CHAPTER V

Conclusion

5.1 Informal and Formal Indexed Methods

We defined, in Chapter II, a semantics for a procedural, imperative programming language with specifications: a language of assertive programs. In Chapter IV, we proved the soundness and relative completeness of the indexed method for proving correctness of assertive programs, the rules of which we established in Chapter III. We do well to ask whether the rules of Chapter III are a fair representation of the method of proof informally described in Chapter I. After all, our argument that it is plausible that the indexed method is more natural than the back substitution method was about the method informally described in Chapter I, not the method formally defined in Chapter III.

The first point of similarity is that, as in Chapter I, when an operational statement is removed by the application in the math direction of a Chapter III proof rule, the statement is replaced by facts and obligations. To see this, we must take **assume** to correspond to **fact**, and **confirm** to **oblig**. The positioning of these replacements in Chapter III is the same as in Chapter I. The content of these replacements is also the same because Chapter I's branch conditions are the same as the conditions occurring in the **whenever** statements of Chapter III; these latter conditions are also called branch conditions.

We marked the between-statement spaces with an increasing sequence of integers in Chapter I. We formally defined our intention regarding where to make these marks with the restricted syntax of Chapter III. Each **stow** statement serves as a between-statement mark.

In Chapter I, once the branch condition for an operational statement has been calculated, it can be removed and replaced with facts and obligations regardless of whether the operational statement immediately preceding it has already been so replaced. As argued in Section 4.6, the simplifying assumption that causes the rules of Chapter III to permit only the first statement within a **whenever** statement to be replaced is without loss of generality. We can augment the set of proof rules with four additional rules that permit operational statements to be processed in any order. We can prove the augmented set of rules to be sound and relatively complete. Therefore, the informal indexed method is, like the formal one, sound and relatively complete.

5.2 Relationships Among Methods

The three formal methods of reasoning about program correctness discussed in Chapter I—the back substitution method, symbolic execution (the forward accumulation method), and the indexed method—share two important properties. They are all sound and relatively complete. They differ chiefly in the direction of action as they are applied in proof discovery, transforming programs to mathematical assertions.

In this context, we understand "direction" with respect to the abstract syntax tree of the program. We distinguish the *front* of the program from its *back*. If statement ST1 is in the same statement sequence as ST2 and ST1 precedes ST2 in the sequence, then ST1 is closer to the program's front than ST2, and ST2 is closer to the back than ST1. When displaying an abstract syntax tree, as in Figure 85, the usual convention is to show the front at the left, the back at the right. Execution of the program is front-to-back. Following the same convention, we label the tree's root as the *top* and its leaves as the *bottom*.

The action of the back substitution method moves from back to front, removing one operational statement at a time. The action of symbolic execution moves from front to back, accumulating a symbolic value for each of the program variables. The action of the indexed method is "perpendicular" to that of either of the other two methods. Branch conditions are calculated as inherited attributes from the top to the bottom. Operational statements are removed and replaced with **assume** and **confirm** statements. Each replacement reduces the depth of the tree; so, the tree is successively flattened. If we choose our frame of reference as the tree's root, then the tree is flattened from bottom to top. According to the indexed method of Chapter I, any operational statement, whether leaf or internal node, may be replaced at any time. This is why we used a two-headed arrow in Figure 85. A strictly downward pointing arrow would be more appropriate for the indexed method of Chapter III because only operational statements at a fixed depth in the tree are replaced; the action is from the top, bringing lower-level constructs higher.

Observation of the methods' differing action directions gives insight into some of their other differences. Because it acts along the dimension of program execution (although in the opposite direction), the back substitution method must result in an assertion whose structure corresponds to a list of program execution paths. The

