An Introduction to Proofs of Program Correctness for Teachers of College-level Introductory Programming Courses

David Gries
Jeff Wadkins*

TR 90-1102
March 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Jeff Wadkins' address is the Educational Testing Service, Princeton, NJ 08541.

An Introduction to Proofs of Program Correctness for Teachers of College-level Introductory Programming Courses

David Gries, Computer Science, Cornell University
Jeff Wadkins, Educational Testing Service[1]

March 1990

Table of contents

0. Introduction, 1
1. Program specifications, 3
2. The rule of consequence, intermediate assertions, and proof outlines, 5
3. The assignment statement, 6
4. Sequencing, 8
5. The conditional statement, 9
6. The while loop, loop invariant and bound function, 10
7. The relation between mathematical induction and loop invariants, 14
8. Developing loops from specifications, 14
9. Example of the development of algorithms, 18
10. Remarks on presenting, documenting, and grading algorithms, 24
11. Conclusion, 26

0. Introduction

Loop invariants and the general topic of program verification appear in the Topic Outline (see p. 11 of [0]) for the College Board's Advanced Placement (AP) Computer Science courses. However, since few elementary textbooks treat this topic with more than a passing nod, many teachers of programming courses find the topic esoteric and devote little time to it.

We find this situation unfortunate, because we view the topic as central to a solid understanding of programming. The earlier the ideas of program verification are introduced to the student, albeit informally, the better. A sound grasp of correctness-proof ideas colors the way we think about programs, how we develop them, and how we look for errors in them. It shapes our rules for documenting programs and our methods of algorithmic presentation. And it drastically influences the way we teach programming and present algorithms.

This paper is directed towards the teachers of introductory programming at the college level, including AP courses in high schools. Our goal is to present the basic concepts of proof of correctness of a program in a way that

[1] The idea for this paper was inspired in 1984 by an offhand remark by Fred Schneider of Cornell, then a member of the Development Committee for the AP Computer Science Examination. In response to a remark by the second author about the similarity of proofs of loops using loop invariants and proofs by mathematical induction, Schneider said that a loop invariant is simply an induction hypothesis for a proof by induction of the correctness of a loop. This remark spawned the first version of the paper. Four years later, after many rewrites by Wadkins, the first author became involved in the project, leading to this collaborative effort.

will make clear their usefulness in programming and in teaching programming. With regard to the loop invariant, which seems to be a stumbling block for many, we describe its relation to mathematical induction, showing how it can be viewed as an induction hypothesis, and illustrate its practical use during the programming process.

Examples

To whet the reader's appetite and motivate them[2] to stick with this paper through some necessary preliminaries that involve unfamiliar notation and terminology, we offer now some examples, which the reader is invited to think about before reading our solutions (given later in the paper). They are little problems, which any programmer with a year's experience should be able to solve crisply and cleanly. Yet, most programmers find them difficult, while those well versed in the methods outlined in this paper have little trouble with them.

(0.0) **Problem. Binary search.** It has been remarked by many that most programmers cannot write a binary search correctly. One search of forty introductory programming texts found only nine correct solutions [6]. Also, experiments conducted in courses for advanced people (e.g. by E.W. Dijkstra) revealed that many in the audience could not develop a correct binary search.

We state the problem as follows. Given is an integer x and a segment $b[1..n]$ of an array b of integers, where $0 \le n$. Assume that $b[0] = -\infty$ and $b[n+1] = \infty$, but note that these virtual array elements may not be referenced in the algorithm. They exist only in our heads, in order to specify the problem, and not in the program. Write an algorithm to store in h an integer to truthify (establish the truth of)

$$0 \le h \le n \textbf{ and } b[h] \le x < b[h+1] .$$

The algorithm should take time proportional to $log(n+1)$.

Note that the specification requires the program to work in the case $n = 0$, i.e. for a search in an empty segment of array b. Note that if b is sorted, the algorithm finds the rightmost occurrence of x in b —or the position where x belongs if x is not in b. But if b is not sorted, it finds only a position h satisfying $b[h] \le x < b[h+1]$. There are contexts where this is useful. □

(0.1) **Problem. Table of cubes.** Given is an integer n, $0 \le n$, and an array b of natural numbers. Write an algorithm to store in $b[0..n-1]$ the cubes of the first n natural numbers. Thus, set $b[0] = 0$, $b[1] = 1$, $b[2] = 8$, ..., $b[n-1] = (n-1)^3$. Use only addition and subtraction, and not multiplication or division. How fast is your algorithm? Is it linear in n? Quadratic? Cubic? □

(0.2) **Problem. Exponentiation.** Given are integers a and b with $b \ge 0$. Consider the following algorithm, which stores in variable z the value a^b (and requires time proportional to $log(b+1)$).

```
z := 1; x := a; y := b;
while y ≠ 0 do if odd(y) then begin z := z*x; y := y−1 end
                         else begin x := x*x; y := y div 2 end
```

The algorithm works with the bit representation of the exponent (stored in y). Each iteration processes the rightmost bit; if it is 0, it is discarded; if it is 1, it is set to 0. The question is: how do we present the algorithm, i.e. how do we convince someone that it is correct? This question is unfair, in that we believe it is our duty always to present a correctness argument with a program, and we have not done so here. However, this example is striking evidence of the effectiveness of the ideas discussed in this paper. □

A Word of warning

Many ideas in this paper may seem new and strange, and the reader will have to spend some time digesting them before feeling at ease with them. For example, in many places, instead of arguing operationally that a statement has a particular effect, we may prove it formally, using some symbol manipulation, because it provides a crisper,

[2] Throughout, instead of *he*, *him*, and *his*, we use the more neutral words *they*, *them* and *their* in the singular to refer to a person of either sex. This practice is supported by the definitions of these words in the Oxford English Dictionary.

shorter, and more satisfying explanation than does a more operational explanation about what happens when the statement is executed. Further, this paper is fairly long and takes effort to read, to understand examples, to try a few exercises, and to digest the material. However, anything of value takes time and effort. We believe that the reader that follows through with this paper will be well rewarded with a better insight into the programming process. When motivation to continue reading the paper wanes, we suggest the reader turn to sections 10 and 11, which provide more arguments concerning the usefulness of this material.

Also, this paper is only an introduction to issues in program correctness and is far from complete. There is far too little space to include all the interesting insights that are gained from a study of correctness-proof issues. The reader is urged to look further at the references mentioned in the last section.

Notation and conventions

For our programming notation, we would prefer to use the guarded-command notation of Dijkstra [1], [2], [4], because it is more amenable to the kind of program development we like to do. However, it is foreign to many and it is not the programming notation used in communicating with the computer, so its use tends to alienate some people. Instead, we use the programming language Pascal (although we use more traditional mathematical symbols like $\neq$ instead of $<>$). We deal only with the assignment statement, the conditional statement, and the while loop, and we assume that the reader is familiar with them. We introduce notation to talk about a segment, or portion, of an array (we used this in earlier examples). Thus, $b[i..j]$ denotes the segment of array b consisting of elements $b[i], b[i+1], ..., b[j]$. If $j = i-1$, then $b[i..j]$ denotes the empty segment beginning at position i. We use the word "program" to denote any portion of a program or subprogram.

We are informal about our assertion language —the notation in which we write and manipulate true-false statements about program variables. We believe that the predicate calculus is an almost indispensable tool of the professional programmer —one that they has dispensed with for too long— , and the student would do well to learn how to use it before learning to program. Unfortunately, the current situation in high schools and colleges does not cater to this point of view. Fortunately, the concepts of program correctness can be useful, although not as effective, even when one has not mastered the predicate calculus. Within this paper, we often use a "calculational" style of showing that something holds, or deriving it, in the hope that the reader will begin to appreciate the predicate calculus and this style. (Hereafter, "calculus" means "predicate calculus".) Nevertheless, one of our goals is to show that program correctness concepts can be used in a less formal manner.

We make two restrictions on our programming language. We assume that the **goto** is not used, and we do not allow side effects —evaluation of an expression changes no variable, and execution of an assignment changes only the variable being assigned. A side effect occurs when invocation of a programmer-defined function f (say) changes the value of a (non-local) variable. Such a side effect may cause the loss of basic mathematical properties, such as $f(i)+f(i) = 2*f(i)$, thus reducing our powers of reasoning. It also drastically complicates the understanding of the assignment statement (see Sect. 3).

We are not so pedantic as to outlaw these two features completely, for they may be useful from time to time. But we have learned to think without them, as if they did not even appear in our everyday programming vocabulary, and we rarely miss them.

Finally, we assume that all expressions are well-defined in the state in which they are evaluated. For example, there is no division by zero and no subscript out of range. Our programs follow this rule. Changing our definitions to take care of undefined expressions does not provide enough new ideas for the additional space it would require.

1. Program specifications

One of the first things to learn about programming is to make precise what a program segment is to do before trying to write it. This is simply an instance of the old adage "don't try to solve a problem until you know what the problem is." It may seems lazy not to take this advice, but if "lazy" implies a desire to spend less time, it is actually the other way around. Being careful and precise at the specification stage for all but the most trivial programs can reduce dramatically the time spent writing and debugging. Nevertheless, it is difficult to become accustomed to working this way. Therefore, the teacher must go overboard in following this rule, emphasizing it over and over again in the classroom, and strict adherence to it should be required on homeworks and exams.

Some scorn this idea of writing precise specifications for small programs because (they feel) it is impossible to write precise specifications for large ones. Do not be swayed by this argument. There are enough small, but important, algorithms and programs being written to make such specifications valuable. Furthermore, every large program consists of many small ones, and if you cannot write the small ones effectively you cannot write the large ones effectively either.

Specifications are given in terms of "conditions" (or "predicates") that represent sets of states. A state in our context is simply the set of the variables of a program and their associated values at some time during execution of the program. It can be considered to be the contents of the memory of the computer during a program execution, but abstractly written in terms of the program instead of as a series of 0's and 1's. For example, suppose the state has three integer variables x, y, and z. Then a state is a set $\{(x,Vx), (y,Vy), (z,Vz)\}$, where Vx, Vy, and Vz are integers. Thus, $\{(x, 1), (y, 8), (z, -4)\}$ is the state in which x has the value 1, y the value 8, and z the value -4. We also consider the input and output files as variables, so their contents can also be included in the state of a program.

