set to x_0 and y_0, respectively, so we have

sometime $x = x_0$ and $y = y_0$ and $x_0, y_0 > 0$ at *more.*

But then the lemma implies that

sometime $y = max\{u : u|x_0$ and $u|y_0\}$ at *enough,*

which is the desired conclusion of the theorem.

It remains to prove the lemma. We assume

sometime $x = a$ and $y = b$ and $a, b > 0$ at *more*
or sometime $x = a$ and $y = b$ and $a, b > 0$ at *reducex*
or sometime $x = a$ and $y = b$ and $a, b > 0$ at *reducey*

and show that

sometime $y = max\{u : u|a$ and $u|b\}$ at *enough.*

The proof employs well-founded induction on the set of pairs of nonnegative integers, under the well-founded ordering $\succ$ defined by

$(a\ b) \succ (a'\ b')$ if $a + b > a' + b'$.

In other words, during the proof we will assume that the lemma holds whenever $x = a'$ and $y = b'$, where $a + b > a' + b'$; i.e., we take as our induction hypothesis that

if sometime $x = a'$ and $y = b'$ and $a', b' > 0$ at *more*
or sometime $x = a'$ and $y = b'$ and $a', b' > 0$ at *reducex*
or sometime $x = a'$ and $y = b'$ and $a', b' > 0$ at *reducey*
then sometime $y = max\{u : u|a'$ and $u|b'\}$ at *enough.*

The proof distinguishes between three cases.

Case $a = b$: Regardless of whether control is at *more*, *reducex*, or *reducey*, control passes to *enough* with $y = b$, so that

sometime $y = b$ at *enough.*

But in this case $b = max\{u : u|b\} = max\{u : u|a$ and $u|b\}$, by Property c) above. Thus,

sometime $y = max\{u : u|a$ and $u|b\}$ at *enough,*

which is the desired conclusion of the lemma.

Case $a > b$: Regardless of whether control is at *more*, *reducex*, or *reducey*, control reaches *reducex* and passes around the top inner loop, resetting x to $a - b$, so that

sometime $x = a - b$ and $y = b$ at *reducex.*

For simplicity, let us denote $a - b$ by a' and b by b'. Note that

$a', b' > 0$,
$a + b > a' + b'$, and
$max\{u : u|a'$ and $u|b'\} = max\{u : u|a - b$ and $u|b\}$
$= max\{u : u|a$ and $u|b\}$.

This last condition follows from Property a) above.

Because $a', b' > 0$ and $a + b > a' + b'$, the induction hypothesis implies that

sometime $y = max\{u : u|a' \text{ and } u|b'\}$ at *enough;*

i.e., by the third condition above,

sometime $y = max\{u : u|a \text{ and } u|b\}$ at *enough.*

This is the desired conclusion of the lemma.

Case $b > a$: This case is disposed of in a manner symmetric to the previous case.

This concludes the proof of the lemma. The total correctness of Program B is thus established.

Let us see how we would prove the correctness and termination of Program B if we were using the methods of the previous sections instead.

The partial correctness of Program B is straightforward to prove using the invariant-assertion method introduced in Section II. The invariant-assertions at *more*, *reducex*, and *reducey*, can all be taken to be

$x > 0$ and $y > 0$
and $max\{u : u|x \text{ and } u|y\} = max\{u : u|x_0 \text{ and } u|y_0\}$.

In contrast, it is awkward to prove the termination of this program by the well-founded ordering approach we discussed in Section III; it is possible to pass from *more* to *reducex*, from *reducex* to *reducey*, or from *reducey* to *more* without altering the value of any program variables. Consequently, it is difficult to find expressions whose values are reduced whenever control passes from one of these labels to the next. One possibility is to take the well-founded set to be the pairs of nonnegative integers ordered by the lexicographical ordering; the termination expressions corresponding to the loop labels are taken to be

$(x + y\ \ 2)$	at *more*,
if $x \neq y$ then $(x + y\ \ 1)$ else $(x + y\ \ 4)$	at *reducex*, and
if $x < y$ then $(x + y\ \ 0)$ else $(x + y\ \ 3)$	at *reducey*.

It can be shown that as control passes from one loop label to the next the values of the corresponding termination expressions decrease. Although this approach is effective, it is unduly complicated.

The above example illustrates that the intermittent-assertion method may be more natural to apply than one of the earlier methods. It can be shown that the reverse is not the case: a proof of partial correctness by either of the methods of Section II or of termination by either of the methods of Section III can be rephrased directly as a proof using intermittent assertions. In this sense, the intermittent assertion method is more powerful than the others.

The intermittent-assertion method was first formulated by Burstall [14] and further developed by Manna and Waldinger [61]. Different approaches to its formalization have been attempted, using predicate calculus (Schwarz [72]), a deductive system (Wang [84]), and modal logic (Pratt [70]).

VI. Correctness of Recursive Programs

So far, we have indicated repeated operations by a particular kind of loop, the iterative loop, which is expressed with the **goto** or **while** statement. We are about to introduce a new looping construct that is in some sense more powerful than the iterative loop. This construct, the *recursive call*, allows a program to use itself as its own subprogram. A recursive call denotes a repeated operation because the subprogram can then use itself again, and so on.

For instance, consider the following recursive version of our subtractive *gcd* algorithm (Program A):

Program A (a recursive version):

gcdminus$(x\ y) \Leftarrow$ **if** $x = 0$
then y
else if $y \geqslant x$
then *gcdminus*$(x\ \ y - x)$
else *gcdminus*$(y\ \ x)$.

In other words, to compute the *gcd* of inputs x and y, test if $x = 0$; if so, return y as the output; otherwise test if $y \geqslant x$; if so, return the value of a recursive call to this same program on inputs x and $y - x$; if not, return the value of another recursive call, with inputs y and x. For example, in computing the *gcd* of 6 and 3 we get the following sequence of recursive calls:

gcdminus$(6\ 3) \Leftarrow$ *gcdminus*$(3\ 6) \Leftarrow$ *gcdminus*$(3\ 3)$
$\Leftarrow$ *gcdminus*$(3\ 0) \Leftarrow$ *gcdminus*$(0\ 3) \Leftarrow 3$.

Thus, the value of *gcdminus*$(6\ 3)$ is 3. Although a recursive definition is apparently circular, it represents a precise description of a computation. Note that *gcdminus* is a "dummy" symbol and, like a loop label, can be replaced by any other symbol without changing the meaning of the program.

A recursive computation can be infinite if the execution of one recursive call leads to the execution of another recursive call, and so on, without ever returning an output. For example, the program

gcdnostop$(x\ y) \Leftarrow$ **if** $x = 0$
then y
else if $y \geqslant x$
then *gcdnostop*$(x\ \ y - x)$
else *gcdnostop*$(x - y\ \ y)$,

which is obtained from Program A by altering the arguments of the second recursive call, computes the *gcd* of those inputs for which it halts. However, this program will not terminate for many inputs, e.g., if $x \neq 0$ and $y = 0$ or if $x \neq 0$ and $y = x$. Thus, for $x = 3$ and $y = 3$ we obtain the infinite computation

gcdnostop$(3\ 3) \Leftarrow$ *gcdnostop*$(3\ 0) \Leftarrow$ *gcdnostop*$(3\ 0)$
$\Leftarrow$ *gcdnostop*$(3\ 0) \Leftarrow \cdots$.

Our recursive version of Program A describes essentially the same computation and produces the same outputs as the iterative version. In fact, it is straightforward to transform any iterative program into a recursive program that performs the same computation. The reverse transformation, however, is not so straightforward; in translating a recursive program into a corresponding iterative one, it is often necessary to introduce devices to simulate the recursion, complicating the program considerably. Some computational problems can be solved quite naturally by a recursive program for which there is no iterative equivalent of comparable simplicity.

As a new specimen for our study of recursion we will intro-

duce a recursive cousin of the greatest common divisor algorithm of Euclid, which appeared in his *Elements* over 2200 years ago.

Program C (the Euclidean algorithm):

$gcdrem(x\ y) \Leftarrow$ **if** $x = 0$
then y
else $gcdrem(rem(y\ x)\ x)$.

Here $rem(y\ x)$ indicates the remainder when y is divided by x. Program C, like Program A, computes the *gcd* of any nonnegative integers x and y, where x and y are not both zero. The correctness of this program will be seen to depend on the following properties of the integers:

a) $u|x$ and $u|y \Leftrightarrow u|x$ and $u|rem(y\ x)$ if $x \neq 0$
(the common divisors of x and y are the same as those of x and $rem(y\ x)$, if $x \neq 0$),

b) $u|0$
(every integer divides 0),

c) $max\{u : u|y\} = y$ if $y > 0$
(every positive integer is its own greatest divisor), and

d) $x > rem(y\ x) \geqslant 0$ if $x > 0$.

The reader may be interested to see a proof of Property a). Suppose that $u|x$ and $u|y$ and that $x \neq 0$. We need to show that $u|rem(y\ x)$. We know that $x = k \cdot u$ and $y = l \cdot u$, for some integers k and l. But $rem(y\ x)$ is defined so that $y = q \cdot x + rem(y\ x)$, where q is the quotient of y and x. Therefore $rem(y\ x) = y - q \cdot x = l \cdot u - q \cdot k \cdot u = u \cdot (l - q \cdot k)$, so that $u|rem(y\ x)$, as we intended to prove. The proof in the opposite direction is similar.

We would like to introduce techniques for proving the correctness and termination of recursive programs. In proving the properties of iterative programs, we often employed the principle of well-founded induction. We distinguished between computational induction, which was based on the number of steps in the computation, and structural induction, which was independent of the computation. These versions of the induction principle have analogues for proving properties of recursive programs. We will illustrate these techniques in proving the correctness and termination of the above recursive Euclidean algorithm (Program C).

To apply computational induction to Program C, we perform induction on the number of recursive calls in the computation of $gcdrem(x\ y)$. (This number is finite if we assume that the computation terminates.) Thus, in proving that some property holds for $gcdrem(x\ y)$, we assume inductively that the property holds for $gcdrem(x'\ y')$, where x' and y' are any nonnegative integers such that the computation of $gcdrem(x'\ y')$ involves fewer recursive calls than the computation of $gcdrem(x\ y)$.

Now, let us use computational induction to show that Program C is partially correct with respect to the *input specification*

$x \geqslant 0$ and $y \geqslant 0$ and $(x \neq 0$ or $y \neq 0)$,

and the *output specification*

$gcdrem(x\ y) = max\{u : u|x$ and $u|y\}$.

Thus, we must prove the property that

for every input x and y such that
$x \geqslant 0$ and $y \geqslant 0$ and $(x \neq 0$ or $y \neq 0)$,
if the computation of $gcdrem(x\ y)$ terminates, then
$gcdrem(x\ y) = max\{u : u|x$ and $u|y\}$.

