Computer Program Verification: Improvements for Human Reasoning

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the Graduate

School of The Ohio State University

By

Wayne David Heym, B.Phil., M.S., M.S.

* * * * *

The Ohio State University

1995

Dissertation Committee:

Bruce W. Weide

William F. Ogden

Stuart H. Zweben

Approved by

Adviser

Department of Computer
and Information Science

For my loving wife, Kimberly Wells Heym

Acknowledgments

My advisor, Bruce W. Weide, has guided and encouraged me through all the stages of graduate study, ever since my first visit to the campus. I am grateful for his wisdom in suggesting the topic for this dissertation. The number of helpful suggestions and ideas Bruce has offered in our regular meetings together is beyond counting, and his steadfast encouragement has helped me to persevere. I am thankful for William F. Ogden and his insightful, skeptical approach to my work. More than anyone else, he has guided the form of the proof rules, the explanation of the semantics, and the presentation of the proofs of soundness and relative completeness. Bill's hearty and healthy skepticism led the way to identifying the crucial ideas in the soundness proofs. Stuart H. Zweben encouraged me to work with him on testing and was instrumental in helping me learn most of what I know about the subject. He persevered in getting me to properly defend, with reference to the literature, the claim that my proposed new method, the indexed method of correctness proof, is more natural than the existing method. Stu's careful reading and corrections of early drafts were invaluable, and he completed his readings with amazing, and much appreciated, speed. These three professors regularly meet with a group of students interested in the effective production of quality software from the standpoint of reusability—the Reusable Software Research Group (RSRG) at The Ohio State University. The

participants in RSRG have helped me learn many of the important issues in software engineering. This thesis has benefited greatly from the RSRG's intelligently dedicated effort to learn how to do software right.

We gratefully acknowledge the support of the National Science Foundation through grants CCR-9111892 and CCR-9311702; the Advanced Research Projects Agency under ARPA contract number F30602-93-C-0243, monitored by the USAF Material Command, Rome Laboratories, ARPA order number A714; and the Army Research Office through grant DAAH04-95-1-0457.

Vita

July 3, 1956	Born - Cleveland, Ohio
1978	B.Phil. Interdisciplinary Studies, Miami University
1980	M.S. Operations Research, Cornell University
1980-1983	Programmer/Analyst I, Eastman Kodak Company
1984-1988	Research Support Specialist, Cornell University
1988-1989	University Fellow, The Ohio State University
1989	M.S. Computer and Information Science, The Ohio State University
1989-1991	Graduate Teaching Associate, The Ohio State University
1991-1995	Graduate Research Associate, The Ohio State University
1994-1995	Instructor of Computer Science, Otterbein College

Publications

Research Publications

E. E. Ewing, W. D. Heym, E. J. Batutis, R. G. Snyder, M. Ben Khedher, K. P. Sandlan, and A. D. Turner. Modifications to the simulation model POTATO for use in New York. *Agricultural Systems*, 33(2):173–192, 1990.

W. D. Heym, E. E. Ewing, A. G. Nicholson, and K. P. Sandlan. Simulation by crop growth models of defoliation, derived from field estimates of percent defoliation. *Agricultural Systems*, 33(3):257–270, 1990.

S. H. Zweben, W. D. Heym, and J. Kimmich. Systematic testing of data abstractions based on software specifications. *The Journal of Software Testing, Verification and Reliability*, 1(4):39–55, 1992.

B. W. Weide, W. D. Heym, and W. F. Ogden. Procedure calls and local certifiability of component correctness. In *Sixth Annual Workshop on Software Reuse*, pages Weide– 1–6. In cooperation with the IEEE Computer Society Technical Committee on Software Engineering, November 1993.

S. H. Edwards, W. D. Heym, T. J. Long, M. Sitaraman, and B. W. Weide. Specifying components in RESOLVE. *Software Engineering Notes*, 19(4):29–51, Oct. 1994.

B. W. Weide, W. D. Heym, and J. E. Hollingsworth. Reverse engineering of legacy code exposed. In *Proceedings 17th International Conference on Software Engineering*, pages 327–331. ACM, Apr. 1995.

Fields of Study

Major Field: Computer and Information Science

Studies in:

Software Engineering	Prof. Bruce W. Weide
Programming Languages	Prof. Wolfgang W. Kuechlin
Theory of Computation	Prof. Timothy J. Long

Table of Contents

List of Tables

List of Figures

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

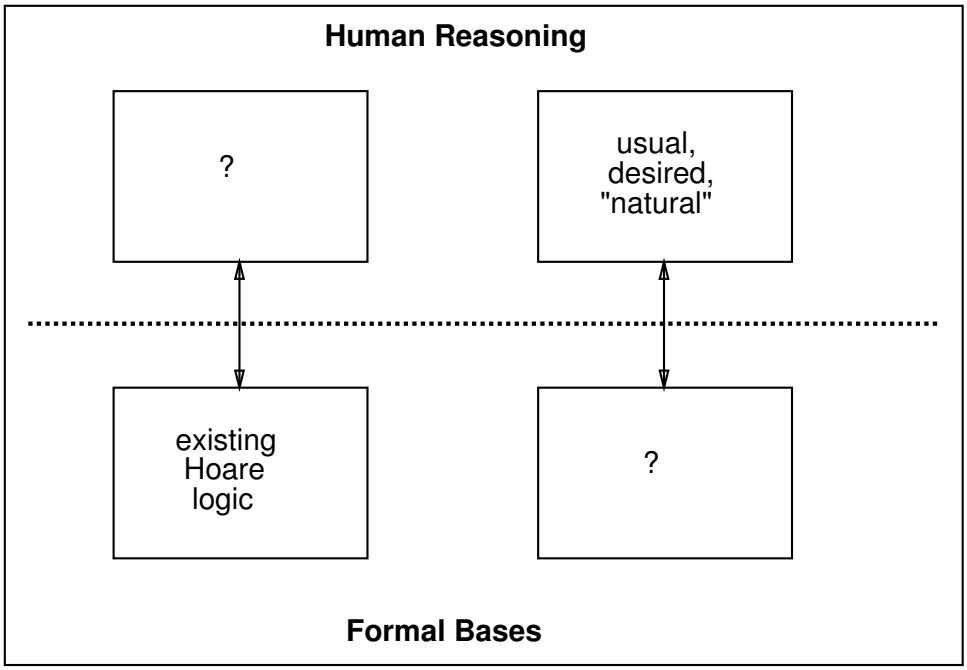

Figure 1: Human Reasoning and Formal Bases

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.

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

3. Show that the procedure body meets its specification.

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

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

```
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

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.

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

{**true**}
max := a
if (b > max) **then**
    max := b
end if
if (c > max) **then**
    max := c
end if
{(max = a ∨ max = b ∨ max = c)
    ∧ max ≥ a ∧ max ≥ b ∧ max ≥ c}

Figure 4: Traditional Back Substitution Method: First Step

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

$\{\mathbf{true}\}$
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

$\{\mathbf{true}\}$
max := a
$\{\ (b > max \Rightarrow ($
$(c > b \Rightarrow ((c = a \vee c = b \vee c = c)$
$\wedge c \geq a \wedge c \geq b \wedge c \geq c))$
$\wedge (c \leq b \Rightarrow ((b = a \vee b = b \vee b = c)$
$\wedge b \geq a \wedge b \geq b \wedge b \geq c))))$
$\wedge (b \leq max \Rightarrow ($
$(c > max \Rightarrow ((c = a \vee c = b \vee c = c)$
$\wedge c \geq a \wedge c \geq b \wedge c \geq c))$
$\wedge (c \leq max \Rightarrow ((max = a \vee max = b \vee max = c)$
$\wedge max \geq a \wedge max \geq b \wedge max \geq c))))\}$

Figure 6: Traditional Back Substitution Method: Third Step

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

Figure 7: Traditional Back Substitution Method: Fourth Step

replacing every occurrence of "max," in the postcondition of Figure 6, with "a"; see 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.

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

Figure 8: Traditional Back Substitution Method: Final Assertion

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 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

```
– – (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

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.

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 "$c_4 > max_4$" 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.

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.

[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.

$$
\begin{array}{l}
\quad -- (0) \\
\quad\ \ \max := a \\
\quad -- (1) \\
\quad\ \ \textbf{if } (b > \max) \textbf{ then} \\
b_1 > \max_1 \left\{ \begin{array}{l} -- (2) \\ \quad\ \ \max := b \\ -- (3) \end{array} \right. \\
\quad\ \ \textbf{end if} \\
\quad -- (4) \\
\quad\ \ \textbf{if } (c > \max) \textbf{ then} \\
c_4 > \max_4 \left\{ \begin{array}{l} -- (5) \\ \quad\ \ \max := c \\ -- (6) \end{array} \right. \\
\quad\ \ \textbf{end if} \\
\quad -- (7)
\end{array}
$$

Figure 10: Indexed Method: Mark Each Branch Condition

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

Figure 11: Indexed Method: Write Obligation at Last Position

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

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

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

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

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

if-then statement, is

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

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

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

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.

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,

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

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

$$
\begin{array}{ll}
 & -- (0) \\
 & \quad \textbf{fact } \max_1 = a_0 \wedge a_1 = a_0 \wedge b_1 = b_0 \wedge c_1 = c_0 \\
 & -- (1) \\
 & \quad \textbf{fact } b_1 > \max_1 \Rightarrow \\
 & \qquad (a_2 = a_1 \wedge b_2 = b_1 \wedge c_2 = c_1 \wedge \max_2 = \max_1) \\
b_1 > \max_1 \left\{ \right. & -- (2) \\
 & \quad \textbf{fact } b_1 > \max_1 \Rightarrow \\
 & \qquad (\max_3 = b_2 \wedge a_3 = a_2 \wedge b_3 = b_2 \wedge c_3 = c_2) \\
 & -- (3) \\
 & \quad \textbf{fact } (b_1 > \max_1 \Rightarrow \\
 & \qquad (a_4 = a_3 \wedge b_4 = b_3 \wedge c_4 = c_3 \wedge \max_4 = \max_3)) \\
 & \quad \wedge (b_1 \leq \max_1 \Rightarrow \\
 & \qquad (a_4 = a_1 \wedge b_4 = b_1 \wedge c_4 = c_1 \wedge \max_4 = \max_1)) \\
 & -- (4) \\
 & \quad \textbf{fact } c_4 > \max_4 \Rightarrow \\
 & \qquad (a_5 = a_4 \wedge b_5 = b_4 \wedge c_5 = c_4 \wedge \max_5 = \max_4) \\
c_4 > \max_4 \left\{ \right. & -- (5) \\
 & \quad \textbf{fact } c_4 > \max_4 \Rightarrow \\
 & \qquad (\max_6 = c_5 \wedge a_6 = a_5 \wedge b_6 = b_5 \wedge c_6 = c_5) \\
 & -- (6) \\
 & \quad \textbf{fact } (c_4 > \max_4 \Rightarrow \\
 & \qquad (a_7 = a_6 \wedge b_7 = b_6 \wedge c_7 = c_6 \wedge \max_7 = \max_6)) \\
 & \quad \wedge (c_4 \leq \max_4 \Rightarrow \\
 & \qquad (a_7 = a_4 \wedge b_7 = b_4 \wedge c_7 = c_4 \wedge \max_7 = \max_4)) \\
 & -- (7) \\
 & \quad \textbf{oblig } (\max_7 = a_7 \vee \max_7 = b_7 \vee \max_7 = c_7) \\
 & \qquad \wedge \max_7 \geq a_7 \wedge \max_7 \geq b_7 \wedge \max_7 \geq c_7
\end{array}
$$

Figure 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

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

Figure 17: Indexed Method: Final Assertion

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

[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

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

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

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.

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 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

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

Figure 20: Evidence of Unsoundness: First Step

procedure Stay_two (x : Integer)
    requires x = 2
    ensures x = 2 ∧ x = #x

Figure 21: A Specification of Procedure Stay_two

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 ≠ 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

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

Figure 22: Evidence of Unsoundness: Second Step

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

call we know two things: that the value of y is 2, and that the call has not changed its 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 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.

CHAPTER II

Syntax and Semantics

An *assertive program* is a computer program that not only gives the instructions to be followed by the computer when it executes the program, but also *asserts*, in the *specification* portions of the program, what is to be accomplished when the program is executed. An assertive program that satisfies its specification in every possible execution is said to be *valid* (or *correct*). The syntax and semantics of a related language (called "RESOLVE") for expressing assertive programs have been discussed in earlier work of the Reusable Software Research Group (RSRG) at The Ohio State University [44, 49, 22, 25, 16, 45].

Krone [25] has already established a method for proving correctness for assertive programs, the major contributions being rules for handling module-level constructs and interactions among modules. The rules governing procedure correctness followed the tradition of Hoare logic, which we have claimed is not as natural as our proposed method. We therefore replace the rules for procedure bodies with new rules that formalize the indexed method—the more natural method of reasoning presented in Section 1.5.2. We are not proposing changes to the rules for modules. The only

change we are proposing to the rules for procedure declaration is a single new rule that provides a bridge between Krone's rules and the indexed method.

Krone's system reduces assertive programs to assertions to be proved in ordinary logic. Furthermore, following Ernst et al. [12, p. 149], her system "views the intermediate objects in the reduction process as extended programs, thereby making verification a much less abstruse process. Treating logical assertions as commands appeals strongly to a programmer's intuition." Morgan [36, p. 403] proposed a similar programming-language extension to aid program refinement. He extended Dijkstra's programming language with *specification statements*.

> The goal is to improve the *development* of programs, making it closer to manipulations within a single calculus. The extension does this by providing one semantic framework for specifications and programs alike: Developments begin with a program (a single specification statement) and end with a program (in the executable language).

The approaches of Ernst et al. and Morgan are similar in that they both establish a single syntax and semantics for programs-with-specifications, i.e., assertive programs. The main difference is that Morgan focuses on developing an executable assertive program given an assertion serving as the specification, while Ernst et al. focus on deriving an assertion from a given assertive program to verify its correctness. We follow the approach of Ernst et al. and Morgan, giving a single syntax and semantics that encompasses all of the following:

1. executable assertive programs,

2. the final, derived, logical assertion, and

3. all the intermediate forms.

More specifically, we follow Ernst et al., focusing on the derivation of assertions from executable assertive programs for the purpose of verification.

The indexed method focuses on procedure bodies—bodies that do not contain declarations. In particular, these procedure bodies do not contain variable, procedure, or module declarations. It will be convenient to redefine our terminology to match this new restricted focus. Henceforth, we use the word "program" (including modifications such as "assertive program" and "executable program") to mean "procedure body." In Section 2.1 we define a program (i.e., a procedure body) to be a sequence of statements. The operational statements—procedure call, selection, and iteration—are likely familiar to the reader; they form the core of executable programs. The statements that permit programs to be assertive may be less familiar. Care will be taken to describe the statements that are important in the intermediate forms of the indexed method because these statements are new with this work. Section 2.1 gives an informal semantics for the statements along with their formal syntax. The formal semantics are presented in Section 2.2.

2.1 Syntax

2.1.1 Aspects of the Syntax That Are Context-Free

The context-free aspects of the syntax are defined here by a grammar in extended Backus-Naur form (EBNF), the extensions being the use of square brackets ("[" and "]") as metasymbols, indicating that the enclosed sequence of symbols is optional, and

Table 1: Nonterminal Symbols Whose Definitions Are Assumed

symbol	*description*	*example*
⟨p_nm⟩	procedure name	Inv_Abs
⟨cur_var⟩	current variable name	catalyst
⟨old_var⟩	old variable name	#catalyst
⟨ind_var⟩	indexed variable name	catalyst_5
⟨cur_var_list⟩	list of current variable names	q1, catalyst
⟨b_p_e⟩	Boolean-valued program expression	q1_empty **and** q2_empty
⟨cur_assert⟩	assertion, current variables	$x = 7$
⟨old_assert⟩	assertion, current and old variables	$\#x = 7 \wedge x = 49$
⟨idx_assert⟩	assertion, indexed variables	$x_0 = 7 \wedge x_3 = 49$
⟨nat_num⟩	natural number in decimal notation	9856

the use of braces ("{" and "}") as metasymbols, indicating that the enclosed sequence of symbols occurs any number of times (zero or more) in succession [39, pages 21–22]. The syntax we propose is generic with respect to appropriate definitions (e.g., grammars) for the nonterminal symbols shown in Table 1, with the restriction that an old variable name is a current variable name preceded by exactly one old sign (#):

$$\langle \text{old\_var} \rangle \quad ::= \quad \#\langle \text{cur\_var} \rangle \tag{2.1}$$

The old sign (#) was introduced at the beginning of Section 1.5.3 on page 32.

A program (i.e., a procedure body) is a sequence of zero or more statements:

$$\langle \text{program} \rangle \quad ::= \quad \{\langle \text{stmt} \rangle\} \tag{2.2}$$

There are nine different statements; we display their nonterminal symbols here in the

rewrite rule for ⟨stmt⟩, and describe each one in the discussion that follows.

$$\begin{aligned} \langle \text{stmt} \rangle \quad ::= \quad & \langle \text{call} \rangle \mid \langle \text{selec} \rangle \mid \langle \text{loop} \rangle \\ & \mid \langle \text{confirm} \rangle \mid \langle \text{assume} \rangle \mid \langle \text{remember} \rangle \\ & \mid \langle \text{whenever} \rangle \mid \langle \text{stow} \rangle \mid \langle \text{alter\_all} \rangle \end{aligned} \tag{2.3}$$

Operational Statements

We use the usual syntax for procedure call:

$$\langle \text{call} \rangle \quad ::= \quad \langle \text{p\_nm} \rangle (\langle \text{cur\_var\_list} \rangle) \tag{2.4}$$

Calling a procedure has the effect of changing the values of the actual parameters in ⟨cur_var_list⟩, and any other referenced state variables, according to the definition provided by the procedure's body. This is the usual meaning accorded to a procedure call. We will augment this meaning when we discuss handling the assertive part of assertive programs (see p. 50).

The selection (⟨selec⟩) and iteration (⟨iter⟩) statements are compound statements in that they each contain statement sequences. The selection statement has the usual syntax and meaning.

$$\begin{aligned} \langle \text{selec} \rangle \quad ::= \quad & \textbf{if } \langle \text{b\_p\_e} \rangle \textbf{ then} \\ & \quad \{\langle \text{stmt} \rangle\} \\ & [\textbf{else} \\ & \quad \{\langle \text{stmt} \rangle\}] \\ & \textbf{end if} \end{aligned} \tag{2.5}$$

The Boolean-valued program expression (⟨b_p_e⟩) is first evaluated. If the result is true, the statement sequence in the **then** part is executed, followed by the statements after the **end if**. If the result is false, the statement sequence in the **else** part (if present) is executed, followed by the statements after the **end if**.

The syntax of the iteration statement is a variation of the **loop** statement of Ada.

⟨iter⟩ ::= **loop** (2.6)

maintaining ⟨old_assert⟩

while ⟨b_p_e⟩ **do**

{⟨stmt⟩}

end loop

A syntactic slot introduced by the keyword **maintaining** provides a place for the loop invariant assertion. This assertion plays a role in the assertive part of assertive programs (see p. 51). Otherwise, the meaning of the iteration statement is the same as for Pascal's **while** statement. An execution begins with evaluation of the Boolean-valued program expression (⟨b_p_e⟩). If the result is true, the statement sequence of the body of the iteration statement is executed, followed by another execution of the iteration statement. If the result is false, execution moves on to the statements after the **end loop**.

The three kinds of operational statements (⟨call⟩, ⟨selec⟩, and ⟨loop⟩) are the only statements that a programmer may write that have the effect of changing the values associated with the current variables during execution. A programmer may write a

confirm statement, but execution of a **confirm** statement cannot change the value of a current variable. Execution of a **confirm** statement is a matter for the assertive part of assertive programs—a promised topic to which we now turn.

Statements That Permit Programs To Be Assertive

In the course of program execution, the assertion in a **confirm** statement may evaluate as false. Such an evaluation may be a witness to the program's invalidity. That is to say, subject to certain assumptions, the assertive program pledges that the assertion in a **confirm** statement will never evaluate as false. If all the assumptions have held true, a false **confirm** statement violates the pledge, rendering the program invalid.

We must have a way to record and explain an assertive program's pledges. The usual method of recording and explaining a program's behavior uses a function from the set of variable names into the universe of values; this function is called the program's *state*. The effect of a program statement is given by explaining how the statement changes the program's state. We use an expanded notion called the program's *environment* to explain the meaning of assertive programs. The environment comprises several components, including the *state* of the current variables. It also includes an *assert-status* to record and explain the pledges an assertive program makes [38, p. 67][12, p. 158][9, p. 8].

When the execution of an assertive program has not yet violated any assumptions, and a **confirm** statement evaluates as false, this fact is recorded by setting the assert-status to the value *categorically false*; the assertive program's pledge has been violated.

The remainder of the program's execution cannot atone for this violation, so the assert-status must remain at categorically false. That is to say, categorically false is a "stuck state" for the assert-status.

An assertive program states its assumptions in **assume** statements. When the execution of an assertive program has not yet set the assert-status to categorically false, and an **assume** statement evaluates as false, one of the program's assumptions has been violated. In this case, the assertive program is relieved of any further obligations. This situation is recorded by setting the assert-status to the stuck-state value of *vacuously true*.

An assert-status value is needed for indicating that neither assumption nor pledge has been violated; call this value *neutral*. Execution of a program statement never changes a non-neutral assert-status. Execution of an **assume** statement that evaluates as false changes a neutral assert-status to vacuously true. Execution of a **confirm** statement that evaluates as false changes a neutral assert-status to categorically false. When the assertion in either of these statements evaluates as true, the assert-status is left unchanged.

A programmer may write a **confirm** statement having an assertion that deals only with current variables. This is permitted as an aid to documentation. Also, some such assertions may be checkable at execution time. A programmer may use neither old nor indexed variables in a **confirm** statement. Programmers never write indexed variables; they arise only during application of the indexed method. The

indexed method permits programmers to use old variables only in the postconditions of procedure specifications and in the loop invariants of the **maintaining** clauses.

Application of Krone's proof rules produces **assume** and **confirm** statements whose assertions may contain old and current variables, while application of the indexed method produces **assume** and **confirm** statements whose assertions may contain indexed variables only. Hence, the general syntax of these two statements permits them to contain any of the three kinds of assertions.

$$\langle \text{confirm} \rangle \quad ::= \quad \textbf{confirm}\ \langle \text{assert} \rangle \tag{2.7}$$

$$\langle \text{assume} \rangle \quad ::= \quad \textbf{assume}\ \langle \text{assert} \rangle \tag{2.8}$$

$$\langle \text{assert} \rangle \quad ::= \quad \langle \text{cur\_assert} \rangle \mid \langle \text{old\_assert} \rangle \mid \langle \text{idx\_assert} \rangle \tag{2.9}$$

The environment we use to explain the meaning of assertive programs includes, of course, the state of indexed variables and the state of old variables.

Because they actively affect the execution of assertive programs by changing the assert status, we chose verbs as the names of the **assume** and **confirm** statements. These two statements correspond, respectively, to the **fact** and **oblig** keywords used in Chapter I. We chose nouns for these keywords because they marked statements of facts and obligations, which are objects.

We are now in a position to explain the effect of a procedure call in connection with the assert-status. In an environment having a non-neutral assert-status, the effect of any statement, including a procedure call, is to leave the environment unchanged. In the case of a neutral assert-status, if the procedure's precondition is violated by the values of the current variables in the environment, the assert-status is set to

categorically false and the remainder of the environment is left unchanged. Otherwise, the new values of the current variables, and the new value of the assert-status, are determined by the semantics of that procedure's declaration.

The effect of the iteration statement on a neutral assert-status is as follows. An execution begins with evaluation of the the assertion in the **maintaining** clause (i.e., the intended loop invariant). If the result is true, the Boolean-valued program expression in the **while** clause (⟨b_p_e⟩) is evaluated. If it evaluates to false, execution moves on to the statements after the **end loop**. Otherwise, the statement sequence of the body of the iteration statement is executed, followed by another execution of the iteration statement. If the result of evaluating the assertion in the **maintaining** clause is false, the assert-status is set to categorically false, and execution moves on to the statements after the **end loop**.

Old variable names (e.g., #x) are used in recording the state of current variables for reference later in the program's execution. An important use of old variable names is recording the values of formal parameters just prior to execution of the procedure body. This is done so that the truth of the procedure's postcondition, which typically asserts a relationship between parameters' initial and final values, can be evaluated. The **remember** statement records the current state of each current variable in its corresponding old variable name. For example, if the current values of x and y are 2 and 8, respectively, then execution of the **remember** statement sets the value of #x to 2 and the value of #y to 8. The syntax of the **remember** statement is simply the

one keyword:

$$\langle \text{remember} \rangle \quad ::= \quad \textbf{remember} \tag{2.10}$$

Programmers may not write **remember** statements. They arise by application of the proof rules of Krone's system.