A condition (predicate) is a true-false description of a set of states; it *represents* the set of states in which it is true. For example, given states with three variables x, y, and z, the condition $x > y$ represents the set of all states in which the value of x is greater than the value of y (no condition is specified for the value of z). We say that any such state "satisfies the condition", or that "the condition holds in those states".

There are two extreme conditions. The set of all states is represented by the condition $1 = 1$, or by any tautology, because $1 = 1$ is true in all states. It is customary to use the constant *true* to represent the set of all states. Similarly, the set of no states can be represented by $1 = 2$, since it is false in all states. The constant *false* represents the set of no states.

Suppose a condition $C1$ implies another condition $C2$: wherever $C1$ is true, $C2$ is true also. We write this as $C1 \Rightarrow C2$ or as $C2 \Leftarrow C1$. Then every state represented by $C1$ is represented by $C2$ as well. In this case, $C1$ is said to be stronger than $C2$ and $C2$ to be weaker than $C1$, since $C1$ makes more restrictions on the set of states it represents. For example, the condition $x = y$ is stronger than $x \leq y$. The constant *false* is the strongest possible condition —it is so restrictive that it refuses to represent any state at all— while *true* is the weakest possible condition —it makes no restriction whatever on the states it represents.

Let us now return to the notion of specification. We can specify a program (segment) S with a command, like "swap the values of x and y" and "Given $n \geq 0$, sort $b[1..n]$". Such a specification can be rewritten as a precondition Q (say) —e.g. $n \geq 0$— which describes the restrictions on the input variables under which S is guaranteed to work, and a postcondition R —e.g. $b[1..n]$ is sorted— which describes the desired result in terms of the output variables. Q and R are true-false statements about the program variables, written in any suitable notation. Writing a specification in this manner not only serves to make it more precise, it can also be helpful during program development. As we shall see later, the shape of the pre- and post-conditions can often guide the program development.

We often write the specification for a program S in the form

$$\{Q\}\ S\ \{R\}$$

with meaning: execution of S begun with Q true is guaranteed to terminate, and with R true. The notation $\{Q\}\ S\ \{R\}$ is called a *Hoare triple*, after C.A.R. Hoare, who first talked about proofs of programs in this fashion [5]. Note that the specification says nothing about execution of S when Q is false.

In summary, we specify a program by describing, using conditions, the set of states in which the program will work correctly and the corresponding set of final states.

Here are four examples of specifications.

(1.0) **Example.** An algorithm to store in x the sum of the elements of $b[1..n]$.

$$\{true\}\ S\ \{x = \text{sum of } b[1..10]\}$$

Precondition *true* indicates that execution of S begun in *any* state will terminate with the stated postcondition true. The postcondition could be written as "x = sum of the first 10 elements of b", or $x = \sum_{i=1..10} b[i]$, or $x = \Sigma(i: 1 \leq i < 11: b[i])$, or in any number of ways. □

(1.1) **Example.** An algorithm to swap x and y. Below, the identifiers A and B are not program variables, but stand for arbitrary fixed values. In this paper, capital letters are used as identifiers for this purpose only. In this case, the Hoare triple can be paraphrased as "whatever the values of x and y, execution of S swaps them". This algorithm is specified by

$\{x = A \textbf{ and } y = B\}\ S\ \{x = B \textbf{ and } y = A\}$ □

(1.2) **Example.** We give two specifications for the kind of program that students are often asked to write in the first few weeks of a course: read values from the input until a zero is read and store their sum in variable x. The first specification uses English and the past tense. The second views the standard input file as a sequence variable *input* and uses the operation of catenation, denoted by ^. We prefer the second because it is shorter, because its precondition and postcondition are easily seen as sets of states, and because the conditions are more amenable to manipulation. Remember our convention that capital letters denote arbitrary values, and not program variables.

(a) {the input begins with N positive values followed by a 0, where $N \geq 0$ }
S
{The first N values and the following zero have been read from the input, and}
{ x is the sum of those N values }

(b) {*input* = T ^0 ^U for some sequences T and U, where T does not contain a 0}
S
{*input* = U **and** x = sum of elements of T} □

(1.3) **Example.** Refer back to example (0.0) of binary search. The precondition Q and postcondition R are given below. In addition, the algorithm is not to change b or n and it is not to refer to virtual elements $b[0]$ and $b[n+1]$, which we assume equal $-\infty$ and $+\infty$, respectively.

Q: $0 \leq n$ **and** $b[0] \leq x < b[n+1]$,
R: $0 \leq h \leq n$ **and** $b[h] \leq x < b[h+1]$.

2. The rule of consequence, intermediate assertions, and proof outlines

At this point, we can give our first rule concerning program proofs using Hoare triples. The validity of this rule depends on our interpretation of the Hoare-triple notation and our knowledge of execution of a program.

(2.0) **Rule of consequence**: Suppose $Q1 \Rightarrow Q$, $R \Rightarrow R1$, and $\{Q\}\ S\ \{R\}$. Then $\{Q1\}\ S\ \{R1\}$.

This rule allows us to strengthen a precondition, i.e. to change Q to a predicate $Q1$ that describes a perhaps-smaller set of states. For example, if $\{x > 5\}\ S\ \{R\}$ holds, then so does $\{x > 6\}\ S\ \{R\}$. In the same way, (2.0) allows us to weaken, or generalize, the postcondition.

The previous section discussed the placement of conditions before and after a program (segment) in order to indicate what is expected to be true at that place during program execution. When so placed, we call them *assertions*, since we are asserting they are true whenever execution reaches that place. We can also place an assertion within braces (which are not part of the assertion) between program statements as well. For example, we might write

{x = sum of $b[1..i]$}
$i := i+1$;
{x = sum of $b[1..i-1]$}
$x := x+b[i]$
{x = sum of $b[1..i]$}

An assertion placed between two statements serves as a postcondition for the preceding statement and a precondition for the following statement. By placing an assertion between two statements, we are asserting that it will be true when execution reaches that place. We place an assertion between two statements in order to give more information; it is a form of documentation that is oriented towards a proof-of-correctness point of view.

We place two assertions next to each other within a program, as in

$\{x > 10\}$
$\{x > 0\}$

to assert that the first implies the second.

A program fully annotated with assertions so that all information needed to prove the program correct is included in the program is called a *proof outline*.

3. The assignment statement

Suppose we need a statement S that satisfies the following (which is actually used in the body of the loop for our solution to problem (0.2)). How do we create a suitable postcondition for S, so that S can be developed independently of the context in which it appears?

(3.0) $\{y > 0$ **and** $z*x^y = c\}$ S; $y := y-1$ $\{z*x^y = c\}$

Or consider (3.1), which indicates that the statement halves y while keeping $z*x^y = c$ true. (This is also used in our solution to problem (0.2).)

(3.1) $\{even(y)$ **and** $y > 0$ **and** $z*x^y = c\}$
$x := x*x$; $y := y$ **div** 2
$\{z*x^y = c\}$

How do we argue that (3.1) is correct? How do we convince students that execution of the assignments in a given initial state terminates with the postcondition true? (If you feel this is easy, stop reading for a moment and write down a careful explanation.)

In elementary algebra, we prove an identity like $(x+1)^3 = x^3+3x^2+3x+1$ by performing manipulations according to various rules, many of which we learned in elementary school and which are so ingrained in us by now that we use them unconsciously. With the assignment statement, however, we are in the different world of change — the value of a variable changes. This is fundamentally different from ordinary elementary algebra, which is usually considered more static. Fortunately, there is a simple way to relate a change of a value back to our old mathematical, or logical, world: the notion of *textual substitution* that appears throughout logic.

Let R be any expression, x an identifier (variable), and e an expression of the same type as x. The notation R^x_e, or, in linear form,

$$[x := e]R,$$

denotes a copy of R with every (free) occurrence of x replaced by e. The process of producing $[x:=e]R$ from R is called textual substitution. Here are three examples.

$[x := 2](x = y) \;=\; (2 = y)$,
$[x := x+2](x < x+y) \;=\; ((x+2) < (x+2)+y)$,
$[i := i+1]$ (x is the sum of $b[1..i]$) $\;=\;$ (x is the sum of $b[1..i+1]$) .

The first example illustrates a rather counter-intuitive fact: expression $[x := 2](x = y)$, which is equal to $2 = y$, is independent of x. This is because the textual substitution requires replacing x everywhere by 2. In general $[x := e]R$ is independent of x if x does not occur in e.

With this notation, we can develop a rule for reasoning about an assignment $x := e$. Based on our knowledge of how it is executed, we look for a precondition ? that makes the following true:

$\{?\}$ $x := e$ $\{R\}$ for all predicates R.

In fact, we will find a condition ? that is true in the initial state *exactly* when R is true in the final state. The initial and final states differ only in their x-values —all other variables have the same value in both states. Therefore,

only the different values of x force R to have different values in the two states. In the final state, the value of x equals the value of e in the initial state. Hence, $[x := e]R$ evaluated in the initial state has the same value as R evaluated in the final state. Therefore, we have

(3.2) **Rule for assignment**: For any variable x and expression e with the same type,

$\{[x := e]R\}\ x := e\ \{R\}$ holds for all predicates R.

When we can calculate a precondition for a statement S and postcondition R, we often write the precondition as an application $pre(S,R)$ of a function pre. Thus, we can write the rule in the form

(3.3) **Rule for assignment**:

$pre(\text{``}x := e\text{''}, R) = [x := e]R$ for all predicates R.

We can now find a suitable postcondition for S of (3.0) by finding the precondition for the statement following S. We have[3]

$pre(\text{``}y := y-1\text{''}, z*x^y = c)$
$=$ «Use rule (3.3)»
$[y := y-1](z*x^y = c)$
$=$ «Apply textual substitution»
$z*x^{y-1} = c$

Hence, statement S of (3.0) must satisfy $\{y > 0$ **and** $z*x^y = c\}\ S\ \{z*x^{y-1} = c\}$.

For the second problem of finding a convincing argument that (3.1) is correct, we apply the Rule of Assignment from end to beginning, as illustrated in the three columns below, and also apply the Rule of Consequence to the last precondition obtained. The reader should attempt to apply these two rules as indicated before looking at our version below.