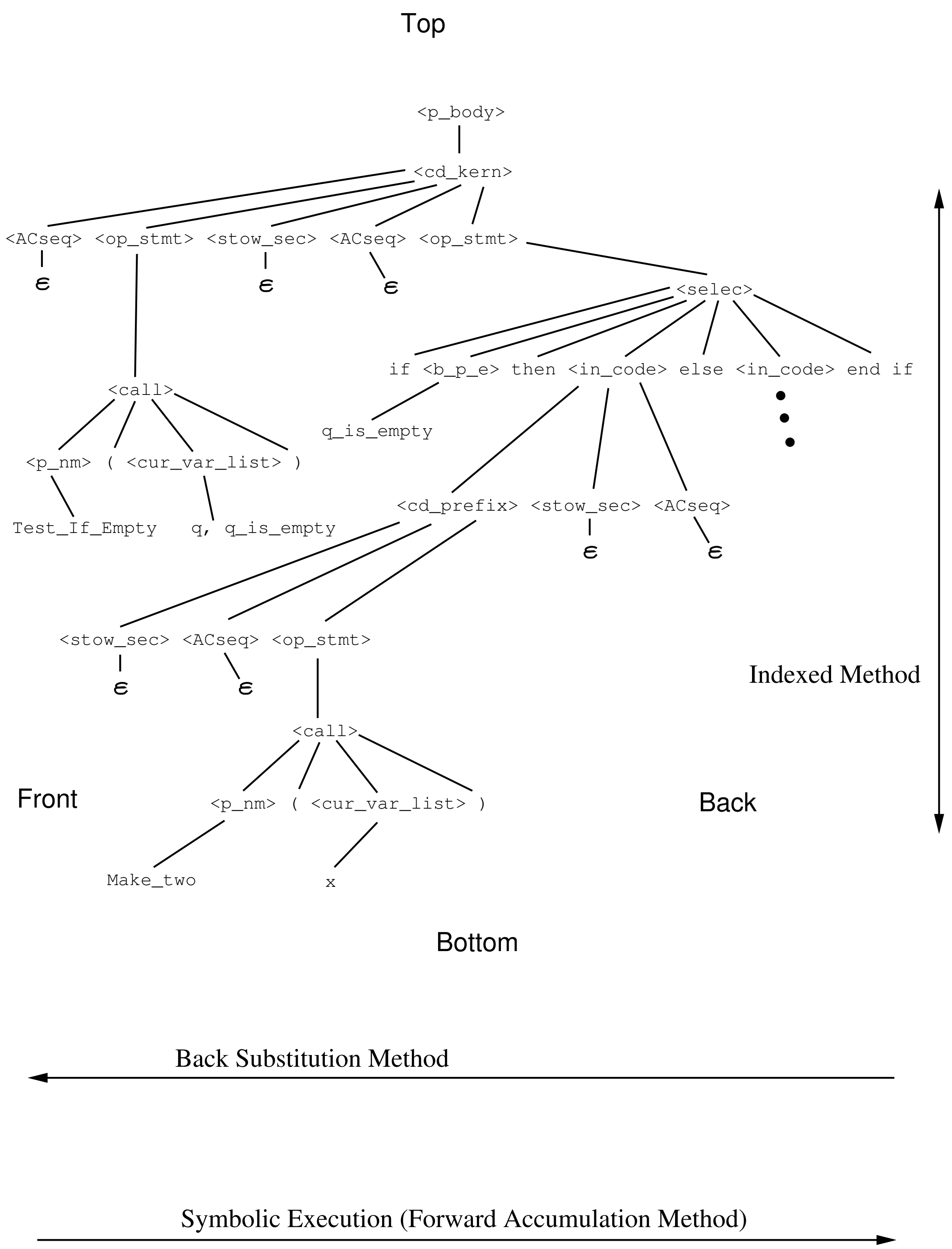
Top
<p_body>
<cd_kern>
<ACseq> <op_stmt> <stow_sec> <ACseq> <op_stmt>
ε
<selec>
if <b_p_e> then <in_code> else <in_code> end if
q_is_empty
<call>
<p_nm> (<cur_var_list>)
Test_If_Empty
q, q_is_empty
<cd_prefix> <stow_sec> <ACseq>
<stow_sec> <ACseq> <op_stmt>
<call>
<p_nm> (<cur_var_list>)
Make_two
x
Front
Back
Indexed Method
Bottom
Back Substitution Method
Symbolic Execution (Forward Accumulation Method)

Figure 85: Action Directions Differ Among the Three Methods

structure of the assertion that the indexed method produces can match the program structure because the method acts in a direction perpendicular to that of program execution; the assertion's structure is a result of flattening the program's structure. In symbolic execution, each "control path" is treated separately; "the endpoints of these paths are (groups of) ASSERT statements, and there must be no loops lacking such statements [5, p. III-1]." That symbolic execution deals with control paths is a consequence of its acting in the same direction as program execution.