Therefore, we consider arbitrary nonnegative integers x and y and attempt to prove that the above property holds for these integers, assuming as our induction hypothesis that the property holds for any nonnegative integers x' and y' such that the computation of $gcdrem(x'\ y')$ involves fewer recursive calls than the computation of $gcdrem(x\ y)$.

Thus, we suppose that

$x \geqslant 0$ and $y \geqslant 0$ and $(x \neq 0$ or $y \neq 0)$,

and that the computation of $gcdrem(x\ y)$ terminates. We would like to show that

$gcdrem(x\ y) = max\{u : u|x$ and $u|y\}$.

Following the definition of *gcdrem*, we distinguish between two cases.

If $x = 0$, then Program C dictates that

$gcdrem(x\ y) = y$.

But because we have assumed that $x \neq 0$ or $y \neq 0$ and that $y \geqslant 0$, we know that $y > 0$. Therefore, by Properties b) and c),

$max\{u : u|x$ and $u|y\} = max\{u : u|y\} = y$.

Thus,

$gcdrem(x\ y) = y = max\{u : u|x$ and $u|y\}$,

as we wanted to prove.

On the other hand, if $x \neq 0$, Program C dictates that

$gcdrem(x\ y) = gcdrem(rem(y\ x)\ x)$.

Because a recursive call to $gcdrem(rem(y\ x)\ x)$ occurs in the computation of $gcdrem(x\ y)$, and because the computation has been assumed to terminate, the computation of $gcdrem(rem(y\ x)\ x)$ involves fewer recursive calls than the computation of $gcdrem(x\ y)$.

Therefore we would like to apply the induction hypothesis, taking x' to be $rem(y\ x)$ and y' to be x. For this purpose, we attempt to prove the antecedent of the induction hypothesis, i.e.,

$rem(y\ x) \geqslant 0$ and $x \geqslant 0$ and $(rem(y\ x) \neq 0$ or $x \neq 0)$

and that the computation of $gcdrem(rem(y\ x)\ x)$ terminates. However, we know that $rem(y\ x) \geqslant 0$ by Property d), that $x \geqslant 0$ by the input specification, and that $x \neq 0$ by our case assumption. Furthermore, we know that the computation of $gcdrem(rem(y\ x)\ x)$ terminates, because it is part of the computation of $gcdrem(x\ y)$, which has been assumed to terminate. Our induction hypothesis therefore allows us to conclude that

$gcdrem(rem(y\ x)\ x) = max\{u : u|rem(y\ x)$ and $u|x\}$.

But, by Property a),

$max\{u : u|rem(y\ x)$ and $u|x\} = max\{u : u|x$ and $u|y\}$,

and therefore

$$gcdrem(x\ y) = max\{u : u|x \text{ and } u|y\},$$

as desired. This concludes the proof of the partial correctness of Program C.

In the above computational-induction proof we were forced to assume that the computation terminates. However, if we choose an appropriate well-founded ordering independent of the computation, we can employ structural induction to prove termination as well as correctness. For example, suppose we want to prove the termination of Program C for all inputs satisfying the input specification; in other words,

For every input x and y such that
$x \geq 0$ and $y \geq 0$ and $(x \neq 0$ or $y \neq 0)$,
the computation of $gcdrem(x\ y)$ terminates.

The well-founded set which will serve as the basis for the structural induction is the set W of all pairs $(w_1\ w_2)$ of nonnegative integers, under the ordering $\succ$ defined by

$$(w_1\ w_2) \succ (w_1'\ w_2') \quad \text{if } w_1 > w_1'.$$

(Yes, the second component is ignored completely.)

To prove the termination property, we consider arbitrary nonnegative integers x and y and attempt to prove that the property holds for these integers, assuming as our induction hypothesis that the property holds for any nonnegative integers x' and y' such that $(x\ y) \succ (x'\ y')$, i.e., $x > x'$.

Thus, we suppose that

$$x \geq 0 \text{ and } y \geq 0 \text{ and } (x \neq 0 \text{ or } y \neq 0).$$

Following the definition of *gcdrem*, we again distinguish between two cases. If $x = 0$, the computation terminates immediately. On the other hand, if $x \neq 0$, the program returns as its output the value of the recursive call $gcdrem(rem(y\ x)\ x)$. Because $x > rem(y\ x)$, by Property d), we have

$$(x\ y) \succ (rem(y\ x)\ x),$$

and therefore we would like to apply the induction hypothesis, taking x' to be $rem(y\ x)$ and y' to be x. For this purpose, we prove the antecedent of the induction hypothesis, that

$$rem(y\ x) \geq 0 \text{ and } x \geq 0 \text{ and } (rem(y\ x) \neq 0 \text{ or } x \neq 0),$$

using Property d), the input specification, and the case assumption, respectively. The consequent of the induction hypothesis tells us that the computation of $gcdrem(rem(y\ x)\ x)$, and therefore of $gcdrem(x\ y)$, terminates. This concludes the proof of the termination of Program C.

Of course, we could have used structural induction, with the same well-founded ordering, to prove the total correctness of Program C. For this purpose we would prove the property that

For every input x and y such that
$x \geq 0$ and $y \geq 0$ and $(x \neq 0$ or $y \neq 0)$,
the computation of $gcdrem(x\ y)$ terminates and
$gcdrem(x\ y) = max\{u : u|x \text{ and } u|y\}$.

The proof would be similar to the above termination proof.

Euclid, himself, presented a "proof" of the properties of his *gcd* algorithm. His termination proof was an informal version of a well-founded ordering proof, but his correctness proof considered only two special cases, in which the recursive call is executed precisely one or three times during the computation. The principle of mathematical induction, which would have been necessary to handle the general case, was unknown at the time.

The reader may have noticed that the proofs of correctness and termination for the recursive program presented here did not require the invention of the intermediate assertions or lemmas that our proofs for iterative programs demanded. He may have been led to conclude that proofs of recursive programs are always simpler than proofs of the corresponding iterative programs; in general, this is not the case. Often, in proving a property by the well-founded induction principle, it is necessary to establish a more general property in order to have the advantage of a stronger induction hypothesis. For example, suppose we wanted to prove that Program C satisfies the property that

$$gcdrem(x\ y)|x.$$

If we tried to apply an inductive proof directly, the induction hypothesis would yield merely that

$$gcdrem(rem(y\ x)\ x)|rem(y\ x);$$

this assumption is not strong enough to imply the desired property. To prove the property we must instead prove a more general property, such as that

$$gcdrem(x\ y)|x \text{ and } gcdrem(x\ y)|y.$$

The induction hypothesis would then yield that

$$gcdrem(rem(y\ x)\ x)|rem(y\ x) \text{ and } gcdrem(rem(y\ x)\ x)|x,$$

which is enough to imply the more general result. It may require considerable ingenuity to find the appropriate stronger property that will enable the inductive proof to go through.

We have used structural induction to show the termination of a program, and we have indicated how it can be used to show the total correctness of a program. We will now show how structural induction can be used to prove an entirely different property: the equivalence of two programs.

We say that two programs are *equivalent* with respect to some input specification if they terminate for precisely the same legal inputs, and if they produce the same outputs when they do terminate. We will write $f(x) \equiv g(x)$ if, either the computations of $f(x)$ and $g(x)$ both terminate and yield the same output, or if they both fail to terminate. Then we can say that f is equivalent to g with respect to a given input specification if, for all x satisfying the input specification, $f(x) \equiv g(x)$.

Let us see how structural induction can be applied to prove the equivalence of the subtractive *gcd* algorithm (Program A) and the Euclidean *gcd* algorithm (Program C) we have introduced in this section. Recall that the Euclidean algorithm is

$$gcdrem(x\ y) \Leftarrow \textbf{if } x = 0 \textbf{ then } y \textbf{ else } gcdrem(rem(y\ x)\ x),$$

and the subtractive algorithm is

$$gcdminus(x\ y) \Leftarrow \textbf{if } x = 0 \textbf{ then } y \textbf{ else if } y \geqslant x \textbf{ then } gcdminus(x\ y - x) \textbf{ else } gcdminus(y\ x).$$

The remainder function *rem* can be defined by the recursive program

$$rem(u\ v) \Leftarrow \textbf{if } u < v \textbf{ then } u \textbf{ else } rem(u - v\ v),$$

where v is assumed not to be zero.

To establish the equivalence of the two *gcd* programs, we need to prove that

$$\text{if } x \geqslant 0 \text{ and } y \geqslant 0 \text{ and } (x \neq 0 \text{ or } y \neq 0) \text{ then } gcdrem(x\ y) \equiv gcdminus(x\ y).$$

The proof of this property is a straightforward application of structural induction, in which the well-founded set is the set of pairs of nonnegative integers ordered by the lexicographic ordering $\succ$. We consider arbitrary nonnegative integers x and y and attempt to prove that the equivalence property holds for these integers, assuming as our induction hypothesis that the property holds for any nonnegative integers x' and y' such that $(x\ y) \succ (x'\ y')$.

Thus, we suppose that

$$x \geqslant 0 \text{ and } y \geqslant 0 \text{ and } (x \neq 0 \text{ or } y \neq 0)$$

and attempt to prove that

$$gcdrem(x\ y) \equiv gcdminus(x\ y).$$

The proof distinguishes between several cases.

If $x = 0$, both programs terminate and yield y as their output.

On the other hand, if $x \neq 0$ and $y < x$, the Euclidean algorithm executes a recursive call

$$gcdrem(rem(y\ x)\ x),$$

or (by the definition of *rem*, since $y < x$)

$$gcdrem(y\ x).$$

In this case, the subtractive algorithm executes a recursive call

$$gcdminus(y\ x).$$

Recall that $x > y$, and therefore that $(x\ y) \succ (y\ x)$. Thus, because y and x satisfy the input specification

$$y \geqslant 0 \text{ and } x \geqslant 0 \text{ and } (y \neq 0 \text{ or } x \neq 0),$$

our induction hypothesis yields that

$$gcdrem(y\ x) \equiv gcdminus(y\ x),$$

i.e., (in this case)

$$gcdrem(x\ y) \equiv gcdminus(x\ y).$$

Finally, if $x \neq 0$ but $y \geqslant x$, the Euclidean algorithm executes a recursive call

$$gcdrem(rem(y\ x)\ x),$$

or (by the definition of *rem*)

$$gcdrem(rem(y - x\ x)\ x),$$

or (by the definition of *gcdrem*)

$$gcdrem(x\ y - x).$$

In this case, the subtractive algorithm executes a recursive call

$$gcdminus(x\ y - x).$$

Note that $x > 0$, and therefore that $(x\ y) \succ (x\ y - x)$. Thus, because here x and $y - x$ satisfy the input specification

$$x \geqslant 0 \text{ and } y - x \geqslant 0 \text{ and } (x \neq 0 \text{ or } y - x \neq 0),$$

the induction hypothesis yields that

$$gcdrem(x\ y - x) \equiv gcdminus(x\ y - x),$$

i.e., (in this case)

$$gcdrem(x\ y) \equiv gcdminus(x\ y).$$

This concludes the proof of the equivalence of the two *gcd* algorithms.