		$\{even(y)$ **and** $y > 0$ **and** $z*x^y = c\}$
	$\{z*(x*x)^{y \text{ div } 2} = c\}$	$\{z*(x*x)^{y \text{ div } 2} = c\}$
	$x := x*x;$	$x := x*x;$
$\{z*x^{y \text{ div } 2} = c\}$	$\{z*x^{y \text{ div } 2} = c\}$	$\{z*x^{y \text{ div } 2} = c\}$
$y := y$ **div** 2	$y := y$ **div** 2	$y := y$ **div** 2
$\{z*x^y = c\}$	$\{z*x^y = c\}$	$\{z*x^y = c\}$

Since each triple in the right column above is a valid Hoare triple, and since the first assertion in the right column implies the second, we have proven that (3.1) is valid. Thus, (3.1) is easily explainable to anyone who understands the Rules of Consequence and Assignment.

The finding of a precondition from the statement and postcondition may seem backwards. There is a rule for calculating a postcondition from the precondition for the assignment statement, but it is much more complicated than simple textual substitution. This is *one* reason why we deal only with the "backward" rule.

Let us for the moment discuss *developing* an assignment statement. In high school, we all learned to solve a word problem like the following one. John has twice as many apples as Mary. John eats one and Mary throws away half of hers because they are rotten. John now has three times as many apples as Mary. How many apples do Mary and John have?

We were taught to translate such problems into a formal mathematical notation, often a system of equations, and were given methods for solving the formal statement of the problem. We practised and practised such exercises.

Now consider the following programming problem. We are given a condition P: $C = i+p$. We would like to

[3] Note the format of this calculation. Between each pair of equivalent formulas appears a hint indicating why they are equal, or what rule is used to transform the first expression into the second. Since equality is transitive, the calculation shows that the first formula is equivalent to the last. We use operations $\Rightarrow$ and $\Leftarrow$, as well as $=$, in such calculations. This format is used throughout the paper.

increase i to $i+p$, but have condition P remain true, and this requires an assigment $p := e$ for some unknown expression e. We can state this formally as solving for e in the Hoare triple

$$\{C = i+p\}\ p := e;\ i := i+p\ \{C = i+p\}\ .$$

While one can solve this problem using seat-of-the-pants methods —and typically do—, there are calculational methods for solving them as well. These calculational methods are based on the fundamentals of the thoery of correcetness. Since this kind of problem occurs frequently in programming, it makes sense to understand and use the calculational methods, especially when the problems get more difficult. And yet, not only do we not teach these calculational methods to our students, most *teachers* of programming do not know them.

4. Sequencing

In Sect. 3, we managed to sneak in the use of a rule before explaining it, mainly because it was so obvious. Consider a sequence of two statements $S1$ and $S2$ for which we want to prove $\{Q\}\ S1; S2\ \{R\}$. To show that this holds, we have to find an intermediate assertion T (say), that acts as the postcondition for $S1$ and the precondition for $S2$. Thus, we have to show

$$\{Q\}\ S1\ \{T\} \quad \text{and} \quad \{T\}\ S2\ \{R\}\ .$$

The first triple $\{Q\}\ S1\ \{T\}$ indicates that execution of $S1$ beginning with Q true terminates with T true, and the second indicates that further execution of $S2$ terminates with R true, so that $\{Q\}\ S1; S2\ \{R\}$ holds.

This gives us the following definition of the semicolon, the operator that joins two statements together into a unit, much the way the semicolon in English joins two independent, adjacent clauses into a sentence:

(4.0) **Rule for sequencing**: for any predicates Q, T, and R,
if $\{Q\}\ S1\ \{T\}$ and $\{T\}\ S2\ \{R\}$ hold, then so does $\{Q\}\ S1; S2\ \{R\}$.

Again using the notation $pre(S, R)$ to denote the precondition calculated according to our rules, we can define

(4.1) **Rule for sequencing**: $pre(\text{“}S1; S2\text{”}, R) = pre(S1, pre(S2, R))$.

Let us apply this rule and the assignment statement rule to a slightly larger example. Consider the sequence $t := x;\ x := y;\ y := t$. Suppose it should terminate with $x = 6$ and $y = 4$ and we want to know what should be true initially. We calculate the precondition:

$pre(\text{“}t := x; x := y; y := t\text{”},\ x = 6$ **and** $y = 4)$
= «Apply rule (4.1)»
$pre(\text{“}t := x; x := y\text{”},\ pre(\text{“}y := t\text{”},\ x = 6$ **and** $y = 4))$
= «Apply rule (3.3) to find $pre(\text{“}y := t\text{”},\ x = 6$ **and** $y = 4)$»
$pre(\text{“}t := x; x := y\text{”},\ x = 6$ **and** $t = 4)$
= «Apply rule (4.1), and then rule (3.3)»
$pre(\text{“}t := x\text{”},\ y = 6$ **and** $t = 4)$
= «Apply rule (4.1)»
$y = 6$ **and** $x = 4$

One could also annotate the sequence of assignments as shown below:

$\{y = 6$ **and** $x = 4\}$
$t := x;$
$\{y = 6$ **and** $t = 4\}$
$x := y;$
$\{x = 6$ **and** $t = 4\}$
$y := t$
$\{x = 6$ **and** $y = 4\}$

In the above, we could replace 6 and 4 by any other constants and have a similar result. This shows that execution of the sequence of assignments swaps the values of x and y.

This example illustrates the goal-directed nature of programming. The goal, the postcondition, is used to find the precondition. This "backwards" nature is characteristic of the calculational, proof-oriented program development. Also, we begin to see more and more that the calculation of a precondition from a sequence of assignments and a postcondition can be viewed as a mechanical process, so much so that we rarely have to place an assertion between two adjacent assignments.

We give a convincing argument, in a different format, that (3.1) is correct:

$$
\begin{array}{ll}
 & pre(\text{"}x := x*x;\ y := y \textbf{ div } 2\text{"},\ z*x^y = c) \\
= & \quad \text{«Rule (4.1) and (3.3)»} \\
 & pre(\text{"}x := x*x\text{"},\ z*x^{y \textbf{ div } 2} = c) \\
= & \quad \text{«Rule (4.1) and (3.3)»} \\
 & z*(x*x)^{y \textbf{ div } 2} = c \\
\Leftarrow & \quad \text{«Arithmetic»} \\
 & even(y) \textbf{ and } y \geq 0 \textbf{ and } z*x^y = c
\end{array}
$$

Note the use of operator $\Leftarrow$. Using $\Rightarrow$ would have required starting with the last formula of the above calculation and deriving the first formula. The calculation would then appear to be far more difficult because there would be no insight for performing each step. Starting from the more complicated formula, the first, makes each step seem easier because the structure of the formula gives insight into the next transformation to apply.

5. The conditional statement

Suppose we wish to prove the following about the Pascal if-statement

$$\{Q\}\ \textbf{if } B \textbf{ then } S\ \{R\}\ .$$

Let us determine what must hold for this to be true, based on our knowledge of how the if-then is executed. First, if B is false and Q is true, then R should already be true, since execution of the statement terminates without changing any variable. That is, we must have $(Q \textbf{ and not } B) \Rightarrow R$. Secondly, if B is true and Q is true, then execution of S should terminate with R true. This simple analysis leads to the following rule for the if:

(5.0) **if-then rule**:

If $(Q \textbf{ and not } B) \Rightarrow R$ and $\{Q \textbf{ and } B\}\ S\ \{R\}$, then $\{Q\}$ **if** B **then** $S\ \{R\}$.

Rule (5.0) simply reflects the normal obligations of the programmer in understanding that the if-then works correctly. In the same manner, we can develop rule (5.1) for the if-then-else.

(5.1) **if-then-else rule**:

If $\{Q \textbf{ and } B\}\ S1\ \{R\}$ and $\{Q \textbf{ and not } B\}\ S2\ \{R\}$,
then $\{Q\}$ **if** B **then** $S1$ **else** $S2\ \{R\}$.

At this point, let us develop an algorithm in order to illustrate our proof-oriented, calculational method. We use an extremely simple example, finding the maximum of two variables, in order to be able to analyze the method without the problem getting in the way. The triviality of the problem should not be taken to mean that the method can only be used on trivial problems. Given precondition Q: *true*, our algorithm is to store in z to establish

(5.2) R: $z = max(x,y)$.

Where should we start? Most of the information about the problem is in postcondition R , and to get some insight into the algorithm, we replace R by an equivalent predicate that does not use function *max* but makes its meaning explicit. Here is one possibility:

(5.3) R: $(z = x$ **and** $z \geq y)$ **or** $(z = y$ **and** $z \geq x)$.

This form of R suggests two possible assignments that, at least in some states, will establish R, namely $z := x$ and $z := y$. However, we have to determine the states in which each will do the job. For this, we can use rule (3.3) for calculating the precondition for an assignment in order to establish a certain postcondition. Let us first find a precondition under which execution of $z := x$ establishes R:

$pre(\text{“}z := x\text{”}, R)$
= «Use form (5.3) of R and apply assignment rule (3.3)»
$(x = x$ **and** $x \geq y)$ **or** $(x = y$ **and** $x \geq x)$
= «$x = x$ and $x \geq x$ are true; simplify»
$x \geq y$ **or** $x = y$
= «simplify»
$x \geq y$

Hence, we have calculated the condition $x \geq y$ under which $z := x$ does the job. This leads directly to a statement **if** $x \geq y$ **then** $z := x$ **else** $\cdots$.

Note how *we used the proof obligation given in rule (5.1) to develop the guard for the statement.* A programmer must use this proof obligation, even when arguing operationally about the correctness of the program, i.e. in terms of how it is executed. Some say that one advantage of our method is that it makes programmers aware of what they knew only subconsciously, making it easier for them to be careful.

To finish the story, we have to deal with the case $x < y$. In a similar manner, or just by using symmetry, we arrive at the else part $z := y$, giving us the if-statement below.

if $x \geq y$ **then** $z := x$ **else** $z := y$.

Remark. Note the asymmetry forced on us by the if-statement. The complete development would show that, in the case $x = y$, either $z := x$ or $z := y$ could be executed, but the form of the if-statement forces us to choose only one of them. The if-statement of Dijkstra's guarded command notation does not force this choice. In his notation, the statement would be written as

if $x \geq y \rightarrow z := x$
[] $y \geq x \rightarrow z := y$
fi.

6. The while loop, loop invariant, and bound function

We now discuss the Pascal loop **while** B **do** S, where S is a statement and B is a boolean expression, which we will call the *guard* of the loop. Execution of the loop can be defined as follows:

Repeat the following until the evaluation of B yields false:
Evaluate B, and if it is true execute S.