5.3 Opportunities for Future Work

5.3.1 Miscellaneous Issues

For the research reported here, we adopted a semantics in which a procedure call's outputs are a function of its inputs—not a relation. While we expect the indexed method to be relatively complete with respect to a relational semantics, it is not relatively complete with respect to a functional semantics if calls to relationally specified external procedures are permitted (see Section 4.2). This situation is one of the reasons that we need a relational semantics. By itself, defining the meaning of a procedure call as a relation is not too difficult, but the implications are a bit daunting for our current understanding of the meaning of loops in terms of the minimum fixed point of a *functional.* Sanderson [42] proposes a relational semantics. However, his approach involves changing the complete partial ordering on the type Boolean. Edwards and Ogden [7] have said "this approach is fine as far as it goes, but changing the partial ordering for Boolean interferes with the 'correct' meaning of assertions that we're used to." Therefore, applying Sanderson's results to produce a relational semantics for assertive programs may require solving some significant problems. We also seek proof rules that are good for modular verification in the relational case. It would be fascinating to discover what form the indexed method would take in the relational case.

The number of components in the environment could be reduced by removing the setup. The index state could serve a new role in addition to its current role of storing the current state at some point with a **stow**(i) statement. It could also serve the role that setup currently serves. Note that, at the top level of top level code, every **alter all** statement immediately precedes a **stow**(i) statement (for some i). This pair of statements could be replaced by one statement, say "**alter stow**(i)" or "**start from**(i)", whose meaning is "change the current state to be the same as the index state at i." The initial index state would then serve as well as the setup. We chose to include the setup as an additional component of the environment so that the

$$
\begin{array}{rcll}
\langle\text{iter}\rangle & ::= & \langle\text{while}\rangle \mid \langle\text{loop}\rangle & (5.1)\\
\langle\text{loop}\rangle & ::= & \textbf{loop} & (5.2)\\
 & & \quad\textbf{maintaining } \langle\text{old\_assert}\rangle & \\
 & & \qquad[\langle\text{in\_code}\rangle] & \\
 & & \quad\textbf{exit when } \langle\text{b\_p\_e}\rangle & \\
 & & \qquad\{\langle\text{in\_code}\rangle & \\
 & & \quad\textbf{exit when } \langle\text{b\_p\_e}\rangle\} & \\
 & & \qquad[\langle\text{in\_code}\rangle] & \\
 & & \textbf{end loop} & \\
\langle\text{while}\rangle & ::= & \textbf{loop} & (5.3)\\
 & & \quad\textbf{maintaining } \langle\text{old\_assert}\rangle & \\
 & & \quad\textbf{while } \langle\text{b\_p\_e}\rangle \textbf{ do} & \\
 & & \qquad\langle\text{in\_code}\rangle & \\
 & & \textbf{end loop} &
\end{array}
$$

Figure 86: Redefinition of ⟨iter⟩

two concerns—storing the current state and changing the current state—would have separate representations in the environment. Separating these two concerns helped our intuition; we trust it has helped the reader's, too.

There are some incremental enhancements that could be added to the indexed method. Here, we adopted the simplifying convention that all procedures are proper procedures, their calls not appearing in expressions as function calls. The indexed method could be enhanced by adding function procedures to the language, and this seems to be straightforward if functions are not allowed to have side effects.

5.3.2 The "loop exit when" Rule

The iteration statement defined in Chapters II and III is a variation of the **loop** statement of Ada. Ada's **loop** statement can also have multiple exit points. Figure 86 shows how to extend the grammars of Chapters II and III by giving the ⟨iter⟩ nonterminal symbol a second alternative. Note that a **loop** statement can have one or more exit points; it has a variable number of **exit when** constructs. A question

```
$\mathcal{P} \overset{\text{def}}{=} C\backslash$  $prec\_top\_lev\_code$                                    (5.4)
          alter all
          stow($i$)
          $ACseq_0$
        | whenever Br_Cd do
        |    loop
        |       maintaining Inv$\lfloor$x, #x$\rfloor$
        |          stow($j_1$)
        |          $cd\_kern_1$   stow($k_1$)   $ACseq_1$
        |       exit when $b\_p\_e_1$
        |          stow($j_2$)
        |          $cd\_kern_2$   stow($k_2$)   $ACseq_2$
        |       exit when $b\_p\_e_2$
        |          stow($j_3$)
        |          $cd\_kern_3$   stow($k_3$)   $ACseq_3$
        |    end loop
        |    stow($l$)
        |    $cd\_suffix$
        | end whenever
          $fol\_top\_lev\_code$
```