The two *gcd* programs we have shown to be equivalent both happen to terminate for all legal inputs. However, the same proof technique could be applied as well to show the equivalence of two programs that do not always terminate, provided that they each fail to terminate for the same inputs.

In general, to solve a programming problem can require not one but a *system of recursive programs*, each of which may call any of the others. Even our simple recursive Euclidean algorithm can be regarded as a system of programs, because *gcdrem* calls the recursive remainder program *rem*. Everything we have done in this section can be extended naturally to treat such systems of programs.

Various forms of computational induction were applied to recursive programs by deBakker and Scott [22], Manna and Pnueli [59], and Morris [64]. The structural induction method was first presented as a technique for proving properties of recursive programs by Burstall [13]. A verification system employing this method was implemented by Boyer and Moore [10].

VII. Program Transformation

Up to now we have been discussing ways of proving the correctness and termination of a given program. We are about to consider logical techniques to transform and improve the given program. These transformations may change the computation performed by the program drastically, but they are guaranteed to produce a program equivalent to the original; we therefore call them *equivalence-preserving transformations*. Usually, a sequence of such transformations is applied to *optimize* the program, i.e., to make it more economical in its use of time or space.

Perhaps the simplest way of expressing a transformation is as a rule that states that a program segment of a certain form can be replaced by a program segment of another form.

For example, an assignment statement of form

$$x \leftarrow f(\alpha\, \alpha \cdots \alpha),$$

which contains several occurrences of a subexpression α, may be replaced by the program segment

$$y \leftarrow \alpha$$
$$x \leftarrow f(y\, y \cdots y),$$

where y is a new variable. This transformation often optimizes the program, because the subexpression α will only be computed once by the latter segment. For instance, the assignment

$$x \leftarrow (a^b)^3 + 2(a^b)^2 + 3(a^b)$$

may be replaced by the segment

$$y \leftarrow a^b$$
$$x \leftarrow y^3 + 2y^2 + 3y.$$

Such *elimination of common subexpressions* is performed routinely by optimizing compilers.

Another transformation: in a program segment of form

if p
then α
else if p
  then β
  else γ

the second test of p, if executed, will always yield *false*; the expression β will never be evaluated. Therefore, this segment can always be replaced by the equivalent segment of form

if p
then α
else γ.

Another example: a **while** loop of form

while $p(x)$ and $q(x\, y)$ **do** $y \leftarrow f(y)$

may be replaced by the equivalent statement of form

if $p(x)$ **then while** $q(x\, y)$ **do** $y \leftarrow f(y)$,

if y does not occur in $p(x)$ and the evaluation of $f(y)$ has no side effects. The former segment will test both $p(x)$ and $q(x\, y)$ and execute the assignment $y \leftarrow f(y)$ repeatedly, even though the outcome of the test $p(x)$ cannot be affected by the assignment statement. The latter segment will test $p(x)$ only once, and execute the **while** loop only if the outcome is *true*. Therefore, this transformation optimizes the program to which it is applied.

An important class of program transformations are those that effect the removal of recursive calls from the given program. Recursion can be an expensive convenience, because its implementation generally requires much time and space. If we can replace a recursive call by an equivalent iterative loop, we may have achieved a great savings.

One transformation for recursion removal states that a recursive program of form α:

$F(u) \Leftarrow$ **if** $p(u)$
  then $g(u)$
  else $F(h(u))$

can be replaced by an equivalent iterative program of form β:

  input(u)
more: **if** $p(u)$ **then output**$(g(u))$
  $u \leftarrow h(u)$
  goto *more*.

To see that the two programs are equivalent, suppose we apply each program to an input a. First, if $p(a)$ is true, each program produces output $g(a)$. Otherwise, if $p(a)$ is false, the iterative program will replace u *by* $h(a)$ and go to *more:* thus, its output will be the same as if its input had been $h(a)$. In this case, the recursive program will return $F(h(a))$; thus, its output, too, is the same as if its input had been $h(a)$.

For example, this transformation will enable us to replace our recursive Euclidean algorithm (Program C)

$gcdrem(x\, y) \Leftarrow$ **if** $x = 0$
  then y
  else $gcdrem(rem(y\, x)\, x)$

by the equivalent iterative program

  input$(x\, y)$
more: **if** $x = 0$ **then output**(y)
  $(x\, y) \leftarrow (rem(y\, x)\, x)$
  goto *more*.

For some forms of recursive programs, the corresponding iterative equivalent is more complex. For instance, a recursive program of form

$F(u) \Leftarrow$ **if** $p(u)$
  then $g(u)$
  else $k(u) + F(h(u))$

can be transformed into the iterative program of form

  input(u)
  $z \leftarrow 0$
more: **if** $p(u)$
  then output$(z + g(u))$
  else $(u\, z) \leftarrow (h(u)\ \ z + k(u))$
  goto *more*.

However, the iterative program requires the use of an additional variable z to maintain a running subtotal. A more complex recursive program, such as one of form

$F(u) \Leftarrow$ **if** $p(u)$
  then $g(u)$
  else $k(F(h_1(u))\, F(h_2(u)))$,

cannot be transformed into an equivalent iterative program without introducing considerable intricacy.

Although not every recursive program can be transformed readily into an equivalent iterative program, an iterative program can always be transformed into an equivalent system of recursive programs in a straightforward way. This transformation involves introducing a recursive program corresponding to each label of the given iterative program. For example, if

the iterative program contains a segment of form

L1: **if** $p(x)$
then output$(g(x))$
else $x \leftarrow h(x)$
goto L2,

the corresponding recursive program will be

$L1(x) \Leftarrow$ **if** $p(x)$
then $g(x)$
else $L2(h(x))$.

The idea behind this transformation is that $L1(a)$ denotes the ultimate output of the given iterative program if control passes through label L1 with $x = a$. By this transformation we can replace our symmetric *gcd* algorithm (Program B) by an equivalent system of recursive programs. The original program may be written as

input$(x\ y)$
start:
more: **if** $x = y$ **then output**(y)
reducex: **if** $x > y$ **then** $x \leftarrow x - y$
goto *reducex*
reducey: **if** $y > x$ **then** $y \leftarrow y - x$
goto *reducey*
goto *more*.

The equivalent system of recursive programs is

$start(x\ y) \Leftarrow more(x\ y)$
$more(x\ y) \Leftarrow$ **if** $x = y$ **then** y **else** $reducex(x\ y)$
$reducex(x\ y) \Leftarrow$ **if** $x > y$ **then** $reducex(x - y\ \ y)$
else $reducey(x\ \ y)$
$reducey(x\ y) \Leftarrow$ **if** $y > x$ **then** $reducey(x\ \ y - x)$
else $more(x\ \ y)$.

The output of the system for inputs x and y is the value of $start(x\ y)$. This transformation does not improve the efficiency of the program, but the simplicity of transforming an iterative program into an equivalent recursive program, and the complexity of performing the opposite transformation, substantiates the folklore that recursion is a more powerful programming feature than iteration.

Paterson and Hewitt [69] have studied the theoretical basis for the difficulty of transforming recursive programs into equivalent iterative programs. The reverse transformation, from iterative to recursive programs, is due to McCarthy [63].

Equivalence-preserving transformations have been studied extensively, and some of these have been incorporated into optimizing compilers. The text of Aho and Ullman [1] on compilers contains a chapter on optimization.

Some more ambitious examples of equivalence-preserving program transformations are discussed by Standish *et al.* [76]. An experimental system for performing such transformations was implemented by Darlington and Burstall [21].

The above transformations are all equivalence preserving: for a given input, the transformed program will always produce the same output as the original program. However, we may be satisfied to produce a program that computes a different output from the original, so long as it still terminates and satisfies the same input-output assertions. For example, if we are optimizing a program to compute the square root of a given real number within a tolerance, we will be satisfied if the transformed program produces any output within that range. In the remainder of this section, we will discuss the *correctness-preserving transformations*; such a transformation yields a program that is guaranteed to be correct, but that is not necessarily equivalent to the original program.

Correctness-preserving transformations are applied to programs that have already been proved to be correct; they use information gathered in constructing the proof as an aid in the transformation process. In particular, suppose we have a partial-correctness proof that employs an invariant assertion $invariant(x\ y)$ at some label L, and a well-founded ordering termination proof that employs a well-founded set W and a termination expression $E(x\ y)$ at L. Then we can insert after L any program segment F with the following characteristics:

1) If $invariant(x\ y)$ holds, then the execution of F terminates and $invariant(x\ y)$ is still true afterwards. (Thus, the altered program will still satisfy the original input-output assertions.)

2) If $invariant(x\ y)$ holds, then the value of $E(x\ y)$ in the well-founded set is reduced or held constant by the execution of F. (Therefore, the altered program will still terminate.)

For example, suppose that we have proved the partial correctness of a program by means of the invariant assertion

L: $\{x \geqslant 0$ and $y \geqslant 0$ and $x \cdot y = k\}$

and that we have proved its termination by means of the termination expression

$E(x\ y) = x$,

over the nonnegative integers, at L. Then we may insert the statement

if $even(x)$ **then** $(x\ y) \leftarrow (x/2\ \ 2 \cdot y)$

after L, without destroying the correctness of the program or its termination.

Note that the above transformation does not dictate what segment F is to be inserted, nor does it guarantee that the altered program will be more efficient than the original. Furthermore, even though it preserves the correctness of the transformed program, it may cause it to produce a different output from the original program.

Let us now apply these techniques to transform our subtractive *gcd* algorithm (Program A) into the so-called *binary gcd algorithm*. We reproduce Program A below, introducing a new invariant assertion in the middle of the loop body:

input$(x_0\ y_0)$
$\{x_0 \geqslant 0$ and $y_0 \geqslant 0$ and $(x_0 \neq 0$ or $y_0 \neq 0)\}$
$(x\ y) \leftarrow (x_0\ y_0)$
more: $\{x \geqslant 0$ and $y \geqslant 0$ and $(x \neq 0$ or $y \neq 0)$
and $gcd(x\ y) = gcd(x_0\ y_0)\}$
if $x = 0$ **then goto** *enough*
$\{x > 0$ and $y \geqslant 0$ and $gcd(x\ y) = gcd(x_0\ y_0)\}$
if $y \geqslant x$ **then** $y \leftarrow y - x$ **else** $(x\ y) \leftarrow (y\ x)$
goto *more*
enough: $\{y = gcd(x_0\ y_0)\}$
output(y).

The new assertion

$x > 0$ and $y \geq 0$ and $gcd(x\ y) = gcd(x_0\ y_0)$

is equivalent to our original loop assertion at *more*, and is included because we want to insert new statements at this point. In formulating the invariant assertions for this program, we have used the abbreviated notation $gcd(x\ y)$ in place of the expression $max\{u : u|x \text{ and } u|y\}$.