Statements Needed for the Indexed Method's Intermediate Forms

The remaining three statements (⟨whenever⟩, ⟨stow⟩, and ⟨alter_all⟩) are not written by programmers, but arise by application of the indexed method's proof rules. They disappear again by the time the final mathematical assertion is produced. The **whenever** statement is a compound statement; it contains a statement sequence. It differs from the selection statement in two ways. Its test expression is a mathematical expression, not a program expression. It does not have an optional **else** part. All the variables in the test expression (⟨idx_assert⟩) are indexed variables.

$$\begin{aligned} \langle \text{whenever} \rangle \quad ::= \quad & \textbf{whenever } \langle \text{idx\_assert} \rangle \textbf{ do} \\ & \{ \langle \text{stmt} \rangle \} \\ & \textbf{end whenever} \end{aligned} \tag{2.11}$$

The meaning of a **whenever** statement is similar to that of an **if-then** statement. The test expression (⟨idx_assert⟩) is first evaluated. If the result is true, the statement sequence between **do** and **end whenever** is executed, followed by the statements after the **end whenever**. If the result is false, the **whenever** statement has no effect (except to continue execution with the statements after the **end whenever**).

The indexed method's proof rules use the **whenever** statement to construct branch conditions, a concept discussed in Chapter I.

Essential to the indexed method is the ability to refer to the values current variables had the last time execution reached any arbitrary point in the program. That is why the method associates each place between consecutive programmer-written statements with an index. This is accomplished with the **stow** statement. The effect of the **stow**(i) statement is to copy the value of each current variable to the corresponding variable with index i. For example, if the current values of x and y are 2 and 8, respectively, then execution of the **stow**(5) statement sets the value of x_5 to 2 and the value of y_5 to 8. The syntax of the **stow** statement is:

$$\langle \text{stow} \rangle \quad ::= \quad \textbf{stow}(\langle \text{nat\_num} \rangle) \tag{2.12}$$

The indexed method transforms programs to mathematical assertions. Doing so means removing operational statements—statements that change the values of current variables. In the intermediate stages, the removed operational statements must be replaced with a statement that permits the values of the current variables to change; the current variables cannot be left frozen at their previous values. The **alter all** statement does what is needed here. The effect of executing the **alter all** statement can be thought of as giving each current variable some arbitrary value of the right type. The syntax of the **alter all** statement is simply the pair of keywords:

$$\langle \text{alter\_all} \rangle \quad ::= \quad \textbf{alter all} \tag{2.13}$$

The context-free aspects of assertive-program syntax are collected in Figure 24.

⟨program⟩ ::= {⟨stmt⟩}
⟨stmt⟩ ::= ⟨call⟩ | ⟨selec⟩ | ⟨loop⟩
| ⟨confirm⟩ | ⟨assume⟩ | ⟨remember⟩
| ⟨whenever⟩ | ⟨stow⟩ | ⟨alter_all⟩
⟨call⟩ ::= ⟨p_nm⟩(⟨cur_var_list⟩)
⟨selec⟩ ::= **if** ⟨b_p_e⟩ **then**
{⟨stmt⟩}
[**else**
{⟨stmt⟩}]
end if
⟨iter⟩ ::= **loop**
maintaining ⟨old_assert⟩
while ⟨b_p_e⟩ **do**
{⟨stmt⟩}
end loop
⟨confirm⟩ ::= **confirm** ⟨assert⟩
⟨assume⟩ ::= **assume** ⟨assert⟩
⟨assert⟩ ::= ⟨cur_assert⟩ | ⟨old_assert⟩ | ⟨idx_assert⟩
⟨remember⟩ ::= **remember**
⟨whenever⟩ ::= **whenever** ⟨idx_assert⟩ **do**
{⟨stmt⟩}
end whenever
⟨stow⟩ ::= **stow**(⟨nat_num⟩)
⟨alter_all⟩ ::= **alter all**

Figure 24: Context-free Grammar of Assertive Programs

procedure Set_State_by_Addition (x, y : Integer)
referenced state variables z : Integer
requires MIN_INT $\leq$ x + y $\wedge$ x + y $\leq$ MAX_INT
ensures z = x + y $\wedge$ x = #x $\wedge$ y = #y

Figure 25: Specification of Procedure Set_State_by_Addition

2.1.2 Aspects of the Syntax That Are Not Context-Free

We adopt the usual non-context-free syntactic restrictions for programs; e.g., each actual parameter must have the same type as its corresponding formal parameter. To be syntactically correct, programs must also obey the additional restriction that, in any given procedure call, an identifier may appear at most once in the list of variable names; duplicate actual parameters are not permitted. The procedure's *referenced state variables* (i.e., "global variables") are considered actual parameters in determining whether there is such duplication. We illustrate what we mean by this restriction, and the reason for it, using two examples.

Consider the procedure Set_State_by_Addition specified in Figure 25. Let us assume for these examples that variables a and b have been declared to be of type Integer. With the above-stated restriction in force, the following procedure call would not be permitted:

Set_State_by_Addition (b, z) (2.14)

The restriction applies to this case due to z, a referenced state variable of Set_State_by_Addition, appearing as an actual parameter. The reason this can be a problem becomes apparent when we substitute the actual parameters into the procedure's

postcondition, yielding three equations.

$$z = b + z \tag{2.15}$$

$$b = \#b \tag{2.16}$$

$$z = \#z \tag{2.17}$$

In general, b can have nonzero values. But this generality is ruled out by equation 2.15, which forces b to zero. One might object that equation 2.15 would be better expressed using the old sign (#):

$$z = b + \#z \tag{2.18}$$

However, equation 2.17 makes this a distinction without a difference.

The trouble arose because we expressed the postcondition with three seemingly independent variables, z, x, and y, but our choice of (duplicate) actual parameters introduced a dependency. The value of z cannot both be preserved (equation 2.17) and increased by b (unless b happens to be zero). One might try to deal with this problem by introducing a complex specification system that produces a different specification depending on the choice of actual parameters. For example, such a system might state that the actual parameter corresponding to the formal parameter y is preserved *unless* it happens to be z, in which case it is increased by the value of the actual parameter corresponding to x. We prefer to deal with this problem, as Cook did [4, p. 76], by prohibiting duplicate actual parameters. Doing so assures that the different variables used to express a postcondition will not obtain dependencies from the choice of actual parameters.

procedure Add (z, x, y : Integer)
    requires MIN_INT $\leq x + y \wedge x + y \leq$ MAX_INT
    ensures $z = x + y \wedge x = \#x \wedge y = \#y$

Figure 26: Specification of Procedure Add

This restriction can rule a procedure call syntactically incorrect even when the called procedure has no referenced state variables. Neither of these calls to procedure Add of Figure 26 is permitted, even though call 2.20 happens to present no problem for this particular specification. (It might, however, be trouble for some implementations of Add!)

Add (a, b, a) (2.19)

Add (b, a, a) (2.20)

2.2 Semantics

The *semantics* of a programming language defines what a computer is supposed to do when it executes a program written in the language. This section gives a semantics for the assertive programs whose syntax is defined in Section 2.1. Only the operational statements (⟨call⟩, ⟨selec⟩, and ⟨iter⟩) are intended to be executed by real computers, although the choice in some systems might be to also execute a class of **confirm** ⟨cur_assert⟩ statements for testing and debugging purposes. The other statements arise temporarily in the translation from procedure body to mathematical assertion when the proof rules of Chapter III are applied. The semantics given here also define how these intermediate forms should be "executed," this definition being important

to our argument for the soundness and relative completeness of the proof rules, which we discuss in Chapter IV.

Earlier in the history of programming languages, some researchers [18, 13] were inclined to define a language's semantics in terms of proof rules. This kind of semantic definition came to be called *axiomatic semantics* [39, p. 193]. These researchers were attracted to the substantial advantages [18, pp. 579, 580, and 583] of having a deductive system—a calculus—consisting of proof rules for reasoning about program behavior. The direct route to obtaining such a deductive system—with its advantages—was simply to let the proposed proof rules define the language's semantics.

It was not too long before some of these same researchers noted shortcomings of this direct approach. It is not entirely clear how to establish whether a given compiler and run-time system satisfies a set of proof rules. A 1974 paper by Hoare and Lauer [21, p. 136] suggested

> ... an approach to the solution of these problems. It is based on the realisation that a single formal definition is unlikely to be equally acceptable to both implementor and user, and that at least two definitions are required, a constructive one to act as a guide and model for the implementor, and an implicit one for the user, who is interested in what his program accomplishes, as much as in how it does it. The doubt arises whether the two descriptions describe the same language; but this doubt can be completely resolved by a mathematical proof of the consistency of the two definitions; and then the pair of complementary definitions can serve together as an interface between implementor and user, which is just as rigorous but possibly more acceptable to each of them than a single definition could be.

Furthermore, if a set of proof rules is inconsistent, then it is useless: incorrect assertions can be proved in such a system. In 1978, Cook [4, p. 70] noted:

> The rules for procedure call statements often (in fact usually) have technical bugs when stated in the literature, and the rules stated in earlier versions of the present paper are not exceptions. In the process of trying to prove the soundness of these rules, I uncovered some of the bugs, and this led me to believe a careful and detailed proof of soundness is necessary to have any confidence that there are no further bugs.

Cook [4, p. 70] approached his justification of the soundness (i.e., consistency) of his axiom system "by introducing an interpretive semantics for the language." By associating each program with a well-defined function, his semantics served to define a notion of *truth*, answering the question: "What exactly does it mean to execute a given program?" Cook's axiom system, on the other hand, served to define the separate notion of *proof*, answering the different question: "Which program-specification pairs can be derived according to the rules of the system?" In the tradition of the twentieth-century development of mathematical logic, he recognized the value of separating these two notions (truth and proof); having been distinguished, they could be compared and contrasted with each other. Cook used comparison to show that his definition of proof is consistent with his definition of truth. Further evidence that Cook separated semantics from proof rules is that, although he included "axiomatic semantics" in his paper's list of key words, he did not use the term in the paper's text, preferring the term "axiom system."

Contrasts between Cook's notions of truth and proof help us see the value gained by separating them and showing proof to be consistent with truth. Defining the association of a function with each program is a straightforward enterprise, admitting

few surprises; hence, his definition of truth is simple. On the other hand, interactions among axioms and rules of an axiom system are complex—not easily predicted. We can understand and evaluate the notion of truth more easily than that of proof. Another contrast is that manipulations and reasoning are more easily performed in an axiom system than they are using the function definitions of the notion of truth. We can work more effectively using proof than we can using truth. This effectiveness is due to the fact that the axiom system (proof) leaves out unnecessary details involved in the semantics (truth); the axiom system is more abstract than the semantics.

Hence, we can evaluate the notion of truth, effectively determining if its definition is adequate and correct. Then, having shown the more abstract notion of proof to be consistent with the notion of truth, we can work effectively in the proof system, confident that all things we properly prove are also true. These are reasons why Cook established a less abstract semantics for the programming language as a basis for justifying the soundness of the proof rules.

We use sets, functions, and relations to define the semantics of assertive programs because set theory is well understood, having been widely used this century for many purposes, including reasoning about mathematics. That is to say, set theory is commonly used for doing *meta*-mathematics, with great success. In contrast, a new deductive system, such as the one proposed in Chapter III, is not well understood. Without further examination, we do not know whether such a deductive system is sound. Our strategy is to define the semantics of assertive programs on the firm foundation of set theory, and, then, in Chapter IV, discuss our set-theoretic proof

that Chapter III's deductive system is sound. Having shown its soundness, we can, henceforth, confidently use the convenience of the deductive system to reason about assertive programs.

2.2.1 Semantic Space

Programs usually have been interpreted as mappings from states into states, where a state gives the current values of the program variables. Such an interpretation is called a *denotational semantics* [39, pp. 167–93]. Navlakha [38, p. 67] and Ernst et al. [12, p. 158][9, p. 8] have shown both the need for and the utility of a richer semantic space for describing the effect of executing programs containing declarations of the specifications of external procedures. We shall see in Chapter IV that their idea of *assert-status* is a crucial tool for showing the soundness and relative completeness of the indexed method. The semantic space we need here is an augmented form of the space Ernst et al. used in [10, pp. 267–270], and its various domains are given in Figure 27. The reader is likely to be unfamiliar with this richer semantic space, so we provide the following explanations associated with various parts of Figure 27.

Figure 27, Part 1: Interpretation $\mathcal{I}$

We interpret programs as mappings from environments to environments. Assertions (the contents of **maintaining** clauses and **assume** and **confirm** statements) and Boolean-valued program expressions ($\langle$b_p_e$\rangle$) are interpreted by function $\mathcal{I}$ as mappings from environments to true or false (Boolean) because they have no effect on the environment.

1. $\mathcal{I}$: (Programs) $\rightarrow$ (Environments $\rightarrow$ (Environments $\cup$ Boolean))
2. Environments = Assert-statuses $\times$ Current-states $\times$ Old-states $\times$ Index-states $\times$ Setups $\times$ Declaration-meanings
3. Assert-statuses = $\{\mathrm{VT}, \mathrm{CF}, \mathrm{NL}\}$
4. Current-states = (Current-variable-names $\rightarrow$ Values)
5. Index-states = (Integers $\rightarrow$ Current-states)
6. Setups = Current-states*
7. Old-states = (Augmented-old-variable-names $\rightarrow$ Values)
8. Declaration-meanings = (Identifiers $\rightarrow$ (Type-meanings $\cup$ Predicate-meanings $\cup$ Procedure-meanings $\cup$ Generic-meanings $\cup$ Module-meanings))
9. Values = $\{v \mid$ there exists t $\in$ Type-meanings such that $v \in$ t$\}$
10. Type-meanings = Base-types $\cup$ Defined-types
11. Predicate-meanings = $\{p \mid p : \mathrm{d} \rightarrow \mathrm{t} \wedge \mathrm{d} \in$ Argument-domains $\wedge$ t $\in$ Type-meanings$\}$
12. Argument-domains = $\{T_1 \times \ldots \times T_n \mid T_i \in$ Type-meanings $\wedge\, 1 \leq n \wedge 1 \leq i \leq n\}$
13. Procedure-meanings = Predicate-meanings $\times$ Predicate-meanings $\times$ Procedure-functions $\times$ Status-functions
14. Procedure-functions = $\{f \mid f : \mathrm{d} \rightarrow \mathrm{r} \wedge \mathrm{d} \in$ Argument-domains $\wedge$ r $\in$ Argument-domains$\}$
15. Status-functions = $\{f \mid f : \mathrm{d} \rightarrow$ Assert-statuses $\wedge\, \mathrm{d} \in$ Argument-domains$\}$

Figure 27: Definition of Domains in the Semantic Space

Figure 27, Part 2: Environment

An environment consists of six kinds of information about the execution of a program. The most obvious part of an environment is the current-state, which maps the names of program variables (i.e., current variables) into their values (part 4 of Figure 27). Because the current-state is part of the environment, and we are interpreting programs as mappings from environments to environments, the kind of semantics we are providing is, still, denotational.

Figure 27, Part 3: Assert-status

The assert-status keeps track of the effect of executing the assertions in a program. Certain assertions in a program must be true in order for it to be correct, e.g., the precondition of a procedure called by the program. When such an assertion is violated, the assert-status becomes categorically false (CF), and remains so for the rest of the program's execution. Thus, the violation of a condition necessary for correctness is indelibly recorded. The **confirm** statement, which arises during application of the proof rules, contains an assertion that must be true.

On the other hand, a program's correctness can rely on certain other assertions being true. For example, if the program includes a specification of a procedure, declaring this procedure to be externally defined, then the program assumes that the environment contains a meaning for that procedure that matches the specification. When such an assumption is violated, the assert-status becomes, and remains, vacuously true (VT). Further execution of the program in an environment that contains

a non-matching procedure meaning is of no use in determining the program's correctness. That is why such environments are "filtered out" by setting the assert-status to vacuously true (VT). The **assume** statement, which arises during application of the proof rules, contains an assertion that can be assumed to be true.

The neutral assert-status (NL) indicates that no assertion in a program has been violated so far. We define a *neutral environment* to be one in which assert-status equals NL. A *categorically false environment* has a CF assert-status, and the assert-status of a *vacuously true environment* equals VT.

Figure 27, Part 5: Index-state

As discussed in Section 1.5.2, the indexed method employs many new variable names, obtained by using the unique integers as subscripts on the names of the program variables. An index-state maps each integer to a current state, which is a mapping from current variable names to values. For example, let x be an Integer program variable, and let x_3 be one of the indexed variables associated with it. Let ns be an index-state. Then ns(3) is a current state. If ns(3) maps x to 275 (i.e., if ns(3)(x) = 275), the value of x_3 is 275.

Figure 27, Part 6: Setup

We employ the "setup" to help define the meaning of the **alter all** statement. A setup is a finite sequence of current-states (as indicated by our use of the Kleene * in the figure). The **alter all** statement changes the current-state and the setup as follows: it makes the new value of the current-state become the state which is at

the front of the setup (sequence), and removes this state from the front of the setup. The term "setup" is a slang expression for a situation (e.g., a trial jury!) that has been "rigged" (arranged in advance). The initial environment contains, in the setup component, a prearrangement of the meaning of each **alter all** statement that might be encountered during execution.

Figure 27, Part 7: Old-state

Depending on whether an old variable, $\#\xi$, occurs in an **ensures** clause or a **maintaining** clause, it refers to the value of parameter ξ at procedure invocation or to the value of variable ξ at the beginning of the loop. The name of an old variable contains exactly one old sign (#), and it is always the first character (see page 45). Because loops occur inside procedures and can be nested within one another, interpreting these programs requires remembering the values of old variables on a last-in-first-out basis. We can use an expanded name space to accomplish this task. Therefore, as an aid to defining old-states, we define a new set, "Augmented-old-variable-names," as the set of all names that can be rewritten from the nonterminal symbol ⟨aug_old_var⟩ in our grammar with the additional rewrite rule:

$$\langle \text{aug\_old\_var} \rangle \quad ::= \quad \{\#\}\#\langle \text{cur\_var} \rangle \tag{2.21}$$

An augmented old variable name has one or more old signs (#) in its prefix. We illustrate the use of augmented old variable names in our discussion of the semantics of iteration, which begins on page 75.

Figure 27, Part 8: Declaration-meaning

A declaration meaning maps certain global names into their semantics. These global names include types, procedures, functions, generics, and modules. Depending on the programming language, other kinds of objects may have to be included in the declaration meanings. This dissertation is concerned only with using the meanings of these objects in defining the meaning of procedure bodies. In our semantics for procedure bodies, their executions make no change to the declaration meanings, but are, of course, affected by them.

How program declarations establish the value of declaration-meanings is discussed in other papers [38, 12, 9, 10, 11, 25]. These papers do not attempt to handle modules because the problem of module semantics has not been adequately solved for their purposes. Neither they nor we deny the importance of modules.

Figure 27, Parts 9 and 10: Value and Type-meaning

The set of values that a variable of a given type can have is the semantics of that type. We call the predefined types "base types." The types that are used to define abstract types vary from application to application; we call them "defined types." The collection of all values of all types is the set we call "Values."

Figure 27, Parts 11 and 12: Predicate-meaning

The only difference between predicates and functions of the specification language is that possible values that can be produced by applying a function to legal arguments can be of any one type, not just Boolean, as is the case for predicates. Therefore,

$$
\begin{array}{rcl}
\text{AE} & : & \text{Environments} \rightarrow \text{Assert-statuses} \\
\text{CSE} & : & \text{Environments} \rightarrow \text{Current-states} \\
\text{ISE} & : & \text{Environments} \rightarrow \text{Index-states} \\
\text{SPE} & : & \text{Environments} \rightarrow \text{Setups} \\
\text{OSE} & : & \text{Environments} \rightarrow \text{Old-states} \\
\text{DME} & : & \text{Environments} \rightarrow \text{Declaration-meanings}
\end{array}
$$

Figure 28: Six Projection Functions on Environments

we combine the semantics of predicates and functions of the specification language in Figure 27, parts 11 and 12. A predicate-meaning whose range is $\{\mathbf{true}, \mathbf{false}\}$ is what is normally called a predicate; other members of predicate-meanings correspond to functions of the specification language. We defer discussion of parts 13, 14, and 15 of Figure 27 (which help define the meaning of procedure calls) to the next section (2.2.2), where we define our language's semantics.

2.2.2 Semantic Definition

We define, in this section, the function $\mathcal{I}$ of Figure 27, part 1. Doing so establishes the meaning of any syntactically correct program (as defined in Section 2.1). In the following definitions, we let env stand for an arbitrary environment, and S1 for a statement ($\langle$stmt$\rangle$). Each of SS, SS1, and SS2 stands for a (possibly empty (ε)) sequence of statements ($\{\langle$stmt$\rangle\}$). We will need to refer to the various components of any environment. The six projection functions shown in Figure 28 allow us to do so. Figure 29 expresses a convenient notation for arbitrary environment env, i.e.,

$$\text{env} = [\text{a}, \text{cs}, \text{ns}, \text{se}, \text{os}, \text{d}] \tag{2.22}$$

$$\begin{array}{rcll} \mathrm{AE(env)} & = & \mathrm{a} & \text{(Assert-status)} \\ \mathrm{CSE(env)} & = & \mathrm{cs} & \text{(Current-state)} \\ \mathrm{ISE(env)} & = & \mathrm{ns} & \text{(Index-state)} \\ \mathrm{SPE(env)} & = & \mathrm{se} & \text{(Setup)} \\ \mathrm{OSE(env)} & = & \mathrm{os} & \text{(Old-state)} \\ \mathrm{DME(env)} & = & \mathrm{d} & \text{(Declaration-meaning)} \end{array}$$

Figure 29: Notation for Environment Named "env"

In the usual fashion, we define the interpretation of a sequence of statements as the interpretation of the tail sequence in the environment resulting from interpreting the first statement in the sequence:

$$\mathcal{I}(\mathrm{S1\ SS2})(\mathrm{env}) \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathrm{SS2})(\mathcal{I}(\mathrm{S1})(\mathrm{env})) \tag{2.23}$$

The interpretation of the empty sequence of statements is the identity function:

$$\mathcal{I}(\varepsilon)(\mathrm{env}) \stackrel{\mathrm{def}}{=} \mathrm{env} \tag{2.24}$$

The interpretation of any statement in either a vacuously true or a categorically false environment is the identity function; the environment remains unchanged by the interpretation:

$$\mathcal{I}(\mathrm{S1})([\mathrm{VT}, \mathrm{cs}, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}]) \stackrel{\mathrm{def}}{=} [\mathrm{VT}, \mathrm{cs}, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}] \tag{2.25}$$

$$\mathcal{I}(\mathrm{S1})([\mathrm{CF}, \mathrm{cs}, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}]) \stackrel{\mathrm{def}}{=} [\mathrm{CF}, \mathrm{cs}, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}] \tag{2.26}$$

We now need only define the interpretation of each nonempty statement in a neutral environment. So, in the remaining definitions, let

$$\mathrm{env}_{\mathrm{NL}} = [\mathrm{NL}, \mathrm{cs}, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}] \tag{2.27}$$

The only effect of the **stow**(i) statement is to change the value of the index-state at i to equal the current-state cs:

$$\mathcal{I}(\textbf{stow}(i))(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} [\text{a}, \text{cs}, \text{ns}', \text{se}, \text{os}, \text{d}] \qquad \text{where} \tag{2.28}$$

$$\text{ns}'(k) = \begin{cases} \text{ns}(k) & \text{if } k \neq i \\ \text{cs} & \text{if } k = i \end{cases} \tag{2.29}$$

The main purpose of the **alter all** statement is to change the value of the current-state; it changes the setup so that the next **alter all** statement executed can change the current-state to yet another state. An **alter all** statement will change a neutral environment whose setup is empty to a vacuously true environment because such an environment does not provide sufficient information to further evaluate the validity of the program. In other words, a characteristic of truly useful initial environments is that their setups are long enough to handle all the **alter all** statements encountered during execution:

$$\mathcal{I}(\textbf{alter all})(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} [\text{a}', \text{cs}', \text{ns}, \text{se}', \text{os}, \text{d}] \qquad \text{where} \tag{2.30}$$

$$\text{a}' = \begin{cases} \text{VT} & \text{if se} = \varepsilon \\ \text{NL} & \text{if se} \neq \varepsilon \end{cases} \tag{2.31}$$

$$\text{cs}' = \begin{cases} \text{cs} & \text{if se} = \varepsilon \\ \text{first(se)} & \text{if se} \neq \varepsilon \end{cases} \tag{2.32}$$

$$\text{se}' = \begin{cases} \text{se} & \text{if se} = \varepsilon \\ \text{tail(se)} & \text{if se} \neq \varepsilon \end{cases} \tag{2.33}$$