Execution of the loop body beginning with guard B true is called an *iteration* of the loop. The first execution of loop body S is called iteration 0, the next one iteration 1, and so on. Note that, if B is false initially, no iterations are executed. Note also that the guard is evaluated once more then the total number of iterations. Finally, note that upon termination the guard is false. Because of our ban on the goto and side-effects, the only way the loop can terminate is by the loop guard becoming false.

Our main goal in this section is to describe a technique for understanding —i.e. proving— that a loop does its job, i.e. that it terminates with its postcondition true if executed with its precondition true. Thus, we aim for a rule like rule (3.2) for the assignment or (5.0) for the if-then statement. We will introduce our technique using the following loop, which stores in s the sum of the first n positive integers.

(6.0) $\{Q\colon\ 0 \le n\}$
$i := 0;\ s := 0;$
while $\{P\}\ i \neq n$ **do begin** $i := i+1;\ s := s+i$ **end**
$\{P$ **and** $i = n\}$
$\{R\colon\ s$ is the sum of $1..n\}$

where P is

(6.1) $P\colon\ 0 \le i \le n$ **and** s is the sum of $1..i$.

Note that we have annotated this little program with its precondition and postcondition, so that we have a precise specification for it. The notation $1..n$ in the postcondition denotes the set of integers $1, 2, \cdots, n$. Note that the case $n = 0$ is included, in which case $1..n$ denotes the empty set. Since the sum over an empty set is 0, s should be 0 upon termination if $n = 0$. (Such cases are rarely handled correctly by the beginner, or even by the advanced programmer. Introducing suitable mathematical notation increases the chances of their being handled correctly.)

Of extreme importance is the assertion P placed just before the loop guard, that is, at the place where the loop guard is evaluated. We are asserting that P is true each time the loop guard is evaluated. P is in a very real sense a definition of variable i and s that holds just before and after each iteration of the loop. An assertion placed at this point plays such an important role in our understanding loops that we give it a name:

(6.2) **Definition**. A condition whose truth is maintained by each iteration is called a *loop invariant*. □

We argue that the program satisfies its specification. First, we show that P (6.1) is true before execution of the loop begins, so that it is true when the loop guard is evaluated the first time. This can be seen informally, and there is little need to say much about it. However, the formal method is not much more trouble. We show that

$\{Q\}\ i := 0;\ s := 0\ \{P\}$

holds by computing $pre(\text{“}i := 0;\ s := 0\text{”},\ P)$ and showing that it is implied by Q.

$pre(\text{“}i := 0;\ s := 0\text{”},\ P)$
= «Use rule (4.1) for semicolon and (3.3) for assignment)»
$pre(\text{“}i := 0\text{”},\ 0 \le i \le n$ **and** 0 is the sum of $1..i)$
= «Use rule (3.3) for assignment)»
$0 \le 0 \le n$ **and** 0 is the sum of $1..0$
= «$0 \le 0$ **and** 0 is the sum of $1..0$ are both true»
$0 \le n$

We now show that P is true each time the loop guard is evaluated. We have proved that it is true after 0 iterations —when execution of the loop begins— and we now show that it is true after any number of iterations. Viewed in this manner, mathematical induction over the natural numbers 0, 1, ... cries out to be used. Assuming that P is true after k (say) iterations, for any $k \ge 0$, we have to prove that it is true after $k+1$ iterations.

Iteration $k+1$ is performed only when B is true. Further, by the inductive assumption, P is true when iteration $k+1$ begins. Hence, we can show that iteration $k+1$ terminates with P true by showing that execution of S begun with B and P true terminates with P true, i.e. by showing that

$\{P$ **and** $B\}\ S\ \{P\}$.

This is the inductive step that has to be proved.

One can prove this informally and operationally, as in the past, or one can prove it more formally. Consider the loop of (6.0). We have to show that $\{P$ and $B\}\ i := i+1;\ s := s+i\ \{P\}$ holds. To do this, we compute as follows:

$pre(\text{“}i := i+1;\ s := s+i\text{”},\ P)$
= «Use rule (4.1) for semicolon and (3.3) for assignment, twice»
$0 \le i+1 \le n$ **and** $s+(i+1)$ is the sum of $1..(i+1)$
⇐ «Arithmetic»

$i \neq n$ **and** $0 \leq i \leq n$ **and** s is the sum of $1..i$

= «Definition of P and B »

P **and** B .

Hence, P **and** B does imply the condition that is necessary for execution of S to terminate with P true, so $\{P$ **and** $B\}$ S $\{P\}$ holds.

We have completed a proof by induction that P is true before and after each loop iteration.

Finally, note that directly after the loop in (6.0) we have placed the assertion $(P$ **and** $i = n)$. P is true upon termination of the loop because it was true when the guard evaluated to false; $i = n$ is true upon termination of the loop because it is the negation of the guard. From P and $i = n$ we conclude that R is true, as implied by the assertions we placed in program (6.0), so we have shown that R is true upon termination.

We have shown that, provided the program terminates, it terminates with R true. It still remains to prove that the loop terminates. To this end, we introduce the function $t = n - i$, which, we claim, is always an upper bound on the number of iterations still to be performed. Because of this, we call t a bound function of the loop. Here are the properties that an integer function t must satisfy in order to be called a bound function:

(a) If there is at least one more iteration to perform, then $t > 0$, i.e. $(P$ **and** $B) \Rightarrow t > 0$, and
(b) Each iteration decreases t by at least 1 (we leave this informal).

Suppose bound function t satisfies these properties. Then the loop terminates, by the following argument. Since each iteration decreases t , at some point a state in which $t \leq 0$ is reached. In that state, since (a) holds, $(P$ **and** $B)$ is false. P is not false, because it has been proven to be true each time the guard is evaluated. Hence B is false and the loop terminates.

We have proved that program (6.0) terminates, and with R true, when executed beginning with its precondition Q true. To prove this, we introduced the concept of a loop invariant P , which is true whenever the loop guard is evaluated, and a bound function t , which is an upper bound on the number of iterations to be performed. We summarize our proof method by stating the general rule for proving a loop correct:

(6.3) **Correctness of a loop while** B **do** S:
If (a) P is a loop invariant, i.e. $\{P$ **and** $B\}$ S $\{P\}$,
(b) integer function t satisfies $(P$ **and** $B) \Rightarrow t > 0$, and
(c) t decreases with each iteration,
then $\{P\}$ **while** B **do** S $\{P$ **and not** $B\}$.

To use rule (6.3) for a loop within a program, one has to show that invariant P is initially true and one has to show that the desired result R (say) of the loop follows from P **and not** B . The latter would be simply an application of the rule of consequence (2.0). Thus, we have the following:

(6.4) **To prove** $\{Q\}$ **while** B **do** S $\{R\}$
using invariant P **and bound function** t :

- Show that Q implies P ,
- Show that P **and not** B implies R ,
- Show that $\{P$ **and** $B\}$ S $\{P\}$,
- Show that P **and** B implies $t > 0$, and
- Show that each iteration decreases t .

One may ask whether a loop invariant and bound function exist for any loop. The answer is yes, provided the language for expressing invariants is rich enough. That is, if execution of a loop in a given set Q of states always terminates with R true, then an invariant and a bound function exist that allow one to prove $\{Q\}$ **while** B **do** S $\{R\}$ using (6.4).

What we really want to know is how practical the use of (6.3) is. Our experience tells us that is indeed very practical. Some algorithms cannot be understood easily without a loop invariant. Other algorithms are more easily presented using loop invariants. Finally, the use of the invariant and the other apparatus of correctness proving can be used to *guide program development*. If a program has to be proved correct, why not use the proof ideas to guide

the development, thus developing proof and program hand-in-hand? We will see a number of examples of this in a later section. First let us solve problem (0.2), understanding the exponentiation algorithm. The algorithm is

$\{Q\colon\ 0 \le b\}$
$z := 1;\ x := a;\ y := b;$
while $\{P\}\ y \neq 0$ **do** **if** $odd(y)$ **then** **begin** $z := z*x;\ y := y-1$ **end**
else **begin** $x := x*x;\ y := y$ **div** 2 **end**
$\{P$ **and** $y = 0\}$
$\{R\colon\ z = a^b\}$

and we are asked to prove it correct. We do so following rule (6.4) and using the following loop invariant and bound function.

(6.5) $P\colon\ 0 \le y$ **and** $z*x^y = a^b$
$t\colon\ y$

- That the initialization beginning with Q true establishes P is obvious (or can be proved easily).
- We have to show that (P **and not** B) implies R.

P **and not** $(y \neq 0)$
$=$ «Definition of P, calculus»
$0 \le y$ **and** $z*x^y = a^b$ **and** $y = 0$
$\Rightarrow$ «Calculus»
$z*x^0 = a^b$
$=$ «Arithmetic»
$z = a^b$

- We prove that P is a loop invariant. Consider the case that y is odd. We show that the **then**-part of the if-statement maintains P, by showing that invariant P together with the condition $odd(y)$ implies $pre(\text{“}z := z*x;\ y := y-1\text{”}, P)$.

$pre(\text{“}z := z*x;\ y := y-1\text{”},\ P)$
$=$ «Use rules of assignment and sequencing»
$0 \le y-1$ **and** $z*x*x^{y-1} = a^b$
$=$ «Arithmetic; use $x*x^{y-1} = x^y$»
$0 < y$ **and** $z*x^y = a^b$
$=$ P **and** $y \neq 0$

Hence, when y is odd, execution of the loop body maintains P. We leave the similar case of even y to the reader. It relies on the fact that, for y even, $(x*x)^{y \text{ div } 2} = x^y$. Hence P is maintained in this case, and for all y.

- It is obvious that (P **and** B) $\Rightarrow t > 0$ holds.
- It is obvious that each iteration decreases y by at least 1.

Using the notion of a loop invariant and bound function, this originally mysterious algorithm has become accessible to all.

Remark. The use of a calculational style to prove the invariance of P in these cases may seem like overkill. However, we encourage the reader to write down a less calculational and more operational argument, probably using English, and then to compare the two arguments. Which is shorter, easier to read, more precise? □

7. The relation between mathematical induction and loop invariants

For a loop **while** B **do** S and loop invariant P, we want to show that P is true upon termination of the loop. We do this by proving that if k iterations are executed, then

(7.0) P is true after k iterations (if there are k iterations), for any $k \geq 0$.

The obvious way to prove such a statement is by mathematical induction. The base case is $k = 0$, that is, P is true after 0 iterations. The base case is proved by showing that P is a valid postcondition of the statement that precedes the loop.

