CHAPTER I

Introduction

Does this computer program do what it is supposed to do? Is this program correct? Does it contain any errors? These are three of the possible ways of phrasing a question for which many people seek a reliable answer. That not everyone demands a good answer to this question is evidence that many computer programs (such as word processors and spelling checkers) have substantial value even when they contain errors. But when errors' consequences to people are injury or death (as in software controlling aircraft or medical equipment [41, 27]), many more people are inclined to demand a good answer to the correctness question.

A program's *behavioral specification* (which we henceforth abbreviate to just "*specification*") states what it is supposed to do. Answering the correctness question means deciding the truth of the *correctness conjecture* that *the program always behaves according to its specification.* We say the program is *correct* when this conjecture is true.

The discipline of mathematics gives us two ways of approaching the problem of deciding the truth of a conjecture. One way is to provide a counter-example, which establishes the conjecture to be false. The other way is to provide an extremely careful argument, or proof, that the conjecture is true. For the correctness conjecture, the search for a counter-example is called software *testing.* Testing usually is used to establish the presence of one or more defects in a program by finding an instance in which the program's behavior does not meet the specification. That a battery of tests has failed to reveal a defect may raise confidence in the correctness of the program, depending on the quality of the battery of tests. However, unless this battery includes every possible input to the program, its failure to reveal any defects is not proof that the program is correct.

When we want to establish the truth of the correctness conjecture with a level of confidence that approaches certainty, we turn to the other mathematical approach: providing an extremely careful argument, or proof, that the conjecture is true. To do this, we must reason very carefully about the behavior of computer programs.

Apart from employing proof in quality assurance efforts, there is another justification for the importance of reasoning about the execution-time behavior of computer programs: This reasoning is an essential part of a programmer's task. The programmer must discover some sequence of statements that, when executed, will accomplish a given objective. He/she thinks about the effects that would be caused by executing the various candidate statements, seldom resorting to the method of trial and error without giving some measure of thought. Although this reasoning is usually invisible, occurring only in the programmer's head, it is among the most important values he/she adds to the production of software.

The major purpose of this dissertation is to provide a good foundation for the reasoning that programmers do. We must note, however, that paradigms and languages for programming vary along a few dimensions. There are concurrent (or parallel) and sequential languages, declarative (for example, functional programming and logic programming) and imperative (or procedural) languages, and languages that do or do not include explicit constructs for non-determinism. We focus our attention here on just one kind of language: a sequential, imperative language having constructs for procedure declaration, specification, and call. Our language of focus also has constructs for selection and iteration, but has no explicit construct for non-determinism. However, we permit a procedure's specification to define a relation, not insisting that it be a function. We adopt a semantics in which a procedure call's outputs are a function of its inputs—not a relation. The semantics deem a procedure to be correct if the function it computes satisfies the relation it is supposed to compute; and they deem a client procedure to be correct if, with any correct implementations of the external procedures it calls, the function it computes satisfies the relation it is supposed to compute. In this sense, the results of calling an external procedure may not be completely determined, being constrained only by the procedure's (relational) specification.

Another dimension to consider when confining the scope of this work within manageable limits is the fact that execution of a given program from given input may not terminate [31]. A program is correct, of course, only if it always terminates with results satisfying its specification whenever its execution is started with input permitted by the specification. However, following Hoare's lead [18, pp. 578–9], the tradition of correctness proof separates the question of termination from that of *partial correctness*. A program has the property of *partial correctness* if and only if the program behaves according to its specification whenever it terminates [39, pp. 194–5]. In other words, partial correctness guarantees that either the program will fail to terminate, or it will behave according to the specification [4, p. 75]. In this tradition, one proves

the *total correctness* of a program by proving its partial correctness and constructing a separate proof of termination by other means [39, p. 195]. This dissertation establishes a good foundation for the partial-correctness portion of programmers' reasoning.

1.1 How People Want to Reason About Program Behavior

Formal bases for methods of reasoning about the behavior of sequential programs already exist [34, 37, 13, 18, 19, 20, 6, 30, 32, 23, 5, 46, 29]. This dissertation's contribution hinges in part on the plausibility of the proposed reasoning method being more natural than existing methods. We consider a reasoning method to be more *natural* (than a competing method) if it is closer to the ways most computer professionals usually use, or would like to use, when they reason about programs' behavior.

The issue of program comprehension is the focus of a growing body of research. Littman et al. [28] named two comprehension strategies: Programmers used either a *systematic* or an *as-needed* strategy.

> Programmers who used the systematic approach to study the program constructed successful modifications; programmers whose used the as-needed approach failed to construct successful modifications. Programmers who used the systematic strategy gathered *knowledge about the causal interactions of the program's functional components.* Programmers who used the as-needed strategy did not gather such causal knowledge and therefore failed to detect interactions among components of the program. [28, p. 80]

Littman et al. focused on the consequences of using one or the other of these strategies, and reported which strategy a manager might well prefer her/his programmers to use, or (to be fair) which strategy a programmer—interested in constructing successful modifications on the first try—might well adopt. A later study that cites Littman et al. chose a different focus. Koenamann and Robertson's [24] goal was to discover which strategy programmers used. Their data suggests that the two strategies' use may not be equal among programmers. "Here we show that subjects follow a pragmatic 'as-needed' strategy rather than a systematic approach, that subjects restrict their understanding to parts of the code they consider to be relevant for the task and, thus, gain only a partial understanding of the program that might lead to misconceptions or errors." Programmers "use bottom-up comprehension only for directly relevant code and in cases of missing, insufficient, or failing hypotheses." "Tools will

have to be developed that facilitate 'as-needed' strategies and help programmers to avoid some of its inherent problems." [24, p. 125]

These two studies differ on the question whether to support programmers' strategies with tools or to change programmers' choice of strategies. Perhaps this difference caused the authors to ask different questions, pursue different methods, and draw different conclusions. Conclusions do vary in the growing—but still young—body of research on the issue of program comprehension. Despite their differences, these two studies both found that a significant number of subjects chose an as-needed strategy rather than a systematic strategy. Both studies examined subjects who were experienced, expert programmers. Therefore, it is plausible that a significant portion of practicing expert computer programmers frequently pursue an as-needed strategy in preference to a systematic strategy for program comprehension. Based on this idea, we make the following two specific claims:

1. When faced with a program consisting of several statements, the computer professional usually reasons independently about different statements, or small groups of statements, before assembling these separate parts into an argument about the entire program. She/he typically does *not* start with the proposed postcondition at the last statement, and step backwards through the program, statement by statement, constructing a mathematical assertion by iteratively substituting formulas into the growing assertion [23], which is the informal counterpart to Hoare logic. Nor does she/he typically employ the informal counterpart of symbolic execution, which requires building the variables' symbolic values from the top of the program [5]. Maurer [33, p. 428] called symbolic execution the *forward accumulation method*; the *back substitution method* [33, p. 429] was his name for the application of Hoare logic. These latter methods are systematic, rather than as-needed. But neither the above-cited studies nor any other we know of gives evidence that the systematic methods actually used by programmers are anything like forward accumulation or back substitution.

2. When reasoning informally about the behavior of a program, the computer professional usually deals with the program's source text, possibly annotating it, but usually does not write a list of the program's execution paths. Such a list is another systematic, not an as-needed, strategy. But, again, there is no reason to believe it is actually among observed systematic strategies.

1.2 Formal Bases for Reasoning

A familiar experience in reasoning is reaching a conclusion that is later contradicted. What went wrong? Did I make a mistake in applying my reasoning method? Did I apply the method faithfully, but produce an error because I assumed a false premise? Or, based on premises all true, did I faithfully apply a method of reasoning that was, itself, faulty? It is difficult to decide whether a method of reasoning is faithfully applied, or whether the method itself is faulty, unless that method is clearly and unambiguously established. The purpose of a precise, formal definition of a reasoning method is to provide such clarity. We say a method of reasoning has a *formal basis* if it is formally defined.