Let Q be an assertion (⟨cur_assert⟩, ⟨old_assert⟩, or ⟨idx_assert⟩). We define $\mathcal{I}(\text{Q})(\text{env})$ to be the usual interpretation of Q as a predicate logic expression, where the assignments to the free variables are determined by cs, ns, and os. Consequently, $\mathcal{I}(\text{Q})(\text{env}) \in \{\text{true}, \text{false}\}$. Interpreting **assume** Q and **confirm** Q in a neutral environment, say env_{NL}, affects only the assert-status, which is changed if and only if

$\mathcal{I}(\mathrm{Q})(\mathrm{env_{NL}})$ is false:

$$\mathcal{I}(\textbf{assume Q})(\mathrm{env_{NL}}) \stackrel{\text{def}}{=} [\mathrm{a}', \mathrm{cs}, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}] \quad \text{where} \tag{2.34}$$

$$\mathrm{a}' = \begin{cases} \mathrm{NL} & \text{if } \mathcal{I}(\mathrm{Q})(\mathrm{env_{NL}}) \\ \mathrm{VT} & \text{if } \neg\mathcal{I}(\mathrm{Q})(\mathrm{env_{NL}}) \end{cases} \tag{2.35}$$

$$\mathcal{I}(\textbf{confirm Q})(\mathrm{env_{NL}}) \stackrel{\text{def}}{=} [\mathrm{a}', \mathrm{cs}, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}] \quad \text{where} \tag{2.36}$$

$$\mathrm{a}' = \begin{cases} \mathrm{NL} & \text{if } \mathcal{I}(\mathrm{Q})(\mathrm{env_{NL}}) \\ \mathrm{CF} & \text{if } \neg\mathcal{I}(\mathrm{Q})(\mathrm{env_{NL}}) \end{cases} \tag{2.37}$$

Interpreting a **whenever** statement in a neutral environment, depending on the interpretation of the assertion Q (all of whose variables, according to the syntax, are indexed variables), either leaves the environment unchanged or changes it just as the body of the **whenever** statement would:

$$\mathcal{I}(\textbf{whenever Q do SS end whenever})(\mathrm{env_{NL}}) \tag{2.38}$$
$$\stackrel{\text{def}}{=} \begin{cases} \mathcal{I}(\mathrm{SS})(\mathrm{env_{NL}}) & \text{if } \mathcal{I}(\mathrm{Q})(\mathrm{env_{NL}}) \\ \mathrm{env_{NL}} & \text{if } \neg\mathcal{I}(\mathrm{Q})(\mathrm{env_{NL}}) \end{cases}$$

Consider now the interpretation of Boolean-valued program expressions ($\langle$b_p_e$\rangle$). Our simple, example programming language includes only *proper* procedures, not *function* procedures, i.e., procedure calls appear only as statements, not in expressions as function calls. Consequently, our Boolean-valued program expressions consist only of program variables, Boolean-operator key words, and parentheses for overriding operator precedence. Therefore, it is easy to define their interpretation to involve no change to the environment. We define $\mathcal{I}(b\_p\_e)(\mathrm{env})$ to be the usual interpretation of a propositional logic expression, where the assignments to the free variables are

determined by cs. Consequently, $\mathcal{I}(b\_p\_e)(\text{env}) \in \{\text{true}, \text{false}\}$.[3] Interpreting an **if-then** statement in a neutral environment, depending on the interpretation of the Boolean-valued program expression b_p_e, either leaves the environment unchanged or changes it just as the body of the **if** statement would:

$$\mathcal{I}(\textbf{if } b\_p\_e \textbf{ then } \text{SS } \textbf{end if})(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} \begin{cases} \mathcal{I}(\text{SS})(\text{env}_{\text{NL}}) & \text{if } \mathcal{I}(b\_p\_e)(\text{env}_{\text{NL}}) \\ \text{env}_{\text{NL}} & \text{if } \neg\mathcal{I}(b\_p\_e)(\text{env}_{\text{NL}}) \end{cases} \tag{2.39}$$

Interpreting an **if-then-else** statement in a neutral environment changes it either just as the body of the **then** portion of the statement or just as the body of the **else** portion of the statement would:

$$\begin{aligned} &\mathcal{I}(\textbf{if } b\_p\_e \textbf{ then } \text{SS1 } \textbf{else } \text{SS2 } \textbf{end if})(\text{env}_{\text{NL}}) \\ &\stackrel{\text{def}}{=} \begin{cases} \mathcal{I}(\text{SS1})(\text{env}_{\text{NL}}) & \text{if } \mathcal{I}(b\_p\_e)(\text{env}_{\text{NL}}) \\ \mathcal{I}(\text{SS2})(\text{env}_{\text{NL}}) & \text{if } \neg\mathcal{I}(b\_p\_e)(\text{env}_{\text{NL}}) \end{cases} \end{aligned} \tag{2.40}$$

Semantics of Procedure Call

This section follows the explanation given by Ernst et al. [10, pp. 268–269].

> The semantics of a procedure has four different components ([part 13, Figure 27]). The first is the domain-predicate which is a predicate on the parameters of the procedure. A procedure is usually designed only for a subset of all possible values for its parameters; the domain-predicate specifies this subset. The second component of a procedure-meaning is the effect-predicate which is an "I/O relation" for the procedure. It is

[3] The possibility of simplifying the example language by excluding function procedures was suggested by Hollingsworth's dissertation [22] where he established forty-one principles for building high-quality software in Ada. Principle number nine demands that all operations be proper procedures. Hollingsworth's discipline would not ban function procedures in all languages; it does not ban them in C++, and the research language, RESOLVE, has function procedures. But there are technical problems with function procedures in Ada [22, pp. 52–56]. In RESOLVE, function procedures are specified as preserving the environment; so, any program expression (including those that are Boolean-valued), when evaluated in a neutral environment, leaves the environment unchanged and returns a value.

a predicate whose arguments are the values of the input and output parameters of the procedure. The purpose of this predicate is to indicate which values of the output parameters are correct for the input parameter values.

Each procedure also has a procedure-function, [definition 14 in Figure 27]. This function maps the values of input parameters into the values for the output parameters. Finally, the status-function maps the input parameters into the new assert-status for the environment produced by executing the procedure ([definition 15, Figure 27]).

A procedure-meaning contains the semantics of both the specification and implementation of a procedure; i.e., the domain- and effect-predicates are the semantics of its specification and the semantics of its implementation are the procedure- and status-functions. The semantics of a correct procedure is a procedure-meaning in which the implementation semantics does not violate the semantics of the specification. Such a procedure-meaning is defined to be *conformal* if the following conditions are true: let $\mathrm{pd} = [\mathrm{dp}, \mathrm{ep}, \mathrm{pf}, \mathrm{sf}]$ be the procedure-meaning. Then,

1. the number and types of arguments of the various components of pd are consistent with one another;
2. for all $\mathrm{x}_1, \ldots, \mathrm{x}_n$ such that $\mathrm{dp}(\mathrm{x}_1, \ldots, \mathrm{x}_n)$ and for all $\mathrm{y}_1, \ldots, \mathrm{y}_m$ such that $\mathrm{pf}(\mathrm{x}_1, \ldots, \mathrm{x}_n) = [\mathrm{y}_1, \ldots, \mathrm{y}_m]$,
   $\mathrm{sf}(\mathrm{x}_1, \ldots, \mathrm{x}_n) \neq \mathrm{CF}$ and
   if $\mathrm{sf}(\mathrm{x}_1, \ldots, \mathrm{x}_n) = \mathrm{NL}$ then $\mathrm{ep}(\mathrm{x}_1, \ldots, \mathrm{x}_n, \mathrm{y}_1, \ldots, \mathrm{y}_m)$.

In these conditions the xs represent inputs to the procedure and the ys represent the outputs of the procedure. The first condition requires that the components of a procedure-meaning have arguments which are consistent with one another. For example, the status-function and the procedure-function must have the same number and types of arguments because their arguments are just the input parameters of the procedure. The xs and ys in the above conditions show the number of arguments to the various components of pd and also when one argument must have the same type as another argument. Condition (2) applies to those input parameters x_i which make the domain-predicate true: the status-function must not be CF and if it is NL, then the effect-predicate must also be true of the output parameters y_i produced by pf. Intuitively this condition requires the outputs of the procedure to satisfy its postcondition whenever its inputs satisfy its precondition. If the procedure does not terminate, then it has no outputs and the condition is true by default. The condition also states

> that there is no violation of required assertions such as the precondition of an externally defined procedure when it is invoked inside the procedure. The semantics of such externally defined procedures are given by the declaration-meanings component of the environment.
>
> A procedure declaration essentially defines the various components of a procedure-meaning. The interpretation of a procedure's precondition is, roughly speaking, the domain-predicate of the procedure; a more precise description is given in Ernst et al. [9]. Similarly the interpretation of a procedure's postcondition gives its effect-predicate. By use of the minimum fixed-point operation, the body of a procedure defines its procedure-function and its status-function in a manner similar to that in Scott and Strachey [43].

The interpretation of a procedure declaration D in a neutral environment env = [NL, cs, ns, se, os, d] is defined as follows: $\mathcal{I}(D)(\text{env}) = [\text{a}', \text{cs}, \text{ns}, \text{se}, \text{os}, \text{d}]$ because the interpretation of D affects only the assert-status component of the environment [10, p. 269].

> The reason it does not affect d is that any env with the wrong meaning for D is filtered out by making $\text{a}' = \text{VT}$. Thus, when a' is not VT, then env has the correct meaning for D. If the procedure-meaning pm defined by D as described above is not conformal, then $\text{a}' = \text{CF}$ because D is incorrect. If pm differs from the procedure-meaning $\text{d}(p)$ stored in the environment, then $\text{a}' = \text{VT}$. In the remaining case $\text{a}' = \text{NL}$ because D is correct and env has the intended interpretation of D. This defines $\mathcal{I}(D)$ for all environments because once the assert-status becomes VT or CF, the environment remains unchanged; i.e., $\mathcal{I}(D)(\text{env}) = \text{env}$ whenever env is not a neutral environment.
>
> This definition applies to all environments, even though some environments are not intended for the declaration. For example, D may be a procedure with three arguments and $\text{d}(p)$ may be a procedure with a single argument. The interpretation "filters out" such meaningless environments by setting the assert-status to VT. The correctness of a program is determined by its interpretation in meaningful environments and the ones that are filtered out are essentially ignored. [10, p. 269]

We are now in position to define the interpretation of a procedure call. We do so for a procedure that has two formal parameters and one referenced state variable. It is straightforward to use this definition as a guide to the semantics of calling a procedure that has a different number of parameters and referenced state variables. We assume that declaration-meaning d of environment env contains $\mathrm{d}(\mathrm{P\_nm}) = [\mathrm{dp}, \mathrm{ep}, \mathrm{pf}, \mathrm{sf}]$, the procedure-meaning corresponding to the following specification:

$$\begin{aligned} &\textbf{procedure } \mathrm{P\_nm}(\mathrm{x}, \mathrm{y} : \mathrm{T}_1) \\ &\quad \textbf{referenced state variables } \mathrm{z} : \mathrm{T}_3 \\ &\quad \textbf{requires } \mathrm{pre}\lfloor \mathrm{x}, \mathrm{y}, \mathrm{z} \rfloor \\ &\quad \textbf{ensures } \mathrm{post}\lfloor \mathrm{x}, \#\mathrm{x}, \mathrm{y}, \#\mathrm{y}, \mathrm{z}, \#\mathrm{z} \rfloor \end{aligned} \tag{2.41}$$

Because we are only defining the intepretation of a call to P_nm in a neutral environment (i.e., $\mathrm{AE}(\mathrm{env}) = \mathrm{NL}$), we have that dp is the same as pre, ep is the same as post, and d(P_nm) is conformal. Let ξ-proj be the projection function for the ξ-component of the range of the procedure function pf, where $\xi \in \{\mathrm{x}, \mathrm{y}, \mathrm{z}\}$. Letting ac and ad be two distinct variables of type T_1, we define:

$$\mathcal{I}(\mathrm{P\_nm}(\mathrm{ac}, \mathrm{ad}))(\mathrm{env}) \stackrel{\mathrm{def}}{=} [\mathrm{a}', \mathrm{cs}', \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}] \qquad \text{where} \tag{2.42}$$

$$\mathrm{a}' = \begin{cases} \mathrm{sf}(\mathrm{cs}(\mathrm{ac}), \mathrm{cs}(\mathrm{ad}), \mathrm{cs}(\mathrm{z})) & \text{if } \mathrm{dp}(\mathrm{cs}(\mathrm{ac}), \mathrm{cs}(\mathrm{ad}), \mathrm{cs}(\mathrm{z})) \\ \mathrm{CF} & \text{otherwise} \end{cases}, \text{ and} \tag{2.43}$$

$$\mathrm{cs}'(\xi) = \begin{cases} \mathrm{cs}(\xi) & \text{if } \neg\mathrm{dp}(\mathrm{cs}(\mathrm{ac}), \mathrm{cs}(\mathrm{ad}), \mathrm{cs}(\mathrm{z})) \\ \mathrm{x\text{-}proj}(\mathrm{pf}(\mathrm{cs}(\mathrm{ac}), \mathrm{cs}(\mathrm{ad}), \mathrm{cs}(\mathrm{z}))) & \text{else if } \xi = \mathrm{ac} \\ \mathrm{y\text{-}proj}(\mathrm{pf}(\mathrm{cs}(\mathrm{ac}), \mathrm{cs}(\mathrm{ad}), \mathrm{cs}(\mathrm{z}))) & \text{else if } \xi = \mathrm{ad} \\ \mathrm{z\text{-}proj}(\mathrm{pf}(\mathrm{cs}(\mathrm{ac}), \mathrm{cs}(\mathrm{ad}), \mathrm{cs}(\mathrm{z}))) & \text{else if } \xi = \mathrm{z} \\ \mathrm{cs}(\xi) & \text{otherwise} \end{cases} \tag{2.44}$$

Semantics of Iteration

In the tradition of denotational semantics [35], we use the minimum fixed point (MFP) of a functional along the way to defining the interpretation of the **while** loop. However, we go further and define the interpretation of an *assertive* **while** loop. Our syntax requires the inclusion of a loop invariant in the **maintaining** clause. We interpret the loop as asserting the truth of the invariant at the start of each iteration. To be useful, the invariant assertion must be able to refer to "old" values of current variables, i.e., values at the start of the loop's execution, or earlier. To keep reasoning about a loop local to that loop, we define "old" values of current variables to be just the values at the start of the loop's execution.

We use the old-state (part 7 of Figure 27) to remember the old values of current variables. Because loops can be nested within one another, at the conclusion of a loop's execution, we must restore the old-state to its value just prior to the loop's execution. We call this restoring operation "forgetting." If ξ is a current variable, its old value is stored in the old-state as the value of the variable $\#\xi$ (ξ preceded by one old sign). The old value for the loop containing the currently executing loop would be stored in the old-state as the value of the variable $\#\#\xi$ (ξ preceded by two old signs). We define the old-state to behave as a last-in-first-out stack, using the mechanism of the number of old signs in a variable name's prefix to express the definition. Recall that variable names having one or more old signs in their prefix are augmented old variable names (page 65).

Figure 30 shows an example of a loop nested inside another. We use this example to illustrate the meaning and use of old variable names, and how augmented old variable names are used in remembering and forgetting old values. The purpose of this assertive program is to set r to the product of m and n without changing m or n, assuming both m and n are nonnegative. The **remember** statement effectively copies the value of m into the old variable #m, but not before it copies the value of #m into the augmented old variable ##m. In this way, the former value of #m is not lost. The **remember** statement even remembers the former values of augmented old variables; i.e., it copies the value of ##m into ###m, and so on. The **remember** statement performs this service for all variables, including n, r, i, and j.

Execution of a loop also begins with this remembering service. Hence, whenever execution reaches index 9, for example, the following three statements are true: the value of #i equals the value i held the last time execution reached index 6; the value of ##i equals the value i held the last time execution reached index 3; and ###i equals the value i held the last time execution reached index 0. This behavior is important because we intend the phrase "i = #i" of the second loop invariant to mean that i remains unchanged from index 6 to index 10, not that i has the same value it had at index 3, i.e., zero!

Execution of a loop concludes with the opposite of the remembering service: forgetting. Forgetting restores the values of the augmented old variables to the values they had when the loop was encountered. Forgetting does not change the values of current variables! Forgetting sets the value of #m to the value of ##m, the value of

```
      remember
-- (0)
      assume 0 ≤ m ∧ 0 ≤ n
-- (1)
      r := 0
-- (2)
      i := 0
-- (3)
      loop
          maintaining i ≤ m ∧ r = i · n ∧ m = #m ∧ n = #n
          while i < m do
-- (4)
              i := i + 1
-- (5)
              j := 0
-- (6)
              loop
                  maintaining j ≤ n ∧ r = (i − 1) · n + j ∧ i = #i ∧ m = #m
                      ∧ n = #n
                  while j < n do
-- (7)
                      j := j + 1
-- (8)
                      r := r + 1
-- (9)
              end loop
-- (10)
      end loop
-- (11)
      confirm r = m · n ∧ m = #m ∧ n = #n
```

Figure 30: Nested Loops and Old Variable Names

##m to that of ###m, and so on. This behavior is important because we intend the phrase "m = #m" of the final **confirm** statement to mean that the value of m at index 11 is the same value it had at index 0.

The meaning of a **while** loop in an environment, env_{NL}, is the composition of three functions: first env_{NL} is remembered, then the minimum fixed point of a functional is applied, and, finally, the "forget" function is applied. Now let us formally define the remember (Rem : Environments $\rightarrow$ Environments) and forget (Fgt : Environments $\rightarrow$ Environments) functions. Let ψ stand for an augmented old variable name; let ξ stand for a current variable name; and let k be a natural number. Let $\#^j$ represent j occurrences of #; e.g., $\#^3$ represents ###.

$$\text{Rem}(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} [\text{a}, \text{cs}, \text{ns}, \text{se}, \text{os}', \text{d}] \qquad \text{where} \tag{2.45}$$

$$\text{os}'(\psi) = \begin{cases} \text{cs}(\xi) & \text{if } \psi = \#\xi \\ \text{os}(\#^{k+1}\xi) & \text{if } \psi = \#^{k+2}\xi \end{cases} \tag{2.46}$$

$$\text{Fgt}(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} [\text{a}, \text{cs}, \text{ns}, \text{se}, \text{os}', \text{d}] \qquad \text{where} \tag{2.47}$$

$$\text{os}'(\psi) = \text{os}(\#\psi) \tag{2.48}$$

The proof rules we present in Chapter III replace some, but not nearly all, of Krone's [25] rules, which employ a programming language statement, **remember**. This statement appears in our rule (Figure 40) that serves as a bridge between our rules and those of Krone that we will still be using. Interpreting this **remember** statement is just the same as applying the Rem function:

$$\mathcal{I}(\textbf{remember})(\text{env}_{\text{NL}}) \stackrel{\text{def}}{=} \text{Rem}(\text{env}_{\text{NL}}) \tag{2.49}$$

In final preparation for defining the interpretation of a **while** loop, let MFP stand for the minimum fixed point operator, let WL denote an arbitrary environment change (WL : Environments $\rightarrow$ Environments), and let the functional Λ_{W} define the next approximation of WL (Λ_{W} : (Environments $\rightarrow$ Environments) $\rightarrow$ (Environments $\rightarrow$ Environments)):

$$\mathcal{I}(\textbf{loop maintaining } \mathrm{Inv} \textbf{ while } b\_p\_e \textbf{ do } \mathrm{SS} \textbf{ end loop})(\mathrm{env}_{\mathrm{NL}}) \tag{2.50}$$

$$\stackrel{\mathrm{def}}{=} \mathrm{Fgt}(\mathrm{MFP}(\Lambda_{\mathrm{W}})(\mathrm{Rem}(\mathrm{env}_{\mathrm{NL}}))) \qquad \text{where}$$

$$\Lambda_{\mathrm{W}}(\mathrm{WL})(\mathrm{env}) = \begin{cases} \mathrm{env} & \text{if } \mathrm{AE}(\mathrm{env}) \neq \mathrm{NL} \\ [\mathrm{CF}, \mathrm{cs}, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}] & \text{else if } \neg\mathcal{I}(\mathrm{Inv})(\mathrm{env}) \\ \mathrm{env} & \text{else if } \neg\mathcal{I}(b\_p\_e)(\mathrm{env}) \\ \mathrm{WL}(\mathcal{I}(\mathrm{SS})(\mathrm{env})) & \text{otherwise} \end{cases} \tag{2.51}$$

Although this definition of the interpretation of the **while** loop only applies when $\mathrm{AE}(\mathrm{env}_{\mathrm{NL}}) = \mathrm{NL}$, as a technical matter Λ_{W} must be total, i.e., we must define $\Lambda_{\mathrm{W}}(\mathrm{WL})$ in all environments. So, for all environments env such that $\mathrm{AE}(\mathrm{env}) \neq \mathrm{NL}$ we define $\Lambda_{\mathrm{W}}(\mathrm{WL})(\mathrm{env}) = \mathrm{env}$. Otherwise, if the invariant is violated ($\neg\mathcal{I}(\mathrm{Inv})(\mathrm{env})$), the next approximation sets the assert-status to categorically false, leaving the remaining components of the environment unchanged. If neither of the above conditions applies and the Boolean program expression in the **while** clause evaluates to false ($\neg\mathcal{I}(b\_p\_e)(\mathrm{env})$), the next approximation leaves the environment unchanged; this represents terminating loop execution. Otherwise, the next approximation is the interpretation of the current approximation in the environment that results from interpreting the body of the loop; this represents performing the first iteration of the loop, and, then, performing the remainder of loop execution.

2.2.3 Validity

Recall that an assertive program is the combination of an executable program and its specification. Furthermore, (see page 42) an assertive program that satisfies its specification in every possible execution is said to be *valid*. The specification *promises* results *assuming* certain conditions hold for the environment. The purpose of the assert-status is to record, for each particular execution, whether it is satisfying the specification.

One of the assumed conditions, discussed on page 73, is that the environment contain the right meaning for procedure D when D's declaration is encountered. If not, then the assert-status is set irrevocably to vacuously true (VT). When the interpretation of a program in a neutral environment yields a vacuously true environment, we know that some assumption has been violated; we know that this environment gives us little information about the validity of the program. The program is computing relative to a specification of the form $H_1 \Rightarrow H_2$ in an environment where H_1 is false.

Among the promised results is that the environment satisfies the precondition of procedure P_nm when P_nm is called. If not, then the assert-status is set irrevocably to categorically false (CF). When the interpretation of a program in a neutral environment yields a categorically false environment, we know that the program has failed to deliver some promised result; this environment has given us the important information that the program is invalid. Because VT and CF are "stuck states," interpretation of a program in an environment that is not neutral gives us little (or no) information.

This discussion of assert-status in our denotational semantics gives rise to an equivalent, but more formal, definition of program validity: an assertive program, Prog, is *valid* if and only if the interpretation of Prog in every neutral environment yields an environment that is not categorically false. We can also state this definition using our mathematical notation:

Definition 2.1 *Program Prog is* valid *if and only if, for every environment env such that* $AE(env) = NL$, $AE(\mathcal{I}(Prog)(env)) \neq CF$.

In our introduction (Chapter I), we wrote of a program-with-specification satisfying the correctness conjecture, meaning that the program is correct with respect to its specification. In this chapter, "programs-with-specification" have become "assertive programs," and "satisfying the correctness conjecture" has become "being valid." We also discussed that formal bases for reasoning about the behavior of computer programs work by transforming a program-with-specification into a formula of classical mathematics. We further pointed out (page 11) that the idea "is that if the final purely mathematical formula is true, then the correctness conjecture for the original program and specification is also true." Consistent with our change in terminology, we want to say that a true mathematical statement is "valid." We also want to carefully define this term. Mathematical statements are expressed in a formal language belonging to a formal mathematical proof system; such a system, including its set of axioms, is often called a *theory*.

Definition 2.2 *A mathematical statement is* valid *if and only if it evaluates as true in every model (see page 8) of its theory.*

We provide, in Chapter III, a purely syntactic method for establishing the validity of assertive programs. This method relies on transforming assertive programs to mathematical statements. If we have used the method to transform assertive program Prog into mathematical statement H, we will know that Prog is valid if H is valid.

CHAPTER III

Proof Rules

The purpose of a system of proof rules is to establish, by method of formal proof (a purely syntactic method), the validity (see Definition 2.1, page 81) of a given assertive program. The typical situation is that a team of programmers has produced a program and its specification. The team wants to know whether the correctness conjecture holds for this assertive program—whether all executions of the program meet the specification. That is to say, the team wants to know whether its assertive program is valid.