Recall that to prove the termination of this program by the well-founded ordering method, we used the termination expression $E(x\ y) = (x\ y)$ over the set of all pairs of nonnegative integers, with the lexicographic ordering.

Now, suppose that we know three additional properties of the *gcd*:

a) $gcd(x\ y) = gcd(x/2\ y)$ if x is even and y is odd,
b) $gcd(x\ y) = gcd(x\ y/2)$ if x is odd and y is even,
c) $gcd(x\ y) = 2 \cdot gcd(x/2\ y/2)$ if x and y are both even.

Then we can use these properties and the above correctness-preserving transformation technique to introduce three new statements into the body of the program loop.

Property a) will allow us to divide x by 2 when x is even and y is odd, without changing the value of $gcd(x\ y)$ and, hence, without affecting the truth of the new invariant

$x > 0$ and $y \geq 0$ and $gcd(x\ y) = gcd(x_0\ y_0)$.

and disturb the invariant. To restore the balance, let us generalize all the invariant assertions, replacing

$gcd(x\ y) = gcd(x_0\ y_0)$

by

$z \cdot gcd(x\ y) = gcd(x_0\ y_0)$,

where z is a new program variable. We can then preserve the truth of the invariant by multiplying z by 2 when we divide both x and y by 2. Thus, we introduce the new statement

if *even*(x) and *even*(y) **then** $(x\ y\ z) \leftarrow (x/2\ \ y/2\ \ 2 \cdot z)$.

The altered program will still terminate, because if x and y are even, the termination expression $(x\ y)$ will then be reduced in the lexicographic ordering.

To introduce the new variable z into the intermediate assertions, we must also adjust the initial and final paths of our program. To ensure that the generalized assertion will hold when control first enters the loop, z must be initialized to 1. Furthermore, when control ultimately leaves the loop with $x = 0$, the output returned by the program must be $z \cdot y$ rather than y, because then $z \cdot y = z \cdot gcd(0\ y) = z \cdot gcd(x\ y) = gcd(x_0\ y_0)$. Therefore, we introduce the assignment $y \leftarrow z \cdot y$ into the final path of the program.

Our transformed program is then

```
        input(x0 y0)
        {x0 ≥ 0 and y0 ≥ 0 and (x0 ≠ 0 or y0 ≠ 0)}
        (x y z) ← (x0 y0 1)
more:   {x ≥ 0 and y ≥ 0 and (x ≠ 0 or y ≠ 0)
          and z · gcd(x y) = gcd(x0 y0)}
        if x = 0 then goto enough
        {x > 0 and y ≥ 0 and z · gcd(x y) = gcd(x0 y0)}
        if even(x) and odd(y) then x ← x/2 . . . . . (1)
        if odd(x) and even(y) then y ← y/2 . . . . . (2)
        if even(x) and even(y) then (x y z) ← (x/2 y/2 2 · z) . . (3)
        {x > 0 and y ≥ 0 and z · gcd(x y) = gcd(x0 y0)}
        if y ≥ x then y ← y - x else (x y) ← (y x)
        goto more
enough: y ← z · y
        {y = gcd(x0 y0)}
        output(y).
```

Furthermore, the value of the termination expression $(x\ y)$ is reduced in the lexicographic ordering if x is divided by 2. Similarly, Property b) will allow us to do the same for y if y is even and x is odd. Consequently, we can apply the correctness-preserving transformation to introduce the two new statements

if *even*(x) and *odd*(y) **then** $x \leftarrow x/2$
if *odd*(x) and *even*(y) **then** $y \leftarrow y/2$

after the new invariant.

Property c), on the other hand, cannot be applied so readily, because dividing both x and y by 2 will divide $gcd(x\ y)$ by 2

(The enumeration on the right has been added for future reference.) The correctness-preserving transformation does not ensure that this program will run faster than the original program, but only that is satisfies the same input-output assertions and that it still terminates.

To improve our program further, we introduce another correctness-preserving transformation. If x is even and y is odd, the assignment statement $x \leftarrow x/2$ preserves the truth of the invariant assertion

$x > 0$ and $y \geq 0$ and $gcd(x\ y) = gcd(x_0\ y_0)$

and, so long as $x > 0$, reduces the value of the termination ex-

pression $(x\ y)$. Therefore, if we replace the conditional statement

if *even*(x) and *odd*(y) **then** $x \leftarrow x/2$ (1)

by the **while** statement

while *even*(x) and *odd*(y) and $x > 0$ **do** $x \leftarrow x/2$, (1′)

we have maintained the correctness and termination of the program. The assignment statement will then be applied repeatedly until x is odd.

Similarly, if x is odd, y is even, and $y > 0$, the assignment $y \leftarrow y/2$ will preserve the invariant assertion and reduce the termination expression; therefore, the conditional statement

if *odd*(x) and *even*(y) **then** $y \leftarrow y/2$ (2)

into

if *odd*(x) and $y > 0$ **then while** *even*(y) **do** $y \leftarrow y/2$ (2″)

and the statement

while *even*(x) and *even*(y) and $(x > 0$ or $y > 0)$
do $(x\ y\ z) \leftarrow (x/2\ \ y/2\ \ 2 \cdot z)$ (3′)

into

if $(x > 0$ or $y > 0)$ **then while** *even*(x) and *even*(y)
do $(x\ y\ z) \leftarrow (x/2\ \ y/2\ \ 2 \cdot z)$. (3″)

Because all of these statements preserve the truth of the invariant $x > 0$, the test $x > 0$ can be dropped from statement (1″), and the test $(x > 0$ or $y > 0)$ can be dropped from statement (3″).

The resulting program is then

Program D (the binary algorithm)

```
        input(x0 y0)
        (x y z) ← (x0 y0 1)
more:   if x = 0 then goto enough
        if odd(y) then while even(x) do x ← x/2
        if odd(x) and y > 0 then while even(y) do y ← y/2
        while even(x) and even(y) do (x y z) ← (x/2 y/2 2 · z)
        if y ≥ x then y ← y - x else (x y) ← (y x)
        goto more
enough: y ← z · y
        output(y).
```

---

can be replaced by the **while** statement

while *odd*(x) and *even*(y) and $y > 0$
do $y \leftarrow y/2$. (2′)

In the same way, the conditional statement

if *even*(x) and *even*(y) **then** $(x\ y\ z) \leftarrow (x/2\ \ y/2\ \ 2 \cdot z)$ (3)

can be replaced by the **while** statement

while *even*(x) and *even*(y) and $(x > 0$ or $y > 0)$
do $(x\ y\ z) \leftarrow (x/2\ \ y/2\ \ 2 \cdot z)$. (3′)

The condition "$x > 0$ or $y > 0$" guarantees that the assignment $(x\ y\ z) \leftarrow (x/2\ \ y/2\ \ 2 \cdot z)$ reduces the value of the expression $(x\ y)$ in the lexicographic ordering.

In the **while** statement

while *even*(x) and *odd*(y) and $x > 0$ **do** $x \leftarrow x/2$, (1′)

the truth of the test "*odd*(y) and $x > 0$" cannot be affected by the assignment statement $x \leftarrow x/2$; therefore, using an equivalence-preserving transformation, we can replace the **while** statement by

if *odd*(y) and $x > 0$ **then while** *even*(x) **do** $x \leftarrow x/2$. (1″)

The same transformation can be used to transform

while *odd*(x) and *even*(y) and $y > 0$ **do** $y \leftarrow y/2$ (2′)

Further transformations are still possible. The **while** statement can be removed from the loop, for example, because once one of the program variables is odd, they will never both be even.

Although the transformations we applied are not all guaranteed to produce optimizations, the final algorithm turns out to be significantly faster than the given subtractive algorithm if implemented on a binary machine, where division and multiplication by 2 can be performed quite quickly by shifting words to the right or left.

The binary *gcd* algorithm is based on one discovered by Silver and Terzian (see Knuth [51, pp. 293–338]). An analysis of the running time of this algorithm has been performed by Knuth and refined by Brent [11].

The correctness-preserving transformations we used to produce the binary *gcd* algorithm are in the spirit of Gerhart [34] and Dijkstra [28].

We have presented program transformations as a means of improving the efficiency of a given program. In fact, the existence of such transformations may aid in ensuring the correctness of programs as well. A programmer can safely ignore efficiency considerations for a while, and produce the simplest and clearest program possible for a given task; the program so produced is more likely to be correct, and can be transformed to a more efficient, if less readable, program at a later stage.

Program transformation as a method for achieving more reliable programming has been advocated by Knuth [52] and Burstall and Darlington [15]. The latter authors implemented an interactive system for the transformation of recursive programs. Wegbreit [85] illustrates how a transformation system can be guided by an analysis of the efficiency of the program being transformed, thus ensuring that the program is improved and not merely transformed.

One area for which the application of program transformations has been particularly well explored is the *representation of data structures*: programs written in terms of abstract data structures, such as sets or graphs, are transformed to employ more concrete representations, such as arrays or bit strings, instead. By delaying the choice of representation for the abstract data structure until after the program is written, one can analyze the program to ensure that an efficient representation is chosen. This process is examined, for example, in Earley [30] and Hoare [43]. Experimental implementations have been constructed by Low [55], Schwartz [71], and Guttag *et al.* [40].

VIII. Program Development

In the previous section we discussed logical techniques for transforming one program into another that satisfies the same specifications. In this section we will go one step further and introduce techniques for developing a program from the specifications themselves. These techniques involve generalizing the notion of transformation to apply to specifications as well as to programs. The programs produced in this way will be guaranteed to satisfy the given specifications, and thus will require no separate verification phase.

To illustrate this process we will present the systematic development of a recursive and an iterative program to compute the *gcd* function. From each derivation we will extract some of the principles frequently used in program development. We will then show how these principles can be applied to extend a given program to achieve an additional task. In particular, we will extend one of our *gcd* programs to compute the "least common multiple" (*lcm*) of two integers as well as their *gcd*.

Let us first develop a recursive program for computing the *gcd*. We require that the desired program *gcdgoal*$(x\ y)$ satisfy the output specification

$$gcdgoal(x\ y) = max\{u : u|x \text{ and } u|y\},$$

where x and y are integers satisfying the input specification

$$x \geqslant 0 \text{ and } y \geqslant 0 \text{ and } (x \neq 0 \text{ or } y \neq 0).$$

The set constructor $\{u: \ldots\}$ is admitted to our specification language but is not a primitive of our programming language. We must find a sequence of transformations to produce an equivalent description of the output that does not use the set constructor or any other nonprimitive construct. This description will be the desired primitive program. In what follows we will exhibit a successful sequence of transformations, without indicating how the next transformation at a given stage is selected.