The past century has seen profound success in providing formal bases for much of the reasoning used in mathematics. This success has been tempered with wisdom gained from results which reveal some inherent limitations of formal systems of logic. Despite these limitations, or, perhaps, because of the power displayed in their being discovered, the formalization of mathematics provides an excellent model to follow in attempts to provide formal bases for reasoning in other fields.

Important to the success of these formal bases for mathematical reasoning is that they are systems of purely syntactic manipulation. Although the subjects of reasoning in mathematics are the meanings of mathematical entities, the subjects of formal proof in a formal basis for mathematics are strings of symbols, or formulas. The rules for what constitutes a formal proof are expressed in terms of the order and arrangement of the symbols, without regard to their intended or possible meanings.

A formal system defines one or more *rules of inference* for constructing one formula from other formulas. It also establishes a distinguished set of formulas called *axioms*. A formal proof is a sequence of formulas in which each formula either is an axiom or is constructed according to a rule of inference from formulas preceding it in the sequence. Each formula in a proof is a formal theorem. In particular, the last formula in a such a proof is a formal theorem, and common usage focuses on the last formula as being the theorem of interest. Clearly, all axioms are theorems.

To achieve clarity without ambiguity, each rule governing a formal proof system is expressed independently of any meaning; however, for such a system to be useful as a basis for reasoning in some field, there must be a way to re-attach it to meanings in the field. This re-attachment is done by associating each formal symbol with an object in the field of interest. If the association of symbols with objects results in a meaning for each axiom that is true in the field of interest, then the association is called a *model* [8, pp. 79–84] of the formal system having that set of axioms.

The usefulness of a formal system depends crucially on its possessing a property called *soundness*. If, in every model of a formal system, each formal theorem is true in the model, then the formal system is *sound* [8, p. 124]. Soundness makes formal proof and its theorems interesting; each formal theorem is necessarily true in any model of a sound formal system. What is more, mathematicians have proven some useful formal systems to be sound. Although it is known that proof cannot establish the soundness of several even more useful formal systems, their soundness is assumed because it has not been contradicted in many years of use.

A formal system can, of course, achieve soundness by having few or no formal theorems. It is easier, then, not to have a formal theorem that is false in some model of the system! Such a system, however, has very little use. So soundness does not, by itself, confer usefulness on a system. We prefer systems to have as many formal theorems as possible. On the other hand, if too many formulas are permitted to be formal theorems, then some of these might be false in some model, rendering the system unsound. A system, in which every formula that is true in every model is also a formal theorem, would be considered to have "enough" theorems. We call such a system *complete* [8, p. 128]. As a characteristic of formal systems, completeness is the dual of soundness. A complete system does not have too few formal theorems; a sound system does not have too many. There are just the right number of formal theorems in a system that is both sound and complete.

In 1931, Gödel [15] showed that any formal system rich enough to express the notion of natural numbers cannot be both sound and complete. He also showed that the soundness of such a system could not be proved within the system itself; therefore, any proof of soundness for the system must employ a more complex logic, whose soundness we should, therefore, question even more than the soundness of the original system. However, mathematicians have heavily exercised some formal systems rich enough to express the notion of natural numbers without discovering unsoundness in them. So they assume these systems to be sound, and, therefore, not complete. These systems, however, do have many formal theorems. When we speak of the incompleteness of mathematics, we are referring to the fact that any sound formal basis for mathematics must be incomplete.

It is in this setting that research beginning with that of Floyd, Hoare, and Dijkstra [13, 18, 19, 20, 6] succeeded in establishing a formal basis for reasoning about the behavior of computer programs. Ideas expressed by Abelson and Sussman [1, p. xvi] help us understand the nature of this formal system.

> The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called procedural epistemology—the study of the

> structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of "what is." Computation provides a framework for dealing precisely with notions of "how to."

Observe that notions of *how* to accomplish something make sense only in relationship to *what* that something is. Recall that a program's behavioral specification states *what* the program is supposed to accomplish. Therefore, the language of choice, when stating specifications precisely, is (classical) mathematics.

The existing formal basis for reasoning about the behavior of computer programs, like the formal bases for mathematics, is a system of purely syntactic manipulation. We can use the rules of this formal basis to transform a formula, consisting of a program with a specification, into a formula of classical mathematics. That is to say, the rules can be used to transform a formula dealing with notions both of "how to" and "what is" into a formula dealing only with notions of "what is." The transformation occurs in many steps, each step justified by a rule of the formal system. The idea of this system is that if the final purely mathematical formula is true, then the correctness conjecture for the original program and specification is also true.

We know this idea is correct because the formal system has been shown to be sound. Furthermore, this soundness is not trivial; researchers also have shown this formal system to be what is called "relatively complete"—that is, complete with respect to the truth of the purely mathematical formulas (see, for example, [4]). Recall, however, that, because mathematics is incomplete, not all true mathematical formulas are provable. Therefore, because the formal system for dealing with the correctness conjecture is built on the formal system(s) for mathematics, it is not a truly complete formal system, but it is complete relative to the incompleteness of mathematics [4, pp. 85–6]. Any incompleteness observed in it is due not to any flaw in the proof rules regarding computation but to the incompleteness of mathematics itself.

In summary, because the system is sound, the correctness conjecture is true for every program/specification pair that can be transformed to a mathematical statement that is true. Because the system is relatively complete, every program/specification pair satisfying the correctness conjecture is transformable to a mathematical assertion that is true. Therefore, reasoning about programs has a solid formal basis. The problem of correctness has been reduced to the problem of mathematical truth.

1.3 The Problem

Thanks to the good work of Floyd, Hoare, Dijkstra, and those who followed them, there now exists a good foundation for reasoning about the behavior of computer programs. Unfortunately, as will be shown in Section 1.5, this traditional formal

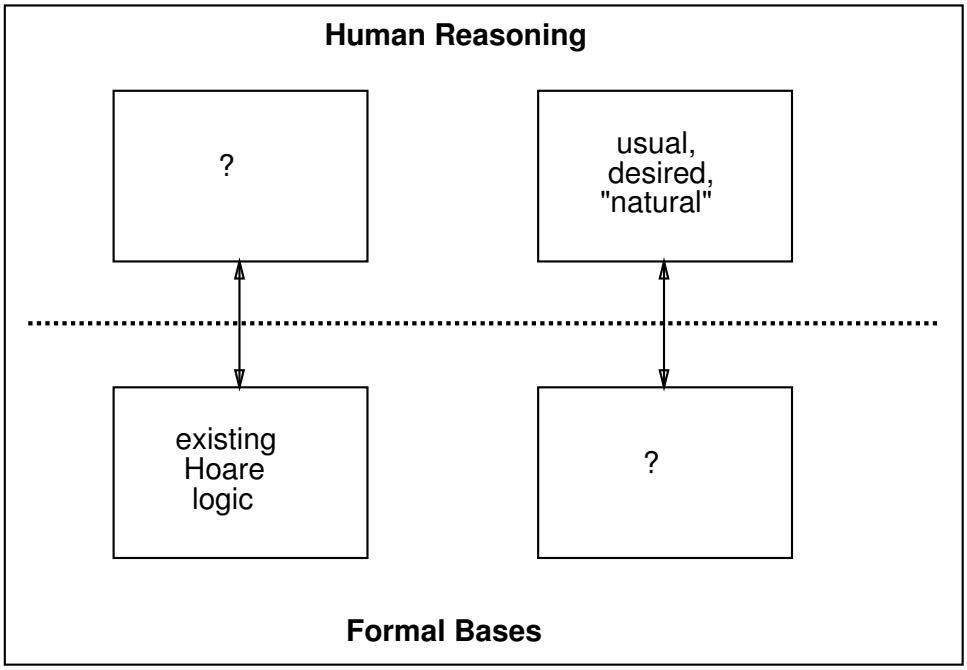

Figure 1: Human Reasoning and Formal Bases

basis does not match closely with the natural ways, described in Section 1.1, that computer professionals usually use, or would like to use, when they reason about programs' behavior. Figure 1 uses question marks ("?") to portray two facts:

1. There does not exist a formal basis for the usual, desired, "natural" way of reasoning about program behavior.

2. Reasoning that matches the traditional formal basis is not widely practiced.

There are, then, two ways to have a formal basis for widely-practiced ways of reasoning about program behavior. One is to establish a mission to teach computer professionals to reason according to the existing formal basis. The other is to provide a formal basis for a method of reasoning that is more natural—closer to the methods used or desired

today. Dijkstra and his disciples have chosen the former option; here we pursue the latter.