This programmer team can apply Krone's [25] proof rules to its assertive program, rewriting it to one or more related assertive programs whose validity would imply the validity of the original program. Krone's rules are capable of transforming the original program all the way to one or more mathematical assertions. However, once Krone's rules have rewritten a procedure declaration, eliminating the procedure header and local variable declarations, and leaving the procedure body in place, the team has the option of rewriting the procedure body with the rules of the indexed method.

What Krone's rules produce is not just the procedure body; the procedure body is preceded by a **remember** and an **assume** statement, and is followed by a **confirm**

statement. The indexed method begins by rewriting this embellished procedure body. Because the body itself was written by the programmers, we have classified, in Figure 31, the embellished procedure body as a programmer-written program. This classification is not strictly correct because programmers are not permitted to write **assume** and **remember** statements, but there is not much harm in this fiction, and it simplifies the illustration. Step 0 is an application of the rule that provides a bridge between Krone's rules and the indexed method.

The points inside the largest ellipse of Figure 31 represent (assertive) programs. Each arrow represents an application of a proof rule. The sequence of points and arrows depicts a path from a programmer-written program (an embellished procedure body) to a mathematical assertion. A property of the indexed method's proof rules, called *soundness*, means that if the mathematical assertion is valid (i.e., true), then the programmer-written program is a valid assertive program, i.e., the programmer-written program is correct. Step 4 is an application of the rule that provides a bridge between the indexed method and predicate logic.

All the intermediate points of the indexed method between Steps 0 and 4 lie in a proper subset of the assertive programs that we call *top level code*. Top level code is partitioned into subsets that correspond to three distinct phases of applying the indexed method. All of the programs in phase 1 contain **whenever** statements, and none of the programs in the other two phases contain them. In top level code, the only occurrences of operational statements (procedure call, selection, and iteration) are in the statement sequences within **whenever** statements. Consequently, none of

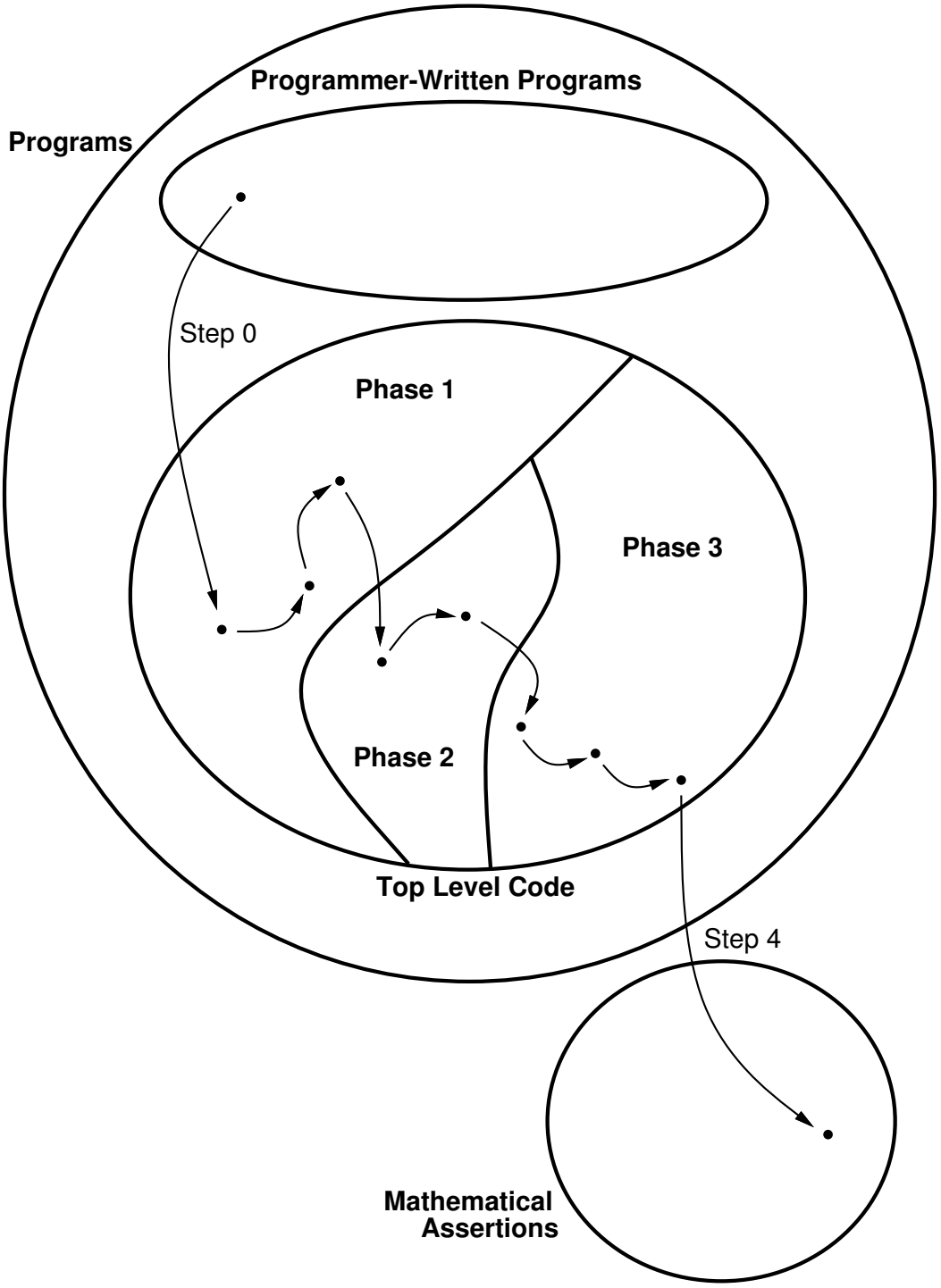

Figure 31: Transforming Programs to Mathematical Assertions in Phases

the programs of phases 2 and 3 contain operational statements. In fact, programs in phase 2 are composed entirely of **alter all**, **stow**, **assume**, and **confirm** statements. Programs in phase 3 contain only **assume** and **confirm** statements. The last program in phase 3 consists of exactly one **confirm** statement.

Step 0 places the procedure body inside a **whenever** statement, decorating the procedure body with **stow** statements. The first program in phase 1 has exactly one **whenever** statement. Each rewrite of phase 1 removes the first statement from a **whenever** statement's statement sequence. This removal is accompanied by introduction of **alter all** and **assume** statements; **confirm** statements also may be introduced. When an iteration or selection statement is removed, one or two additional **whenever** statements may be introduced. As phase 1 proceeds, the statement sequences inside the **whenever** statements shrink and, inevitably, become empty. Empty **whenever** statements are removed in phase 1. When a program has zero **whenever** statements, it enters phase 2.

Phase 2 consists of removing all the **alter all** and **stow** statements. When this removal is completely accomplished, the program enters phase 3, and comprises **assume** and **confirm** statements only. Phase 3 consolidates these statements into one **confirm** statement. Step 4 simply produces the mathematical assertion contained in the remaining **confirm** statement.

3.1 Assertive Program Language Subsets

We defined a liberal syntax for assertive programs in Section 2.1; any statement in the language may appear in any statement sequence. Therefore, all of these programs (members of the set depicted by the largest ellipse of Figure 31) have a well-defined meaning according to the semantics of Section 2.2. However, as we have already mentioned, the language of programmer-written assertive programs is restricted to be a subset of the liberal language. The language of programmer-written procedure bodies is of particular importance to the indexed method. We shall establish the subset language of procedure bodies in this section. We shall do so in the context of establishing the subset languages of top level code and the three phases discussed in Figure 31.

We begin by defining a new nonterminal symbol, ⟨op_stmt⟩, for the operational statements:

$$\langle \text{op\_stmt} \rangle \quad ::= \quad \langle \text{call} \rangle \mid \langle \text{selec} \rangle \mid \langle \text{iter} \rangle \tag{3.1}$$

The **assume** and **confirm** statements appear together in sequences. Programmers may not write **assume** statements. The **confirm** statements that a programmer may write use ⟨cur_assert⟩s. The **assume** and **confirm** statements introduced by indexed method proof rules use ⟨idx_assert⟩s, and those introduced by Krone's rules use ⟨cur_assert⟩s and ⟨old_assert⟩s:

$$\langle \text{ACseq} \rangle \quad ::= \quad \{\textbf{assume } \langle \text{assert} \rangle \mid \textbf{confirm } \langle \text{assert} \rangle\} \tag{3.2}$$

$$\langle \text{assert} \rangle \quad ::= \quad \langle \text{cur\_assert} \rangle \mid \langle \text{old\_assert} \rangle \mid \langle \text{idx\_assert} \rangle \tag{3.3}$$

The **stow** and **alter all** statements may not be written by a programmer, but arise in the proof rules. The **alter all** statement, like the operational statements, has the effect of changing the values associated with the current variables during execution. It appears only in connection with a **stow** statement, so the following nonterminal symbol is named for the **stow** statement, "stow section":

$$\langle \text{stow\_sec} \rangle \quad ::= \quad \varepsilon \mid \textbf{stow}(\langle \text{nat\_num} \rangle) \mid \textbf{alter all} \; \textbf{stow}(\langle \text{nat\_num} \rangle) \tag{3.4}$$

The restricted syntax makes a distinction between statement sequences (i.e., "code") internal to a selection or iteration statement (*internal code*) and *top level code*. Internal code (⟨in_code⟩) is a pattern of nonterminal symbols, cycling repeatedly through ⟨stow_sec⟩, ⟨ACseq⟩, and ⟨op_stmt⟩, beginning with ⟨stow_sec⟩ and concluding with ⟨ACseq⟩. Precise definitions of the proof rules can be conveniently stated in terms of the following portions of ⟨in_code⟩:

1. a prefix (⟨cd_prefix⟩, concludes with ⟨op_stmt⟩),
2. a suffix (⟨cd_suffix⟩, begins and ends with ⟨ACseq⟩), and
3. a kernal (⟨cd_kern⟩, begins with ⟨ACseq⟩ and, if there is an ⟨op_stmt⟩, concludes with an ⟨op_stmt⟩).

We define these here, with ⟨in_code⟩ defined in terms of ⟨cd_prefix⟩:

$$\langle \text{in\_code} \rangle \quad ::= \quad \langle \text{cd\_prefix} \rangle \langle \text{stow\_sec} \rangle \langle \text{ACseq} \rangle \tag{3.5}$$

$$\langle \text{cd\_prefix} \rangle \quad ::= \quad \{ \langle \text{stow\_sec} \rangle \langle \text{ACseq} \rangle \langle \text{op\_stmt} \rangle \} \tag{3.6}$$

$$\langle \text{cd\_suffix} \rangle \quad ::= \quad \{\langle \text{ACseq} \rangle \langle \text{op\_stmt} \rangle \langle \text{stow\_sec} \rangle\} \langle \text{ACseq} \rangle \tag{3.7}$$

$$\langle \text{cd\_kern} \rangle \quad ::= \quad \langle \text{ACseq} \rangle [\langle \text{op\_stmt} \rangle \{\langle \text{stow\_sec} \rangle \langle \text{ACseq} \rangle \langle \text{op\_stmt} \rangle\}] \tag{3.8}$$

The grammar of top level code ($\langle$top_lev_code$\rangle$) is very similar to that of $\langle$cd_prefix$\rangle$. The difference is that $\langle$top_lev_code$\rangle$ contains guarded blocks ($\langle$gd_blk$\rangle$) in place of operational statements ($\langle$op_stmt$\rangle$).

$$\langle \text{gd\_blk} \rangle \quad ::= \quad \varepsilon \mid \textbf{whenever } \langle \text{idx\_assert} \rangle \textbf{ do} \tag{3.9}$$

$$\langle \text{cd\_suffix} \rangle$$

$$\textbf{end whenever}$$

$$\langle \text{top\_lev\_code} \rangle \quad ::= \quad \{\langle \text{stow\_sec} \rangle \langle \text{ACseq} \rangle \langle \text{gd\_blk} \rangle\} \tag{3.10}$$

Each nonempty $\langle$gd_blk$\rangle$ is derivable in the grammar of Chapter II from the nonterminal symbol $\langle$whenever$\rangle$. Note that the statement sequence inside a guarded block is a portion of internal code ($\langle$cd_suffix$\rangle$). In phase 1, each $\langle$gd_blk$\rangle$ is nonempty; in phases 2 and 3, each $\langle$gd_blk$\rangle$ is empty.

The procedure body ($\langle$p_body$\rangle$) of Krone's work [25] has a syntax compatible with that of $\langle$cd_kern$\rangle$, so that is how we define the syntax of $\langle$p_body$\rangle$:

$$\langle \text{p\_body} \rangle \quad ::= \quad \langle \text{cd\_kern} \rangle \tag{3.11}$$

We have expressed a single, unified grammar (collected in Figure 32) for $\langle$in_code$\rangle$, $\langle$p_body$\rangle$, and $\langle$top_lev_code$\rangle$. That these three nonterminals share the symbols $\langle$stow_sec$\rangle$ and $\langle$ACseq$\rangle$ in their rewrite rules aids our expression and explanation of the proof rules. Please note that the statement sequences inside the selection

⟨top_lev_code⟩ ::= {⟨stow_sec⟩⟨ACseq⟩⟨gd_blk⟩}
⟨gd_blk⟩ ::= ε | **whenever** ⟨idx_assert⟩ **do**
⟨cd_suffix⟩
end whenever
⟨p_body⟩ ::= ⟨cd_kern⟩
⟨cd_kern⟩ ::= ⟨ACseq⟩[⟨op_stmt⟩{⟨stow_sec⟩⟨ACseq⟩⟨op_stmt⟩}]
⟨cd_suffix⟩ ::= {⟨ACseq⟩⟨op_stmt⟩⟨stow_sec⟩}⟨ACseq⟩
⟨cd_prefix⟩ ::= {⟨stow_sec⟩⟨ACseq⟩⟨op_stmt⟩}
⟨in_code⟩ ::= ⟨cd_prefix⟩⟨stow_sec⟩⟨ACseq⟩
⟨stow_sec⟩ ::= ε | **stow**(⟨nat_num⟩) | **alter all**
stow(⟨nat_num⟩)
⟨ACseq⟩ ::= {**assume** ⟨assert⟩ | **confirm** ⟨assert⟩}
⟨assert⟩ ::= ⟨cur_assert⟩ | ⟨old_assert⟩ | ⟨idx_assert⟩
⟨op_stmt⟩ ::= ⟨call⟩ | ⟨selec⟩ | ⟨iter⟩
⟨selec⟩ ::= **if** ⟨b_p_e⟩ **then**
⟨in_code⟩
[**else**
⟨in_code⟩]
end if
⟨iter⟩ ::= **loop**
maintaining ⟨old_assert⟩
while ⟨b_p_e⟩ **do**
⟨in_code⟩
end loop
⟨call⟩ ::= ⟨p_nm⟩(⟨cur_var_list⟩)

Figure 32: Context-free Grammar of Subsets of Assertive Programs

$$\langle \text{stow\_sec} \rangle ::= \varepsilon \quad (3.12)$$
$$\langle \text{ACseq} \rangle ::= \{\textbf{confirm } \langle \text{cur\_assert} \rangle\} \quad (3.13)$$

Figure 33: Grammar Productions Restricted for ⟨p_body⟩

$$\langle \text{stow\_sec} \rangle ::= \textbf{stow}(\langle \text{nat\_num} \rangle) \quad (3.14)$$
$$\langle \text{ACseq} \rangle ::= \{\textbf{confirm } \langle \text{cur\_assert} \rangle\} \quad (3.15)$$

Figure 34: Grammar Productions Restricted for ⟨in_code⟩, ⟨cd_prefix⟩, ⟨cd_suffix⟩, and ⟨cd_kern⟩ Inside Selection or Iteration, But Not Part of Procedure Body

(⟨selec⟩) and iteration (⟨iter⟩) statements are restricted to be internal code (⟨in_code⟩). Each of the languages defined by rewriting the nonterminals ⟨in_code⟩, ⟨p_body⟩, and ⟨top_lev_code⟩ according to this grammar is a subset of the language defined by rewriting the nonterminal ⟨program⟩ according to Chapter II's grammar. However, the grammar for each of these kinds of code (top level code, internal code, and procedure body) is really a further restriction of the unified grammar. Figures 33, 34, and 36 show how ⟨stow_sec⟩ and ⟨ACseq⟩ are restricted for each kind of code.

Procedure bodies (which are programmer-written) have empty ⟨stow_sec⟩s, and contain only **confirm** statements of current variables in every ⟨ACseq⟩ (Figure 33). Portions of internal code (each of ⟨in_code⟩, ⟨cd_prefix⟩, ⟨cd_suffix⟩, and ⟨cd_kern⟩, Figure 34) that are not part of a procedure body but are inside a selection or iteration statement have exactly one **stow** statement in every ⟨stow_sec⟩, and contain only **confirm** statements of current variables in every ⟨ACseq⟩. Portions of internal code

$$\langle \text{stow\_sec} \rangle ::= \textbf{stow}(\langle \text{nat\_num} \rangle) \tag{3.16}$$

$$\langle \text{ACseq} \rangle ::= \{\textbf{assume } \langle \text{idx\_assert} \rangle \mid \textbf{confirm } \langle \text{cur\_assert} \rangle \mid \textbf{confirm } \langle \text{idx\_assert} \rangle\} \tag{3.17}$$

Figure 35: Grammar Productions Restricted for ⟨in_code⟩, ⟨cd_prefix⟩, ⟨cd_suffix⟩, and ⟨cd_kern⟩ Outside Selection and Iteration, and Not Part of Procedure Body

$$\langle \text{stow\_sec} \rangle ::= \varepsilon \mid \textbf{alter all } \textbf{stow}(\langle \text{nat\_num} \rangle) \tag{3.18}$$

$$\langle \text{ACseq} \rangle ::= \{\textbf{assume } \langle \text{idx\_assert} \rangle \mid \textbf{confirm } \langle \text{idx\_assert} \rangle\} \tag{3.19}$$

Figure 36: Grammar Productions Restricted for ⟨top_lev_code⟩

that are not part of a procedure body and are *not* inside any selection or iteration statement (Figure 35) also have exactly one **stow** statement in every ⟨stow_sec⟩, but both **assume** and **confirm** statements are permitted in their ⟨ACseq⟩s. The variables in **assume** statements are all indexed; those in **confirm** statements are either all indexed or all current. The ⟨stow_sec⟩ of top level code (Figure 36) is either empty or contains both an **alter all** statement and a **stow** statement. In phase 1, ⟨stow_sec⟩s contain both an **alter all** statement and a **stow** statement. In phase 2, a ⟨stow_sec⟩ may or may not be empty. In phase 3, ⟨stow_sec⟩s are empty. The **assume** and **confirm** statements of top level code refer to indexed variables only.

We also place on the language some additional syntactic restrictions that are not context-free:

1. If **stow**(i) appears earlier than **stow**(j) in a complete in-order traversal of ⟨top_lev_code⟩, then $i < j$. (Indexes are everywhere increasing.)

2. If a statement, S, appears earlier than **stow**(i) in a complete in-order traversal of ⟨top_lev_code⟩, then S contains no reference to any variable indexed with i. (References to index i occur only after **stow**(i).)

3.2 How the Rules are Defined

The indexed method has one rule governing step 0 of Figure 31, and one governing step 4. The rest of the rules transform programs in phases 1, 2, and 3. These latter rules use two equations to define two symbols: $\mathcal{P}$ (meaning "more like a program") and $\mathcal{M}$ (meaning "more like a mathematical statement"). Each of these rules means that if there is an instantiation Inst such that

$$top\_lev\_code_{\mathcal{P}} = \text{Inst}(\mathcal{P}) \tag{3.20}$$

$$top\_lev\_code_{\mathcal{M}} = \text{Inst}(\mathcal{M}) \tag{3.21}$$

and both $top\_lev\_code_{\mathcal{M}}$ and $top\_lev\_code_{\mathcal{P}}$ are syntactically correct top level code (⟨top_lev_code⟩), then $top\_lev\_code_{\mathcal{M}}$ may be derived from $top\_lev\_code_{\mathcal{P}}$.

The proof rules include symbols such as *prec_top_lev_code*. The meaning of these symbols is that, for example, *prec_top_lev_code* is properly instantiated only by code that can be rewritten, according to the grammar of Section 3.1, from the nonterminal symbol ⟨top_lev_code⟩. Correct instantiations must also obey the non-context-free restrictions of Section 3.1. The only occurrence in our proof rules of a procedure

body (⟨p_body⟩) is in the bridge rule (Figure 40). Code that is an instantiation of a schema ($\mathcal{P}$ or $\mathcal{M}$) of any other proof rule is not part of a procedure body. For example, *cd_kern*$_1$ is properly instantiated only by code that can be rewritten from the nonterminal symbol ⟨cd_kern⟩, respecting the restrictions of Figure 34. That is to say, the code instantiating *cd_kern*$_1$ must be rewritten from ⟨cd_kern⟩ with exactly one **stow** statement for each ⟨stow_sec⟩.

Derivation of a mathematical statement from an assertive program is called *proof discovery*. If the resulting mathematical statement is valid, the syntax-directed process has *discovered* a proof of the assertive program's validity. Derivation of *top_lev_code*$_{\mathcal{M}}$ from *top_lev_code*$_{\mathcal{P}}$ is called an application of a proof rule in the *math direction*. Figure 31 depicts proof discovery by applying proof rules in the math direction. If we record each of the steps in a proof discovery and write them down in reverse order, we will have the orthodox form of a traditional formal proof—a list beginning with a known mathematical theorem (the valid mathematical statement) in which each succeeding line in the list is justified by one of the proof rules. This order of writing down the steps is called *proof construction*, and each of the rules is applied in the *program direction*. If we reversed the arrows of Figure 31, we would have a picture of proof construction by applying proof rules in the program direction. The proof rules of the indexed method are defined to be applicable in *both* the math and program directions: each of these rules means that if there is an instantiation, Inst, such that equations 3.20 and 3.21 hold and both *top_lev_code*$_{\mathcal{M}}$ and *top_lev_code*$_{\mathcal{P}}$ are syntactically correct (including any additional syntactic restrictions stated for the

rule), then, in the math direction, $top\_lev\_code_{\mathcal{M}}$ may be derived from $top\_lev\_code_{\mathcal{P}}$, and, in the program direction, $top\_lev\_code_{\mathcal{P}}$ may be derived from $top\_lev\_code_{\mathcal{M}}$.

The soundness and relative completeness of the proof rules is based on the rules' preserving validity, not semantics. A rule is sound if validity is preserved in the program direction. We say that validity is *preserved* in the direction of a rule application if and only if the validity of the original implies the validity of the result. The rule need not preserve semantics. Soundness is not ruined if the result has different behavior than the original—if the result means something different than the original, as long as validity is preserved. We shall emphasize this point again when we present the proof rules.

The relative completeness of the rules depends partly on all but two of them preserving validity in the math direction.[4] Also important for showing relative completeness is that the process of rewriting programs in the math direction always terminates with a mathematical assertion.

3.3 The Context Attribute

Our definitions of the symbols $\mathcal{P}$ and $\mathcal{M}$ in the forthcoming proof rules begin with a preamble of the form "$C\backslash$". The name C stands for the value of the assertive program's "Context" attribute. The Context contains the types, variables, procedures, and mathematical theory definitions that have been declared for the program. For example, suppose the process of applying Krone's proof rules [25] in the math direction

[4] We will see in Chapter IV that the procedure call rule and the **loop while** rule do not necessarily preserve validity in the math direction.

```
C\ procedure Change_X ( q: Queue
                        x: Integer )
            ensures "(q = #q) ∧ (q = Λ ⇒ x = 2) ∧ (q ≠ Λ ⇒ x = 3)"
    var
        q_is_empty: Boolean
    begin
        Test_If_Empty(q, q_is_empty)
        if q_is_empty then
          Make_two(x)
        else
          Make_three(x)
        end if
    end Change_X
    code
    confirm Q
```

Figure 37: Example Subgoal

is in progress. The current subgoal, shown in Figure 37, begins with a declaration for a procedure Change_X. There is some *code* following this declaration, and the last statement in the program is a **confirm** statement. Let us assume that the Context C, which has been built up by the process so far, includes at least the definitions of mathematical string theory and the three procedure headers shown in Figure 38.

Application of Krone's procedure declaration rule in the math direction produces two hypotheses. We will know that the program in Figure 37 is valid only if we establish both programs in Figure 39 to be valid. Krone's rules must be used to make further progress with the second program. We will use the indexed method to show that the first program is valid.

The procedure declaration rule used to produce Figure 39 from Figure 37 is an adaptation of the rule that appears in Krone's dissertation [25, pp. 34, 39, 214].