The transformations we employ for this example embody no knowledge of the *gcd* function itself, but some sophisticated knowledge about functions simpler than the *gcd*, such as the following:

For any integers u, v, and w,

a) $u|v \Longrightarrow true$ if $v = 0$
(any integer divides zero),

b) $u|v$ and $u|w \Longrightarrow u|v$ and $u|w - v$
(the common divisors of v and w are the same as those of v and $w - v$),

c) $max\{u : u|v\} \Longrightarrow v$ if $v > 0$
(any positive integer is its own greatest divisor).

In applying these transformations, we will produce a sequence of goals; the first will be derived directly from the output specification, and the last will be the desired program itself. Our initial goal is

Goal 1. Compute $max\{u : u|x \text{ and } u|y\}$,

for any x and y satisfying the input specification. Transformation b) above,

$$u|v \text{ and } u|w \Longrightarrow u|v \text{ and } u|w - v$$

applies directly to a subexpression of Goal 1, yielding

Goal 2. Compute $max\{u : u|x \text{ and } u|y - x\}$.

Note that Goal 2 is an instance of our output specification, Goal 1, but with x and $y - x$ in place of the arguments x and y. This suggests achieving Goal 2 with a recursive call to *gcdgoal*$(x\ y - x)$, because the *gcdgoal* program is intended to satisfy its output specification for any arguments satisfying its input specification.

To see that the input specification is indeed satisfied for the arguments x *and* $y - x$ of the proposed recursive call, we establish a subgoal to prove the *input condition*

Goal 3. Prove $x \geqslant 0$ and $y - x \geqslant 0$ and $(x \neq 0$ or $y - x \neq 0)$.

This input condition is formed from the original input specification by substituting the arguments x and $y - x$ for the given arguments x and y.

Furthermore, we must ensure that the proposed recursive call will terminate. For this purpose, we will use the well-founded ordering method of Section IV; we establish a subgoal to achieve the following *termination condition*

Goal 4. Find a well-founded set W with ordering $\succ$ such that

$$(x\ y) \in W \text{ and } (x\ y - x) \in W$$
$$\text{and } (x\ y) \succ (x\ y - x).$$

Let us consider the input condition (Goal 3) first. Because x has been assumed nonnegative by our original input specification, Goal 3 can be reduced to the two subgoals,

Goal 5. Prove $y \geqslant x$,

and

Goal 6. Prove $(x \neq 0$ or $y \neq x)$.

We cannot prove or disprove Goal 5—it will be true for some inputs and false for others—so we will consider separately the case for which this condition is false, i.e., $y < x$. This case analysis will yield a conditional expression, testing if $y < x$, in the final program.

Case $y < x$: We cannot achieve Goal 5 in this case. In fact, the proposed recursive call does not satisfy its input condition; therefore, we try to find some other way of achieving one of our higher goals.

Using the logical identity

$$\text{P and Q} \Longrightarrow \text{Q and P},$$

we see that Goal 1 is an instance of itself, with x replaced by y and y by x. This suggests achieving Goal 1 with the recursive call *gcdgoal*$(y\ x)$. For this purpose we must establish the input condition

Goal 7. $y \geqslant 0$ and $x \geqslant 0$ and $(y \neq 0$ or $x \neq 0)$

and the termination condition

Goal 8. Find a well-founded set W with ordering $\succ$ such that
$(x\ y) \in W$ and $(y\ x) \in W$
and $(x\ y) \succ (y\ x)$.

Goal 7 is achieved at once; it is a simple reordering of our original input specification. We can achieve Goal 8 by taking W to be the set of pairs of nonnegative integers, because x and y are known to be nonnegative by our input specification. In this case $y < x$, so we take our well-founded ordering $\succ$ to be the usual $>$ ordering applied to the first components of the pairs. (In other words, $(u_1\ u_2) \succ (v_1\ v_2)$ if $u_1 > v_1$.) Having established the input condition and the termination condition, we are justified in returning the recursive call *gcdgoal*$(y\ x)$. Thus, the partial program completed at this stage is

$$\begin{aligned} \textit{gcdgoal}(x\ y) \Longleftarrow\ & \textbf{if } y < x \\ & \textbf{then } \textit{gcdgoal}(y\ x) \\ & \textbf{else } \ldots . \end{aligned}$$

It remains to consider the alternate branch of the case analysis, in which $y \geqslant x$. This case corresponds to the **else** branch of the final program.

Case $y \geqslant x$: Here, we have established Goal 5, a subgoal of the input condition for the proposed recursive call *gcdgoal*$(x\ y - x)$. It remains to prove the other subgoal of the input condition, Goal 6, that $x \neq 0$ or $y \neq x$. Again, we cannot prove or disprove either disjunct of this goal because they each will be true for some inputs and false for others. Thus, we can make either $x \neq 0$ or $y \neq x$ a basis for a case analysis; we choose the former disjunct and consider the case in which $x \neq 0$ is false.

Case $x = 0$: We cannot achieve Goal 6 here, so we are prevented from introducing the recursive call *gcdgoal*$(x\ y - x)$. We therefore again attempt to apply alternate transformations to the higher level goals. Because in this case $x = 0$, Transformation a),

$$u \mid v \Longrightarrow \textit{true} \quad \text{if } v = 0$$

applies to the subexpression $u \mid x$ of Goal 1, yielding

Goal 9. Compute $\textit{max}\{u : \textit{true}$ and $u \mid y\}$.

Applying the logical transformation

$$\textit{true} \text{ and P} \Longrightarrow \text{P}$$

produces

Goal 10. Compute $\textit{max}\{u : u \mid y\}$.

Because $y \geqslant 0$ and $(x \neq 0$ or $y \neq 0)$, by our original input specification, and $x \neq 0$, by our case condition, we know that $y > 0$ at this point; therefore, we can apply Transformation c)

$$\textit{max}\{u : u \mid v\} \Longrightarrow v \quad \text{if } v > 0$$

yielding

Goal 11. Compute y.

We have thus reduced the goal in this case to the task of computing y, which involves no nonprimitive constructs. The desired program may simply output y. The partial program we have constructed so far is

$$\begin{aligned} \textit{gcdgoal}(x\ y) \Longleftarrow\ & \textbf{if } y < x \\ & \textbf{then } \textit{gcdgoal}(y\ x) \\ & \textbf{else if } x = 0 \\ & \quad \textbf{then } y \\ & \quad \textbf{else } \ldots . \end{aligned}$$

Finally, we consider the remaining branch in our case analysis.

Case $x \neq 0$: Here, the input condition (Goal 3) for our proposed recursive call *gcdgoal*$(x\ \ y - x)$ is satisfied; it remains, therefore, to consider the termination condition (Goal 4):

Find a well-founded set W with ordering $\succ$ such that

$(x\ \ y) \in W$ and $(x\ \ y - x) \in W$
and $(x\ \ y) \succ (x\ \ y - x)$.

For the previous recursive call, *gcdgoal*$(y\ x)$, we have taken W to be the set of pairs of nonnegative integers, and $\succ$ to be the usual $>$ relation on the first components of the pairs. To ensure the termination of the final program, it is necessary that W and $\succ$ be the same for both recursive calls. Unfortunately, the first argument of the proposed recursive call *gcdgoal*$(x\ y - x)$ is x itself, and it is not so that $(x\ y) \succ (x\ y - x)$ in the ordering $\succ$ we have employed. We therefore attempt to alter $\succ$ to establish the termination conditions of both recursive calls *gcdgoal*$(y\ x)$ and *gcdgoal*$(x\ y - x)$.

Because in this case it is known that $x > 0$ (i.e., $x \neq 0$ and $x \geqslant 0$), we have that $y > y - x$. We therefore extend the ordering to examine the second components if it happens that the first components are equal; in other words, we revise $\succ$ to be the lexicographic ordering on the pairs of nonnegative integers. With the new ordering $\succ$, both recursive calls can be shown to terminate. We have thereby established Goal 4, and the program can output *gcdgoal*$(x\ y - x)$ in this case.

Our final program is

$gcdgoal(x\ y) \Leftarrow$ **if** $y < x$
then $gcdgoal(y\ x)$
else if $x = 0$
then y
else $gcdgoal(x\ y - x)$.

This program is similar to our subtractive *gcd* algorithm (Program A), but its tests are performed in the reverse order.

Note that in performing the above derivation, we have ensured that the derived program terminates and satisfies the given specifications; thus, we have proved the total correctness of the program in the course of its construction.

From the above example, we may extract some of the basic principles that are frequently used in program development.

1) *Transformation Rules:* The program is developed by applying successive transformation rules to the given specifications. The rules preserve the meaning of the specifications, but try to replace the nonprimitive contructs of the specification language by primitive constructs of the programming language.

2) *Conditional Introduction:* Some transformation rules require that certain conditions be true before the rules can be applied. When a transformation requires a condition that we cannot prove or disprove, we introduce a case analysis based on that condition, yielding a conditional expression in the ultimate program.

3) *Recursion Introduction:* When a subgoal is an instance of the top goal (or any higher level subgoal), a recursive call can be introduced, provided that the input specification of the desired program is satisfied by the new arguments, and the termination of the recursion can be guaranteed.

The above example illustrated the construction of a recursive program from given specifications. If we wish to construct an iterative program instead, alternate techniques are necessary. In our next example we will illustrate some of these techniques.

In constructing the recursive program we did not allow ourselves to use any of the properties we know about the *gcd* function itself, but only the properties of subsidiary functions such as division and subtraction. In constructing the iterative program, however, we facilitate the process by admitting the use of several properties of the *gcd* function itself:

For any integers u and v

a) $gcd(u\ v) = v$ if $u = 0$ and $v > 0$
b) $gcd(u\ v) = gcd(rem(v\ u)\ u)$ if $u > 0$ and $v \geq 0$,

where $rem(v\ u)$ is the remainder of dividing v by u. We further simplify the task by assuming the stronger input assertion

$x_0 > 0$ and $y_0 > 0$.

We write our goal directly in terms of the *gcd* function

Goal 1. **input**$(x_0\ y_0)$
$\{x_0 > 0$ and $y_0 > 0\}$
achieve $z = gcd(x_0\ y_0)$
$\{z = gcd(x_0\ y_0)\}$
output(z).

Here, to **achieve** a relation means to construct a program segment assigning values to the program variables so that the relation holds. Note that we have annotated the goal with the program's input and output assertions.

It is understood that "*gcd*" is part of the assertion language but not a primitive construct of our programming language, so it does not suffice merely to set z to be $gcd(x_0\ y_0)$; we are forced to rephrase our goal in terms of the primitive constructs.

Because x_0 and y_0 are input values, which we will want to refer to later, we introduce new program variables x and y whose values can be manipulated. Consequently, the above goal is replaced by the equivalent subgoal

Goal 2. **input**$(x_0\ y_0)$
$\{x_0 > 0$ and $y_0 > 0\}$
achieve $z = gcd(x\ y)$ and $gcd(x\ y) = gcd(x_0\ y_0)$
$\{z = gcd(x_0\ y_0)\}$
output(z).