1.4 The Thesis

This dissertation defends the following three-part thesis:

1. The traditional formal method of reasoning about the behavior of programs is not natural.

2. There is a sound formal basis for (the partial-correctness portion of) a more natural reasoning method.

3. The soundness of this new formal basis is strong in the sense that the method is also logically complete (relative to the (in)completeness of the mathematical theories used in the program's behavioral specification and explanation).

Part 1 of this thesis is supported in Section 1.5 by using an example to compare the traditional formal method of reasoning with the new method we propose—a method we call the *indexed method* for reasoning about program behavior. We discuss our example in the light of literature concerning empirical studies of programmers, showing the indexed method to be more natural. Parts 2 and 3 are supported in subsequent chapters by defining the target language and the formal basis for the indexed method, and proving this basis sound and relatively complete.

1.5 Traditional Formal Reasoning Is Not Natural

In the context of an example program-with-specification, we describe here how reasoning is performed according to the traditional formal method, the back substitution method. We then sketch how reasoning about the same program/specification pair would be done in the indexed method, a way that is more comfortable to today's computer professionals, according to our claims in Section 1.1.

Our example program/specification pair arises out of a three-part problem statement:

1. Specify a procedure that has four formal parameters, a, b, c, and max, all of type Integer, such that the maximum among a, b, and c becomes the value of max.

procedure Set_Maximum (a, b, c, max : Integer)
    requires true
    ensures (max = a ∨ max = b ∨ max = c)
            ∧ max ≥ a ∧ max ≥ b ∧ max ≥ c

Figure 2: A Specification of Procedure Set_Maximum

2. Provide a sequence of statements to compose the body of this procedure.

3. Show that the procedure body meets its specification.

Figure 2 shows a possible specification of this procedure, calling it "Set_Maximum." We specify procedures by prescribing conditions on the parameters' values. The condition in the **requires** clause is called a *precondition*; it states what may be assumed about the formal parameters at the beginning of the procedure body. Therefore, from the point of view of a client that may call the procedure, the precondition on the corresponding actual parameters is a necessary condition for such a procedure call to be legal. Figure 2 specifies the weakest possible precondition: **true**. We may assume nothing special about the values of the formal parameters at the beginning of the procedure body, except that all four are integers. Stated another way (from the client's viewpoint), the values of the actual parameters in a client have no bearing on the legality of a call to this procedure. When a precondition is **true**, we may abbreviate the specification by omitting the clause "**requires true**."

We call the condition in the **ensures** clause a *postcondition*; it states what the procedure body must make true of the values of the formal parameters upon completion of the body's execution. Therefore, a client that has legally called this procedure may assume that the postcondition is true for the corresponding actual parameters after the call. Figure 2 specifies that, at the procedure's conclusion, max's value must equal that of at least one of a, b, and c, and that the value of max must be at least as great as the value of a, b, and c.

Part 2 of the problem asks that we provide a sequence of statements that we believe will always behave according to the procedure's specification. Figure 3 shows procedure Set_Maximum with its abbreviated specification (no **requires** clause) and a body containing a sequence of statements.

```
procedure Set_Maximum (a, b, c, max : Integer)
    ensures (max = a ∨ max = b ∨ max = c)
                ∧ max ≥ a ∧ max ≥ b ∧ max ≥ c
begin
    max := a
    if (b > max) then
        max := b
    end if
    if (c > max) then
        max := c
    end if
end Set_Maximum
```

Figure 3: An Implementation for Procedure Set_Maximum

1.5.1 An Example of Traditional Reasoning

Next we answer the problem's part 3, showing, according to the traditional back substitution method, that this procedure body meets its specification. Figure 4 shows that we begin by surrounding the precondition and postcondition with braces ("{" and "}") and writing the results, respectively, before and after the statement sequence. (These braces are not to be confused with set notation.) The traditional method works from the back of the sequence toward the front. We first alter the postcondition to remove the last statement.

The last statement is an **if-then** statement. We must concern ourselves both with the Boolean condition, "c > max," and its negation, "c ≤ max." Under the negated condition, execution skips the statement's body, so here we do not alter the postcondition. However, to handle the condition "c > max," we do alter the postcondition according to the statement's body, which here consists of the single assignment statement "max := c." The back substitution method has us replace every occurrence of "max" in the postcondition with "c." Figure 5 shows the new postcondition at the end of the statement sequence now shortened by the removal of its last **if-then** statement. The new postcondition is the conjunction of the two parts arising from the Boolean condition, "c > max," and its negation, "c ≤ max." Figure 6 shows the result of applying this process to the new postcondition of Figure 5 and the "**if** (b > max) **then**" statement. We remove the assignment statement "max := a" by replacing every occurrence of "max," in the postcondition of Figure 6, with "a"; see

$\{\textbf{true}\}$
max := a
if (b > max) **then**
    max := b
end if
if (c > max) **then**
    max := c
end if
$\{(\max = a \vee \max = b \vee \max = c)$
    $\wedge \max \geq a \wedge \max \geq b \wedge \max \geq c\}$

Figure 4: Traditional Back Substitution Method: First Step

$\{\textbf{true}\}$
max := a
if (b > max) **then**
    max := b
end if
$\{\ (c > \max \Rightarrow ((c = a \vee c = b \vee c = c)$
    $\wedge c \geq a \wedge c \geq b \wedge c \geq c))$
  $\wedge (c \leq \max \Rightarrow ((\max = a \vee \max = b \vee \max = c)$
    $\wedge \max \geq a \wedge \max \geq b \wedge \max \geq c))\}$

Figure 5: Traditional Back Substitution Method: Second Step

$$
\begin{array}{l}
\{\textbf{true}\} \\
\max := a \\
\{\ (b > \max \Rightarrow (\\
\qquad (c > b \Rightarrow ((c = a \vee c = b \vee c = c) \\
\qquad\qquad \wedge c \geq a \wedge c \geq b \wedge c \geq c)) \\
\qquad \wedge (c \leq b \Rightarrow ((b = a \vee b = b \vee b = c) \\
\qquad\qquad \wedge b \geq a \wedge b \geq b \wedge b \geq c)))) \\
\quad \wedge (b \leq \max \Rightarrow (\\
\qquad (c > \max \Rightarrow ((c = a \vee c = b \vee c = c) \\
\qquad\qquad \wedge c \geq a \wedge c \geq b \wedge c \geq c)) \\
\qquad \wedge (c \leq \max \Rightarrow ((\max = a \vee \max = b \vee \max = c) \\
\qquad\qquad \wedge \max \geq a \wedge \max \geq b \wedge \max \geq c))))\}
\end{array}
$$

Figure 6: Traditional Back Substitution Method: Third Step

Figure 7. The assertion of classical mathematics we are seeking, shown in Figure 8, is the assertion that the precondition of Figure 7 (**true**) implies the postcondition of Figure 7. If this assertion is true (and it is), then Set_Maximum's body does, indeed, meet its specification.