Figure 87: Definition of $\mathcal{P}$ for a Proof Rule That Would Handle the "**loop exit when**" Statement Having Exactly Two **exit when** Constructs

seeking an answer is: what are good ways to express a proof rule for a statement that can have a variable number of constructs?

Figures 87 and 88 respectively define $\mathcal{P}$ and $\mathcal{M}$ for a proof rule that would handle the "**loop exit when**" statement having exactly two **exit when** constructs. The proofs of soundness and relative completeness would need to be adapted to accommodate this rule or a more general rule that would handle the "**loop exit when**" statement having any number of **exit when** constructs. Another variation is that the **maintaining** clause containg the loop invariant need not be the first construct; it could appear anywhere in the loop. A rule or rules to handle this variation would need to be developed.

```
$\mathcal{M} \overset{\text{def}}{=} C\backslash$ $\mathit{prec\_top\_lev\_code}$                                    (5.5)
      alter all
      stow($i$)
      $\mathit{ACseq}_0$
    | confirm $(\text{Br\_Cd}) \Rightarrow (\text{Inv}[\text{x} \rightsquigarrow \text{x}_i, \#\text{x} \rightsquigarrow \text{x}_i])$
    | alter all   stow($j_1$)
    | whenever Br_Cd do
    |    assume $\text{Inv}[\text{x} \rightsquigarrow \text{x}_{j_1}, \#\text{x} \rightsquigarrow \text{x}_i]$
    |    $\mathit{cd\_kern}_1$   stow($k_1$)   $\mathit{ACseq}_1$
    | end whenever
    | alter all   stow($j_2$)
    | whenever $(\text{Br\_Cd}) \wedge (\neg\text{MExp}(\mathit{b\_p\_e}_1[\text{y} \rightsquigarrow \text{y}_{k_1}]))$ do
    |    assume $\text{x}_{j_2} = \text{x}_{k_1}$
    |    $\mathit{cd\_kern}_2$   stow($k_2$)   $\mathit{ACseq}_2$
    | end whenever
    | alter all   stow($j_3$)
    | whenever $(\text{Br\_Cd}) \wedge (\neg\text{MExp}(\mathit{b\_p\_e}_1[\text{y} \rightsquigarrow \text{y}_{k_1}]))$
    |     $\wedge (\neg\text{MExp}(\mathit{b\_p\_e}_2[\text{y} \rightsquigarrow \text{y}_{k_2}]))$ do
    |    assume $\text{x}_{j_3} = \text{x}_{k_2}$
    |    $\mathit{cd\_kern}_3$   stow($k_3$)   $\mathit{ACseq}_3$
    |    confirm $\text{Inv}[\text{x} \rightsquigarrow \text{x}_{k_3}, \#\text{x} \rightsquigarrow \text{x}_i]$
    | end whenever
    | alter all   stow($l$)
    | whenever Br_Cd do
    |    assume
    |       $((\text{MExp}(\mathit{b\_p\_e}_1[\text{y} \rightsquigarrow \text{y}_{k_1}])) \vee (\text{MExp}(\mathit{b\_p\_e}_2[\text{y} \rightsquigarrow \text{y}_{k_2}])))$
    |       $\wedge ((\text{MExp}(\mathit{b\_p\_e}_1[\text{y} \rightsquigarrow \text{y}_{k_1}])) \Rightarrow (\text{x}_l = \text{x}_{k_1}))$
    |       $\wedge (((\neg\text{MExp}(\mathit{b\_p\_e}_1[\text{y} \rightsquigarrow \text{y}_{k_1}]))$
    |          $\wedge (\text{MExp}(\mathit{b\_p\_e}_2[\text{y} \rightsquigarrow \text{y}_{k_2}]))) \Rightarrow (\text{x}_l = \text{x}_{k_2}))$
    |    $\mathit{cd\_suffix}$
    | end whenever
      $\mathit{fol\_top\_lev\_code}$
```