Using Property a), that

$gcd(u\ v) = v$ if $u = 0$ and $v > 0$,

we can reduce Goal 2 to the following goal,

Goal 3. **input**$(x_0\ y_0)$
$\{x_0 > 0$ and $y_0 > 0\}$
achieve $z = y$ and $gcd(x\ y) = gcd(x_0\ y_0)$
and $x = 0$ and $y > 0$
$\{z = gcd(x_0\ y_0)\}$
output(z).

We can now achieve $z = y$ by setting z to be y before exiting from the program. We choose to achieve the remaining conjunction by introducing a loop whose exit test is $x = 0$, and whose invariant assertion is $gcd(x\ y) = gcd(x_0\ y_0)$ and $y > 0$. (To be certain that $gcd(x\ y)$ is defined, we must add the invariant $x \geq 0$, as well.) On exiting from such a loop, we can be sure that all the conjuncts are satisfied. The desired program will be of the form

Goal 4. **input**$(x_0\ y_0)$
$\{x_0 > 0$ and $y_0 > 0\}$
achieve $gcd(x\ y) = gcd(x_0\ y_0)$ and $x \geq 0$ and $y > 0$
more: $\{gcd(x\ y) = gcd(x_0\ y_0)$ and $x \geq 0$ and $y > 0\}$
if $x = 0$ **then goto** *enough*
achieve $gcd(x\ y) = gcd(x_0\ y_0)$ and $x \geq 0$ and $y > 0$
while guaranteeing termination
goto *more*
enough: $z \leftarrow y$
$\{z = gcd(x_0\ y_0)\}$
output(z).

The variables x and y can be initialized to satisfy the invariant assertion easily enough by setting x to x_0 and y to y_0. In constructing the loop body, we must ensure not only that the invariant is maintained, but also that the values of the program variables x and y are altered so that the program will ultimately terminate, i.e., so that eventually $x = 0$. For this purpose, we require that x be strictly reduced with each iteration.

To reduce x while maintaining the invariant assertion, we use the above Property b) of the *gcd* function, that

$$gcd(u\ v) = gcd(rem(v\ u)\ u) \quad \text{if } u > 0 \text{ and } v \geqslant 0,$$

and an additional property of the remainder function, that

$$0 \leqslant rem(v\ u) < u \quad \text{if } u > 0 \text{ and } v \geqslant 0.$$

Because we know that x and y are positive (by the exit test and the invariant assertion), we can achieve the requirements for the loop body by updating x and y to be $rem(y\ x)$ and x, respectively. The final program, complete with its annotations, is

```
          input(x0 y0)
          {x0 > 0 and y0 > 0}
          (x y) ← (x0 y0)
more:     {gcd(x y) = gcd(x0 y0) and x ⩾ 0 and y > 0}
          if x = 0 then goto enough
          (x y) ← (rem(y x) x)
          goto more
enough:   z ← y
          {z = gcd(x0 y0)}
          output(z).
```

This is an iterative version of the Euclidean *gcd* algorithm (Program C).

The above example allows us to extract some additional principles of program development:

Variable Introduction: Introduce program variables that can be manipulated in place of input values, and rewrite the goal in terms of the program variables.

Iteration Introduction: If a goal is expressed as a conjunction of several conditions, attempt to introduce an iterative loop whose exit test is one of the conditions and whose invariant assertion is the conjunction of the others.

There are many other program development techniques besides those encountered in the two examples above. Some of these are listed here:

1) *Generalization:* We have observed earlier that in proving a theorem by mathematical induction, it is sometimes necessary to strengthen the theorem, so that a stronger induction hypothesis can be used in the proof. By the same token, in deriving a recursive program it is sometimes necessary to generalize the program's specifications, so that a recursive call to the program will satisfy a desired subgoal. Thus, in constructing a program to sort an array with elements $A_0, A_1, \cdots, A_n$, we may be led to construct a more general program to sort an arbitrary segment $A_i, A_{i+1}, \cdots, A_j$. Similarly, in constructing an iterative program we may need to generalize a proposed invariant assertion, much as we were forced to generalize the invariant assertion $gcd(x\ y) = gcd(x_0\ y_0)$ to be $z \cdot gcd(x\ y) = gcd(x_0\ y_0)$ in developing the binary *gcd* algorithm (Program D) in Section VII.

2) *Simultaneous Goals:* Often we need to construct a program whose specifications involve achieving a conjunction of two or more interdependent conditions at the same time. The difficulty is that in the course of achieving the second condition we may undo the effects of achieving the first, and so on. One approach to this problem is to construct a program to achieve the first condition, and then *extend* that program to achieve the second condition as well; in modifying the program we must *protect* the first condition so that it will still be achieved by the altered program. For instance, a program to sort the values of three variables x, y, and z must permute their values to achieve the output specification "$x \leqslant y$ and $y \leqslant z$." To construct such a program, we may first construct a program to achieve $x \leqslant y$ and then extend that program to achieve $y \leqslant z$ as well, while protecting $x \leqslant y$.

3) *Efficiency:* To ensure that the program we construct will be efficient, we must be able to decide between alternate means of achieving a given subgoal. We must consider the effects of the chosen transformations on the time and space requirements of the ultimate program. For example, in constructing a *gcd* program, if we were given a variety of transformations based on different properties of the *gcd* function, we might need to decide between achieving the subgoal "compute $max\{u : u|x \text{ and } u|y - x\}$" and the subgoal "compute $max\{u : u|x \text{ and } u|(y/2)\}$."

The synthesis of the iterative Euclidean algorithm above follows Dershowitz and Manna [24]. A discussion of generalization in program synthesis is found in Siklossy [73]. An approach to the simultaneous goal problem appears in Waldinger [82].

The systematic development of programs has been regarded from two points of view: as a discipline to be adhered to by human programmers in order to construct correct and transparent programs, and as a method by which programs can be generated automatically by computer systems. The first aspect, referred to as *structured programming* (see, for example, Dahl *et al.* [19], Wirth [86], and Dijkstra [28]), has been advocated as a practical method for achieving reliability in large computer programs. The second aspect of program development, called *program synthesis*, is currently being pursued as a research activity (e.g., see Buchanan and Luckham [12], Manna and Waldinger [60], and Darlington [20]).

Although the techniques of structured programming are sufficiently well-specified to serve as a guide to the human programmer, much needs to be done before human performance can be imitated by an automatic system. For instance, at each point in the development of a program, a synthesis system must decide what portion of the specifications will be the next to be transformed and select an appropriate transformation from many plausible candidates. In introducing a loop or recursive call it may need to find a suitable generalization of the goal or the proposed invariant assertion. Furthermore, a synthesis system must have access to knowledge of the properties of the operations involved in the program being constructed and be able to use this knowledge to reason about the program. To some extent these problems are shared by verification systems, but the synthesis task is more difficult than verification, because it receives less help from the human programmer and demands more from the computer system. Consequently, automatic program synthesis is still in an experimental stage of development, and does not seem likely to be applied to practical programming problems in the near future.

In the examples of program development we have seen so far, we have used the given specification as a basis for constructing a completely new program. We have introduced no mechanisms for taking advantage of work we may have done previously in solving some related problem. This situation conflicts sharply with ordinary programming practice, where we are often altering or extending old programs to suit new purposes. In our next example we will assume that we are given a program with its original specifications plus some additional specifications; we will *extend* the program to satisfy the new specifications as well as the original ones. Thus, although we may add new statements or change old ones in the existing program to achieve the new goal, we will always be careful that the program still achieves the purpose for which it was originally intended.

We suppose we are given a program to compute the *gcd* of two positive integers, and we want to extend it to compute their *least common multiple* as well. The least common multiple of x and y, or $lcm(x\ y)$, is defined to be the smallest positive integer that is a multiple of both x and y; for example, $lcm(12\ 18) = 36$. Now, of course we could construct a completely separate program to compute $lcm(x\ y)$, but in fact the *gcd* and the *lcm* are closely related by the identity

a) $gcd(x\ y) \cdot lcm(x\ y) = x \cdot y$.

(For example, $gcd(12\ 18) \cdot lcm(12\ 18) = 6 \cdot 36 = 216 = 12 \cdot 18$.) We would like to take advantage of the work being done in the *gcd* program by adding new statements that will enable it to compute the *lcm* at the same time.

Suppose the given *gcd* program, annotated with its assertions, is as follows:

```
          input(x0 y0)
          {x0 > 0 and y0 > 0}
          (x y) ← (x0 y0)
more:     {gcd(x y) = gcd(x0 y0) and x ≥ 0 and y > 0}
          if x = 0 then goto enough
          if y > x then y ← y - x else x ← x - y
          goto more
enough:   {y = gcd(x0 y0)}
          output(y).
```

This is a version of our subtractive algorithm (Program A) for computing the *gcd* of positive integers only.

The extension task is to achieve the additional output assertion

$$x' = lcm(x_0\ y_0)$$

as well as the original output assertion

$$y = gcd(x_0\ y_0).$$

In the light of the identity (a) relating the *gcd* and the *lcm*, the most straightforward way to achieve this new assertion is to assign

$$x' \leftarrow (x_0 \cdot y_0)/y$$

at the end of Program A. However, Program A itself computes the *gcd* without using multiplication or division; let us see if we can extend the program to compute the *lcm* using only addition and subtraction.

One approach to program extension reflects a technique we already used in developing a new program: we try to find an additional intermediate assertion for the program, usually involving new variables, that will imply the new output assertion when the program halts. We then alter the program by initializing the new variables so that the additional intermediate assertion will be satisfied the first time we enter the loop, and by updating these variables in the loop body so that the assertion will be maintained as an invariant every time we travel around the loop. As in proving the correctness of a program, the choice of suitable intermediate assertion may require some ingenuity.

For instance, it would suffice if we could extend the program by introducing the relation

$$x' \cdot y = x_0 \cdot y_0$$

as a new intermediate assertion in addition to our original assertion

$$gcd(x\ y) = gcd(x_0\ y_0) \text{ and } x \geq 0 \text{ and } y > 0.$$

This relation implies the new output assertion, because when the program halts, y will be $gcd(x_0\ y_0)$, and therefore x' will be $lcm(x_0\ y_0)$. If we initialize x' to be x_0, this relation will be satisfied the first time we enter the loop, because y is initialized to y_0. However, we still need to update the value of x' as we travel around the loop so that the relation is maintained; this turns out to be a very difficult task.