Please note that the structure of Figure 8's assertion reflects the number of execution paths through the statement sequence. There are four execution paths in this example, and the original postcondition is repeated (with appropriately substituted variables) four times. This is the way the back substitution method handles selection statements like **if-then** and **if-then-else** statements. With the aid of a loop invariant, the back substitution method treats each **while** loop, however, as a single execution path.

1.5.2 An Example of More Natural Reasoning

We now return to Figure 3, and give a different argument that the body of Set_Maximum meets its specification. This argument is closer to the methods computer professionals usually use, or would like to use, when they reason about programs' behavior (see Section 1.1). The first thing we do is mark each between-statement space with a unique integer. For convenience, we use an increasing sequence of integers, as shown in Figure 9. We can then refer to the value each program variable had the last time execution reached position 4, say, with the new names a_4, b_4, c_4, and $\max_4$. Many new variable names, obtained by using the unique integers as subscripts on the

$$
\begin{array}{l}
\{\mathbf{true}\} \\
\{\ (b > a \Rightarrow (\\
\qquad (c > b \Rightarrow ((c = a \vee c = b \vee c = c) \\
\qquad\qquad \wedge c \geq a \wedge c \geq b \wedge c \geq c)) \\
\qquad \wedge (c \leq b \Rightarrow ((b = a \vee b = b \vee b = c) \\
\qquad\qquad \wedge b \geq a \wedge b \geq b \wedge b \geq c)))) \\
\quad \wedge (b \leq a \Rightarrow (\\
\qquad (c > a \Rightarrow ((c = a \vee c = b \vee c = c) \\
\qquad\qquad \wedge c \geq a \wedge c \geq b \wedge c \geq c)) \\
\qquad \wedge (c \leq a \Rightarrow ((a = a \vee a = b \vee a = c) \\
\qquad\qquad \wedge a \geq a \wedge a \geq b \wedge a \geq c))))\}
\end{array}
$$

Figure 7: Traditional Back Substitution Method: Fourth Step

$$
\begin{array}{l}
\mathbf{true} \ \Rightarrow \\
\quad (\ (b > a \Rightarrow (\\
\qquad (c > b \Rightarrow ((c = a \vee c = b \vee c = c) \\
\qquad\qquad \wedge c \geq a \wedge c \geq b \wedge c \geq c)) \\
\qquad \wedge (c \leq b \Rightarrow ((b = a \vee b = b \vee b = c) \\
\qquad\qquad \wedge b \geq a \wedge b \geq b \wedge b \geq c)))) \\
\quad \wedge (b \leq a \Rightarrow (\\
\qquad (c > a \Rightarrow ((c = a \vee c = b \vee c = c) \\
\qquad\qquad \wedge c \geq a \wedge c \geq b \wedge c \geq c)) \\
\qquad \wedge (c \leq a \Rightarrow ((a = a \vee a = b \vee a = c) \\
\qquad\qquad \wedge a \geq a \wedge a \geq b \wedge a \geq c)))))
\end{array}
$$

Figure 8: Traditional Back Substitution Method: Final Assertion

```
-- (0)
     max := a
-- (1)
     if (b > max) then
-- (2)
          max := b
-- (3)
     end if
-- (4)
     if (c > max) then
-- (5)
          max := c
-- (6)
     end if
-- (7)
```

Figure 9: Indexed Method: Mark Between-Statement Spaces

names of the program variables, are now available to the reasoning process. The name "indexed method" comes from the fact that the numbers marking between-statement spaces—and appearing as subscripts in names—function as indexes. The assertion "$a_4 = a_0$" says that "the value of program variable a, whenever execution reaches position 4, is the same as the value variable a had the most recent time execution was at position 0." Both "$a_4 = a_0$" and "$a_7 = a_0$" are true statements in our example.

Use of these subscripted variables reduces the problem of aliasing variable names from one involving dimensions that include both time and location to one involving only time. That is to say, the variable a stands for (is an alias for) an integer value at some arbitrary location in the program text at some time during execution of the program at that location in the program. On the other hand, variable a_4 stands for an integer value at some *fixed* location in the program text at some time during execution of the program at that location in the program. In fact, when comparing, say a_7, with a_4, a_4 stands for the *most recent* time execution reached position 4 in the program. Therefore, there are fewer things to keep track of when reasoning about subscripted variables than when reasoning about program variables. The programmer's working memory [2] carries a lesser burden with subscripted variables than with program variables. Hence, subscripted variables should be preferred.

$$
\begin{array}{ll}
 & -- (0) \\
 & \quad \text{max} := \text{a} \\
 & -- (1) \\
 & \quad \textbf{if } (\text{b} > \text{max}) \textbf{ then} \\
\text{b}_1 > \text{max}_1 & \left\{ \begin{array}{l} -- (2) \\ \quad \text{max} := \text{b} \\ -- (3) \end{array} \right. \\
 & \quad \textbf{end if} \\
 & -- (4) \\
 & \quad \textbf{if } (\text{c} > \text{max}) \textbf{ then} \\
\text{c}_4 > \text{max}_4 & \left\{ \begin{array}{l} -- (5) \\ \quad \text{max} := \text{c} \\ -- (6) \end{array} \right. \\
 & \quad \textbf{end if} \\
 & -- (7)
\end{array}
$$

Figure 10: Indexed Method: Mark Each Branch Condition

We can use these subscripted variables to reason about the execution of **if-then** statements. If execution is at position 5 or 6, we know that the last time execution was at position 4, c was greater than max. That is to say, the condition "$\text{c}_4 > \text{max}_4$" is necessary for execution at positions 5 or 6. When selection and/or looping statements are nested, so are these conditions. A necessary condition for execution inside one of these nested scopes is the conjunction of the nested conditions associated with the containing statements. We call these conjunctions *branch conditions*[1]. Figure 10 shows the second step in our reasoning process—marking each branch condition.

Recall that our goal in reasoning is to establish the correctness conjecture for this procedure's specification and body. To do this, we must prove something. Here we must prove that the postcondition holds at position 7; that is our obligation. We abbreviate "obligation" to the key word **oblig**. Figure 11 shows that we write this obligation at position 7.

[1]We have been used to calling the statement sequence within a nested scope a "branch." The conjunction is the "condition" for executing the branch. Earlier uses of these two terms have not been in combination; they have been used separately in a closely-related fashion. For example, Pressman's discussion of software testing [40, p. 612] states that branch testing "is probably the simplest condition testing strategy." That is to say, branch testing is one kind of condition testing.