The inductive case consists of assuming that (7.0) holds for the case $k = n$ (≥ 0) and proving that it holds for $k = n+1$. That is, if P is true after $k = n$ iterations, then P is true after $k = n+1$ iterations (provided there are $n+1$ iterations). Since P **and** B is true before execution of the loop body, i.e. before iteration $n+1$, the inductive step requires proving that

(7.1) $\{P \textbf{ and } B\}\ S\ \{P\}$

holds. The interesting point in this is that k is not used in describing the inductive case. The proof of (7.1) is all that is needed to show the inductive case; no reference to k is necessary. Nevertheless, we are indeed performing an induction proof when showing that P is true upon termination of the loop.

Interestingly enough, each of our programming constructs can be understood using some mathematical technique. The assignment statement, as mentioned earlier, is brought under control through the use of the logical notion of textual substitution. Sequencing " $S1$; $S2$ " is understood through function composition —to see this, just look at rule (4.1) for sequencing:

$$pre(\text{"}S1; S2\text{"}, R) = pre(S1, pre(S2, R)) .$$

The if-statement **if** B **then** $S1$ **else** $S2$ is understood using the mathematical technique of "case analysis" —we investigate the cases B and **not** B separately. The loop is understood using mathematical induction. The procedure call, which is outside the scope of this paper, is dealt with using "abstraction", in that once we prove a procedure correct, thereafter, in writing calls on it, we deal only with what it does and not how it does it. (That is, one views the procedure as a black box, relying only on its specification. This is akin to relying on a theorem, once it is proved, and not reading the proof every time one refers to the theorem.)

A discussion of this kind with students, looking at the relationship between mathematical techniques and understanding programs, can help provide a more systematic view of programming. Naturally, a discussion has to be done with care, so as not to lose the student through the use of too much mathematics.

8. Developing loops from specifications

As illustrated by example program (0.2) for exponentiation, proving a program correct after the fact is a difficult chore, so difficult that we recommend that students never be asked to do it —except as we have done to make the point that it is difficult. It is the programmer's responsibility to include enough details of a correctness argument for each program they write. We should never have to look at a program bare of a correctness argument. The purpose of "documentation" is to provide that correctness argument. The sooner this is learned, the better.

If we do not want to concoct a correctness argument after the fact, and if we require a correctness argument, then the only alternative is to construct the correctness argument while constructing the program. Program and proof should come hand-in-hand, with the latter usually using the way. This subject is the topic of this section. We concentrate mainly on developing loops, because this task (along with developing recursive procedures or functions) is the difficult part of writing relatively small programs.

Probably, the most common method taught for writing a loop is: (1) vizualize a repetitive process, (2) write out several repetitions of the process until one sees a pattern emerging, and (3) generalize the specific steps of the pattern into a single piece of code for the loop body. One problem with this method is that it requires us to vizualize the repetitive process before we begin, and that is often difficult or even impossible to do. Many times, we know that iteration (or recursion) is needed, but we have absolutely no idea what form it will take until the problem is almost solved. Other times, we have an idea what the repetitive process will be like, but we don't know how to

formulate it simply, and generalizing based on seeing the repetitive pattern is not simple.

We propose an alternative method that is based on what has to be proven about the loop, as given by rule (6.3). This rule requires a loop invariant and a bound function, so let us discuss how we can find these, or at least an approximation to them, before writing the loop.

8.0 Finding (an approximation to) the loop invariant.

Consider the following facts about a loop invariant P :

(0) Using rule (6.3) for a loop within a program requires that P be true initially, before execution of the loop. (This often requires initialization before the loop to establish P .)
(1) P is true upon termination of the loop, when the postcondition is true.
(2) Hence, P is true in more places than either the pre- or the post-condition.
(3) Hence, P can be viewed as a generalization of the precondition, the postcondition, or both, and one that is easy to establish initially.

Therefore, the basic method of developing a loop invariant is to generalize the precondition, the postcondition, or both in such a way that the generalization is easily established. More often than not, the postcondition provides the most information for this task, since it contains the most structure.

There are many ways to generalize a condition, or assertion; but two stand out as useful in many cases. (But do not expect to limit yourself to them.)

(8.0) **Method for generalizing a postcondition.** Replace a constant or expression of the postcondition by a fresh variable, typically putting suitable bounds on the variable.

As an example, consider the postcondition for program (6.0):

R: s is the sum of $1..n$.

Setting s to establish R is difficult, since the segment $1..n$ can contain many integers. However, setting s to the sum of 0 integers is easy. If we replace n in R by a fresh variable k with range $0 \le k \le n$, we arrive at the invariant

P: $0 \le k \le n$ **and** s is the sum of $1..k$,

which is easily established by $k := 0; s := 0$. This leads directly to the loop given in Section 6. (Note that we did not use the initialization $k := 1; s := 1$, since it fails to establish P in the case $n = 0$.) On the other hand, suppose we decide to replace the constant 1 of R by a fresh variable k . Then, we place bounds $1 \le k \le n+1$ on k so that $k..n$ can denote empty segment (i.e. when $k = n+1$) and use the loop invariant

P': $1 \le k \le n+1$ and s is the sum of $k..n$.

P' leads to a loop that accumulates the values of $1..n$ in reverse order, since P' is most easily established using $k := n+1; s := 0$.

(8.1) **Method for generalizing a postcondition.** For a postcondition of the form $R1$ **and** $R2$, delete $R1$ or $R2$.

Let us look at an example of the use of this method. Consider a program that is to set a variable i to the index of the first occurrence of x in an array segment $b[m..n-1]$, where $m \le n$, setting i to n if x is not in the array segment. This is known as a linear search. We can write the pre- and post-condition for the problem as follows:

Q: $m \le n$,
R: $m \le i \le n$ **and** x is not in $b[m..i-1]$ **and** $(i = n$ **or** $x = b[i])$.

The first conjunct $m \le i \le n$ is easily established by setting i to either m or n , the second by setting i to m , and the third by setting i to n . There is a conflict in establishing the second and third conjuncts easily, because establishing either one easily does not establish the other. Therefore, let us delete the third one and use the invariant

P: $m \le i \le n$ **and** x is not in $b[m..i-1]$.

As we will see later, this leads directly to the following program, where the loop guard is the negation of the deleted conjunct.

(8.2) $i := m$; **while** $i \neq n$ **and** $x \neq b[i]$ **do** $i := i+1$

Methods (8.0) and (8.1) are the major ones for finding a first approximation to a loop invariant. However, do not be so rigid as to impose only these two methods on yourself. From time to time, and with more experience, the shape of the pre- and post-conditions will provoke other methods as well. Important here is that the shape, the structure, the content of the pre- and post- conditions provide the insight. This makes sense. After all, where would we hope to find more insight than in the definition of the problem itself?

8.1 Developing a loop from a given invariant and bound function

We now show how rule (6.3) can be used to *guide* the development of a loop that satisfies

$\{Q1\}$ *initialization*; **while** B **do** S $\{R\}$

This is actually rather a straightforward idea, once one understands (6.3). Note that (6.3) concerns the loop in isolation, whereas now we are concerned with developinmg a loop and some initialization that truthifies the invariant.

Consider writing a loop, with initialization, given the following information:

$Q1$: $1 \le n$
R: s is the sum of $1..n$
P: $1 \le k \le n+1$ **and** s is the sum of $k..n$
t: $k-1$

Step 1. Develop initialization that truthifies invariant P . This is easiest to do if we make the range $k..n$ as small as possible (i.e. empty), and this is when $k = n+1$. Hence, we use the initialization $k := n+1$; $s := 0$.

Step 2. Develop loop guard B to satisfy $(P$ **and not** $B) \Rightarrow R$, so that R is true upon termination. In this case, it is easy to see from P and R that **not** B is $k = 1$, so that B is $k \neq 1$.

Step 3. At this point, we can verify easily that (6.3b) holds.

Step 4. Begin developing the loop body by finding a way to make progress towards termination (see (6.3c)). We introduce the obvious bound function k and choose the progress $k := k-1$.

Step 5. Modify the loop body, if necessary, so that invariant P remains true with each iteration (see (6.3a)). To determine what is to be done, we write the loop body as

$S1$; $k := k-1$

and attempt to find $S1$. The only other variable mentioned of the problem is s , and we see whether some assignment $s := e$ (for some e) will do the trick. Thus, we have to solve for e in

$\{P$ **and** $B\}$ $s := e$; $k := k-1$ $\{P\}$.

which, using the definition of P and B and rule (3.2) for assignment, is equivalent to solving for e in

$\{1 \le k \le n+1$ **and** s is the sum of $k..n$ **and** $k \neq 0\}$
$s := e$
$\{1 \le k-1 \le n+1$ **and** s is the sum of $k-1..n\}$

Hence, we have a subproblem to solve. The solution is $s := s+(k-1)$, and we have the program

$k := n+1$; $s := 0$;
while $k \neq 1$ **do begin** $s := s+k-1$; $k := k-1$ **end** .

At this point, we could transform the loop body into the simpler version $k := k-1; s := s+k$.

One might think that we could have more straightforwardly looked for a loop body of the form

$$\{P \textbf{ and } B\}\ k := k-1;\ S2\ \{P\}\ .$$

However, in general this is harder to do. Finding the precondition for $S2$ requires finding a postcondition for $\{P \textbf{ and } B\}\ k := k-1\ \{?\}$. While such a rule exists, it is so much more complicated than the simple textual-replacement rule of (3.2) that we do not give it here. A program is indeed executed from beginning to end, but this operational way of thinking is not the best for reasoning about the program.

We now summarize this method of developing a loop.

(8.3) **Method for developing initialization and a loop** (given invariant P and bound function t) to satisfy

$$\{Q\}\ \textit{initialization};\ \textbf{while } B \textbf{ do } S\ \{R\}\ .$$

Step 1. Create initialization that satisfies $\{Q\}$ *initialization* $\{P\}$.
Step 2. Find guard B by solving the equation $(P \textbf{ and not } B) \Rightarrow R$ for B, so that R is true upon termination.
Step 3. Prove $(P \textbf{ and } B) \Rightarrow t > 0$.
Step 4. Find a way to reduce bound function t, call it *progress*. (*progress* is often a simple assignment, like $i := i+1$.)
Step 5. If $\{P \textbf{ and } B\}$ *progress* $\{P\}$ holds, the loop body is complete. Otherwise, find a statement $S1$ for the rest of the body, so that the body is $S1$; *progress*. The body must satisfy $\{P \textbf{ and } B\}\ S1;$ *progress* $\{P\}$. Therefore, find a condition $P1$ that satisfies

$$\{P1\}\ \textit{progress}\ \{P\}$$

and solve the new problem $\{P \textbf{ and } B\}\ S1\ \{P1\}$.