Figure 88: Definition of $\mathcal{M}$ for a Proof Rule That Would Handle the "**loop exit when**" Statement Having Exactly Two **exit when** Constructs

5.3.3 Investigating the Worth of the Indexed Method

In Chapter I we showed the plausibility that the indexed method is more natural than the other existing methods of proving program correctness. Now that the indexed method is known to be sound and relatively complete, investigations into its practical value can be pursued without fear that the method is otherwise flawed. Researchers could perform empirical studies to compare practitioners' performance when using the competing methods to prove program correctness. Such use could be tool-supported or not.

While tool support is essential for making proofs of program correctness economically feasible, there is a reason for people to prove some programs correct without the use of automated tools. Practitioners need to learn how to write specifications; this learning can be acquired, in part, by practicing proofs by hand. Teachers could explore uses of the indexed method in the classroom.

The choice of proof method may also be important for the speed and power of the tool support. Perhaps the indexed method has the benefit of procrastinating expression elaboration (substitution) so that appropriate decisions for simplification can be made later. As Deutsch observed: "One interesting aspect of Scott's attempt to reduce the idea of a program to its mathematical essence is that it essentially removes the idea of 'control' from programs and converts them to static objects." This conversion "may greatly simplify their analysis [5, p. VII-9]." On the other hand, perhaps the structure of the program itself could be used by an automatic tool to make appropriate decisions to reduce the number of theorems that must be proved later. Deutsch [5, p. III-3] based his work on this premise.

The question facing us here is how/where to factor the problem of program verification. The indexed method is "based strictly on theorem-proving power [5, p. VII-5]" because it moves directly from the realm of assertive programs to that of mathematical assertions. It preserves the relationship between the assertion and the program through the use of indices. The problem of proving the assertion is saved for later. Such procrastination may be a liability or an asset. It could be an asset if the automation of proofs of simple, tedious theorems is not much harder than doing the same in the context of the programs themselves. Preserving the relationship between the assertion and the program certainly is an asset if the proof fails—because the parts of the assertion that cause it to be invalid correspond directly to the parts of the program that are at fault.

5.4 Contributions

We have defended the following 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).

We showed that the traditional formal method supports reasoning according to systematic strategies only, whereas the indexed method also supports as-needed strategies. Evidence suggests that people use both as-needed and systematic strategies to reason about programs. Hence, the indexed method provides better support for people's natural tendencies. We formally defined the syntax and denotational semantics of an imperative, procedural, assertive progamming language. We gave a formal definition for the validity of these assertive programs. We established the formal proof rules of the indexed method. Finally, based on our definition of validity, we proved the system of proof rules to be both sound and relatively complete.

Beyond establishing the three-part thesis we set out to defend, a chief contribution of this work is the novelty of the proof of soundness and relative completeness. It is easy to be skeptical about the indexed method; how can an **if-then-else** statement be correctly transformed into a sequence of statements in which both the former "**then**" and "**else**" parts are always executed? The branch conditions play a key role, of course, but how are we to show that their role is proper? Much of the answer lies in the assert status, originally invented to handle external procedures [38]. The assert status mimics logical implication, enabling an implication to represent all possible executions. When coupled with the assert status, the indexed variables (and index state) permit simultaneous (parallel) processing of statements arranged in a sequence of statements. In other words, they permit simplifying the abstract syntax tree from top to bottom rather than from back to front or front to back.

Another puzzle that required solution was the removal of executable statements, replacing them with **assume** and **confirm** statements. Executable statements change the current state; if an executable statement is simply removed, the current state no longer changes at that point, retaining its value from the last time it was changed.

This dilemma was solved by including one or more **alter all** statements among the replacements for an executable statement. The "setup" component of the environment was introduced to explain the semantics of **alter all**.

The level of detail we have used in presenting the syntax, semantics, proof rules, and proofs is rare. We hope this presentation of detail reduces the mystery sometimes associated with the logic of computer programming.