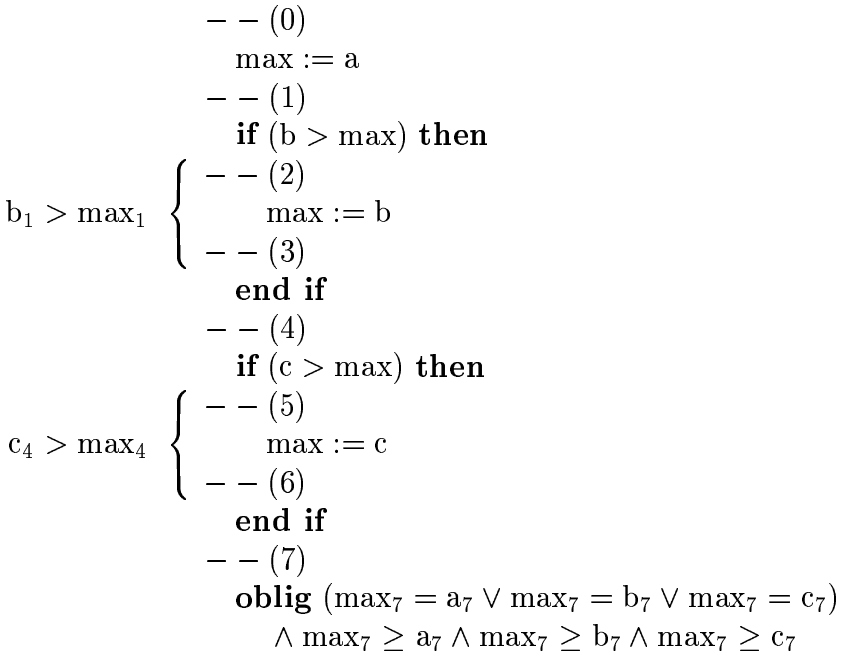

Figure 11: Indexed Method: Write Obligation at Last Position

$$
\begin{array}{l}
\qquad\qquad -- (0) \\
\qquad\qquad \textbf{fact } \max_1 = a_0 \wedge a_1 = a_0 \wedge b_1 = b_0 \wedge c_1 = c_0 \\
\qquad\qquad -- (1) \\
\qquad\qquad \textbf{if } (b > \max) \textbf{ then} \\
b_1 > \max_1 \left\{ \begin{array}{l} -- (2) \\ \qquad \max := b \\ -- (3) \end{array} \right. \\
\qquad\qquad \textbf{end if} \\
\qquad\qquad -- (4) \\
\qquad\qquad \textbf{if } (c > \max) \textbf{ then} \\
c_4 > \max_4 \left\{ \begin{array}{l} -- (5) \\ \qquad \max := c \\ -- (6) \end{array} \right. \\
\qquad\qquad \textbf{end if} \\
\qquad\qquad -- (7) \\
\qquad\qquad \textbf{oblig } (\max_7 = a_7 \vee \max_7 = b_7 \vee \max_7 = c_7) \\
\qquad\qquad\qquad \wedge \max_7 \geq a_7 \wedge \max_7 \geq b_7 \wedge \max_7 \geq c_7
\end{array}
$$

Figure 12: Indexed Method: Replacing an Assignment Statement with a Fact

Fortunately, the statement sequence gives us facts to help prove the obligation. We can replace the first statement, the assignment of a to max, with the key word **fact** followed by everything this statement makes true. This statement changes the value of max ($\max_1 = a_0$), leaving the values of all other variables unchanged ($a_1 = a_0 \wedge b_1 = b_0 \wedge c_1 = c_0$). Figure 12 shows this Fact replacing the assignment statement.

We replace an **if-then** statement with two Facts. The first fact is that, if the branch condition is true, then the values of all variables after the key word **then** equal their values before the **if**. With the second **if-then** statement, this first fact is

$$\textbf{fact } c_4 > \max_4 \Rightarrow (a_5 = a_4 \wedge b_5 = b_4 \wedge c_5 = c_4 \wedge \max_5 = \max_4). \qquad (1.1)$$

The second fact establishes variables' values after the key words **end if**. These values are the same as before the **end if** when the branch condition is true, but they are the same as before the **if** when the branch condition is false. This fact, for the second

$$
\begin{array}{l}
\quad -- (0) \\
\quad \textbf{fact } \max_1 = a_0 \wedge a_1 = a_0 \wedge b_1 = b_0 \wedge c_1 = c_0 \\
\quad -- (1) \\
\quad \textbf{if } (b > \max) \textbf{ then} \\
b_1 > \max_1 \left\{ \begin{array}{l} -- (2) \\ \quad \max := b \\ -- (3) \end{array} \right. \\
\quad \textbf{end if} \\
\quad -- (4) \\
\quad \textbf{fact } c_4 > \max_4 \Rightarrow \\
\qquad (a_5 = a_4 \wedge b_5 = b_4 \wedge c_5 = c_4 \wedge \max_5 = \max_4) \\
c_4 > \max_4 \left\{ \begin{array}{l} -- (5) \\ \quad \max := c \\ -- (6) \end{array} \right. \\
\quad \textbf{fact } (c_4 > \max_4 \Rightarrow \\
\qquad (a_7 = a_6 \wedge b_7 = b_6 \wedge c_7 = c_6 \wedge \max_7 = \max_6)) \\
\quad \wedge (c_4 \leq \max_4 \Rightarrow \\
\qquad (a_7 = a_4 \wedge b_7 = b_4 \wedge c_7 = c_4 \wedge \max_7 = \max_4)) \\
\quad -- (7) \\
\quad \textbf{oblig } (\max_7 = a_7 \vee \max_7 = b_7 \vee \max_7 = c_7) \\
\quad \wedge \max_7 \geq a_7 \wedge \max_7 \geq b_7 \wedge \max_7 \geq c_7
\end{array}
$$

Figure 13: Indexed Method: Replacing the Second **if-then** Statement with Two Facts

if-then statement, is

$$
\begin{array}{lll}
\textbf{fact} & & (c_4 > \max_4 \Rightarrow (a_7 = a_6 \wedge b_7 = b_6 \wedge c_7 = c_6 \wedge \max_7 = \max_6)) \\
& \wedge & (c_4 \leq \max_4 \Rightarrow (a_7 = a_4 \wedge b_7 = b_4 \wedge c_7 = c_4 \wedge \max_7 = \max_4))
\end{array} \quad (1.2)
$$

Figure 13 shows these facts replacing the second **if-then** statement.

When a statement is inside a branch condition, we only know that statement's fact when the condition holds. So we replace the statement with a Fact that is an implication; the branch condition is the left-hand side of the implication, and the effect of the statement is the right-hand side. For the assignment "max := b" we have the fact

$$
\textbf{fact } b_1 > \max_1 \Rightarrow (\max_3 = b_2 \wedge a_3 = a_2 \wedge b_3 = b_2 \wedge c_3 = c_2). \quad (1.3)
$$

$$
\begin{array}{ll}
 & -- (0) \\
 & \quad \textbf{fact } \max_1 = a_0 \wedge a_1 = a_0 \wedge b_1 = b_0 \wedge c_1 = c_0 \\
 & -- (1) \\
 & \quad \textbf{if } (b > \max) \textbf{ then} \\
b_1 > \max_1 \left\{ \right. & -- (2) \\
 & \quad \textbf{fact } b_1 > \max_1 \Rightarrow \\
 & \qquad (\max_3 = b_2 \wedge a_3 = a_2 \wedge b_3 = b_2 \wedge c_3 = c_2) \\
 & -- (3) \\
 & \quad \textbf{end if} \\
 & -- (4) \\
 & \quad \textbf{fact } c_4 > \max_4 \Rightarrow \\
 & \qquad (a_5 = a_4 \wedge b_5 = b_4 \wedge c_5 = c_4 \wedge \max_5 = \max_4) \\
c_4 > \max_4 \left\{ \right. & -- (5) \\
 & \quad \textbf{fact } c_4 > \max_4 \Rightarrow \\
 & \qquad (\max_6 = c_5 \wedge a_6 = a_5 \wedge b_6 = b_5 \wedge c_6 = c_5) \\
 & -- (6) \\
 & \quad \textbf{fact } (c_4 > \max_4 \Rightarrow \\
 & \qquad (a_7 = a_6 \wedge b_7 = b_6 \wedge c_7 = c_6 \wedge \max_7 = \max_6)) \\
 & \quad \wedge (c_4 \leq \max_4 \Rightarrow \\
 & \qquad (a_7 = a_4 \wedge b_7 = b_4 \wedge c_7 = c_4 \wedge \max_7 = \max_4)) \\
 & -- (7) \\
 & \quad \textbf{oblig } (\max_7 = a_7 \vee \max_7 = b_7 \vee \max_7 = c_7) \\
 & \qquad \wedge \max_7 \geq a_7 \wedge \max_7 \geq b_7 \wedge \max_7 \geq c_7
\end{array}
$$

Figure 14: Indexed Method: Replacing Statements Inside Branch Conditions with Facts

For the assignment "max := c" the fact is similar:

$$\textbf{fact } c_4 > \max_4 \Rightarrow (\max_6 = c_5 \wedge a_6 = a_5 \wedge b_6 = b_5 \wedge c_6 = c_5). \tag{1.4}$$

We replace these statements in precisely the same fashion whether or not we have already replaced the containing **if-then** statement (please see Figure 14).