There are a few points to make about this method of development. First, a caveat. Not all loops should be or even can be developed in this fashion. Depending on the situation, different ideas will arise and a different ordering of the steps may be more convenient. Also, at each point of development something may arise to require a change. For example, in step 3 it may be necessary to change invariant P in order to prove (6.3a). Neverthess, it is advantageous to think in terms of this process of developing a loop and to adhere to it as strictly as possible at the beginning in order to gain experience. In this way, one begins to see how correctness ideas can influence the development of a loop.

The first author remembers quite vividly trying to teach the development of a loop as early as 1966 and having almost nothing to say about the topic. It was a black art. He ended up inventing the old method of writing down a series of similar statements to represent the idea of repetition and then trying to generalize into a single statement, the body of a loop, but it was unsatisfactory. Now, it is an easier task, for there is a method to use that works in many cases. With practice, the student gains experience; once the experience is gained, they can begin to break away from the rigorous method in various contexts, but still use the correctness-proof ideas in the development.

We now analyze the steps of method (8.3), one by one. Step 1 should be obvious. On the other hand, we fail to see how one teaches students how to "initialize a loop" without knowing what is to be established. The old repetitive-pattern method outlined in the introduction to Sect. 8 is rarely described in precise enough terms to be useful to the student in this regard.

Step 2 is rather striking. It used to be the case that we determined when a loop should terminate by determining how many iterations it should execute. But the latter method is flawed, because often we cannot easily determine how many iterations should be executed. The latter method is also more prone to errors because of the operational thinking that is going on. On the other hand, step 2 is usually quite simple to perform, for it concerns solving a formula —a formula that is completely in terms of a single state— something the student should know how to do. Usually, the shape of invariant P and postcondition R make the determination of **not** B quite easy, and then one just has to complement it to get B.

Step 3, proving $(P \textbf{ and } B) \Rightarrow t > 0$, is usually straightforward, although it may require the introduction of restrictions on the ranges of newly introduced variables (as will be seen later).

Consider steps 4 and 5 of (8.3). The loop body must maintain the invariant and must decrease the bound function. Which is most important in *beginning* the development of a loop? When asked this question, many people answer "maintain the invariant". However, the simplest way to maintain the invariant is to do nothing, so using this idea would result in an empty procedure body! This points out clearly that making progress towards termination is the key idea in beginning the development of the loop body, which runs counter to the usual programmer's ideas. They often think of termination as an afterthought, something to be verified after everything is complete, while here we see that it is an important consideration in beginning the development of the loop body. Thus, we see that thinking about correctness concerns changes our thoughts about the development process.

9. Examples of the development of algorithms

We present aexamples of the development of algorithms, some more detailed than others, in order to give the reader a feeling for the generality of the methods outlined thus far. We solve the two remaining example problems given in the introduction. As we progress, we will introduce a few extra points concerning the development process.

9.0 Linear search

Consider writing a program that finds the first occurrence of x in array segment $b[m..n]$, where it is known that x is in the segment. We specify this as follows. Given

Q: x is in $b[m..n]$,

store a value in i to establish

R: $m \le i \le n$ **and** x is not in $b[m..i-1]$ **and** $x = b[i]$.

We need a loop (or recursion), and we decide on a loop. We look for a suitable loop invariant. The first two terms of R are easily established (see step 1 of (8.3)) using $i := m$, while the third term is more difficult to establish. Therefore, we construct loop invariant P by deleting the third term. (Note how step 1, the need to establish the invariant, is used to help us find the invariant.) The bound function $n-i$ is fairly obvious, so we have

invariant P: $m \le i \le n$ and x is not in $b[m..i-1]$
bound function t: $n-i$

Step 1. As mentioned earlier, the initialization is $i := m$.

Step 2. The guard B of a loop has to satisfy $(P$ **and not** $B) \Rightarrow R$ (step 2 of (8.3)). Looking at P and R, we see that for **not** B we can use the deleted term $x = b[i]$, so B is $x \neq b[i]$.

Step 3. $(P$ **and** $B) \Rightarrow t > 0$ must hold. This is

$(m \le i \le n$ **and** x is not in $b[m..i-1]$ **and** $x \neq b[i]) \Rightarrow n-i > 0$.

This we can prove easily, using $i \le n$ and the fact that x is in $b[m..n]$, which was given to us as part of the problem description. (Strictly speaking, we should place the term (x is in $b[m..n]$) in the loop invariant. However, we often leave unmentioned in the invariant facts about variables that do not change; like mathematicians, we omit the obvious in order to reduce the number of details.)

Step 4. To begin writing the loop body, we look for a simple way to decrease bound function t; the statement $i := i+1$ will do.

Step 5. We look for a statement $S1$ that satisfies $\{P$ **and** $B\}$ $S1$; $i := i+1$ $\{P\}$. Using the assignment statement rule, we see that statement $S1$ should satisfy

$\{P$ **and** $B\}$ $S1$ $\{m \le i+1 \le n$ **and** x is not in $b[m..i+1-1]\}$.

Here, $(P$ **and** $B)$ implies the postcondition for $S1$, so we take the empty statement for $S1$, giving us the loop

$i := m$; **while** $x \neq b[i]$ **do** $i := i+1$.

9.1 Binary search

We now solve problem (0.0) of the introduction. The precondition is

$$Q\colon\ 0 \le n \ \textbf{and}\ b[0] \le x < b[n+1]$$

and our algorithm should store an integer in h to establish

$$R\colon\ 0 \le h \le n \ \textbf{and}\ b[h] \le x < b[h+1]\ .$$

We need a loop or recursion for this algorithm, and we decide on a loop. Our first task is to develop an invariant P. The term $0 \le h \le n$ is easily established by setting h to 0 or n. On account of precondition Q, the term $b[h] \le x$ is easily established by setting h to 0 and the term $x < b[h+1]$ by setting h to n. However, it is difficult to establish these two terms simultaneously. So we arrive at an invariant by breaking this interdependence by replacing $h+1$ in R by a fresh variable k to obtain P':

$$P'\colon\ 0 \le h \le n \ \textbf{and}\ b[h] \le x < b[k]\ .$$

(Bounds have to be placed on k; these will come later.)

Step 1. Establish P' using the initialization $h := 0;\ k := n+1$.

Step 2. To find B, we solve the equation $(P' \ \textbf{and not}\ B) \Rightarrow R$. In this case, we have $\textbf{not}\ B \equiv (h+1 = k)$, so that B is $h+1 \neq k$.

Further development requires a bound function. Variable k starts at $n+1$ ($\ge h+1$) and ends at $h+1$. Hence, we can take $k-(h+1)$ as the bound function.

Step 3. This step forces us to place bounds on k. Since initially $0 = h < k \le n+1$ and upon termination $h+1 = k$, it is reasonable to restrict k as given in the following revision of invariant P':

$$P\colon\ 0 \le h < k \le n+1 \ \textbf{and}\ b[h] \le x < b[k]\ .$$

At this point, it is obvious that $(P \ \textbf{and}\ B) \Rightarrow k-(h+1) > 0$ holds.

Steps 4 and 5. We look for a way to make progress within the loop body. The guard and invariant imply that there exists an integer e satisfying $h < e < k$, and setting either h or k to any such e reduces the bound function. We calculate the condition under which setting h to e maintains the invariant and then manipulate it, *under the condition that the invariant and loop guard hold*:

$$\begin{array}{ll} & pre(\text{“}h := e\text{”}, P) \\ = & \quad \text{«Definition of } P \text{ and assignment rule (3.3)»} \\ & 0 \le e < k \le n+1 \ \textbf{and}\ b[e] \le x < b[k] \\ \Leftarrow & \quad \text{«}P \text{ and } h < e < k \text{ imply the first term; } x < b[k] \text{ appears in } P \text{»} \\ & b[e] \le x \end{array}$$

Hence, a precondition for execution of $h := e$ to maintain P is $b[e] \le x$. Similarly, a precondition for execution of $k := e$ to maintain P is $b[e] > x$. Hence, the conditional statement $\textbf{if}\ b[e] \le x\ \textbf{then}\ h := e\ \textbf{else}\ k := e$ maintains the invariant and decreases the bound function, for any e satisfying $h < e < k$.

Finally, we are left with the choice of e. Remember that there exists an integer e that satisfies $h < e < k$, and any such e will do. We could use $e := h+1$, which would yield a linear search that starts at the beginning. We could use $e := k-1$, which yield a linear search that starts at the end. But we can also choose $e := (h+k)\ \textbf{div}\ 2$, which yields a search, called binary search, that cuts the bound function in half at each step, yielding an algorithm that is logarithmic in n. Thus, we end up with the algorithm

```
h := 0; k := n+1;
while h+1 ≠ k do begin e := (h+k) div 2;
                        {h < e < k}
                        if b[e] ≤ x then h := e else k := e
                  end .
```

Note that precondition $h < e < k$ of the conditional statement together with the invariant implies that the reference $b[e]$ is not to one of the virtual values $b[0]$ and $b[n+1]$.

This development and algorithm invites several remarks. First, nowhere was an argument used concerning the fact that the array segment was sorted. But note also that the algorithm is not guaranteed to find x; it is guaranteed to find only an index h satisfying $b[h] \leq x < b[h+1]$. If b is in ascending order, then we can conclude that the final value h is the rightmost position of x in b (if x is indeed in b) or the position where x belongs (if x is not in b). The fact that the sortedness of b is not required in the proof is interesting in its own right. It actually makes the proof simpler. Secondly, there are contexts in which binary search can be used with good effect on an unordered array segment.

The more conventional binary search algorithm does not present a meaningful result if x is not in b; further, it rarely works for the case that the array segment is empty. The binary search above provides a meaningful answer in all these cases. Further, the fact that it finds the rightmost occurrence of x, instead of an arbitrary occurrence of x, is useful in many contexts.

The more conventional binary search algorithm terminates as soon as it finds x in b. This is usually viewed as a time-saving device. But terminating as soon as x is found does require an extra test within the loop body —on the average 1.5 tests instead of 1 test will be performed at each iteration. Moreover, on the average, premature termination saves only about one iteration. Hence, the conventional binary search trades one iteration for an extra test within the loop, which makes the conventional one generally slower than the one developed here.