Let $C \supseteq$ {
procedure Test_If_Empty (q: Queue
empty: Boolean)
ensures "(#q = q) $\wedge$ (q = $\Lambda \Leftrightarrow$ empty)"
procedure Make_two (x: Integer)
ensures "x = 2"
procedure Make_three (x: Integer)
ensures "x = 3"
}

Figure 38: Example Context

$C' \setminus$ **remember**
assume true $\wedge$ **is_initial**(q_is_empty)
Test_If_Empty(q, q_is_empty)
if q_is_empty **then**
Make_two(x)
else
Make_three(x)
end if
confirm (q = #q) $\wedge$ (q = $\Lambda \Rightarrow$ x = 2) $\wedge$ (q $\neq \Lambda \Rightarrow$ x = 3)

and $C'' \setminus$ *code*
confirm Q

where $C'' =$ {
procedure Change_X (q: Queue
x: Integer)
ensures "(q = #q) $\wedge$ (q = $\Lambda \Rightarrow$ x = 2) $\wedge$ (q $\neq \Lambda \Rightarrow$ x = 3)"
} $\cup$ C.
and $C' =$ {q_is_empty: Boolean} $\cup$ C''.

Figure 39: Application of Krone's Procedure Declaration Rule

Application of the procedure declaration rule removes Change_X's header and variable declaration from the code. Change_X's header is placed in a Context, C'', to be used with the second program. Both Change_X's header and the variable declaration are placed in a Context, C', to be used with the first program (see Figure 39). This rule also removes the keywords "**begin**" and "**end** Change_X", and introduces a **remember**, an **assume**, and a **confirm** statement into the first program. The stage is now set for presentation of the proof rules of the indexed method.

3.4 The Bridge Rule

The bridge rule provides a bridge between the rules of the indexed method and the rules already stated by Krone [25]. Step 0 of Figure 31 represents an application of this rule. The form of the bridge rule is a variation of the general form stated on page 95. The bridge rule uses two equations to define two symbols: $\mathcal{P}$ and $\mathcal{M}$. This rule means that if there is an instantiation, Inst, such that

$$code_{\mathcal{P}} = \text{Inst}(\mathcal{P}) \tag{3.22}$$

$$top\_lev\_code_{\mathcal{M}} = \text{Inst}(\mathcal{M}) \tag{3.23}$$

and $top\_lev\_code_{\mathcal{M}}$ and $code_{\mathcal{P}}$ are both syntactically correct, then, in the math direction, $top\_lev\_code_{\mathcal{M}}$ may be derived from $code_{\mathcal{P}}$, and, in the program direction, $code_{\mathcal{P}}$ may be derived from $top\_lev\_code_{\mathcal{M}}$. The syntax of $code_{\mathcal{P}}$ is that of Krone with the exception that the syntax of *p_body* is the compatible syntax given above in Section 3.1. The syntax of $top\_lev\_code_{\mathcal{M}}$ is that of Section 3.1.

$$\mathcal{P} \stackrel{\text{def}}{=} C\backslash \quad \textbf{remember} \qquad (3.24)$$
$$\textbf{assume } \mathrm{pre}\lfloor \mathrm{x} \rfloor \wedge \textbf{is\_initial}(\mathrm{z})$$
$$p\_body$$
$$\textbf{confirm } \mathrm{post}\lfloor \#\mathrm{x}, \mathrm{x} \rfloor$$

$$\mathcal{M} \stackrel{\text{def}}{=} C\backslash \quad \textbf{alter all} \qquad (3.25)$$
$$\textbf{stow}(i)$$
$$\textbf{whenever true do}$$
$$\quad \textbf{assume } \mathrm{pre}[\mathrm{x} \rightsquigarrow \mathrm{x}_i] \wedge \textbf{is\_initial}(\mathrm{z}_i)$$
$$\quad \mathrm{Stows\_added}(p\_body)$$
$$\quad \textbf{stow}(j)$$
$$\quad \textbf{confirm } \mathrm{post}[\#\mathrm{x} \rightsquigarrow \mathrm{x}_i, \mathrm{x} \rightsquigarrow \mathrm{x}_j]$$
$$\textbf{end whenever}$$

Figure 40: Equations Defining the Bridge Rule

The equations of Figure 40 define the bridge rule. Equation 3.25 uses the relation Stows_added. Stows_added is a relation from *p_body*s to *cd_kern*s. Stows_added(*p_body*) is the same as *p_body* except that the derivation of Stows_added(*p_body*) from ⟨cd_kern⟩ rewrites each nonterminal symbol ⟨stow_sec⟩ to **stow**(⟨nat_num⟩) rather than to the empty string, ε. The instantiation of i and j, and the indexes chosen in the Stows_added relation must satisfy the syntactic restriction that indexes are everywhere increasing (see page 93). Such an instantiation and appropriate choices for the Stows_added indexes always exist.

The example of Figures 37 and 39 illustrates the meaning of the Stows_added relation. Figure 41 shows a derivation according to the grammar of Section 3.1 of the body of procedure Change_X. A derivation of Stows_Added(body of Change_X), shown in Figure 42, begins with the nonterminal symbol ⟨cd_kern⟩, not ⟨p_body⟩. It

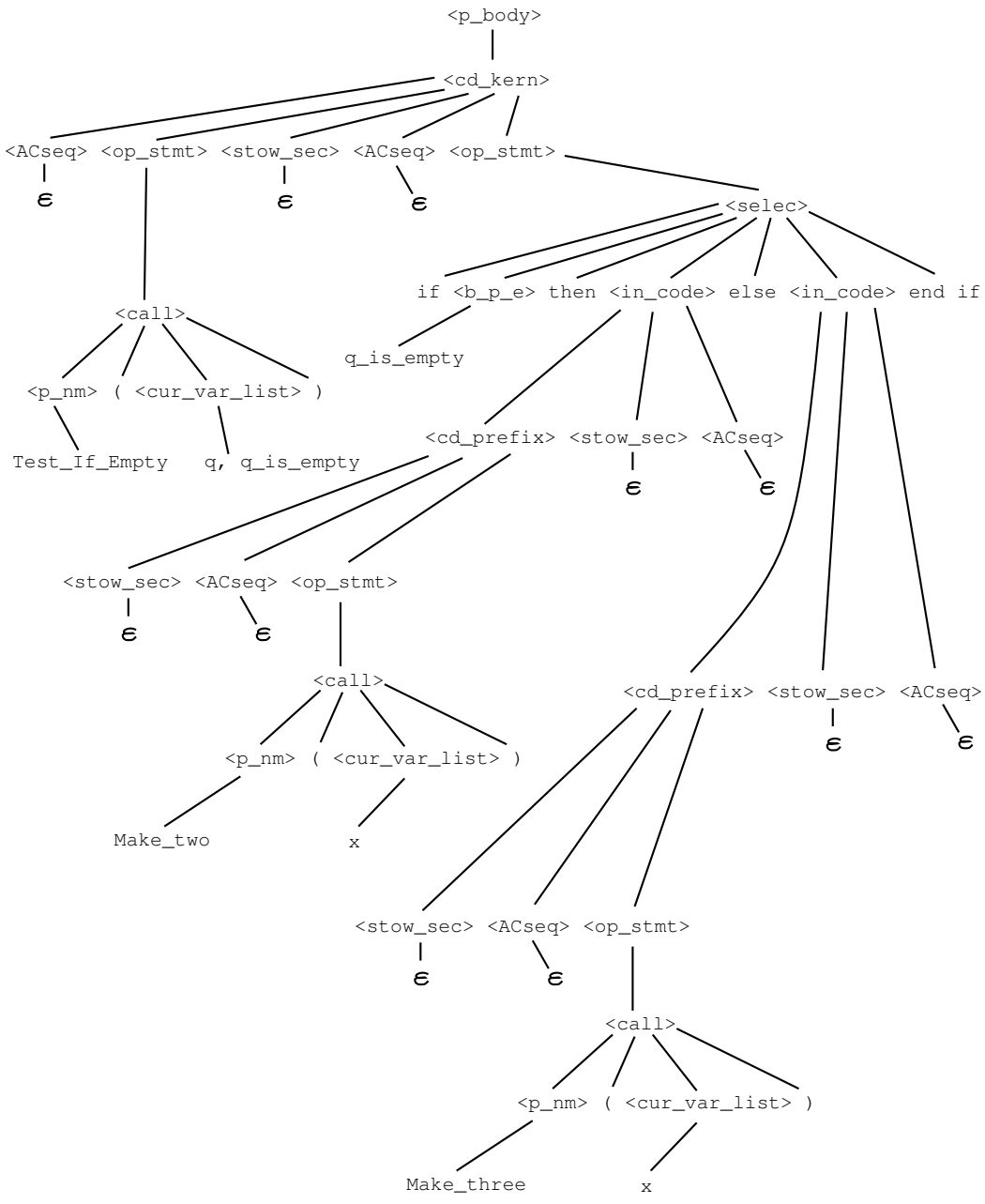

Figure 41: Grammatic Derivation of Body of Change_X

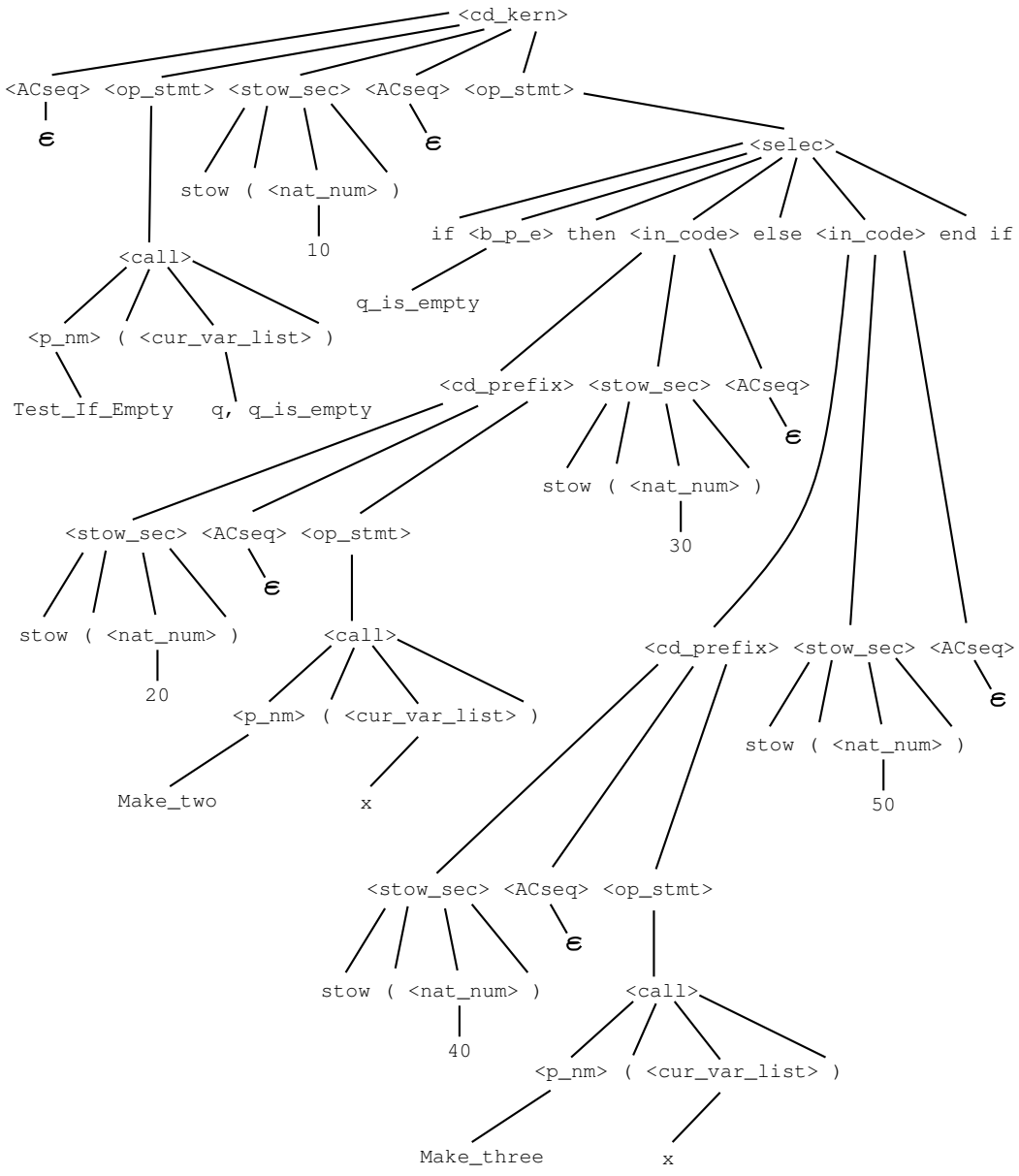

Figure 42: Grammatic Derivation of Stows_Added(Body of Change_X)

is same as the body of Change_X except that each nonterminal symbol ⟨stow_sec⟩ rewrites to **stow**(⟨nat_num⟩) rather than to the empty string, ε. There is some freedom in the choice of the numbers to rewrite from the nonterminal symbols ⟨nat_num⟩, but they must make the result of applying the bridge rule in the math direction syntactically correct. The alternative choice of 55 for the **stow** statement following the call to Make_three would be all right, but a choice of 70 would be incompatible with the choice of 60 for the **stow** just prior to the **confirm** statement of Figure 43. This latter **stow** statement is not part of Stows_Added(body of Change_X), but is the instantiation of the **stow**(j) in the bridge rule.

Returning to the definition of the bridge rule in Figure 40, the notation involving the symbols "⌊" and "⌋" means, in the example of "post⌊#x, x⌋", that it is helpful to think of the expression post as containing free occurrences of the variables #x and x. This notation does not serve as a syntactic restriction on post. The related notation that uses the symbols "[", "$\rightsquigarrow$", and "]" has important syntactic implications. It means, for example, that "post[#x $\rightsquigarrow$ x_i, x $\rightsquigarrow$ x_j]" is the expression obtained from post by replacing every free occurrence of #x with x_i and every free occurrence of x with x_j—not only for variable x, but similarly for every free variable of post. The old variables (e.g., #y) are replaced by the corresponding i-subscripted variables (y_i), and the current variables (y) by the corresponding j-subscripted variables (y_j).

Let us return to the example and apply the bridge rule in the math direction to the first program of Figure 39. We can easily find an instantiation Inst such that Inst($\mathcal{P}$) equals this program. We lose no generality in supposing that Inst instantiates i to 0

$C'\backslash$ **alter all**
stow(0)
whenever true do
assume true $\wedge$ **is_initial**($\mathrm{q\_is\_empty}_0$)
Test_If_Empty(q, q_is_empty)
stow(10)
if q_is_empty **then**
stow(20)
Make_two(x)
stow(30)
else
stow(40)
Make_three(x)
stow(50)
end if
stow(60)
confirm $(\mathrm{q}_{60} = \mathrm{q}_0) \wedge (\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2) \wedge (\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3)$
end whenever

Figure 43: Application of the Bridge Rule: Inst($\mathcal{M}$)

and j to 60. We are able to choose Stows_added indexes of 10, 20, 30, 40, and 50. The resulting Inst($\mathcal{M}$) is shown in Figure 43.

The reader's intuition may be helped by an informal explanation of why the bridge rule preserves validity in the program direction—of why the bridge rule is sound. The definition of validity requires us to think about every environment, while the concept of invalidity focuses on the existence of some one environment. Lemma 3.1 captures the concept of invalidity and follows immediately from definition 2.1 on page 81.

Lemma 3.1 *Program Prog is* invalid *if and only if, there exists environment env such that* $AE(env) = NL$ *and* $AE(\mathcal{I}(Prog)(env)) = CF$.

Because invalidity deals with the existence of some one environment, it is easier to argue that a rule preserves invalidity in the math direction than it is to argue (equivalently because it is the contrapositive) that it preserves validity in the program direction.

So we suppose that $\mathcal{P}$ of Figure 40 is invalid, intending to show that $\mathcal{M}$ must be invalid. By our supposition, there exists a neutral environment, env, such that execution of $\mathcal{P}$ results in a categorically false environment when started in environment env. From env we can construct another neutral environment env$'$ that is a witness to the invalidity of $\mathcal{M}$. We just make the front (first, or head) state of the setup (page 64) equal to the current state of env. That will make the current state of the environment after execution of the **alter all** statement in environment env$'$ the same as the current state of env. Then the **stow**(i) statement will make the index-state map i to the value of the current state.

The body of the **whenever** statement will be executed because the condition is the constant **true**. The **assume** statement keeps the assert status neutral because the **assume** statement of $\mathcal{P}$ kept it neutral and because the index-state at i is the same as the current state of env. Stows_added(p_body) has the same effect on the current state and the assert status as does p_body because it is being executed in the same current state and because the additional **stow** statements do not produce a disturbance. Due to the semantics of **remember** and **stow**(j), **confirm** post$[\#\mathrm{x} \rightsquigarrow \mathrm{x}_i, \mathrm{x} \rightsquigarrow \mathrm{x}_j]$ has the same effect on the assert status in its environment as **confirm** post$\lfloor \#\mathrm{x}, \mathrm{x} \rfloor$ did in its. In fact, at this point, the assert status must be categorically false. Hence,

env′ is a witness to the invalidity of $\mathcal{M}$, and we are done. Note that $\mathcal{M}$ does not have the same meaning as $\mathcal{P}$; it does not do the same thing. The **alter all** statement has clearly changed its meaning. What is important is that if $\mathcal{P}$ is invalid, then $\mathcal{M}$ is certainly invalid. Thus we have shown the bridge rule to be sound; it preserves invalidity in the math direction—validity in the program direction.

3.5 The Rule for assume

The example program in Figure 43 is in phase 1 of the diagram in Figure 31. Recall that each rewrite of phase 1 removes the first statement from a **whenever** statement's statement sequence. The rule for **assume**, defined in Figure 44, is used to remove an **assume** statement when it is the first statement of a **whenever** statement. The long vertical bars at the left side of the figure indicate the parts of the two schemas ($\mathcal{P}$ and $\mathcal{M}$) that differ one from the other.

Figure 45 shows the result of applying the **assume** rule to the example of Figure 43. This application removes the **assume** statement just prior to the call to Test_If_Empty, inserting a modified **assume** statement just prior to the **whenever** statement.

Suppose, for an informal explanation of the soundness of the rule for **assume**, that $\mathcal{P}$ is invalid. Thus, there exists a neutral environment env such that execution of $\mathcal{P}$ in env yields a categorically false environment. We are to show that $\mathcal{M}$ is invalid. That is, we need to construct a neutral environment env′ such that execution of $\mathcal{M}$

$$\mathcal{P} \stackrel{\text{def}}{=} C\backslash \quad \begin{array}{l} \mathit{prec\_top\_lev\_code} \\ \textbf{alter all} \\ \textbf{stow}(i) \\ \mathit{ACseq}_0 \\ \quad | \ \textbf{whenever } \text{Br\_Cd } \textbf{do} \\ \quad \quad \textbf{assume } H \\ \quad \quad \mathit{cd\_suffix} \\ \quad \textbf{end whenever} \\ \mathit{fol\_top\_lev\_code} \end{array} \tag{3.26}$$

$$\mathcal{M} \stackrel{\text{def}}{=} C\backslash \quad \begin{array}{l} \mathit{prec\_top\_lev\_code} \\ \textbf{alter all} \\ \textbf{stow}(i) \\ \mathit{ACseq}_0 \\ \quad | \ \textbf{assume } (\text{Br\_Cd}) \Rightarrow (H) \\ \quad \textbf{whenever } \text{Br\_Cd } \textbf{do} \\ \quad \quad \mathit{cd\_suffix} \\ \quad \textbf{end whenever} \\ \mathit{fol\_top\_lev\_code} \end{array} \tag{3.27}$$

Figure 44: Equations Defining the Rule for **assume**

$C'\backslash$ **alter all**
stow(0)
assume (**true**) $\Rightarrow$ (**true** $\wedge$ **is_initial**(q_is_empty$_0$))
whenever true do
Test_If_Empty(q, q_is_empty)
stow(10)
if q_is_empty **then**
stow(20)
Make_two(x)
stow(30)
else
stow(40)
Make_three(x)
stow(50)
end if
stow(60)
confirm $(q_{60} = q_0) \wedge (q_{60} = \Lambda \Rightarrow x_{60} = 2) \wedge (q_{60} \neq \Lambda \Rightarrow x_{60} = 3)$
end whenever

Figure 45: First Application of Rule for **assume**

in env′ yields a categorically false environment. In this case, env itself is a witness to $\mathcal{M}$'s invalidity; i.e., we just set env′ equal to env.

Because an **assume** statement can affect only the assert status of an environment, all we need show is that "**assume** (Br_Cd) $\Rightarrow$ (H)" does not change the assert status to VT (vacuously true) when $\mathcal{M}$ is executed beginning in environment env. In other words, we need to show that (Br_Cd) $\Rightarrow$ (H) evaluates as true. It does evaluate as true if Br_Cd evaluates as false, by the meaning of mathematical implication ($\Rightarrow$). If, on the other hand, Br_Cd evaluates as true, we know that H evaluates as true because execution of $\mathcal{P}$ from env results in a categorically false—not a vacuously true—environment. Hence, (Br_Cd) $\Rightarrow$ (H) evaluates as true, and we are done.

3.6 The Rule for Procedure Call

The rule for procedure call rewrites phase 1 programs by removing a procedure call when it is the first statement of a **whenever** statement. The result is another phase 1 program. The equations and the additional syntactic restriction of Figure 46 define the rule for procedure call—where the called procedure has two formal parameters and one referenced state variable. This definition gives a clear indication of what the rule is when the called procedure has a different number of parameters and referenced state variables. The variable b represents any program variable that is neither a referenced state variable of procedure P_nm nor an actual parameter in the call. This programming language is designed, according to the principles of RESOLVE [17], so

$$\mathcal{P} \stackrel{\text{def}}{=} C\backslash \quad \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textbf{alter all} \\ \textbf{stow}(i) \\ \textit{ACseq}_0 \\ \quad | \; \textbf{whenever } \text{Br\_Cd } \textbf{do} \\ \quad\quad \text{P\_nm}(\text{ac}, \text{ad}) \\ \quad\quad \textbf{stow}(j) \\ \quad\quad \textit{cd\_suffix} \\ \quad \textbf{end whenever} \\ \textit{fol\_top\_lev\_code} \end{array} \tag{3.28}$$

$$\mathcal{M} \stackrel{\text{def}}{=} C\backslash \quad \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textbf{alter all} \\ \textbf{stow}(i) \\ \textit{ACseq}_0 \\ \quad | \; \textbf{confirm } (\text{Br\_Cd}) \Rightarrow (\text{pre}[\text{x} \rightsquigarrow \text{ac}_i, \text{y} \rightsquigarrow \text{ad}_i, \text{z} \rightsquigarrow \text{z}_i]) \\ \quad \textbf{alter all} \\ \quad \textbf{stow}(j) \\ \quad \textbf{whenever } \text{Br\_Cd } \textbf{do} \\ \quad\quad \textbf{assume } \text{b}_j = \text{b}_i \wedge (\text{post}[\#\text{x} \rightsquigarrow \text{ac}_i, \text{x} \rightsquigarrow \text{ac}_j, \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \#\text{y} \rightsquigarrow \text{ad}_i, \text{y} \rightsquigarrow \text{ad}_j, \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \#\text{z} \rightsquigarrow \text{z}_i, \text{z} \rightsquigarrow \text{z}_j]) \\ \quad\quad \textit{cd\_suffix} \\ \quad \textbf{end whenever} \\ \textit{fol\_top\_lev\_code} \end{array} \tag{3.29}$$

Additional Syntactic Restriction:

$$C \supseteq \{\text{ac}, \text{ad} : \text{T}_1, \text{z} : \text{T}_3, \text{b} : \text{T}_4\} \cup \left\{ \begin{array}{l} \textbf{procedure } \text{P\_nm}(\text{x}, \text{y} : \text{T}_1) \\ \quad \textbf{referenced state variables } \text{z} : \text{T}_3 \\ \quad \textbf{requires } \text{pre}\lfloor \text{x}, \text{y}, \text{z} \rfloor \\ \quad \textbf{ensures } \text{post}\lfloor \text{x}, \#\text{x}, \text{y}, \#\text{y}, \text{z}, \#\text{z} \rfloor \end{array} \right\}$$

Figure 46: Equations and Additional Syntactic Restriction Defining the Rule for Procedure Call

$C'\backslash$ **alter all**
stow(0)
assume (**true**) $\Rightarrow$ (**true** $\wedge$ **is_initial**($\mathrm{q\_is\_empty}_0$))
confirm (**true**) $\Rightarrow$ (**true**)
alter all
stow(10)
whenever true do
assume $\mathrm{x}_{10} = \mathrm{x}_0 \wedge ((\mathrm{q}_0 = \mathrm{q}_{10}) \wedge (\mathrm{q}_{10} = \Lambda \Leftrightarrow \mathrm{q\_is\_empty}_{10}))$
if q_is_empty **then**
stow(20)
Make_two(x)
stow(30)
else
stow(40)
Make_three(x)
stow(50)
end if
stow(60)
confirm $(\mathrm{q}_{60} = \mathrm{q}_0) \wedge (\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2) \wedge (\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3)$
end whenever

Figure 47: First Application of Procedure Call Rule

that we know that the call to P_nm does not change b. That is why the **assume** statement includes the assertion that $\mathrm{b}_j = \mathrm{b}_i$.