This example shows that, in the indexed method, the programming statements can be replaced by facts in different orders of succession. We chose for this example an order that jumps around in the sequence of statements. We replaced the first assignment statement, the second **if-then** statement, the second assignment, the third assignment, and, finally, as shown in Figure 15, the first **if-then** statement.

$$
\begin{array}{l}
\quad -- (0) \\
\quad\ \textbf{fact}\ \mathrm{max}_1 = \mathrm{a}_0 \wedge \mathrm{a}_1 = \mathrm{a}_0 \wedge \mathrm{b}_1 = \mathrm{b}_0 \wedge \mathrm{c}_1 = \mathrm{c}_0 \\
\quad -- (1) \\
\quad\ \textbf{fact}\ \mathrm{b}_1 > \mathrm{max}_1 \Rightarrow \\
\qquad\qquad (\mathrm{a}_2 = \mathrm{a}_1 \wedge \mathrm{b}_2 = \mathrm{b}_1 \wedge \mathrm{c}_2 = \mathrm{c}_1 \wedge \mathrm{max}_2 = \mathrm{max}_1) \\
\mathrm{b}_1 > \mathrm{max}_1 \left\{ \begin{array}{l} -- (2) \\ \quad\ \textbf{fact}\ \mathrm{b}_1 > \mathrm{max}_1 \Rightarrow \\ \qquad\qquad (\mathrm{max}_3 = \mathrm{b}_2 \wedge \mathrm{a}_3 = \mathrm{a}_2 \wedge \mathrm{b}_3 = \mathrm{b}_2 \wedge \mathrm{c}_3 = \mathrm{c}_2) \end{array} \right. \\
\quad -- (3) \\
\quad\ \textbf{fact}\ (\mathrm{b}_1 > \mathrm{max}_1 \Rightarrow \\
\qquad\qquad (\mathrm{a}_4 = \mathrm{a}_3 \wedge \mathrm{b}_4 = \mathrm{b}_3 \wedge \mathrm{c}_4 = \mathrm{c}_3 \wedge \mathrm{max}_4 = \mathrm{max}_3)) \\
\qquad\ \wedge (\mathrm{b}_1 \leq \mathrm{max}_1 \Rightarrow \\
\qquad\qquad (\mathrm{a}_4 = \mathrm{a}_1 \wedge \mathrm{b}_4 = \mathrm{b}_1 \wedge \mathrm{c}_4 = \mathrm{c}_1 \wedge \mathrm{max}_4 = \mathrm{max}_1)) \\
\quad -- (4) \\
\quad\ \textbf{fact}\ \mathrm{c}_4 > \mathrm{max}_4 \Rightarrow \\
\qquad\qquad (\mathrm{a}_5 = \mathrm{a}_4 \wedge \mathrm{b}_5 = \mathrm{b}_4 \wedge \mathrm{c}_5 = \mathrm{c}_4 \wedge \mathrm{max}_5 = \mathrm{max}_4) \\
\mathrm{c}_4 > \mathrm{max}_4 \left\{ \begin{array}{l} -- (5) \\ \quad\ \textbf{fact}\ \mathrm{c}_4 > \mathrm{max}_4 \Rightarrow \\ \qquad\qquad (\mathrm{max}_6 = \mathrm{c}_5 \wedge \mathrm{a}_6 = \mathrm{a}_5 \wedge \mathrm{b}_6 = \mathrm{b}_5 \wedge \mathrm{c}_6 = \mathrm{c}_5) \end{array} \right. \\
\quad -- (6) \\
\quad\ \textbf{fact}\ (\mathrm{c}_4 > \mathrm{max}_4 \Rightarrow \\
\qquad\qquad (\mathrm{a}_7 = \mathrm{a}_6 \wedge \mathrm{b}_7 = \mathrm{b}_6 \wedge \mathrm{c}_7 = \mathrm{c}_6 \wedge \mathrm{max}_7 = \mathrm{max}_6)) \\
\qquad\ \wedge (\mathrm{c}_4 \leq \mathrm{max}_4 \Rightarrow \\
\qquad\qquad (\mathrm{a}_7 = \mathrm{a}_4 \wedge \mathrm{b}_7 = \mathrm{b}_4 \wedge \mathrm{c}_7 = \mathrm{c}_4 \wedge \mathrm{max}_7 = \mathrm{max}_4)) \\
\quad -- (7) \\
\quad\ \textbf{oblig}\ (\mathrm{max}_7 = \mathrm{a}_7 \vee \mathrm{max}_7 = \mathrm{b}_7 \vee \mathrm{max}_7 = \mathrm{c}_7) \\
\qquad\ \wedge \mathrm{max}_7 \geq \mathrm{a}_7 \wedge \mathrm{max}_7 \geq \mathrm{b}_7 \wedge \mathrm{max}_7 \geq \mathrm{c}_7
\end{array}
$$

Figure 15: Indexed Method: Replacing the First **if-then** Statement with Two Facts

$$
\begin{aligned}
&\textbf{fact } \max_1 = a_0 \wedge a_1 = a_0 \wedge b_1 = b_0 \wedge c_1 = c_0 \\
&\textbf{fact } b_1 > \max_1 \Rightarrow \\
&\qquad (a_2 = a_1 \wedge b_2 = b_1 \wedge c_2 = c_1 \wedge \max_2 = \max_1) \\
&\textbf{fact } b_1 > \max_1 \Rightarrow \\
&\qquad (\max_3 = b_2 \wedge a_3 = a_2 \wedge b_3 = b_2 \wedge c_3 = c_2) \\
&\textbf{fact } (b_1 > \max_1 \Rightarrow \\
&\qquad (a_4 = a_3 \wedge b_4 = b_3 \wedge c_4 = c_3 \wedge \max_4 = \max_3)) \\
&\quad \wedge (b_1 \leq \max_1 \Rightarrow \\
&\qquad (a_4 = a_1 \wedge b_4 = b_1 \wedge c_4 = c_1 \wedge \max_4 = \max_1)) \\
&\textbf{fact } c_4 > \max_4 \Rightarrow \\
&\qquad (a_5 = a_4 \wedge b_5 = b_4 \wedge c_5 = c_4 \wedge \max_5 = \max_4) \\
&\textbf{fact } c_4 > \max_4 \Rightarrow \\
&\qquad (\max_6 = c_5 \wedge a_6 = a_5 \wedge b_6 = b_5 \wedge c_6 = c_5) \\
&\textbf{fact } (c_4 > \max_4 \Rightarrow \\
&\qquad (a_7 = a_6 \wedge b_7 = b_6 \wedge c_7 = c_6 \wedge \max_7 = \max_6)) \\
&\quad \wedge (c_4 \leq \max_4 \Rightarrow \\
&\qquad (a_7 = a_4 \wedge b_7 = b_4 \wedge c_7 = c_4 \wedge \max_7 = \max_4)) \\
&\textbf{oblig } (\max_7 = a_7 \vee \max_7 = b_7 \vee \max_7 = c_7) \\
&\quad \wedge \max_7 \geq a_7 \wedge \max_7 \geq b_7 \wedge \max_7 \geq c_7
\end{aligned}
$$