A successful new intermediate assertion is the much less obvious choice

b) $x' \cdot y + x \cdot y' = x_0 \cdot y_0$

where x' and y' are both new variables. This relation does imply the output assertion, because $x = 0$ and $y = gcd(x_0\ y_0)$ when the program halts. Furthermore, because y is initialized to y_0, we can ensure that the relation will be true the first time we enter the loop by initially assigning

$$(x'\ y') \leftarrow (x_0\ 0).$$

Finally, we can maintain the relation when control passes around the loop: Considering the case in which $y > x$, let us rewrite Relation b) as

$$x' \cdot ((y - x) + x) + x \cdot y' = x_0 \cdot y_0.$$

After y is reset to $y - x$, a new relation holds:

$$x' \cdot (y + x) + x \cdot y' = x_0 \cdot y_0,$$

i.e.,

$$x' \cdot y + x \cdot (y' + x') = x_0 \cdot y_0.$$

Hence, to restore our intended invariant assertion, it is only necessary to assign

$$y' \leftarrow y' + x'$$

in this branch of the loop body.

In considering the other branch, for which $y \leq x$, we merely

reverse the roles of x and y, and of x' and y'; thus, we can restore our intended invariant by assigning

$$x' \leftarrow x' + y'$$

in this case.

It is clear that the changes we have introduced do not affect the truth of the program's original assertions, because we have only altered the values of the new variables x' and y', which do not appear in those assertions. The complete program, which computes both the *gcd* and the *lcm* at the same time, is

Program E (the extended algorithm):

input$(x_0\ y_0)$
$\{x_0 > 0$ and $y_0 > 0\}$
$(x\ y\ x'\ y') \leftarrow (x_0\ y_0\ x_0\ 0)$
more: $\{gcd(x\ y) = gcd(x_0\ y_0)$ and $x \geqslant 0$ and $y > 0$
and $x' \cdot y + x \cdot y' = x_0 \cdot y_0\}$
if $x = 0$ **then goto** *enough*
if $y > x$ **then** $(y\ y') \leftarrow (y - x\ y' + x')$
else $(x\ x') \leftarrow (x - y\ x' + y')$
goto *more*
enough: $\{y = gcd(x_0\ y_0)$ and $x' = lcm(x_0\ y_0)\}$
output$(y\ x')$.

This program computes the *lcm* as a byproduct of computing the *gcd*, using only the addition operation. Given the intermediate assertion b), it is purely mechanical to extend Program A to Program E. Choosing a successful intermediate assertion, however, is still a mysterious process.

In the above example, we were careful that the program being extended still achieved its original purpose, computing the *gcd* of its arguments. It sometimes happens that we need to *adapt* a program to perform a new but analogous task. For example, a program that computes the square root of a number by the method of "successive approximations" might be adapted to compute the quotient of two numbers by the same method. In adapting a program we want to maintain as much as possible of its original structure, but we change as much as necessary of its details to ensure that the altered program will satisfy the new specifications. If we have proved the correctness of the original program, it is possible that we may also be able to adapt the proof in the same way to show the correctness of the new program. *Program debugging* may be considered as a special case of adaptation, in which we alter an incorrect program to conform with its intended specifications.

Program adaptation has been studied by Ulrich and Moll [80], and an experimental program adaptation system has been produced by Dershowitz and Manna [25]. Automatic debugging has been discussed by von Henke and Luckham [81] and by Katz and Manna [47].

In this section, we have discussed logical techniques for program development from given input-output specifications. Other approaches to the construction of programs, under the general rubric of *automatic programming*, have used more informal methods of program specification and less systematic techniques for program development; a survey of the entire field of automatic programming is provided by Biermann [6]. Alternate approaches to automatic programming include the following:

1) Giving typical pairs of inputs and outputs; e.g., (A (B C) D) $\Longrightarrow$ (D (B C) A) suggests a program to reverse a list. A system that accepts such specifications must be able to generalize from examples (e.g., see Hardy [41] and Summers [77]). Sample input-output pairs are natural and easy to formulate, but they may yield ambiguities, even if several pairs are given.

2) Giving typical traces of the execution of the algorithm to be encoded; e.g., the trace (12 18) $\longrightarrow$ (6 12) $\longrightarrow$ (0 6) $\longrightarrow$ 6 suggests that the Euclidean *gcd* algorithm is to be constructed (see Biermann and Krishnaswamy [7]). To formulate such a specification, we must have a particular algorithm in mind.

3) Engaging in a natural-language dialogue with the system. For instance, in specifying an operating system or airline reservation system, we are unlikely to formulate a complete and correct description all at once. In the course of an extended dialogue, we may resolve inconsistencies and clarify details (see Balzer [5], Green [38]). The use of natural language avoids the necessity to communicate through an artificial formalism, but requires the existence of a system capable of understanding such dialogue.

4) Constructing a program that "almost" achieves the specifications, but is not completely correct, and then debugging it (see Sussman [78]). This technique is similar to the way human programmers proceed and is particularly appropriate in conjunction with the natural-dialogue approach, in which the specifications themselves are likely to be incorrect at first.

Acknowledgment

We would like to thank J. Derksen, N. Dershowitz, C. Ellis, J. Guttag, J. King, D. Knuth, J. van Leeuwen, D. Oppen, and A. Pnueli for discussions helpful in preparing this paper. We are also grateful to the following members of the Theory of Computing Panel of the NSF Computer Science and Engineering Research Study (COSERS)—R. Karp, A. Meyer, J. Reynolds, R. Ritchie, J. Ullman, and S. Winograd—for their critical comments on parts of the manuscript.

In our writing we have drawn on the wealth of material about the greatest common divisor and the algorithms that compute it included in Knuth [51]. Information about catastrophic bugs in spacecraft guidance systems was provided by members of the staff of the Jet Propulsion Laboratory, Pasadena, CA.

References

Many recent introductory programming texts touch upon the topics we have discussed here; furthermore, there are several textbooks that are devoted exclusively to these issues. Manna [58], Greibach [39], and Bird [8] all give a fairly theoretical treatment of the correctness and termination of programs. Dijkstra [28] emphasizes the development and optimization of programs, in his own inimitable style.

[1] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vol. 2: Compiling*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[2] E. A. Ashcroft, "Proving assertions about parallel programs," *J. Comput. Syst. Sci*, vol. 10, no. 1, pp. 110-135, Feb. 1975.
[3] E. A. Ashcroft, M. Clint, and C. A. R. Hoare, "Remarks on 'Program proving: Jumps and functions by M. Clint and C. A. R. Hoare'," *Acta Informatica*, vol. 6, pp. 317-318, 1976.
[4] E. A. Ashcroft and W. Wadge, "Lucid, a nonprocedural language with iteration," *Commun. Assoc. Comput. Mach.*, vol. 20, no. 7, pp. 519-526, July 1977.
[5] R. M. Balzer, "Automatic programming," Information Science Institute, University of Southern California, Marina del Rey, CA, Tech. Rep., Sept. 1972.
[6] A. W. Biermann, "Approaches to automatic programming," in *Advances in Computers*, vol. 15. New York: Academic Press, 1976, pp. 1-63.
[7] A. W. Biermann and R. Krishnaswamy, "Constructing programs from example computations," *IEEE Trans. Software Eng.*, vol. 2, pp. 141-153, Sept. 1976.
[8] R. Bird, *Programs and Machines–An Introduction to the Theory of Computation*. London, England: Wiley, 1976.
[9] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT–A formal system for testing and debugging programs by symbolic execution," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, pp. 234-245, Apr. 1975.
[10] R. S. Boyer and J S. Moore, "Proving theorems about LISP functions," *J. Assoc. Comput. Mach.*, vol. 22, pp. 129-144, Jan. 1975.
[11] R. P. Brent, "Analysis of the binary Euclidean algorithm," in *New Directions and Recent Results in Algorithms and Complexity*, J. F. Traub, Ed. New York: Academic Press, 1976.
[12] J. R. Buchanan and D. C. Luckham, "On automating the construction of programs," Artificial Intelligence Lab., Stanford Univ., Stanford, CA, Tech. Rep. May 1974.
[13] R. M. Burstall, "Proving properties of programs by structural induction," *Comput. J.*, vol. 12, no. 1, pp. 41-48, Feb. 1969.
[14] —, "Program proving as hand simulation with a little induction," in *1974 Proc. IFIP Congr*. Amsterdam, The Netherlands: North-Holland, pp. 308-312,
[15] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *J. Assoc. Comput. Mach.*, vol. 24, pp. 44-67, Jan. 1977.
[16] E. M. Clarke, Jr. "Programming language constructs for which it is impossible to obtain good Hoare-like axiom systems," in *Proc. 4th Symp. Principles of Programming Languages*, Los Angeles, CA, pp. 10-20, Jan. 1977.
[17] M. Clint and C. A. R. Hoare, "Program proving: jumps and functions," *Acta Informatica*, vol. 1, pp. 214-224, 1972.
[18] S. A. Cook, "Soundness and completeness of an axiom system for program verification," University of Toronto, Toronto, Canada, Tech. Rep., June 1976.
[19] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. New York: Academic Press, 1972.
[20] J. Darlington, "Applications of program transformation to program synthesis," in *Proc. IRIA Symp. Proving and Improving Programs*, Arc-et-Senans, France, pp. 133-144, July 1975.
[21] J. Darlington and R. M. Burstall, "A system which automatically improves programs," *Acta Informatica*, vol. 6, no. 1, pp. 41-60, 1976.
[22] J. W. deBakker and D. Scott, "A theory of programs," IBM Seminar, Vienna, Austria, unpublished notes, Aug. 1969.
[23] R. A. DeMillo, R. J. Lipton, and A. J. Perlis, "Social processes and proofs of theorems and programs," in *Proc. 4th Symp. Principles of Programming Languages*, Los Angeles, CA, pp. 206-214, Jan. 1977.
[24] N. Dershowitz and Z. Manna, "On automating structured programming," in *Proc. IRIA Symp. Proving and Improving Programs*, Arc-et-Senans, France, July 1975, pp. 167-193.
[25] N. Dershowitz and Z. Manna, "The evolution of programs: A system for automatic program modification," *IEEE Trans. Software Eng.*, vol. 3, pp. 377-385, Nov. 1977.
[26] L. P. Deutsch, "An interactive program verifier," Ph.D. dissertation, University of California, Berkeley, CA, May 1973.
[27] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. Assoc. Comput. Mach.*, vol. 18, pp. 453-457, Aug. 1975.
[28] —, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976.
[29] —, "Programming: From craft to scientific discipline," Int. Computing Symposium, E. Morlet and D. Ribbens, Ed. Amsterdam, The Netherlands: North-Holland, 1977, pp. 23-30.
[30] J. Earley, "Toward an understanding of data structures," *Commun. Assoc. Comput. Mach.*, vol. 14, no. 10, pp. 617-627, Oct. 1971.
[31] B. Elspas, K. N. Levitt, and R. J. Waldinger, "An interactive system for the verification of computer programs," Stanford Research Institute, Menlo Park, CA, Tech. Rep., Sept. 1973.
[32] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Applied Math.*, vol. 19, J. T. Schwartz, Ed. Providence, RI: American Mathematical Society, 1967, pp. 19-32.
[33] N. Francez and A. Pnueli, "A proof method for cyclic programs," Dep. Comput. Sci., Tel Aviv Univ., Tel Aviv, Israel, Tech. Rep., Nov. 1975.
[34] S. L. Gerhart, "Correctness-preserving program transformations," in *Proc. 2nd Symp. on Principles of Programming Languages*, Palo Alto, CA, pp. 54-66, Jan. 1975.
[35] S. L. Gerhart and L. Yelowitz, "Observations of fallibility in applications of modern programming methodologies," *IEEE Trans. Software Eng.*, vol. 2, pp. 195-207, Sept. 1976.
[36] S. M. German and B. Wegbreit, "A synthesizer of inductive assertions," *IEEE Trans. Software Eng.*, vol. 1, pp. 68-75, Mar. 1975.
[37] D. I. Good, R. L. London, and W. W. Bledsoe, "An interactive program verification system," *IEEE Trans. Software Eng.*, vol. 1, pp. 59-67, Mar. 1975.
[38] C. Green, "The design of PSI program synthesis system," in *Proc. 2nd Int. Conf. Software Eng.*, San Francisco, CA, pp. 4-18, Oct. 1976.
[39] S. A. Greibach, *Theory of Program Structures: Schemes, Semantics, Verification*. Berlin, Germany: Springer-Verlag, 1975.
[40] J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract data types and software validation," Tech. Rep., Information Sciences Institute, Marina del Rey, CA, Aug. 1977.
[41] S. Hardy, "Synthesis of LISP programs from examples," in *Proc. 4th Int. Joint Conf. on Artificial Intelligence*, Tbilisi, Georgia, USSR, pp. 240-245, Sept. 1975.
[42] C. A. R. Hoare, "An axiomatic basis of computer programming," *Commun. Assoc. Comput. Mach.*, vol. 12, pp. 576-580, 583, Oct. 1969.
[43] —, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, no. 4, pp. 271-281, 1972.
[44] —, "Parallel programming: An axiomatic approach," *Comput. Languages*, vol. 1, no. 2, pp. 151-160, June 1975.
[45] T. E. Hull, W. H. Enright, and A. E. Sedgwick, "The correctness of numerical algorithms," in *Proc. Conf. Proving Assertions about Programs*, Las Cruces, NM, pp. 66-73, Jan. 1972.
[46] S. Igarashi, R. L. London, and D. C. Luckham, "Automatic program verification 1: A logical basis and its implementation," *Acta Informatica*, vol. 4, no. 2, pp. 145-182, May 1975.
[47] S. M. Katz and Z. Manna, "Logical analysis of programs," *Commun. Assoc. Comput. Mach.*, vol. 19, no. 4, pp. 188-206, Apr. 1976.
[48] J. C. King, "A program verifier," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1969.
[49] —, "Symbolic execution and program testing," *Commun. Assoc. Comput. Mach.*, vol. 19, no. 7, pp. 385-391, July 1976.
[50] D. E. Knuth, *The Art of Computer Programming, Volume 1*. Reading, MA: Addison-Wesley, 1968.
[51] —, *The Art of Computer Programming, Volume 2*. Reading, MA: Addison-Wesley, 1969.
[52] —, Structured programming with go to statements, *Comput. Surveys*, vol. 6, no. 4, pp. 261-301, Dec. 1974.
[53] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek, "Report on the programming language EUCLID," *SIGPLAN Notices*, vol. 12, no. 2, Feb. 1976.
[54] B. H. Liskov, "An introduction to CLU," in *New Directions in Algorithmic Languages*, S. A. Schuman, Ed. Paris, France: Institut de Recherche D'Informatique et d'Automatique, 1976, pp. 139-156.
[55] J. R. Low, *Automatic Coding: Choice of Data Structures*. Basle, Switzerland: Birkhauser Verlag, 1976.
[56] D. C. Luckham and N. Suzuki, "Proof of termination within a weak logic of programs," *Acta Informatica*, vol. 8, no. 1, pp. 21-36, 1977.
[57] Z. Manna, "Mathematical theory of partial correctness," *J. Comput. Syst. Sci*, vol. 5, no. 3, pp. 239-253, June 1971.
[58] —, *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.