The development of our binary search, along with this analysis, should convince the reader that developing proof and program hand-in-hand can be more effective than the usual ad hoc approach to programming.

9.2 Table of cubes

Example problem (0.1) requested an algorithm to build a table of the first n cubes, without using multiplication. We can write the pre- and post-conditions as

Q: $0 \leq n$
R: for all i, $0 \leq i < n$, $b[i] = i^3$.

This example will be used to introduce a new technique, (9.0) below, to be used in developing the bodies of some loops. The technique as stated may seem rather general and perhaps vague. Yet, used consciously, it is indeed powerful. In reading (9.0), remember that "strengthening" means restricting to represent fewer states. This can be done, for example, by adding more conjuncts or adding variables and giving definitions for them.

(9.0) If the body of the loop is too inefficient or complicated, try to strengthen the loop invariant in some way in order to be able to eliminate the inefficiency.

We begin developing the algorithm. The obvious order in which to calculate the elements $b[i]$ is in order of increasing subscript. Hence, we will write such a loop. The loop invariant is found by replacing variable n in postcondition R by a fresh variable k and placing suitable bounds on k. For later purposes, we break the invariant up into two terms, $P0$ and $P1$ and also give the obvious bound function:

$P0$: $0 \leq k \leq n$
$P1$: for all i, $0 \leq i < k$, $b[i] = i^3$
t: $n-k$

Using the methods discussed thus far, we easily develop the following algorithm.

$k := 0$;
while $k \neq n$ **do begin** $b[k] := k^3$; $k := k+1$ **end**.

However, this algorithm uses multiplication (or exponentiation), which is not allowed according to the problem specification, so we have to transform it to eliminate the multiplication. We do this using technique (9.0). We eliminate the offending expression k^3 by introducing a fresh variable x with definition (a new term of the invariant)

$P2$: $x = k^3$,

and replacing k^3 within the loop body by x. At the same time, we have to ensure that $P2$ is indeed a loop invariant. $P2$ can be initially established using $x := 0$. Within the loop body, execution of $k := k+1$ will falsify $P2$, and in order to be sure it is reestablished we insert a statement just before $k := k+1$ that establishes $[k := k+1]P2$. This is simply the statement $x := (k+1)^3$. So we have the algorithm

$k := 0;\ x := 0;$
while $k \neq n$ **do begin** $b[k] := x;$
$x := (k+1)^3;$
$k := k+1$
end .

At this point, we don't seem to have made much headway. However, the expression $(k+1)^3$ can be rewritten to take advantage of the fact that $P2$ holds just before its execution, as follows:

$(k+1)^3$
= «Arithmetic»
k^3+3k^2+3k+1
= «Use $P2$ »
$x+3k^2+3k+1$.

Hence, we can rewrite the algorithm as

$k := 0;\ x := 0;$
while $k \neq n$ **do begin** $b[k] := x;$
$x := x+3k^2+3k+1;$
$k := k+1$
end .

We have eliminated a cubic term, which is some progress. Now, perhaps we can repeat the process, by introducing another fresh variable. We leave it to the reader to carry out the rest of the development in the same manner. After introducing two replacements using the definitions (i.e. invariants)

$P3$: $y = 3k^2+3k+1$
$P4$: $z = 6k+6$

we end up with the algorithm

$k := 0;\ x := 0;\ y := 1;\ z := 6;$
while $k \neq n$ **do begin** $b[k] := x;$
$x := x+y;\ y := y+z;\ z := z+6;$
$k := k+1$
end .

Knowing about finite difference tables might allow one to develop this linear algorithm without knowing about the methodology that we used in developing it. However, experience shows that the typical upper-level undergraduate or even first-year graduate student in computer science does not develop a linear algorithm for this problem —most develop a cubic or quadratic algorithm and are unable to reason about its correctness effectively. On the other hand, those familar with our methods of developing programs have little trouble with this problem.

9.3 A revision of logarithmic exponentation

The second author developed a different loop body for logarithmic exponentiation. With the same invariant and bound function (see (6.5)), to develop the loop body, he took $y := y$ **div** 2 as the way to make progress and wrote

$$
\begin{array}{ll}
 & \mathit{pre}(\text{``}y := y \textbf{ div } 2\text{''}, P) \\
= & \quad \text{«definition of assignment»} \\
 & 0 \le y \textbf{ div } 2 \textbf{ and } z*x^{y \textbf{ div } 2} = a^b \\
= & \quad \text{«}y \textbf{ div } 2 = (y - (y \textbf{ mod } 2))/2\text{»} \\
(9.1) & 0 \le y \textbf{ and } z*x^{(y - (y \textbf{ mod } 2))/2} = a^b
\end{array}
$$

Now note that substituting $x*x$ for x in (9.1) eliminates the fractional exponent, and this influences us to use the sequence of assignments $x := x*x;\ y := y \textbf{ div } 2$. Redoing our calculation of the precondition yields

$$
\begin{array}{ll}
 & \mathit{pre}(\text{``}x := x*x;\ y := y \textbf{ div } 2\text{''}, P) \\
= & \quad \text{«Rules of consequence and assignment and the above derivation»} \\
 & \mathit{pre}(\text{``}x := x*x\text{''}, 0 \le y \textbf{ and } z*x^{(y - (y \textbf{ mod } 2))/2} = a^b \\
= & \quad \text{«Rule of assignment, simplification»} \\
 & 0 \le y \textbf{ and } z*x^{y - (y \textbf{ mod } 2)} = a^b \\
= & \quad \text{«}y \textbf{ mod } 2 \text{ is } 0 \text{ or } 1\text{»} \\
 & 0 \le y \textbf{ and } ((\mathit{even}(y) \textbf{ and } z*x^y = a^b) \textbf{ or } (\mathit{odd}(y) \textbf{ and } z*x^{y-1} = a^b) \\
= & \quad \text{«Arithmetic»} \\
(9.2) & 0 \le y \textbf{ and } ((\mathit{even}(y) \textbf{ and } z*x^y = a^b) \textbf{ or } (\mathit{odd}(y) \textbf{ and } z*x*x^y = a^b)
\end{array}
$$

Now note that $\mathit{even}(y)$ **and** P implies (9.2), while, in the case $\mathit{odd}(y)$ **and** P, (9.2) can be established using $z := z*x$. Hence, we are led to the following loop with initialization, which saves up to half of the evaluations of the loop condition (over the original exponentiation algorithm).

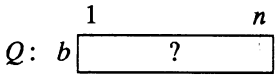
```
z := 1; x := a; y := b;
while {P} y ≠ 0 do begin if odd(y) then z := z*x;
                         x := x*x;
                         y :=y div 2
                   end .
```

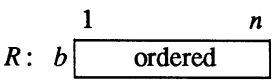
9.4 Memorizing algorithms —three simple sorting algorithms

Students are often asked to memorize some simple sorting algorithms, such as selection sort and insertion sort. This can be a difficult task if the student is given no method for memorizing such algorithms, except to memorize the code itself. However, the task is easy if one knows how to develop algorithms as outlined in this paper. Then, instead of memorizing the code, one simply develops it whenever it is needed.

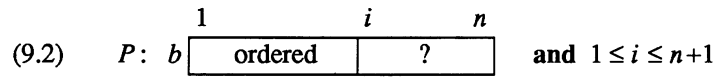
We illustrate this idea using three sorting algorithms. We will use diagrams of array segments instead of formulae in order to show that diagrams can be used as well. We do not suggest that diagrams be used, actually, but we know that students may have trouble with formulae and may feel more comfortable with diagrams, and we want to be sure that the reader understands that the methods proposed are almost independent of the notation used for expressing specifications.

Algorithm Insertion sort sorts an array $b[1..n]$, where $0 \le n$. The precondition Q and postcondition R can be specified using diagrams as follows, where "?" denotes a segment of values whose order is unknown:





As mentioned in Sect. 8.1, a loop invariant can be viewed as a generalization of Q and R. A generalization of the two has to have both an unknown segment and an ordered segment. One possibility, then, for the invariant is the following, where we have introduced a fresh variable i to mark the position that follows the end of the ordered segment.

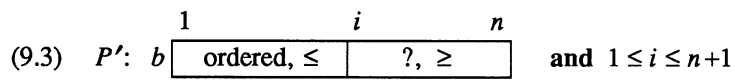


(The upper bound for i is $n+1$ because P **and** $i = n+1$ implies R, but P **and** $i = n$ does not imply R.) With this invariant, it is straightforward to write down the following algorithm:

```
i := 1;
while i ≤ n do begin  Sort b[1..i], given that b[1..i−1] is ordered;
                      i := i+1
                end.
```

It is now a simple task to refine the English statement at the beginning of the loop body so that it moves the value initially in $b[i]$ down to its correct position. (We leave the details to the reader.) At each iteration, $b[i]$ is being *inserted* into its correct position; hence the name for the algorithm.

Selection sort. Let us change invariant (9.2) slightly as shown below (keeping Q and R unchanged), where the additional notation means that all values of the first segment are no larger than those of the second:



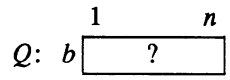
P' is the invariant of the following algorithm, which is known as Selection sort:

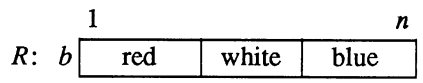
```
i := 1;
while i < n do begin Store in k the position of a smallest element of b[i..n];
                     Swap b[i] and b[k];
                     i := i+1
                end.
```

where the two statements that are written in English can be refined or left as they are. ((9.3) is also the invariant of one version of algorithm Bubble sort.)

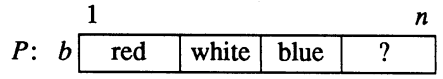
In each of these cases, the student who understands the method of program development has no difficulty "memorizing" the algorithm; the invariant is enough to keep in one's head, because the algorithm flows from it.

The Dutch National Flag. As a final case, suppose we are sorting an array $b[1..n]$ that has at most three different values, say red, white and blue balls. We can write the precondition and postcondition as follows:



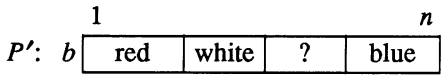


That is, the red balls should be first, then the white, and then the blue. We can write an invariant for a loop to sort the balls by color by generalizing the pre- and post-conditions as follows,



where for the moment we have omitted variables that describe the boundaries of the four segments. Here, when the first three segments are empty, we have the initial condition, and when the last segment is empty we have the final condition. Each iteration of a loop will place one element from the last segment of unknowns into one of the other segments, by performing 0, 1, or 2 swaps.