Figure 16: Indexed Method: Sequence of Facts and Obligations

$$
\begin{array}{l}
(\max_1 = a_0 \wedge a_1 = a_0 \wedge b_1 = b_0 \wedge c_1 = c_0 \wedge \\
(b_1 > \max_1 \Rightarrow \\
\quad (a_2 = a_1 \wedge b_2 = b_1 \wedge c_2 = c_1 \wedge \max_2 = \max_1)) \wedge \\
(b_1 > \max_1 \Rightarrow \\
\quad (\max_3 = b_2 \wedge a_3 = a_2 \wedge b_3 = b_2 \wedge c_3 = c_2)) \wedge \\
(b_1 > \max_1 \Rightarrow \\
\quad (a_4 = a_3 \wedge b_4 = b_3 \wedge c_4 = c_3 \wedge \max_4 = \max_3)) \\
\quad \wedge (b_1 \leq \max_1 \Rightarrow \\
\quad (a_4 = a_1 \wedge b_4 = b_1 \wedge c_4 = c_1 \wedge \max_4 = \max_1)) \wedge \\
(c_4 > \max_4 \Rightarrow \\
\quad (a_5 = a_4 \wedge b_5 = b_4 \wedge c_5 = c_4 \wedge \max_5 = \max_4)) \wedge \\
(c_4 > \max_4 \Rightarrow \\
\quad (\max_6 = c_5 \wedge a_6 = a_5 \wedge b_6 = b_5 \wedge c_6 = c_5)) \wedge \\
(c_4 > \max_4 \Rightarrow \\
\quad (a_7 = a_6 \wedge b_7 = b_6 \wedge c_7 = c_6 \wedge \max_7 = \max_6)) \\
\quad \wedge (c_4 \leq \max_4 \Rightarrow \\
\quad (a_7 = a_4 \wedge b_7 = b_4 \wedge c_7 = c_4 \wedge \max_7 = \max_4))) \Rightarrow \\
((\max_7 = a_7 \vee \max_7 = b_7 \vee \max_7 = c_7) \\
\quad \wedge \max_7 \geq a_7 \wedge \max_7 \geq b_7 \wedge \max_7 \geq c_7)
\end{array}
$$

Figure 17: Indexed Method: Final Assertion

Figure 16 shows the resulting sequence of facts and obligations (seven facts and one obligation) without the branch conditions and the position numbers. At this point we almost have an assertion in classical mathematics. The indexed method specifies a syntactic tranformation from a sequence of facts and obligations to a single mathematical assertion. The idea behind this transformation is that the truth of an obligation in the sequence depends just on the facts appearing earlier in the sequence, i.e., each obligation is to be proved using only the facts that have already appeared. The indexed method's rules transform the sequence of Figure 16 into the assertion of Figure 17. If we have made no mistakes in applying the indexed method, and if the indexed method is sound, then, if we can prove the obligation given the preceding facts (and we can!), we have proved the correctness conjecture true for this implementation of procedure Set_Maximum.

Please note that the list of seven facts and one obligation in Figure 16 has the same structure as the original sequence of statements. This is so because we derived the list by replacing each program statement with one or more facts—one simple fact,

derived from the first assignment statement, followed by two sections of facts, derived from the two **if-then** statements.[2] Therefore, the final assertion (Figure 17) also has the same structure as the original sequence of program statements.

Its structure contrasts with the structure of the final assertion we obtained using the back substitution method (Figure 7)—a structure having the form of four similar conclusions each guarded by a different antecedent. Figure 7's structure is a list of the procedure's four possible execution paths. Please imagine the original program to have contained a sequence of *six*, rather than two, **if-then** statements. Then the final assertion we would have obtained according to the back substitution method would have had the form of *sixty-four* similar conclusions each guarded by a different antecedent. This number "sixty-four" is a characteristic of the procedure's execution paths, but is otherwise not evident in the structure of the procedure's sequence of statements. On the other hand, the final assertion we would have obtained according to the indexed method would have started with the single fact associated with the first assignment statement. Following this would have been six groups (for six **if-then** statements) of three facts each, and the obligation, for a total of nineteen facts preceding the obligation. These are, therefore, two distinct structures.

1.5.3 Changing the Postcondition

Another difference between the two reasoning methods comes to our attention when we realize that we may want to change the postcondition of Set_Maximum. Figure 18 shows an alternative implementation for Set_Maximum. The three assignment statements shown leave max's value unchanged, possibly changing each of a, b, and c. However, this new procedure body is correct with respect to our original specification. We now realize that this specification did not say what we meant. We also want to say that Set_Maximum must not change the values of a, b, and c. New notation, "#" (the "old sign"), will help us say this.

When "#b" (pronounced "old bee") appears in a postcondition, it refers to the value of formal parameter b just before execution of the procedure body. When "b" appears in a postcondition without the old sign, it means the value of formal parameter b just after execution of the procedure body. An old sign must not appear in a precondition, and a "b" in a precondition refers to the value of formal parameter b just before execution of the procedure body.

[2] A procedure call would be replaced by one obligation and one fact. The key words of a **while** loop would be replaced by two obligations and two facts. Other steps would replace the body of the loop with facts and obligations.

procedure Set_Maximum (a, b, c, max : Integer)
    ensures (max = a ∨ max = b ∨ max = c)
            ∧ max ≥ a ∧ max ≥ b ∧ max ≥ c
begin
    a := max
    b := max
    c := max
end Set_Maximum

Figure 18: Another Correct Implementation for Procedure Set_Maximum

procedure Useful_Set_Max (a, b, c, max : Integer)
    ensures (max = a ∨ max = b ∨ max = c)
            ∧ max ≥ a ∧ max ≥ b ∧ max ≥ c
            ∧ a = #a ∧ b = #b ∧ c = #c

Figure 19: An Improved Specification: Procedure Useful_Set_Max

By conjoining to Set_Maximum's postcondition the phrase "a = #a∧b = #b∧c = #c," we cause the body of Figure 18 to become an incorrect implementation of the new specification. Because this new specification, shown in Figure 19, establishes different behavior for the procedure, we give it the new name "Useful_Set_Max." We hope that the body of Figure 3 remains a correct implementation of the new specification.

How do the two reasoning methods cope with this change to the postcondition? The indexed method handles this change easily by changing only the final obligation. We change the obligation by simply conjoining to it the phrase "$a_7 = a_0 \wedge b_7 = b_0 \wedge c_7 = c_0$." This new obligation can still be established from the existing facts.

It is significantly more difficult to adjust the reasoning we already did according to the back substitution method to fit the new postcondition. We have to take the new phrase in the postcondition and back it through the two **if-then** statements and the first assignment statement. This process introduces four new phrases into the classical mathematical assertion—one for each execution path. We might get confused trying

to make this adjustment, and decide simply to apply the back substitution method again to the entire new postcondition.

In either case, before trying to prove the resulting classical mathematical assertion, we would remove all the old signs from it. We do this because, after having backed through the program, the assertion that remains is a statement about the values at the start of the program. In other words, because all programming statements have been removed, the final values of the variables equal their initial values.

1.5.4 Conclusions

Hoare Logic

The back substitution method is an algorithmic way of producing between-statement assertions that can be used to build a proof within Hoare logic. By itself, Hoare logic is not a method; it is a logic that defines what proofs are. Hoare logic permits any method that could produce between-statement assertions that can be used to build a proof. A good oracle or good guesses could provide appropriate between-statement assertions for Hoare logic. Hence, we can imagine the existence of a good algorithmic method—for producing between-statement assertions for Hoare logic—whose characteristics differ from the back substitution method.