[59] Z. Manna and A. Pnueli, "Formalization of properties of functional programs," *J. Assoc. Comput. Mach.*, vol. 17, no. 3, pp. 555-569, July 1970.
[60] Z. Manna and R. Waldinger, "Knowledge and reasoning in program synthesis," *Artificial Intelligence*, vol. 6, no. 2, pp. 175-208, Summer, 1975.
[61] —, "Is 'sometime' sometimes better than 'always'? Intermittent assertions in proving program correctness," *Commun. Assoc. Comput. Mach.*, vol. 21, no. 2, pp. 159-172, Feb. 1978.
[62] J. McCarthy, "Towards a mathematical science of computation," in *Proc. of IFIP Congress 1962*, C. M. Popplewell, Ed. Amsterdam, The Netherlands: North-Holland, pp. 21-28, 1962.
[63] —, "A basis for a mathematical theory of computation," in *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, Ed. Amsterdam, The Netherlands: North-Holland, 1963, pp. 33-70.
[64] J. H. Morris, "Another recursion induction principle," *Commun. Assoc. Comput. Mach.*, vol. 14, no. 5, pp. 351-354, May 1971.
[65] J. H. Morris and B. Wegbreit, "Subgoal induction," *Commun. Assoc. Comput. Mach.*, vol. 20, no. 4, pp. 209-222, Apr. 1977.
[66] D. C. Oppen and S. A. Cook, "Proving assertions about programs that manipulate data structures," in *Proc. 7th Annu. Symp. Theory of Computing*, Albuquerque, NM, pp. 107-116, May 1975.
[67] S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Commun. Assoc. Comput. Mach.*, vol. 19, no. 5, pp. 279-285, May 1976.
[68] D. L. Parnas, "A technique for software module specification with examples," *Commun. Assoc. Comput. Mach.*, vol. 15, no. 5, pp. 330-336, May 1972.
[69] M. S. Paterson and C. E. Hewitt, "Comparative schematology," in *Rec. Proj. MAC Conf. Concurrent Systems and Parallel Computation*, Association for Computing Machinery, NY, Dec. 1970, pp. 119-228.
[70] V. Pratt, "Semantical considerations on Floyd-Hoare Logic," in *Proc. 17th Annu. Symp. Foundations of Computer Science*, Houston, TX, pp. 109-121, Oct. 1976.
[71] J. T. Schwartz, "Automatic and semiautomatic optimization of SETL," in *Proc. Symp. on Very High Level Languages*, Santa Monica, CA, pp. 43-49, Mar. 1974.
[72] J. Schwarz, "Event-based reasoning—a system for proving correct termination of programs," in *Proc. 3rd Int. Colloqu. Automata, Languages and Programming*, Edinburgh, Scotland, pp. 131-146, July 1976.
[73] L. Siklossy, "The synthesis of programs from their properties, and the insane heuristic," in *Proc. 3rd Texas Conf. Computing Systems*, Austin, TX, 1974.
[74] R. L. Sites, "Proving that computer programs terminate cleanly," Ph.D. dissertation, Stanford Univ., Stanford, CA, May 1974.
[75] J. Spitzen, K. N. Levitt, and L. Robinson, "An example of hierarchical design and proof," *Commun. Assoc. Comput. Mach.*, to be published.
[76] T. A. Standish, D. C. Harriman, D. F. Kibler, and J. M. Neighbors, "Improving and refining programs by program manipulation," Tech. Rep., University of California, Irvine, CA, Feb. 1976.
[77] P. D. Summers, "A methodology for LISP program construction from examples," *J. Assoc. Comput. Mach.*, vol. 24, no. 1, pp. 161-175, Jan. 1977.
[78] G. J. Sussman, *A Computer Model of Skill Acquisition*. New York: American Elsevier, 1975.
[79] N. Suzuki, "Verifying programs by algebraic and logical reduction," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, pp. 473-481, Apr. 1975.
[80] J. W. Ulrich and R. Moll, "Program synthesis by analogy," in *Proc. SIGART-SIGPLAN Conf. Artificial Intelligence and Programming Languages*, Rochester, NY, pp. 22-28, Aug. 1977.
[81] F. W. von Henke and D. C. Luckham, "A methodology for verifying programs," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, pp. 156-164, Apr. 1975.
[82] R. J. Waldinger, "Achieving several goals simultaneously," in *Machine Intelligence 8: Machine Representations of Knowledge*, E. W. Elcock and D. Michie, Ed. Chichester, England: Ellis Horwood, 1977, pp. 94-136.
[83] R. J. Waldinger and K. N. Levitt, "Reasoning about programs," *Artificial Intelligence*, vol. 5, no. 3, pp. 235-316, Fall, 1974.
[84] A. Wang, "An axiomatic basis for proving total correctness of goto-programs," *BIT*, vol. 16, pp. 88-102, 1976.
[85] B. Wegbreit, "Goal-directed program transformation," in *Proc. 3rd Symp. Principles of Programming Languages*, Atlanta, GA, pp. 153-170, Jan. 1976.
[86] N. Wirth, "On the composition of well-structured programs," *Comput. Surveys*, vol. 6, no. 4, pp. 247-259, Dec. 1974.
[87] W. A. Wulf, R. L. London, and M. Shaw, An introduction to the construction and verification of ALPHARD programs," *IEEE Trans. Software Eng.*, vol. 2, pp. 253-265, Dec. 1976.

Zohar Manna received the B.S. and M.S. degrees in mathematics from the Technion, Israel Institute of Technology, Haifa, Israel, in 1961 and 1965, respectively, and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1968.

From 1968 to 1972 he was an Assistant Professor of Computer Science at Stanford University, Stanford, CA. From 1972 to 1975 he was an Associate Professor, and from 1975 to date a Professor, at the Weizmann Institute of Science, Rehovot, Israel. For the last two years he has been on leave from the Weizmann Institute, and works at the Artificial Intelligence Laboratory, Stanford University, Stanford, CA. His current research interests include program verification, methodology of programming, program synthesis, and the various areas of the mathematical theory of computation. He is the author of the book *Mathematical Theory of Computation* (New York: McGraw-Hill).

Richard Waldinger was born in Brooklyn, NY, on March 1, 1944. He received the A.B. degree in mathematics from Columbia College, New York, NY in 1964. and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1969.

He has worked as a programmer for the IBM Field Engineering Division, White Plains, NY, and as a researcher for the NIH Heuristics Laboratory, Bethesda, MD. Since 1969 he has been employed by the Artificial Intelligence Center, SRI International, Menlo Park, CA. He has also been a visiting lecturer at Stanford University, Stanford, CA. His principal research interest is in automatic program synthesis, but he has also worked in the related topics of program verification, theorem proving, automatic planning, and programming languages for artificial intelligence.