Figure 47 shows the result of applying the procedure call rule to the example of Figure 43 to replace the call to Test_If_Empty. We actually use one of the obvious variants of the rule shown in Figure 46, because Test_If_Empty has no referenced state variables, and its two formal parameters have two different types. In this application, we instantiate i to 0 and j to 10. Because it is neither an actual parameter of Test_If_Empty nor a referenced state variable, x plays the role of b in the rule; that is why we write "$\mathrm{x}_{10} = \mathrm{x}_0$".

To indicate informally why the procedure call rule is sound, we suppose that env is a witness to $\mathcal{P}$'s invalidity, and discuss how to construct env$'$ to be a witness to $\mathcal{M}$'s invalidity. Because $\mathcal{M}$ has an additional **alter all** statement (just prior to **stow**(j)), we obtain env$''$s setup by inserting an additional state into env's setup just after the state that gets consumed by the **alter all** statement that precedes **stow**(i). All other parts of env$'$ are the same as those of env. We choose the additional state of the setup according to the evaluation of Br_Cd. If Br_Cd evaluates as false, we make the additional state the same as the current state when execution of $\mathcal{P}$—starting from env—just reaches the **whenever** statement. Otherwise, we make it the same as the current state at execution of the **stow**(j) statement.

If Br_Cd is false, execution of $\mathcal{M}$ beginning in env$'$ will have nearly the same environment just prior to *fol_top_lev_code* as execution of $\mathcal{P}$ beginning in env. The only difference is in the index-state at j. The difference, however, does not prevent the continued execution of $\mathcal{M}$ from producing a categorically false environment. This fact follows from the negative-branch-condition independence lemma, which we will present and prove in Chapter IV.

If Br_Cd is true and the precondition of P_nm is violated in $\mathcal{P}$, then "**confirm** (Br_Cd) $\Rightarrow$ (pre[x $\rightsquigarrow$ ac$_i$, y $\rightsquigarrow$ ad$_i$, z $\rightsquigarrow$ z$_i$])" of $\mathcal{M}$ will take the assert status to CF (categorically false). Otherwise, $\mathcal{M}$ will have the same environment just prior to *cd_suffix* as $\mathcal{P}$. Hence, env$'$ is what we were seeking—a witness to $\mathcal{M}$'s invalidity.

3.7 Rules for Selection

The rules for selection produce phase 1 programs by rewriting other phase 1 programs. The selection statement has an optional **else** clause. The equations of Figure 48 define the rule for selection in the absence of an **else** clause, and Figures 49 and 50 respectively define $\mathcal{P}$ and $\mathcal{M}$ for the rule for selection in the presence of an **else** clause.

These rules employ a new function symbol, MExp. MExp is a function from Boolean-valued program expressions into the set of mathematical expressions. Because we have not included function procedures in the languages of this dissertation, the purpose of MExp here is strictly for explicit type conversion from program expressions to mathematical expressions. There is some typographic evidence of this conversion; for example, MExp changes the program symbol "**and**" to the mathematical symbol "$\wedge$". When function calls are included in the language, MExp, when applied to a function call, will produce the specification's "return" expression with the actual parameters substituted for the formal parameters.

Figure 51 shows an application of the rule for **assume** to Figure 47 of our running example. Having applied the rule for **assume**, it is now possible to apply the rule for selection in the presence of an **else** clause, producing Figure 52. In this application, we instantiate i to 10, j to 20, k to 30, l to 40, m to 50, and n to 60.

To indicate informally why the rule for selection in the presence of an **else** clause is sound, we suppose that env is a witness to $\mathcal{P}$'s invalidity, and discuss how to construct env$'$ to be a witness to $\mathcal{M}$'s invalidity. Because $\mathcal{M}$ has three additional

$\mathcal{P} \stackrel{\text{def}}{=} C\backslash$ *prec_top_lev_code* (3.30)
alter all
stow(i)
$ACseq_0$
whenever Br_Cd **do**
if *b_p_e* **then**
stow(j)
cd_kern
stow(k)
ACseq
end if
stow(n)
cd_suffix
end whenever
fol_top_lev_code

$\mathcal{M} \stackrel{\text{def}}{=} C\backslash$ *prec_top_lev_code* (3.31)
alter all
stow(i)
$ACseq_0$
alter all
stow(j)
whenever $(\text{Br\_Cd}) \wedge (\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i])$ **do**
assume $\text{x}_j = \text{x}_i$
cd_kern
stow(k)
ACseq
end whenever
alter all
stow(n)
assume $((\text{Br\_Cd}) \wedge (\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i])) \Rightarrow (\text{x}_n = \text{x}_k)$
assume $((\text{Br\_Cd}) \wedge \neg(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i])) \Rightarrow (\text{x}_n = \text{x}_i)$
whenever Br_Cd **do**
cd_suffix
end whenever
fol_top_lev_code

Figure 48: Equations Defining the Rule for Selection in the Absence of an **else** Clause

$$
\begin{array}{lll}
\mathcal{P} & \stackrel{\text{def}}{=} & C\backslash \\
 & & \quad \textit{prec\_top\_lev\_code} \\
 & & \quad \textbf{alter all} \\
 & & \quad \textbf{stow}(i) \\
 & & \quad \textit{ACseq}_0 \\
 & & \quad \textbf{whenever } \text{Br\_Cd } \textbf{do} \\
 & & \quad\quad \textbf{if } \textit{b\_p\_e } \textbf{then} \\
 & & \quad\quad\quad \textbf{stow}(j) \\
 & & \quad\quad\quad \textit{cd\_kern}_1 \\
 & & \quad\quad\quad \textbf{stow}(k) \\
 & & \quad\quad\quad \textit{ACseq}_1 \\
 & & \quad\quad \textbf{else} \\
 & & \quad\quad\quad \textbf{stow}(l) \\
 & & \quad\quad\quad \textit{cd\_kern}_2 \\
 & & \quad\quad\quad \textbf{stow}(m) \\
 & & \quad\quad\quad \textit{ACseq}_2 \\
 & & \quad\quad \textbf{end if} \\
 & & \quad\quad \textbf{stow}(n) \\
 & & \quad\quad \textit{cd\_suffix} \\
 & & \quad \textbf{end whenever} \\
 & & \quad \textit{fol\_top\_lev\_code}
\end{array}
\tag{3.32}
$$

Figure 49: Definition of $\mathcal{P}$ for the Rule for Selection in the Presence of an **else** Clause

$$
\begin{array}{lll}
\mathcal{M} & \stackrel{\text{def}}{=} & C\backslash \;\; \textit{prec\_top\_lev\_code} \\
 & & \textbf{alter all} \\
 & & \textbf{stow}(i) \\
 & & \textit{ACseq}_0 \\
 & & \quad \textbf{alter all} \\
 & & \quad \textbf{stow}(j) \\
 & & \quad \textbf{whenever } (\text{Br\_Cd}) \wedge (\text{MExp}(\textit{b\_p\_e})[\text{y} \rightsquigarrow \text{y}_i]) \textbf{ do} \\
 & & \quad\quad \textbf{assume } \text{x}_j = \text{x}_i \\
 & & \quad\quad \textit{cd\_kern}_1 \\
 & & \quad\quad \textbf{stow}(k) \\
 & & \quad\quad \textit{ACseq}_1 \\
 & & \quad \textbf{end whenever} \\
 & & \quad \textbf{alter all} \\
 & & \quad \textbf{stow}(l) \\
 & & \quad \textbf{whenever } (\text{Br\_Cd}) \wedge \neg(\text{MExp}(\textit{b\_p\_e})[\text{y} \rightsquigarrow \text{y}_i]) \textbf{ do} \\
 & & \quad\quad \textbf{assume } \text{x}_l = \text{x}_i \\
 & & \quad\quad \textit{cd\_kern}_2 \\
 & & \quad\quad \textbf{stow}(m) \\
 & & \quad\quad \textit{ACseq}_2 \\
 & & \quad \textbf{end whenever} \\
 & & \quad \textbf{alter all} \\
 & & \quad \textbf{stow}(n) \\
 & & \quad \textbf{assume } ((\text{Br\_Cd}) \wedge (\text{MExp}(\textit{b\_p\_e})[\text{y} \rightsquigarrow \text{y}_i])) \Rightarrow (\text{x}_n = \text{x}_k) \\
 & & \quad \textbf{assume } ((\text{Br\_Cd}) \wedge \neg(\text{MExp}(\textit{b\_p\_e})[\text{y} \rightsquigarrow \text{y}_i])) \Rightarrow (\text{x}_n = \text{x}_m) \\
 & & \quad \textbf{whenever } \text{Br\_Cd} \textbf{ do} \\
 & & \quad\quad \textit{cd\_suffix} \\
 & & \quad \textbf{end whenever} \\
 & & \textit{fol\_top\_lev\_code}
\end{array}
\tag{3.33}
$$

Figure 50: Definition of $\mathcal{M}$ for the Rule for Selection in the Presence of an **else** Clause

$C'\backslash$ **alter all**
stow(0)
assume (**true**) $\Rightarrow$ (**true** $\wedge$ **is_initial**(q_is_empty$_0$))
confirm (**true**) $\Rightarrow$ (**true**)
alter all
stow(10)
assume (**true**) $\Rightarrow$ ($x_{10} = x_0 \wedge ((q_0 = q_{10}) \wedge (q_{10} = \Lambda \Leftrightarrow$ q_is_empty$_{10}$)))
whenever true do
if q_is_empty **then**
stow(20)
Make_two(x)
stow(30)
else
stow(40)
Make_three(x)
stow(50)
end if
stow(60)
confirm ($q_{60} = q_0$) $\wedge$ ($q_{60} = \Lambda \Rightarrow x_{60} = 2$) $\wedge$ ($q_{60} \neq \Lambda \Rightarrow x_{60} = 3$)
end whenever

Figure 51: Second Application of Rule for **assume**

$C'\backslash$ **alter all**
stow(0)
assume (**true**) $\Rightarrow$ (**true** $\wedge$ **is_initial**($\mathrm{q\_is\_empty}_0$))
confirm (**true**) $\Rightarrow$ (**true**)
alter all
stow(10)
assume (**true**) $\Rightarrow$ ($\mathrm{x}_{10} = \mathrm{x}_0 \wedge ((\mathrm{q}_0 = \mathrm{q}_{10}) \wedge (\mathrm{q}_{10} = \Lambda \Leftrightarrow \mathrm{q\_is\_empty}_{10}))$)
alter all
stow(20)
whenever (**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$) **do**
  assume $\mathrm{q\_is\_empty}_{20} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{20} = \mathrm{q}_{10} \wedge \mathrm{x}_{20} = \mathrm{x}_{10}$
  Make_two(x)
  stow(30)
end whenever
alter all
stow(40)
whenever (**true**) $\wedge \neg$($\mathrm{q\_is\_empty}_{10}$) **do**
  assume $\mathrm{q\_is\_empty}_{40} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{40} = \mathrm{q}_{10} \wedge \mathrm{x}_{40} = \mathrm{x}_{10}$
  Make_three(x)
  stow(50)
end whenever
alter all
stow(60)
assume ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
  ($\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{30} \wedge \mathrm{q}_{60} = \mathrm{q}_{30} \wedge \mathrm{x}_{60} = \mathrm{x}_{30}$)
assume ((**true**) $\wedge \neg$($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
  ($\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{50} \wedge \mathrm{q}_{60} = \mathrm{q}_{50} \wedge \mathrm{x}_{60} = \mathrm{x}_{50}$)
whenever true do
  confirm ($\mathrm{q}_{60} = \mathrm{q}_0$) $\wedge$ ($\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2$) $\wedge$ ($\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3$)
end whenever

Figure 52: Application of Rule for Selection in the Presence of an **else** Clause

alter all statements, we obtain env′'s setup by inserting three additional states into env's setup just after the state that gets consumed by the **alter all** statement that precedes **stow**(i). All other parts of env′ are the same as those of env. We make the first two additional states the same as the current state when execution of $\mathcal{P}$—starting from env—just reaches the **whenever** statement. We choose the third additional state of the setup according to the evaluation of Br_Cd. If Br_Cd evaluates as false, we make the third additional state the same as the first two. Otherwise, we choose the third additional state according to the evaluation of *b_p_e*. If *b_p_e* evaluates as true, we make the third additional state the same as the current state when execution of $\mathcal{P}$—starting from env—just reaches **stow**(k); otherwise, we use the current state at **stow**(m).

If Br_Cd is false, execution of $\mathcal{M}$ beginning in env′ will have nearly the same environment just prior to *fol_top_lev_code* as execution of $\mathcal{P}$ beginning in env. The only differences are in the index-state at j, l, and n. These differences, however, do not prevent the continued execution of $\mathcal{M}$ from producing a categorically false environment. This fact follows from the negative-branch-condition independence lemma, which—as we mentioned above in Section 3.6 —we will present and prove in Chapter IV.

If Br_Cd is true, cd_kern_1 will be executed in $\mathcal{M}$ if and only if *b_p_e* evaluates as true in $\mathcal{P}$; otherwise, cd_kern_2 will be executed in $\mathcal{M}$. Either way, $\mathcal{M}$ will have nearly the same environment just prior to *cd_suffix* as $\mathcal{P}$. The only difference is in the index-state at an index internal to the selection statement: either j or l. The difference, however, does not prevent the continued execution of $\mathcal{M}$ from producing

$$
\begin{array}{lll}
\mathcal{P} & \stackrel{\text{def}}{=} & C\backslash \;\; prec\_top\_lev\_code \\
 & & \quad \textbf{alter all} \\
 & & \quad \textbf{stow}(i) \\
 & & \quad ACseq_0 \\
 & & \quad \textbf{whenever } \text{Br\_Cd } \textbf{do} \\
 & & \quad\quad \textbf{loop} \\
 & & \quad\quad\quad \textbf{maintaining } \text{Inv}\lfloor \text{x}, \#\text{x} \rfloor \\
 & & \quad\quad\quad \textbf{while } b\_p\_e \textbf{ do} \\
 & & \quad\quad\quad\quad \textbf{stow}(j) \\
 & & \quad\quad\quad\quad cd\_kern \\
 & & \quad\quad\quad\quad \textbf{stow}(k) \\
 & & \quad\quad\quad\quad ACseq \\
 & & \quad\quad \textbf{end loop} \\
 & & \quad\quad \textbf{stow}(l) \\
 & & \quad\quad cd\_suffix \\
 & & \quad \textbf{end whenever} \\
 & & \quad fol\_top\_lev\_code
\end{array}
\tag{3.34}
$$

Figure 53: Definition of $\mathcal{P}$ for the **loop while** Rule

a categorically false environment. This fact follows from the internal index independence lemma, which we will present and prove in Chapter IV. Hence, env$'$ is what we were seeking—a witness to $\mathcal{M}$'s invalidity.

3.8 The loop while Rule

The **loop while** rule produces phase 1 programs by rewriting other phase 1 programs. Figures 53 and 54 respectively define $\mathcal{P}$ and $\mathcal{M}$ for the **loop while** rule. Because it is not applicable to the running example, we will show an application of this rule in the later Section 3.16.

$$
\begin{array}{lll}
\mathcal{M} & \stackrel{\text{def}}{=} & C\backslash \quad \textit{prec\_top\_lev\_code} \\
 & & \textbf{alter all} \\
 & & \textbf{stow}(i) \\
 & & \textit{ACseq}_0 \\
 & & \quad \textbf{confirm } (\text{Br\_Cd}) \Rightarrow (\text{Inv}[\text{x} \rightsquigarrow \text{x}_i, \#\text{x} \rightsquigarrow \text{x}_i]) \\
 & & \quad \textbf{alter all} \\
 & & \quad \textbf{stow}(j) \\
 & & \quad \textbf{whenever } (\text{Br\_Cd}) \wedge (\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i]) \textbf{ do} \\
 & & \quad\quad \textbf{assume } (\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_j]) \wedge (\text{Inv}[\text{x} \rightsquigarrow \text{x}_j, \#\text{x} \rightsquigarrow \text{x}_i]) \\
 & & \quad\quad \textit{cd\_kern} \\
 & & \quad\quad \textbf{stow}(k) \\
 & & \quad\quad \textit{ACseq} \\
 & & \quad\quad \textbf{confirm } \text{Inv}[\text{x} \rightsquigarrow \text{x}_k, \#\text{x} \rightsquigarrow \text{x}_i] \\
 & & \quad \textbf{end whenever} \\
 & & \quad \textbf{alter all} \\
 & & \quad \textbf{stow}(l) \\
 & & \quad \textbf{whenever } \text{Br\_Cd} \textbf{ do} \\
 & & \quad\quad \textbf{assume } (\neg(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_l])) \wedge (\text{Inv}[\text{x} \rightsquigarrow \text{x}_l, \#\text{x} \rightsquigarrow \text{x}_i]) \\
 & & \quad\quad \textit{cd\_suffix} \\
 & & \quad \textbf{end whenever} \\
 & & \textit{fol\_top\_lev\_code}
\end{array}
\tag{3.35}
$$

Figure 54: Definition of $\mathcal{M}$ for the **loop while** Rule

To indicate informally why the **loop while** rule is sound, we suppose that env is a witness to $\mathcal{P}$'s invalidity, and discuss how to construct env$'$ to be a witness to $\mathcal{M}$'s invalidity. Because $\mathcal{M}$ has two additional **alter all** statements, we obtain env$''$s setup by inserting two additional states into env's setup just after the state that gets consumed by the **alter all** statement that precedes **stow**(i). All other parts of env$'$ are the same as those of env. If Br_Cd evaluates as false, we make these two additional states the same as the current state when execution of $\mathcal{P}$—starting from env—just reaches the **whenever** statement. If Br_Cd evaluates as true, our choice for the first additional state depends upon when execution of $\mathcal{P}$ first took the assert status to CF. If this action happened during an iteration of the loop, we choose the first additional state to be the same as the current state at the start of that iteration. In this case, execution of $\mathcal{M}$ is guaranteed to set the assert status to CF by the time "**confirm** Inv$[\mathrm{x} \rightsquigarrow \mathrm{x}_k, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i]$" finishes executing. Here, the choice of the second additional state is immaterial.

On the other hand, if execution of $\mathcal{P}$ first took the assert status to CF after execution of the loop had completed, we choose the second additional state to be the same as the current state when execution of $\mathcal{P}$ reached **stow**(l). In this latter case, it is convenient to choose the first additional state to be the same as the index-state at i. As a consequence of these choices, $\mathcal{M}$ will have nearly the same environment just prior to *cd_suffix* as $\mathcal{P}$. The only difference is in the index-state at an index internal to the **loop while** statement: j. The difference, however, does not prevent

the continued execution of $\mathcal{M}$ from producing a categorically false environment. This fact follows from the internal index independence lemma (see Chapter IV).

If Br_Cd is false, execution of $\mathcal{M}$ beginning in env′ will have nearly the same environment just prior to *fol_top_lev_code* as execution of $\mathcal{P}$ beginning in env. The only differences are in the index-state at j and l. These differences, however, do not prevent the continued execution of $\mathcal{M}$ from producing a categorically false environment. This fact follows from the negative-branch-condition independence lemma (see Chapter IV). Hence, env′ is what we were seeking—a witness to $\mathcal{M}$'s invalidity.

3.9 The Rule for confirm

The rule for **confirm** produces phase 1 programs by rewriting other phase 1 programs. The equations of Figure 55 define the rule for **confirm**. Figure 56 shows the result of applying the **confirm** rule to the example of Figure 52. This application removes the **confirm** statement in the very last **whenever** statement, inserting a modified **confirm** statement just prior to the **whenever** statement. The assertion in this **confirm** statement contains no current variables because it was not written by a programmer but generated by a proof rule. If it had been written by a programmer, the **confirm** rule directs that all free occurrences of current variables be replaced by indexed variables. No replacements had to be made in this application. The soundness of the **confirm** rule follows easily from the observation that if env is a witness to the invalidity of $\mathcal{P}$, it is also a witness to the invalidity of $\mathcal{M}$. This fact

$$\mathcal{P} \stackrel{\text{def}}{=} C\backslash \quad \begin{array}{l} \mathit{prec\_top\_lev\_code} \\ \textbf{alter all} \\ \textbf{stow}(i) \\ \mathit{ACseq}_0 \\ \quad | \ \textbf{whenever } \text{Br\_Cd } \textbf{do} \\ \quad | \quad \textbf{confirm } H\lfloor \text{x} \rfloor \\ \quad | \quad \mathit{cd\_suffix} \\ \quad | \ \textbf{end whenever} \\ \mathit{fol\_top\_lev\_code} \end{array} \tag{3.36}$$

$$\mathcal{M} \stackrel{\text{def}}{=} C\backslash \quad \begin{array}{l} \mathit{prec\_top\_lev\_code} \\ \textbf{alter all} \\ \textbf{stow}(i) \\ \mathit{ACseq}_0 \\ \quad | \ \textbf{confirm } (\text{Br\_Cd}) \Rightarrow (H[\text{x} \rightsquigarrow \text{x}_i]) \\ \quad | \ \textbf{whenever } \text{Br\_Cd } \textbf{do} \\ \quad | \quad \mathit{cd\_suffix} \\ \quad | \ \textbf{end whenever} \\ \mathit{fol\_top\_lev\_code} \end{array} \tag{3.37}$$

Figure 55: Equations Defining the Rule for **confirm**

$C'\backslash$ **alter all**
stow(0)
assume (**true**) $\Rightarrow$ (**true** $\wedge$ **is_initial**($\mathrm{q\_is\_empty}_0$))
confirm (**true**) $\Rightarrow$ (**true**)
alter all
stow(10)
assume (**true**) $\Rightarrow$ ($\mathrm{x}_{10} = \mathrm{x}_0 \wedge ((\mathrm{q}_0 = \mathrm{q}_{10}) \wedge (\mathrm{q}_{10} = \Lambda \Leftrightarrow \mathrm{q\_is\_empty}_{10}))$)
alter all
stow(20)
whenever (**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$) **do**
  assume $\mathrm{q\_is\_empty}_{20} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{20} = \mathrm{q}_{10} \wedge \mathrm{x}_{20} = \mathrm{x}_{10}$
  Make_two(x)
  stow(30)
end whenever
alter all
stow(40)
whenever (**true**) $\wedge \neg$($\mathrm{q\_is\_empty}_{10}$) **do**
  assume $\mathrm{q\_is\_empty}_{40} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{40} = \mathrm{q}_{10} \wedge \mathrm{x}_{40} = \mathrm{x}_{10}$
  Make_three(x)
  stow(50)
end whenever
alter all
stow(60)
assume ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
  ($\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{30} \wedge \mathrm{q}_{60} = \mathrm{q}_{30} \wedge \mathrm{x}_{60} = \mathrm{x}_{30}$)
assume ((**true**) $\wedge \neg$($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
  ($\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{50} \wedge \mathrm{q}_{60} = \mathrm{q}_{50} \wedge \mathrm{x}_{60} = \mathrm{x}_{50}$)
confirm (**true**) $\Rightarrow$ (($\mathrm{q}_{60} = \mathrm{q}_0$) $\wedge$ ($\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2$) $\wedge$ ($\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3$))
whenever true do
end whenever

Figure 56: Application of Rule for **confirm**

is true due to the meanings of mathematical implication ($\Rightarrow$) and the **whenever** statement.

Figure 57 shows the result of applying the **assume** rule to the example of Figure 56. This application removes the **assume** statement just prior to the call to Make_three, inserting a modified **assume** statement just prior to the **whenever** statement. At this point, at least two different rule applications are possible. We may either apply the **assume** rule to the first **whenever** statement, or remove the call to Make_three with an application of the procedure call rule to the second **whenever** statement. To emphasize that no specific order of application need be followed when several rules may be applied, we now use the procedure call rule to replace the call to Make_three. This result, instantiating i to 40 and j to 50, is shown in Figure 58.