The algorithm using the proposed invariant will perform up to two swaps per iteration, and we look for a modification of the algorithm that will perform fewer in the worst case. With the given invariant, nothing better can be achieved. But consider the following invariant, in which the segment of unknowns has been moved before the segment of blues:



Now, if at each iteration the leftmost unknown is put in place (depending on its color), we have an algorithm that uses at most one swap per iteration. It remains only to introduce three fresh variables to mark the boundaries between the segments and to write the loop; we leave this to the reader.

The point to make here is that we are able to discuss algorithms without writing and reading code, because we have a standard way of dealing with loops, which is based on proof-of-correctness concerns. By introducing formal notions of correctness, we have greatly enhanced our ability to reason about programs.

10. Remarks on remembering, presenting, documenting, and grading algorithms

Remembering and presenting algorithm

Anyone involved deeply in programming will spend some time presenting algorithms to others. Sometimes this occurs in a one-on-one situation, sometimes before a larger group. For instructors of programming courses and courses in data structures, this is a frequent and important task.

In designing the presentation of an algorithm, one is trying to make the presentation as simple as possible, so as to minimize the time needed to understand the algorithm and to maximize that understanding. But there is another point to consider: the time required to prepare the presentation. This time is most important for instructors who have to prepare daily for lectures.

Too often, we have seen lecturers stumble through the presentation of some intricate program, using an operational style of reasoning. They refer to their notes often, directly copying pieces of code from their notes onto the blackboard. Their style suggests to the audience that they haven't really *mastered* the algorithm and its presentation. It simply does not work well.

We believe that *presenting a derivation of the algorithm*, using the techniques outlined in this paper and analyzed in detail in references [1], [2], [4], can be extremely effective on all counts. The general prerequisites for doing this are, of course, mastery of this style of program derivation. However, once that mastery is obtained, the process of preparing an algorithmic presentation is greatly simplified; one has only to commit to memory the program specification and one or two important points concerning the derivation. Then, in the lecture, one simply *derives the algorithm afresh.* For many algorithms, no notes are needed at all!

For example, consider presenting algorithm binary search, whose purpose is to store in h a value to establish

$$R\colon\ 0 \le h \le n \ \textbf{ and } \ b[h] \le x < b[h+1]\ .$$

where we have virtual values $b[0] = -\infty$ and $b[n+1] = +\infty$.

What must one remember in order to present this algorithm, besides the specification? Two simple facts: (0) the loop invariant is determined by replacing $h+1$ in R by a fresh variable k so that the invariant is easily established, and (1) in developing the body of the loop, progress can be made by setting h or k to any value e satisfying $h < e < k$. With these facts in hand, during the presentation one simply *derives* the algorithm, as discussed in Sect. 9.1.

Remark. For binary search, it can be exciting to develop the algorithm without mentioning the virtual values $b[0]$ and $b[n+1]$, assuming instead that $b[1] \le x < b[n]$, and *after* the presentation, asking the students how they would change the specification, and subsequently the algorithm, to remove the restriction $b[1] \le x < b[n]$. Most will invent a three-case analysis for the specification: $x < b[1]$, $b[1] \le x < b[n]$, and $b[n] \le x$, and this case analysis will then find its way into the algorithm. It is that this point that one mentions the need to avoid case

analysis like the plague and introduces the virtual values for this purpose, leading to an algorithm with no case analysis. There are *many* such techniques to be learned about presenting programs and teaching programming. Unfortunately, this paper cannot discuss them all. □

The first author has found this method of preparing for the presentation of algorithms and algorithmic concepts invaluable. Preparation time is minimal. The method leads to a more dynamic, interesting lecture, with the audience taking part (at the suggestion of the lecturer from time to time) in deriving the algorithm. The algorithmic ideas are presented far more precisely than could be done otherwise, and the ideas are grasped more easily by the audience.

The first author has given three-day courses and has participated in five-day courses on these methods for developing algorithms, in which essentially no notes were used during the lectures. The algorithms presented ranged from binary search to iterative inorder traversal of a binary tree, to the Schorr-Waite algorithm for marking the nodes of a binary directed graph, to transitive reduction of a relation. This would have been impossible without our method of derivation of programs.

In summary, then, we urge the teaching of imperative programming in this manner not only because it appears to be a more effective programming method but also because it allows the instructor to be far more effective in their lecturing, with less preparation time.

To good effect, the first author also requires students in his programming course to memorize various algorithms of central interest —logarithmic exponentation, selection sort, the partition algorithm of quicksort, quicksort itself, and so on, so that the students can repeat them on a test. Of course, the students are told to memorize not the code, but only the specification and one or two important points about the development, and to redevelop the algorithm whenever they need it. This forces them to practice the development on algorithms they have seen developed already, thus providing reinforcing practice with the method.

Documenting algorithms and programs

Every so often, another experiment is performed to determine effective documentation methods for programs. Groups of students or practicing programmers are given the same program but documented in a different fashion (or not documented at all) and asked to find errors in it or to modify it to reflect a change in specification.

In our opinion, such experiments have been utterly useless until now, since none of the methods of documentation used in the experiments have been based on principles of program correctness. Unless one understands what it means to prove a program correct, how can one determine what information an educated programmer needs to understand a given program?

Based on the methods we outlined for developing and understanding programs, we lean to the following minimal documentation for a sequential program, which forms the basis of a proof outline of the program.

- A rigorous specification is needed, in terms of a precondition Q and a postcondition R.
- If the program is a sequence of assignments, little else is needed, because proving $Q \Rightarrow pre(S, R)$ will be simple. If there are difficulties in this proof, then the proof should be given or at least the subtleties explained).
- If the program is a conditional statement, the precondition and the if-condition can be used to form the preconditions for the if-part and then-part, and the postcondition of the conditional statement is the postcondition of the then-part. Hence, nothing is needed in the way of documentation for the conditional statement itself, other than its specification, although one may need documentation for the then-part and else-part.
- If the program is a loop (with initialization), one needs the loop invariant and bound function that are used in proving it correct, as outlined in Sect. 6. One could also give hints as to how the loop invariant was obtained, thus allowing the reader to see hints as to how the algorithm was developed. (Note that the loop invariant and loop guard automatically give the precondition and postcondition for the loop body.)

Naturally, along with this proof outline should be a discussion of any subtle or difficult points of the proof.

In some programs, some variables have a global function, in that their values are referenced by all parts of the program. Their declarations should be grouped by logical relation (and not by what type of variable they are) and should be accompanied by a statement that defines them. Basically, this is a true-false statement that is true at all the "important" places of a program. The purpose of the program is, generally, to make progress in some fashion

while maintaining this definition.

Documentation is often difficult to extract from a programmer. One probable reason for this is that the programmer does not *use* their own documentation to develop the program; the documentation is simply something extra that has to be created for others. If we can convince programmers of the viability of writing the documentation hand-in-hand with the program, with the documentation leading the way, and if we can convince them that such a practice actually is more effective, then documentation will no longer be a problem.

Grading algorithms

One of the most frustrating and time-consuming tasks for instructors has been grading an exam question that deals with the development of a loop (or, in general, a program). With 50 students in a class, one could get 50 different designs, each without documentation, and understanding and grading in a consistent manner is difficult.

However, if the students are expected to provide documentation, for example, by annotating each loop with a loop invariant and bound function, the task becomes easier: simply use 20% of the credit for each of the five points concerning the correctness of the loop and grade the five parts independently.

The first author uses this method repeatedly on tests in order to impress on students the need for consistency between program and its documentation, a fact that, for some odd reason, is hard for them to grasp. They are asked to write a loop (with initialization), given the specification, the invariant, and the bound function. They are told explicitly that each of the five points concerning loop correctness will be given equal weight. There is always a group of students that dismiss the given invariant entirely and write the loop the way they want. They usually end up getting zero on the question, because they simply were not trying to solve the stated problem.

11. Conclusion

In outlining a theory for proving programs correct and a methodology of deriving programs that stems from the theory, we have only scratched the surface. The reader has to see many more examples of program developments, from many different areas. The reader has to *practice* the methodology on many examples, in order to become familiar with it, to really understand it. Without such practice, one simply *cannot* judge the method fairly.

The reader should also be shown how the method extends to assignment to arrays, to pointers, to procedures and functions, including recursive ones, and to the development and proof of implementations of "abstract data types". A paper this size is, however, no place for so much material.

We do hope that the reader has begun to see how experience with this material can change radically one's view of programs and the programming task. Programming becomes more calculational, with the various formulae that are involved helping to provide the insight for the next step of the development. It should also be clear that knowledge of this material can change radically how one presents algorithms. This holds not only for the presentation of typical algorithms that one presents in a class on data structures, or for the documentation of the typical programmer's work, but also for new algorithms that appear in our computing journals. As an editor of several of these journals for over 15 years, the first author has helped simplify and shorten many algorithmic presentations in journal articles through the use of the techniques discussed in this paper. It is because of the wide range of applicability of this material —be the application formal or informal— that we believe that *every* undergraduate computer science major should become proficient in this material.

The problem, however, is that teaching this proficiency calls for a different style of teaching. First, we should teach the use of the predicate calculus as a calculational tool, to be used wherever applicable in mathematical work, and not simply as an object of study. Second, we have to teach method, and not just facts. Third, we have to begin instilling a sense of discrimination and judgement, for example, a sense of how one compares two different methods or two notational styles or two proof methods, and a sense of how one decides which is more appropriate in a given context. We have to emphasize the search for brevity and conciseness and even beauty.

The problem is that none of our introductory texts —in either programming or discrete mathematics— has the necessary orientation. Our hope is that the next decade, 1991-2000, will see the field turning in this direction.

References

[0] *1990 Advanced Placement Course Description in Computer Science*. College Entrance Examination Board, New York 1989.

[1] Dijkstra, E.W. *A Discipline of Programming*. Prentice Hall, New Jersey, 1976.

[2] Dijkstra, E.W., and W. Feijen. *A Method of Programming*. Addison Wesley, Reading, 1988.

[4] Gries, D. *A Science of Programming*. Springer Verlag, New York, 1981.

[5] Hoare, C.A.R. An axiomatic basis for computer programming. *CACM 12* (Oct 1969), 576-580, 583.

[6] Pattis, R.E. A survey of texbook errors in binary searching. Dept. of Computer Science, Univ. of Washington, 1988.