Consequently, we are not claiming that all possible methods for producing between-statement assertions for Hoare logic are necessarily less natural than the indexed method. The following comparisons show that the traditional method of producing between-statement assertions that can be used to build a proof within Hoare logic—namely, the back substitution method—is not as natural as the indexed method.

Comparison of Indexed and Back Substitution Methods

Corresponding to the two points of Section 1.1, the example explored here in Section 1.5 shows the following:

1. The indexed method permits the computer professional to select the order in which he/she reasons independently about groups of statements before easily assembling these arguments into an argument about the whole program. In contrast, the back substitution method requires the mathematical assertion to be built by iterative substitution in reverse order of the statements. The back substitution method enforces a systematic strategy for reasoning about programs. The indexed method also supports a systematic strategy, but, additionally, it facilitates an as-needed strategy for reasoning about programs. Littman et al.

[28] and Koenamann and Robertson [24] have shown that both systematic and as-needed strategies are used among programmers.

2. The structure of the mathematical assertion built in the indexed method matches the *static* structure of the program, while the assertion's structure in the back substitution method matches a list of the program's *dynamic* execution paths. A list of execution paths contains a bias toward systematic reasoning because it is not clear that one can learn what he needs to know from examination of just one or a few of the execution paths. Due to the match, in the indexed method, between assertion and program structure, a programmer has the option of reasoning about the mathematical assertion with the same kind of as-needed strategy she might have preferred when reasoning about the program [24]. Furthermore, discoveries about the assertion will have a direct correspondence with the program text.

These two benefits for the indexed method come at the cost of using more names—the names with subscripts. However, as we discussed in the second paragraph of Section 1.5.2, using these extra names decreases the burden on the programmer's working memory; the extra names are aids in stating relationships among the values of program variables following different statements. Furthermore, our experience with the Larch Proof Assistant (LP) [14, 3] indicates that such tools can handle the load of these extra variables with ease. Given, say, the facts that $x_5 = x_6$, $x_6 = x_7$, and $x_7 = x_8$, LP quickly deduces that $x_5 = x_8$ and reduces any facts about x_6, x_7, or x_8 to facts about x_5. (Under different instructions, LP could instead reduce any facts about x_5, x_6, or x_7 to facts about x_8.) The cost of using the additional names is, therefore, manageable. We conclude that, when compared with the indexed method, the back substitution method is not so natural.

1.6 Importance of Proving Soundness and Completeness

Every designer of a system of proof rules includes soundness among the characteristics to be achieved. Any unsoundness is unintentional. The indexed method is no exception. Its rules are not capricious; they are designed to preserve validity. Experience with this method, such as the example explored in this chapter, seems to indicate that it is sound. It looks reasonable. Shall we stop there? Is it necessary to go through the trouble of *proving* the system's soundness? What is to be gained by doing so?

The history of the indexed method's design provides evidence for the necessity of proving its soundness. This history includes serious consideration of a rule that

{**true**}
Stay_two (y)
{y = 2}

Figure 20: Evidence of Unsoundness: First Step

has a subtle flaw. Inclusion of this rule would have made the system unsound. This flaw was recognized and corrected. Did this correction make the system sound, or are there still-unrecognized flaws lurking in the system? Proof of soundness can greatly increase our confidence that there are no remaining undiscovered inconsistencies. The understanding of proof we have learned from Lakatos [26] causes us to write of "confidence" rather than "certainty."

While they were designing a direct ancestor of the indexed method in the mid-1980s, Doug Harms and Bruce Weide [48] considered permitting us to prove each obligation with the help of all the facts. They discovered that this method of proof was unsound, and corrected it before Weide included the method in his course notes [47]. The correction was that the facts that can be used to prove an obligation must derive from points in the program text that precede the point from which the obligation was derived; the relative order of facts and obligations is important.

Even earlier, Douglas Maurer had invented a method of proof, called the modification index method, that is very similar to the indexed method. An indication of the subtlety of the flaw noticed by Harms and Weide is that Maurer's paper includes the recommendation to use all the facts when proving an obligation [33, p. 430]. (Harms and Weide were not aware of Maurer's work when they were designing the indexed method. Maurer's paper came to this author's attention in 1994.)

A counterexample will show plainly that it is unsound to use all the facts when proving an obligation. Suppose the program variable y is of type Integer in the program segment shown in Figure 20. This segment consists of a precondition, a call to the procedure Stay_two, and a postcondition. Figure 21 shows the specification of procedure Stay_two. Examination of this specification reveals that an implementation of Stay_two that does nothing is correct. Given this implementation of Stay_two, if initially y = 3 in Figure 20, then obviously y $\neq$ 2 after Stay_two is called.

When we mark the between-statement spaces in Figure 20, change the precondition to a fact, and change the postcondition to an obligation, we obtain Figure 22. We have an obligation to show that when Stay_two is called, y has the value 2. After the call we know two things: that the value of y is 2, and that the call has not changed its

procedure Stay_two (x : Integer)
    requires $x = 2$
    ensures $x = 2 \wedge x = \#x$

Figure 21: A Specification of Procedure Stay_two

fact true
$--$ (0)
Stay_two (y)
$--$ (1)
oblig $y_1 = 2$

Figure 22: Evidence of Unsoundness: Second Step

value. Figure 23 shows the result of replacing the call to Stay_two with the obligation and the fact. The obligation $y_0 = 2$ cannot be proven from the bare fact of **true**. This is why the program segment of Figure 20 is incorrect. On the other hand, we could "prove" the program segment correct if we were permitted to use all the facts in proving this obligation. Given $y_1 = 2$ and $y_1 = y_0$, substitution of y_0 for y_1 in $y_1 = 2$ gives us $y_0 = 2$. Therefore, permission to use all the facts when establishing an obligation leads to unsoundness; with this permission, we are able to prove an incorrect program correct. It turns out to be a sound practice to use any *earlier* fact

fact true
$--$ (0)
oblig $y_0 = 2$
fact $y_1 = 2 \wedge y_1 = y_0$
$--$ (1)
oblig $y_1 = 2$

Figure 23: Evidence of Unsoundness: Third Step

when establishing an obligation. Based on the formalizations of Chapters II and III, we prove, in Chapter IV, that the indexed method is sound.

Our effort to prove in detail the relative completeness of the indexed method provided more evidence for the benefit of so doing. In the late stages of this effort, while attempting to prove relative completeness for the procedure call rule, the author discovered, contrary to his wishful expectations, that the indexed method is *not* relatively complete with respect to a functional semantics if external procedures are permitted to have relational specifications (see Sections 4.2 and 5.3.1). The lesson here is that detailed proof is one effective method of directing our attention to subtle yet important matters that might otherwise be overlooked.

In support of our thesis, we hasten to add that the indexed method *is* relatively complete with respect to a functional semantics if all external procedures must have functional specifications. Furthermore, the back substitution method, like the indexed method, is not relatively complete with respect to a functional semantics if external procedures are permitted to have relational specifications. Future work could correct these problems by developing a satisfactory relational semantics and (without restricting the specification of external procedures) proving these methods sound and relatively complete with respect to it.

1.7 Outline of Dissertation

The remainder of the dissertation is concerned with establishing a formal basis for the indexed method. Chapter II establishes the syntax and semantics of a simple imperative programming language, and extends this language with constructs useful only in the formal basis for the indexed method. Chapter III defines the indexed method's formal basis by specifying its proof rules. Chapter IV presents detailed proofs of the soundness and relative completeness of the indexed method's formal basis; a typical reader will be unlikely to enjoy this chapter. Good literature affords the reader's imagination room to playfully fill in its own details. Each block of a good foundation is present and laid firmly in the right place. Alas, in chapter IV, we are building not good literature, but a good foundation. Chapter V discusses possible future work and conclusions drawn from the work so far.