3.10 The Rule for Empty Guarded Blocks

The rule for empty guarded blocks rewrites phase 1 programs by removing one **whenever** statement whose statement sequence is empty. The result is in phase 1 if the program still contains at least one **whenever** statement. Otherwise, the result is in phase 2. The equations of Figure 59 define the rule for empty guarded blocks. Figure 60 shows an application of this rule to remove the **whenever** statement from the back end of Figure 58's program. The soundness of the rule for empty guarded blocks follows easily from the observation that if env is a witness to the invalidity of $\mathcal{P}$, it is also a witness to the invalidity of $\mathcal{M}$. This fact is true because the meaning of a **whenever** statement whose statement sequence is empty is the identity function

$C'\backslash$ **alter all**
stow(0)
assume (**true**) $\Rightarrow$ (**true** $\wedge$ **is_initial**($\mathrm{q\_is\_empty}_0$))
confirm (**true**) $\Rightarrow$ (**true**)
alter all
stow(10)
assume (**true**) $\Rightarrow$ ($\mathrm{x}_{10} = \mathrm{x}_0 \wedge ((\mathrm{q}_0 = \mathrm{q}_{10}) \wedge (\mathrm{q}_{10} = \Lambda \Leftrightarrow \mathrm{q\_is\_empty}_{10}))$)
alter all
stow(20)
whenever (**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$) **do**
  assume $\mathrm{q\_is\_empty}_{20} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{20} = \mathrm{q}_{10} \wedge \mathrm{x}_{20} = \mathrm{x}_{10}$
  Make_two(x)
  stow(30)
end whenever
alter all
stow(40)
assume ((**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10}) \Rightarrow$
  ($\mathrm{q\_is\_empty}_{40} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{40} = \mathrm{q}_{10} \wedge \mathrm{x}_{40} = \mathrm{x}_{10}$)
whenever (**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})$ **do**
  Make_three(x)
  stow(50)
end whenever
alter all
stow(60)
assume ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
  ($\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{30} \wedge \mathrm{q}_{60} = \mathrm{q}_{30} \wedge \mathrm{x}_{60} = \mathrm{x}_{30}$)
assume ((**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow$
  ($\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{50} \wedge \mathrm{q}_{60} = \mathrm{q}_{50} \wedge \mathrm{x}_{60} = \mathrm{x}_{50}$)
confirm (**true**) $\Rightarrow ((\mathrm{q}_{60} = \mathrm{q}_0) \wedge (\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2) \wedge (\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3))$
whenever true do
end whenever

Figure 57: Third Application of Rule for **assume**

$C'\backslash$ **alter all**
stow(0)
assume (**true**) $\Rightarrow$ (**true** $\wedge$ **is_initial**($\mathrm{q\_is\_empty}_0$))
confirm (**true**) $\Rightarrow$ (**true**)
alter all
stow(10)
assume (**true**) $\Rightarrow$ ($\mathrm{x}_{10} = \mathrm{x}_0 \wedge ((\mathrm{q}_0 = \mathrm{q}_{10}) \wedge (\mathrm{q}_{10} = \Lambda \Leftrightarrow \mathrm{q\_is\_empty}_{10}))$)
alter all
stow(20)
whenever (**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$) **do**
  assume $\mathrm{q\_is\_empty}_{20} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{20} = \mathrm{q}_{10} \wedge \mathrm{x}_{20} = \mathrm{x}_{10}$
  Make_two(x)
  stow(30)
end whenever
alter all
stow(40)
assume ((**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow$
  $(\mathrm{q\_is\_empty}_{40} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{40} = \mathrm{q}_{10} \wedge \mathrm{x}_{40} = \mathrm{x}_{10})$
confirm ((**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow$ (**true**)
alter all
stow(50)
assume ((**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow$
  $(\mathrm{q\_is\_empty}_{50} = \mathrm{q\_is\_empty}_{40} \wedge \mathrm{q}_{50} = \mathrm{q}_{40} \wedge \mathrm{x}_{50} = 3)$
whenever (**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})$ **do**
end whenever
alter all
stow(60)
assume ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
  $(\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{30} \wedge \mathrm{q}_{60} = \mathrm{q}_{30} \wedge \mathrm{x}_{60} = \mathrm{x}_{30})$
assume ((**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow$
  $(\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{50} \wedge \mathrm{q}_{60} = \mathrm{q}_{50} \wedge \mathrm{x}_{60} = \mathrm{x}_{50})$
confirm (**true**) $\Rightarrow ((\mathrm{q}_{60} = \mathrm{q}_0) \wedge (\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2) \wedge (\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3))$
whenever true do
end whenever

Figure 58: Second Application of Procedure Call Rule

$$\mathcal{P} \stackrel{\text{def}}{=} C\backslash \begin{array}{l} \mathit{prec\_top\_lev\_code} \\ \textbf{alter all} \\ \textbf{stow}(i) \\ \mathit{ACseq}_0 \\ \quad \mid \textbf{whenever } \text{Br\_Cd } \textbf{do} \\ \quad \mid \textbf{end whenever} \\ \mathit{fol\_top\_lev\_code} \end{array} \tag{3.38}$$

$$\mathcal{M} \stackrel{\text{def}}{=} C\backslash \begin{array}{l} \mathit{prec\_top\_lev\_code} \\ \textbf{alter all} \\ \textbf{stow}(i) \\ \mathit{ACseq}_0 \\ \mathit{fol\_top\_lev\_code} \end{array} \tag{3.39}$$

Figure 59: Equations Defining the Rule for Empty Guarded Blocks

from environments to environments. At this point, we pick up the pace and apply the **assume** rule, the procedure call rule (for Make_two), and the rule for empty guarded blocks (twice) to produce Figure 61.

3.11 The Rule for alter all

The rule for **alter all** rewrites phase 2 programs by removing one **alter all-stow** two-statement sequence. The result is in phase 2 if the program still contains at least one **alter all** statement. Otherwise, the result is in phase 3. The equations and the additional syntactic restriction of Figure 62 define the rule for **alter all**. We can only apply this rule after all **whenever** statements have been removed. Applying it too soon—in phase 1—could keep other rules from being applicable, preventing the rewrite process from leaving phase 1. Figure 63 shows what we obtain from Figure 61 with seven applications of the **alter all** rule.

$C'\backslash$ **alter all**
stow(0)
assume (**true**) $\Rightarrow$ (**true** $\wedge$ **is_initial**($\mathrm{q\_is\_empty}_0$))
confirm (**true**) $\Rightarrow$ (**true**)
alter all
stow(10)
assume (**true**) $\Rightarrow$ ($\mathrm{x}_{10} = \mathrm{x}_0 \wedge ((\mathrm{q}_0 = \mathrm{q}_{10}) \wedge (\mathrm{q}_{10} = \Lambda \Leftrightarrow \mathrm{q\_is\_empty}_{10}))$)
alter all
stow(20)
whenever (**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$) **do**
  assume $\mathrm{q\_is\_empty}_{20} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{20} = \mathrm{q}_{10} \wedge \mathrm{x}_{20} = \mathrm{x}_{10}$
  Make_two(x)
  stow(30)
end whenever
alter all
stow(40)
assume ((**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow$
  ($\mathrm{q\_is\_empty}_{40} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{40} = \mathrm{q}_{10} \wedge \mathrm{x}_{40} = \mathrm{x}_{10}$)
confirm ((**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow$ (**true**)
alter all
stow(50)
assume ((**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow$
  ($\mathrm{q\_is\_empty}_{50} = \mathrm{q\_is\_empty}_{40} \wedge \mathrm{q}_{50} = \mathrm{q}_{40} \wedge \mathrm{x}_{50} = 3$)
whenever (**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})$ **do**
end whenever
alter all
stow(60)
assume ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10})) \Rightarrow$
  ($\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{30} \wedge \mathrm{q}_{60} = \mathrm{q}_{30} \wedge \mathrm{x}_{60} = \mathrm{x}_{30}$)
assume ((**true**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow$
  ($\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{50} \wedge \mathrm{q}_{60} = \mathrm{q}_{50} \wedge \mathrm{x}_{60} = \mathrm{x}_{50}$)
confirm (**true**) $\Rightarrow ((\mathrm{q}_{60} = \mathrm{q}_0) \wedge (\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2) \wedge (\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3))$

Figure 60: First Application of Rule for Empty Guarded Blocks

$C'\backslash$ **alter all**
stow(0)
assume (**true**) $\Rightarrow$ (**true** $\wedge$ **is_initial**($\mathrm{q\_is\_empty}_0$))
confirm (**true**) $\Rightarrow$ (**true**)
alter all
stow(10)
assume (**true**) $\Rightarrow$ ($\mathrm{x}_{10} = \mathrm{x}_0 \wedge ((\mathrm{q}_0 = \mathrm{q}_{10}) \wedge (\mathrm{q}_{10} = \Lambda \Leftrightarrow \mathrm{q\_is\_empty}_{10}))$)
alter all
stow(20)
assume ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
($\mathrm{q\_is\_empty}_{20} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{20} = \mathrm{q}_{10} \wedge \mathrm{x}_{20} = \mathrm{x}_{10}$)
confirm ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$ (**true**)
alter all
stow(30)
assume ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
($\mathrm{q\_is\_empty}_{30} = \mathrm{q\_is\_empty}_{20} \wedge \mathrm{q}_{30} = \mathrm{q}_{20} \wedge \ \mathrm{x}_{30} = 2$)
alter all
stow(40)
assume ((**true**) $\wedge \neg$($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
($\mathrm{q\_is\_empty}_{40} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{40} = \mathrm{q}_{10} \wedge \mathrm{x}_{40} = \mathrm{x}_{10}$)
confirm ((**true**) $\wedge \neg$($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$ (**true**)
alter all
stow(50)
assume ((**true**) $\wedge \neg$($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
($\mathrm{q\_is\_empty}_{50} = \mathrm{q\_is\_empty}_{40} \wedge \mathrm{q}_{50} = \mathrm{q}_{40} \wedge \mathrm{x}_{50} = 3$)
alter all
stow(60)
assume ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
($\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{30} \wedge \mathrm{q}_{60} = \mathrm{q}_{30} \wedge \mathrm{x}_{60} = \mathrm{x}_{30}$)
assume ((**true**) $\wedge \neg$($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
($\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{50} \wedge \mathrm{q}_{60} = \mathrm{q}_{50} \wedge \mathrm{x}_{60} = \mathrm{x}_{50}$)
confirm (**true**) $\Rightarrow$ (($\mathrm{q}_{60} = \mathrm{q}_0$) $\wedge$ ($\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2$) $\wedge$ ($\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3$))

Figure 61: Application of Three Different Rules

$$\mathcal{P} = C\backslash \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textbf{alter all} \\ \textbf{stow}(i) \\ \textit{fol\_top\_lev\_code} \end{array} \tag{3.40}$$

$$\mathcal{M} = C\backslash \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textit{fol\_top\_lev\_code} \end{array} \tag{3.41}$$

Additional Syntactic Restriction:

Each of the nonterminal symbols ⟨gd_blk⟩ of *prec_top_lev_code* and *fol_top_lev_code* is rewritten to the empty string (ε); i.e., $\mathcal{P}$ is a phase 2 program—one that contains no **whenever** statements.

Figure 62: Equations and Additional Syntactic Restriction Defining the Rule for **alter all**

$C'\backslash$ **assume** (**true**) $\Rightarrow$ (**true** $\wedge$ **is_initial**($\mathrm{q\_is\_empty}_0$))
confirm (**true**) $\Rightarrow$ (**true**)
assume (**true**) $\Rightarrow$ ($\mathrm{x}_{10} = \mathrm{x}_0 \wedge ((\mathrm{q}_0 = \mathrm{q}_{10}) \wedge (\mathrm{q}_{10} = \Lambda \Leftrightarrow \mathrm{q\_is\_empty}_{10}))$)
assume ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
($\mathrm{q\_is\_empty}_{20} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{20} = \mathrm{q}_{10} \wedge \mathrm{x}_{20} = \mathrm{x}_{10}$)
confirm ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$ (**true**)
assume ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
($\mathrm{q\_is\_empty}_{30} = \mathrm{q\_is\_empty}_{20} \wedge \mathrm{q}_{30} = \mathrm{q}_{20} \wedge \ \mathrm{x}_{30} = 2$)
assume ((**true**) $\wedge \neg$($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
($\mathrm{q\_is\_empty}_{40} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{40} = \mathrm{q}_{10} \wedge \mathrm{x}_{40} = \mathrm{x}_{10}$)
confirm ((**true**) $\wedge \neg$($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$ (**true**)
assume ((**true**) $\wedge \neg$($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
($\mathrm{q\_is\_empty}_{50} = \mathrm{q\_is\_empty}_{40} \wedge \mathrm{q}_{50} = \mathrm{q}_{40} \wedge \mathrm{x}_{50} = 3$)
assume ((**true**) $\wedge$ ($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
($\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{30} \wedge \mathrm{q}_{60} = \mathrm{q}_{30} \wedge \mathrm{x}_{60} = \mathrm{x}_{30}$)
assume ((**true**) $\wedge \neg$($\mathrm{q\_is\_empty}_{10}$)) $\Rightarrow$
($\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{50} \wedge \mathrm{q}_{60} = \mathrm{q}_{50} \wedge \mathrm{x}_{60} = \mathrm{x}_{50}$)
confirm (**true**) $\Rightarrow$ ($(\mathrm{q}_{60} = \mathrm{q}_0) \wedge (\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2) \wedge (\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3)$)

Figure 63: Seven Applications of the Rule for **alter all**

$$\mathcal{P} \;=\; C\backslash \quad \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textbf{assume}\ H_1 \\ \textbf{assume}\ H_2 \\ \textit{fol\_top\_lev\_code} \end{array} \tag{3.42}$$

$$\mathcal{M} \;=\; C\backslash \quad \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textbf{assume}\ (H_1) \wedge (H_2) \\ \textit{fol\_top\_lev\_code} \end{array} \tag{3.43}$$

Figure 64: Equations Defining the Rule for Consecutive **assume** Statements

To indicate informally why the rule for **alter all** is sound, we suppose that env is a witness to $\mathcal{P}$'s invalidity, and discuss how to construct env$'$ to be a witness to $\mathcal{M}$'s invalidity. All we have to do is make env$'$'s index-state at i equal the current state at the execution in $\mathcal{P}$ of **stow**(i). All other parts of env$'$ are the same as those of env. Because references to index i occur only after **stow**(i) (see page 93), execution of *prec_top_lev_code* **alter all stow**(i) beginning in env produces the same environment as execution of *prec_top_lev_code* beginning in env$'$. Hence, env$'$ is what we were seeking—a witness to $\mathcal{M}$'s invalidity.

3.12 The Rule for Consecutive assume Statements

The rule for consecutive **assume** statements may be applied in any phase, but will typically be applied in phase 3. The equations of Figure 64 define this rule. When working in the math direction, this rule is useful anytime there are two or more **assume** statements in a row. Four applications of this rule were used to produce Figure 65 from Figure 63.

$$
\begin{array}{ll}
C'\backslash & \textbf{assume}\ (\textbf{true}) \Rightarrow (\textbf{true} \wedge \textbf{is\_initial}(\mathrm{q\_is\_empty}_0)) \\
& \textbf{confirm}\ (\textbf{true}) \Rightarrow (\textbf{true}) \\
& \textbf{assume}\ ((\textbf{true}) \Rightarrow (\mathrm{x}_{10} = \mathrm{x}_0 \wedge ((\mathrm{q}_0 = \mathrm{q}_{10}) \wedge (\mathrm{q}_{10} = \Lambda \Leftrightarrow \mathrm{q\_is\_empty}_{10})))) \\
& \qquad \wedge (((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\quad (\mathrm{q\_is\_empty}_{20} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{20} = \mathrm{q}_{10} \wedge \mathrm{x}_{20} = \mathrm{x}_{10})) \\
& \textbf{confirm}\ ((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow (\textbf{true}) \\
& \textbf{assume}\ (((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\quad (\mathrm{q\_is\_empty}_{30} = \mathrm{q\_is\_empty}_{20} \wedge \mathrm{q}_{30} = \mathrm{q}_{20} \wedge\ \mathrm{x}_{30} = 2)) \\
& \qquad \wedge (((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\quad (\mathrm{q\_is\_empty}_{40} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{40} = \mathrm{q}_{10} \wedge \mathrm{x}_{40} = \mathrm{x}_{10})) \\
& \textbf{confirm}\ ((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow (\textbf{true}) \\
& \textbf{assume}\ (((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\quad (\mathrm{q\_is\_empty}_{50} = \mathrm{q\_is\_empty}_{40} \wedge \mathrm{q}_{50} = \mathrm{q}_{40} \wedge \mathrm{x}_{50} = 3)) \\
& \qquad \wedge ((((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\quad (\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{30} \wedge \mathrm{q}_{60} = \mathrm{q}_{30} \wedge \mathrm{x}_{60} = \mathrm{x}_{30})) \\
& \qquad \wedge (((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\quad (\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{50} \wedge \mathrm{q}_{60} = \mathrm{q}_{50} \wedge \mathrm{x}_{60} = \mathrm{x}_{50}))) \\
& \textbf{confirm}\ (\textbf{true}) \Rightarrow ((\mathrm{q}_{60} = \mathrm{q}_0) \wedge (\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2) \wedge (\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3))
\end{array}
$$

Figure 65: Four Applications of the Rule for Consecutive **assume** Statements

$$\mathcal{P} = C\backslash \begin{array}{l} top\_lev\_code \\ \textbf{assume } H_1 \\ \textbf{confirm } H_2 \end{array} \tag{3.44}$$

$$\mathcal{M} = C\backslash \begin{array}{l} top\_lev\_code \\ \textbf{confirm } (H_1) \Rightarrow (H_2) \end{array} \tag{3.45}$$

Figure 66: Equations Defining the **assume-confirm** Rule

The soundness of the rule for consecutive **assume** statements follows easily from the observation that if env is a witness to the invalidity of $\mathcal{P}$, it is also a witness to the invalidity of $\mathcal{M}$. We know that neither execution of **assume** H_1 nor **assume** H_2 in $\mathcal{P}$ starting from env changed the assert status to VT. Therefore, execution of **assume** $(H_1) \wedge (H_2)$ in $\mathcal{M}$ starting from env does not change the assert status to VT. So, execution of $\mathcal{M}$ from env produces the same environment as execution of $\mathcal{P}$ from env.

3.13 The assume-confirm Rule

The **assume-confirm** rule may be applied in any phase, but will typically be applied in phase 3. The equations of Figure 66 define this rule. When working in the math direction, this rule is useful only when the last statement is a **confirm** statement, and it is preceded by an **assume** statement. We used this rule to produce Figure 67 from Figure 65. The soundness of the **assume-confirm** rule follows easily from the observation that if env is a witness to the invalidity of $\mathcal{P}$, it is also a witness to the invalidity of $\mathcal{M}$. This fact is true due to the meanings of mathematical implication ($\Rightarrow$) and the **assume** and **confirm** statements.

$$
\begin{array}{l}
C'\backslash\ \textbf{assume}\ (\textbf{true}) \Rightarrow (\textbf{true} \wedge \textbf{is\_initial}(\mathrm{q\_is\_empty}_0)) \\
\quad \textbf{confirm}\ (\textbf{true}) \Rightarrow (\textbf{true}) \\
\quad \textbf{assume}\ ((\textbf{true}) \Rightarrow (\mathrm{x}_{10} = \mathrm{x}_0 \wedge ((\mathrm{q}_0 = \mathrm{q}_{10}) \wedge (\mathrm{q}_{10} = \Lambda \Leftrightarrow \mathrm{q\_is\_empty}_{10})))) \\
\qquad\qquad \wedge (((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
\qquad\qquad\quad (\mathrm{q\_is\_empty}_{20} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{20} = \mathrm{q}_{10} \wedge \mathrm{x}_{20} = \mathrm{x}_{10})) \\
\quad \textbf{confirm}\ ((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow (\textbf{true}) \\
\quad \textbf{assume}\ (((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
\qquad\qquad\quad (\mathrm{q\_is\_empty}_{30} = \mathrm{q\_is\_empty}_{20} \wedge \mathrm{q}_{30} = \mathrm{q}_{20} \wedge\ \mathrm{x}_{30} = 2)) \\
\qquad\qquad \wedge (((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
\qquad\qquad\quad (\mathrm{q\_is\_empty}_{40} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{40} = \mathrm{q}_{10} \wedge \mathrm{x}_{40} = \mathrm{x}_{10})) \\
\quad \textbf{confirm}\ ((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow (\textbf{true}) \\
\quad \textbf{confirm}\ ((((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
\qquad\qquad\quad (\mathrm{q\_is\_empty}_{50} = \mathrm{q\_is\_empty}_{40} \wedge \mathrm{q}_{50} = \mathrm{q}_{40} \wedge \mathrm{x}_{50} = 3)) \\
\qquad\qquad \wedge ((((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
\qquad\qquad\quad (\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{30} \wedge \mathrm{q}_{60} = \mathrm{q}_{30} \wedge \mathrm{x}_{60} = \mathrm{x}_{30})) \\
\qquad\qquad \wedge (((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
\qquad\qquad\quad (\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{50} \wedge \mathrm{q}_{60} = \mathrm{q}_{50} \wedge \mathrm{x}_{60} = \mathrm{x}_{50})))) \Rightarrow \\
\qquad\qquad ((\textbf{true}) \Rightarrow ((\mathrm{q}_{60} = \mathrm{q}_0) \wedge (\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2) \wedge (\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3)))
\end{array}
$$

Figure 67: An Application of the **assume-confirm** Rule

$$\mathcal{P} = C\backslash \begin{array}{l} prec\_top\_lev\_code \\ \textbf{confirm } H_1 \\ \textbf{confirm } H_2 \\ fol\_top\_lev\_code \end{array} \quad (3.46)$$

$$\mathcal{M} = C\backslash \begin{array}{l} prec\_top\_lev\_code \\ \textbf{confirm } (H_1) \wedge (H_2) \\ fol\_top\_lev\_code \end{array} \quad (3.47)$$

Figure 68: Equations Defining the Rule for Consecutive **confirm** Statements

3.14 The Rule for Consecutive confirm Statements

The rule for consecutive **confirm** statements may be applied in any phase, but will typically be applied in phase 3. The equations of Figure 68 define this rule. When working in the math direction, this rule is useful anytime there are two or more **confirm** statements in a row. However, these will typically be the last two statements, a situation arising from application of the **assume-confirm** rule. An application of the rule for consecutive **confirm** statements produced Figure 69 from Figure 67. The soundness of the rule for consecutive **confirm** statements follows easily from the observation that if env is a witness to the invalidity of $\mathcal{P}$, it is also a witness to the invalidity of $\mathcal{M}$. If execution of either **confirm** H_1 or **confirm** H_2 changes the assert status form NL to CF, then, due to the meaning of mathematical conjunction ($\wedge$), execution of **confirm** $(H_1) \wedge (H_2)$ will change the assert status from NL to CF. Returning to our running example, we used two alternating applications of the **assume-confirm** rule and the rule for consecutive **confirm** statements, and a

$$
\begin{aligned}
C'\setminus\ & \textbf{assume}\ (\textbf{true}) \Rightarrow (\textbf{true} \wedge \textbf{is\_initial}(\mathrm{q\_is\_empty}_0)) \\
& \textbf{confirm}\ (\textbf{true}) \Rightarrow (\textbf{true}) \\
& \textbf{assume}\ ((\textbf{true}) \Rightarrow (\mathrm{x}_{10} = \mathrm{x}_0 \wedge ((\mathrm{q}_0 = \mathrm{q}_{10}) \wedge (\mathrm{q}_{10} = \Lambda \Leftrightarrow \mathrm{q\_is\_empty}_{10})))) \\
& \qquad \wedge (((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\quad (\mathrm{q\_is\_empty}_{20} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{20} = \mathrm{q}_{10} \wedge \mathrm{x}_{20} = \mathrm{x}_{10})) \\
& \textbf{confirm}\ ((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow (\textbf{true}) \\
& \textbf{assume}\ (((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\quad (\mathrm{q\_is\_empty}_{30} = \mathrm{q\_is\_empty}_{20} \wedge \mathrm{q}_{30} = \mathrm{q}_{20} \wedge\ \mathrm{x}_{30} = 2)) \\
& \qquad \wedge (((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\quad (\mathrm{q\_is\_empty}_{40} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{40} = \mathrm{q}_{10} \wedge \mathrm{x}_{40} = \mathrm{x}_{10})) \\
& \textbf{confirm}\ (((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow (\textbf{true})) \\
& \qquad \wedge (((((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\quad (\mathrm{q\_is\_empty}_{50} = \mathrm{q\_is\_empty}_{40} \wedge \mathrm{q}_{50} = \mathrm{q}_{40} \wedge \mathrm{x}_{50} = 3)) \\
& \qquad \wedge ((((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\quad (\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{30} \wedge \mathrm{q}_{60} = \mathrm{q}_{30} \wedge \mathrm{x}_{60} = \mathrm{x}_{30})) \\
& \qquad \wedge (((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\quad (\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{50} \wedge \mathrm{q}_{60} = \mathrm{q}_{50} \wedge \mathrm{x}_{60} = \mathrm{x}_{50})))) \Rightarrow \\
& \qquad ((\textbf{true}) \Rightarrow ((\mathrm{q}_{60} = \mathrm{q}_0) \wedge (\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2) \wedge (\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3))))
\end{aligned}
$$

Figure 69: An Application of the Rule for Consecutive **confirm** Statements

$$
\begin{aligned}
C'\backslash\ \textbf{confirm}\ &((\textbf{true}) \Rightarrow (\textbf{true} \wedge \textbf{is\_initial}(\text{q\_is\_empty}_0))) \Rightarrow \\
&(((\textbf{true}) \Rightarrow (\textbf{true})) \\
&\wedge ((((\textbf{true}) \Rightarrow (x_{10} = x_0 \wedge ((q_0 = q_{10}) \wedge (q_{10} = \Lambda \Leftrightarrow \text{q\_is\_empty}_{10})))) \\
&\wedge (((\textbf{true}) \wedge (\text{q\_is\_empty}_{10})) \Rightarrow \\
&\quad (\text{q\_is\_empty}_{20} = \text{q\_is\_empty}_{10} \wedge q_{20} = q_{10} \wedge x_{20} = x_{10}))) \Rightarrow \\
&((((\textbf{true}) \wedge (\text{q\_is\_empty}_{10})) \Rightarrow (\textbf{true})) \\
&\wedge (((((\textbf{true}) \wedge (\text{q\_is\_empty}_{10})) \Rightarrow \\
&\quad (\text{q\_is\_empty}_{30} = \text{q\_is\_empty}_{20} \wedge q_{30} = q_{20} \wedge\ x_{30} = 2)) \\
&\wedge (((\textbf{true}) \wedge \neg(\text{q\_is\_empty}_{10})) \Rightarrow \\
&\quad (\text{q\_is\_empty}_{40} = \text{q\_is\_empty}_{10} \wedge q_{40} = q_{10} \wedge x_{40} = x_{10}))) \Rightarrow \\
&((((\textbf{true}) \wedge \neg(\text{q\_is\_empty}_{10})) \Rightarrow (\textbf{true})) \\
&\wedge (((((\textbf{true}) \wedge \neg(\text{q\_is\_empty}_{10})) \Rightarrow \\
&\quad (\text{q\_is\_empty}_{50} = \text{q\_is\_empty}_{40} \wedge q_{50} = q_{40} \wedge x_{50} = 3)) \\
&\wedge ((((\textbf{true}) \wedge (\text{q\_is\_empty}_{10})) \Rightarrow \\
&\quad (\text{q\_is\_empty}_{60} = \text{q\_is\_empty}_{30} \wedge q_{60} = q_{30} \wedge x_{60} = x_{30})) \\
&\wedge (((\textbf{true}) \wedge \neg(\text{q\_is\_empty}_{10})) \Rightarrow \\
&\quad (\text{q\_is\_empty}_{60} = \text{q\_is\_empty}_{50} \wedge q_{60} = q_{50} \wedge x_{60} = x_{50})))) \Rightarrow \\
&((\textbf{true}) \Rightarrow ((q_{60} = q_0) \wedge (q_{60} = \Lambda \Rightarrow x_{60} = 2) \wedge (q_{60} \neq \Lambda \Rightarrow x_{60} = 3)))))))))
\end{aligned}
$$

Figure 70: Five Rule Applications

final application of the **assume-confirm** rule (five rule applications all together) to produce Figure 70 from Figure 69.

3.15 The Rule of Inference Bridging Predicate Logic and the Indexed Method

The rule of inference presented here as equation 3.48 enables one to establish the validity of the block, $C\backslash$ **confirm** H, from the truth of the assertion, H, in every model for the theories needed to define the symbols used in H, according to the theory definitions in context, C.

$$\frac{C\backslash \quad H}{C\backslash \quad \textbf{confirm}\ H} \tag{3.48}$$

$$
\begin{aligned}
C'\backslash\ & ((\mathbf{true}) \Rightarrow (\mathbf{true} \wedge \mathbf{is\_initial}(\mathrm{q\_is\_empty}_0))) \Rightarrow \\
& (((\mathbf{true}) \Rightarrow (\mathbf{true})) \\
& \wedge ((((\mathbf{true}) \Rightarrow (\mathrm{x}_{10} = \mathrm{x}_0 \wedge ((\mathrm{q}_0 = \mathrm{q}_{10}) \wedge (\mathrm{q}_{10} = \Lambda \Leftrightarrow \mathrm{q\_is\_empty}_{10})))) \\
& \wedge (((\mathbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad (\mathrm{q\_is\_empty}_{20} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{20} = \mathrm{q}_{10} \wedge \mathrm{x}_{20} = \mathrm{x}_{10}))) \Rightarrow \\
& ((((\mathbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow (\mathbf{true})) \\
& \wedge (((((\mathbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad (\mathrm{q\_is\_empty}_{30} = \mathrm{q\_is\_empty}_{20} \wedge \mathrm{q}_{30} = \mathrm{q}_{20} \wedge\ \mathrm{x}_{30} = 2)) \\
& \wedge (((\mathbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad (\mathrm{q\_is\_empty}_{40} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{40} = \mathrm{q}_{10} \wedge \mathrm{x}_{40} = \mathrm{x}_{10}))) \Rightarrow \\
& ((((\mathbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow (\mathbf{true})) \\
& \wedge (((((\mathbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad (\mathrm{q\_is\_empty}_{50} = \mathrm{q\_is\_empty}_{40} \wedge \mathrm{q}_{50} = \mathrm{q}_{40} \wedge \mathrm{x}_{50} = 3)) \\
& \wedge ((((\mathbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad (\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{30} \wedge \mathrm{q}_{60} = \mathrm{q}_{30} \wedge \mathrm{x}_{60} = \mathrm{x}_{30})) \\
& \wedge (((\mathbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad (\mathrm{q\_is\_empty}_{60} = \mathrm{q\_is\_empty}_{50} \wedge \mathrm{q}_{60} = \mathrm{q}_{50} \wedge \mathrm{x}_{60} = \mathrm{x}_{50}))) \Rightarrow \\
& ((\mathbf{true}) \Rightarrow ((\mathrm{q}_{60} = \mathrm{q}_0) \wedge (\mathrm{q}_{60} = \Lambda \Rightarrow \mathrm{x}_{60} = 2) \wedge (\mathrm{q}_{60} \neq \Lambda \Rightarrow \mathrm{x}_{60} = 3)))))))))
\end{aligned}
$$

Figure 71: Mathematical Assertion in Context C'

For example, establishing the mathematical assertion of Figure 71 to be true in context C' assures the validity of the one-statement assertive program of Figure 70. Equation 3.48 states this rule in the program direction. Its application in the opposite direction—the math direction—is step 4 of Figure 31. The soundness of this rule follows easily from the observation that if env is a witness to the invalidity of $C\backslash$ **confirm** H, it is also a witness to the invalidity of $C\backslash\ H$. In other words, if execution of the **confirm** statement in the neutral environment env takes the assert status to CF, then env contains an assignment of values to the free variables of H that makes H false.

$C''\backslash$ **alter all**
stow(100)
assume $q_{100} = \Lambda \Leftrightarrow$ q_is_empty$_{100}$
whenever true do
loop
maintaining $q = \Lambda \Leftrightarrow$ q_is_empty
while not q_is_empty **do**
stow(110)
Dequeue(q, y)
stow(120)
Test_If_Empty(q, q_is_empty)
stow(130)
end loop
stow(140)
confirm $q_{140} = \Lambda$
end whenever

where $C'' = \{$ y: Item
procedure Dequeue (q: Queue
x: Item)
requires "$q \neq \Lambda$"
ensures "$\#q = \langle x \rangle * q$"$\} \cup C'$.

Figure 72: Example: Iteration

3.16 Example Application of the loop while Rule

Figure 72 shows an example assertive program to which we can apply the **loop while** rule, which was defined in Figures 53 and 54. C' is the context shown in Figure 39. The result of this application, having instantiated i to 100, j to 110, k to 130, and l to 140, is shown in Figure 73.

$C''\backslash$ **alter all**
stow(100)
assume $q_{100} = \Lambda \Leftrightarrow$ q_is_empty$_{100}$
confirm (**true**) $\Rightarrow (q_{100} = \Lambda \Leftrightarrow$ q_is_empty$_{100})$
alter all
stow(110)
whenever (**true**) $\wedge (\neg$q_is_empty$_{100})$ **do**
assume $(\neg$q_is_empty$_{110}) \wedge (q_{110} = \Lambda \Leftrightarrow$ q_is_empty$_{110})$
Dequeue(q, y)
stow(120)
Test_If_Empty(q, q_is_empty)
stow(130)
confirm $q_{130} = \Lambda \Leftrightarrow$ q_is_empty$_{130}$
end whenever
stow(140)
alter all
whenever true do
assume $(\neg(\neg$q_is_empty$_{140})) \wedge (q_{140} = \Lambda \Leftrightarrow$ q_is_empty$_{140})$
confirm $q_{140} = \Lambda$
end whenever

Figure 73: Application of the **loop while** Rule

3.17 Summary

In this chapter we have defined the proof rules of the indexed method. We have presented an example application of each of those rules. Furthermore, we have argued informally why each rule preserves invalidity when applied in the math direction. Because two contrapositives are equivalent, our arguments established informally that each rule preserves validity when applied in the program direction. Thus, we have completed an informal argument that shows the indexed method to be sound. Chapter IV argues more precisely for the soundness and relative completeness of the indexed method.

CHAPTER IV

Soundness and Relative Completeness

4.1 Soundness

In Chapter I (page 11), we said that soundness of the formal proof system implies the truth of the correctness conjecture for every assertive program (i.e., program-with-specification) that can be transformed to a mathematical statement that is true. By now we are referring to correct assertive programs (Definition 2.1, page 81) and true mathematical statements (Definition 2.2, page 81) as *valid*; hence, we have the following definition:

Definition 4.1 *A proof system for assertive programs is* sound *if and only if every program that (using only the rules of the proof system) can be transformed to a valid mathematical statement is, itself, valid.*

Recall from Section 3.2 that validity is *preserved* in the direction of a rule application if and only if the validity of the original implies the validity of the result. We have designed each of the rules to preserve validity in the program direction. Suppose we use the rules to transform an assertive program Prog into mathematical assertion H, and that H is valid (i.e., it is true in every model of its theory). Further

suppose that every rule preserves validity when applied in the program direction. Then we can apply the rules in the reverse order to transform H into Prog. The result of applying each rule will be a valid assertive program. In particular, Prog is valid. Hence, assuming every rule preserves validity in the program direction, every assertive program that (using only the rules of the proof system) can be transformed to a valid mathematical statement is necessarily valid. Therefore, we have just proven the following lemma:

Lemma 4.1 *If every rule presented in Chapter III preserves validity in the program direction, then the proof system composed of those rules is sound.*

The following theorem states one of the two important results discussed in the present chapter.

Theorem 1 (Soundness) *The proof system composed of the rules of Chapter III is sound.*

Theorem 1 follows from Lemmas 4.1 and 4.2.

Lemma 4.2 *Every rule presented in Chapter III preserves validity in the program direction.*

We define invalidity to be *preserved* in the direction of a rule application if and only if the invalidity of the original implies the invalidity of the result. Using the contrapositive, we obtain the fact that invalidity is preserved in the direction of a rule application if and only if the validity of the result implies the validity of the

original. Therefore, a rule preserves validity in a given direction if and only if it preserves invalidity in the opposite direction. Hence Lemma 4.2 follows immediately from Lemma 4.3.

Lemma 4.3 *Every rule presented in Chapter III preserves invalidity in the math direction.*

We prove Lemma 4.3 by establishing invalidity preservation in the math direction for each rule, one at a time, in Section 4.4.

4.2 Relative Completeness

Turning to the question of relative completeness, there are two problems that make the simplistic definition we made in Chapter I (page 11) not quite serviceable: when a "system is relatively complete, every program/specification pair satisfying the correctness conjecture is transformable to a mathematical assertion that is true." In the language of validity, this statement translates to the following possible definition: a proof system for assertive programs is relatively complete if and only if every valid program can be transformed (using only the rules of the proof system) to a valid mathematical statement.

This first problem is that we did not design the indexed method so that it could transform *all* assertive programs. It is designed specifically for transforming those programs that are a result of applying Krone's [25] procedure declaration rule. These programs begin with a **remember** statement and conclude with a **confirm** statement. Application of the bridge rule to such a program produces a program that conforms

to the syntax of top level code. All the other rules of the indexed method transform only top level code, producing only top level code. We, therefore, make the following definition.

Definition 4.2 *A* well-prepared assertive program *is an assertive program that conforms to the syntax of either (1) top level code or (2) $\mathcal{P}$ defined in the bridge rule (Figure 40).*

Then, provisionally, a proof system for well-prepared assertive programs is relatively complete if and only if every valid well-prepared program can be transformed (using only the rules of the proof system) to a valid mathematical statement.

The second problem is that the existence of internal assertions such as procedure specifications and loop invariants causes our proof system not to satisfy even this definition of relative completeness. For example, application of the **loop while** rule to the valid program shown in Figure 74 produces the invalid program shown in Figure 75.

First we argue that the program in Figure 74 is valid. It is valid if its interpretation takes every neutral environment to an environment that is not categorically false. Let env $= [\mathrm{a}, \mathrm{cs}, \mathrm{os}, \mathrm{ns}, \mathrm{se}, \mathrm{d}]$ be the environment that results from interpreting the first three statements of the program (**alter all**, **stow**(0), and **assume** (**true**) $\Rightarrow (\mathrm{x}_0 < 8)$) in an arbitrary neutral environment. Because interpreting each of those statements in a neutral environment never produces a categorically false environment, $\mathrm{a} \neq \mathrm{CF}$. The result of interpreting the entire program is a vacuously true environment (and, therefore, not categorically false) if $\mathrm{a} = \mathrm{VT}$; we, therefore, must assume that $\mathrm{a} = \mathrm{NL}$.

$C\backslash$ **alter all**
stow(0)
assume (**true**) $\Rightarrow$ ($x_0 < 8$)
whenever true do
loop
maintaining $x < 9$
while $x < 7$
stow(1)
Inc(x)
stow(2)
end loop
stow(3)
confirm $x_3 = 7$
end whenever

where $C \supseteq \{x : \text{Integer}\} \cup \left\{ \begin{array}{c} \textbf{procedure } \text{Inc(y: Integer)} \\ \textbf{ensures } y = \#y + 1 \end{array} \right\}$

Figure 74: A Valid Assertive Program

Then, due to the interpretation of the first three statements, especially the **assume** statement, we have $\mathrm{cs(x)} = \mathrm{ns(x,0)} < 8$. Let $\mathcal{F}$ be the interpretation of the loop in the program. Because env is a neutral environment,

$$\mathcal{F}(\mathrm{env}) = [\mathrm{NL}, \mathrm{cs}', \mathrm{os}, \mathrm{ns}', \mathrm{se}, \mathrm{d}] \tag{4.1}$$

$$\text{where } \mathrm{cs}'(\xi) = \begin{cases} \mathrm{cs}(\xi) & \text{if } \xi \neq \mathrm{x} \\ 7 & \text{else if } \mathrm{cs(x)} \leq 7 \\ \mathrm{cs(x)} & \text{otherwise} \end{cases} \tag{4.2}$$

$$\text{and } \mathrm{ns}'(\xi, i) = \begin{cases} \mathrm{ns}(\xi, i) & \text{if } i \notin \{1,2\} \vee 7 \leq \mathrm{cs(x)} \\ \mathrm{ns}(\xi, 0) & \text{else if } \xi \neq \mathrm{x} \\ 6 & \text{else if } i = 1 \\ 7 & \text{otherwise} \end{cases} \tag{4.3}$$

Because cs(x) is an integer and $\mathrm{cs(x)} < 8$, we have $\mathrm{cs(x)} \leq 7$, and, by equation 4.2, $\mathrm{cs}'(\mathrm{x}) = 7$. Hence, the interpretation of the sequence **stow**(3) **confirm** $\mathrm{x}_3 = 7$ in the environment $\mathcal{F}(\mathrm{env})$ necessarily yields a neutral environment. Therefore, the program in Figure 74 is valid.

The only rule applicable to the program in Figure 74 is the **loop while** rule. Figure 75 shows the result of this rule application. We can show the program in Figure 75 to be invalid. Let $\mathrm{env} = [\mathrm{NL}, \mathrm{cs}, \mathrm{os}, \mathrm{ns}, \mathrm{se}, \mathrm{d}]$ be a neutral environment such that the setup se has at least three states in its sequence that have the following properties. The first state of se maps x to the value 7, and the third state of se maps x to the value 8. The second state's value does not matter. Then, after execution of **stow**(0) in Figure 75, $\mathrm{ns(x,0)} = 7$ (i.e., $\mathrm{x}_0 = 7$). Therefore, the assert status is still NL after execution of **stow**(1). The statement sequence inside the "**whenever** (**true**) $\wedge$ $(\mathrm{x}_0 < 7)$ **do**" statement is not executed, so the assert status is still NL after execution of **stow**(3). Due to the setup, we now have $\mathrm{ns(x,3)} = 8$ (i.e., $\mathrm{x}_3 = 8$). Hence, the

$C\backslash$ **alter all**
stow(0)
assume (**true**) $\Rightarrow$ ($x_0 < 8$)
confirm (**true**) $\Rightarrow$ ($x_0 < 9$)
alter all
stow(1)
whenever (**true**) $\wedge$ ($x_0 < 7$) **do**
  assume ($x_1 < 7$) $\wedge$ ($x_1 < 9$)
  Inc(x)
  stow(2)
  confirm $x_2 < 9$
end whenever
alter all
stow(3)
whenever true do
  assume ($\neg$($x_3 < 7$)) $\wedge$ ($x_3 < 9$)
  confirm $x_3 = 7$
end whenever

Figure 75: An Invalid Assertive Program

assert status is still NL after execution of "**assume** $(\neg(x_3 < 7)) \wedge (x_3 < 9)$". We still have $x_3 = 8$, so execution of "**confirm** $x_3 = 7$" produces a categorically false environment. We have shown the program in Figure 75 to be invalid.

The proof rules of Chapter III can be applied in the math direction to rewrite the program in Figure 75 to a mathematical assertion. This assertion will be invalid (i.e., false) because, as we will learn in Section 4.4, all the rules preserve invalidity in the math direction. Once rewriting produces an invalid program, the programs produced by all further rewriting are also invalid. We have demonstrated that our rules cannot always preserve validity in the math direction. The valid program of Figure 74 cannot be transformed (using only the rules of the proof system) to a valid mathematical statement. The traditional proof systems do not differ from ours in this respect.

Not all is lost, however. Although the **maintaining** clause's claim about the loop in Figure 74 is true, it is not true enough! The loop itself accomplishes more than what its stated invariant advertises. Had the loop invariant been $x < 8$ rather than $x < 9$, the program would have been transformed by the rules to a true mathematical statement. Perhaps for every well-prepared valid assertive program Prog there exists a related valid program Prog$'$, which differs from Prog at most in the internal assertions (loop invariants and procedure specifications), such that Prog$'$ can be transformed (using only the rules of the proof system) to a valid mathematical statement. Such a proof system would still be quite useful. The authors of a valid assertive program could discover loop invariants and procedure specifications that are "tight" enough

so that their program with these modified assertions could be shown to be valid—by using the rules to transform the modified program into a valid mathematical statement. This characterization is our definition of relative completeness:

Definition 4.3 *A proof system for well-prepared assertive programs is* relatively complete *if and only if for every well-prepared valid assertive program Prog there exists a related valid program Prog′, which differs from Prog at most in the internal assertions (loop invariants and procedure specifications), such that Prog′ can be transformed (using only the rules of the proof system) to a valid mathematical statement.*

The **loop while** rule is sort of an exception among our proof rules. Nearly all the other rules do, in fact, preserve validity in the math direction. The other exception is the procedure call rule, which can fail to preserve validity in the math direction when the procedure's postcondition is weaker than what Cook defines to be "the *post relation corresponding to*" the procedure's precondition and its body [4, p. 85]. By use of the minimum fixed-point operation, the procedure's body defines its procedure-function. This procedure-function plays an important role in the validity of the assertive program. When the procedure call rule removes the call and replaces it with an **assume** statement constructed from the procedure's weaker postcondition, the resulting program may be invalid.

A procedure that is declared and defined in the program is an *internal* procedure; others are *external*. The body of an internal procedure is part of the assertive program; the body of an external procedure is not. Because researchers have not yet produced a satisfactory relational semantics for assertive programs (see Section 5.3.1),

$C\backslash$ **alter all**
stow(0)
assume true
whenever true do
Make_1_Or_Minus_1(x)
stow(1)
Make_1_Or_Minus_1(y)
stow(2)
confirm $x_2 + y_2 \neq 0$
end whenever

where $C \supseteq \{x, y : \text{Integer}\} \cup \left\{ \begin{array}{l} \textbf{procedure}\ \text{Make\_1\_Or\_Minus\_1}(z\text{: Integer}) \\ \quad \textbf{ensures}\ (z = 1) \vee (z = -1) \end{array} \right\}$

Figure 76: An Assertive Program That Is Valid According to Functional Semantics

we have adopted a well-understood functional semantics. This choice causes a technical problem for relative completeness with respect to external procedures. The indexed method's rules, like Krone's rules, are designed so that external procedures can have relational behavior. We do not want programs to be considered correct whose validity depends crucially on the external procedures' adhering to functional semantics. For example, the program in Figure 76, given our functional semantics, and assuming that Make_1_Or_Minus_1 is an external procedure, is valid. Because the semantics of Make_1_Or_Minus_1 is functional, it either sets both x and y to 1, or it sets both x and y to -1. In either case $x + y \neq 0$. However, this program would not be valid if the semantics of Make_1_Or_Minus_1 were relational; the result of the procedure call could be 1 for x and -1 for y, making $x + y = 0$.

We want our proof rules to be sound whether the semantics is functional or relational; consequently, the indexed method, like Krone's method, is not relatively complete if the external procedures have functional semantics. Both methods are designed to be relatively complete if the external procedures have relational semantics. A careful proof that this is so will have to wait for the development of a satisfactory relational semantics. These technical problems do not arise if every external procedure called is functionally specified. Therefore, at present, we are in a position to provide a careful proof that the indexed method is relatively complete if all called procedures are either internal procedures or functionally specified external procedures.

Cook defines the assertion language to be *expressive* if, for every precondition and statement sequence, there is a formula of the language that expresses the post relation corresponding to the precondition and statement sequence [4, p. 86]. He also shows that if the assertion language is expressive, then, for every **while** loop, there is a formula of the language that expresses the tightest loop invariant—that fully expresses the behavior of the loop [4, p. 87]. We claim, then, that given a valid assertive program Prog, we can construct from it a related valid program Prog′ that satisfies the conditions of Definition 4.3 above by (1) replacing the postcondition of each internal procedure with the post relation corresponding to the procedure's precondition and its body and (2) replacing each loop invariant with a tightest loop invariant. We will prove the following lemma in Section 4.5.

Lemma 4.4 (Related Valid Program Differing in Assertions Only) *Assuming the assertion language is expressive, if a well-prepared assertive program Prog is valid, then there exists a well-prepared assertive program Prog′ that can be obtained from Prog by (1) replacing the postcondition of each internal procedure with the post relation corresponding to the procedure's precondition and its body and (2) replacing each loop invariant with a tightest loop invariant; furthermore, Prog′ is also valid.*

Relative completeness will follow from Lemma 4.4 if we can show that every such Prog′ can be transformed (using only the rules of the proof system) to a valid mathematical statement. We can ignore the matter of validity for the moment and wonder whether every well-prepared assertive program can be transformed to a mathematical assertion. We answer this question in the affirmative by proving the following lemma in Section 4.5.

Lemma 4.5 *The process of rewriting any well-prepared assertive program in the math direction using the rules of Chapter III always terminates with a mathematical assertion.*

Returning to the matter of validity, we will prove each of the following lemmas in Section 4.5.

Lemma 4.6 *An application of the procedure call rule that involves a call either to (1) a functionally specified external procedure or (2) to an internal procedure, whose postcondition is the post relation corresponding to the procedure's precondition and its body, preserves invalidity in the program direction.*

Lemma 4.7 *An application of the* **loop while** *rule that involves a loop whose tightest loop invariant is expressed in the* **maintaining** *clause preserves invalidity in the program direction.*

Lemma 4.8 *Excluding the procedure call rule and the* **loop while** *rule (which are treated according to Lemmas 4.6 and 4.7), application of each rule of Chapter III preserves invalidity in the program direction.*

Recalling that preserving invalidity in the program direction is equivalent to preserving validity in the math direction, the preceding lemmas establish this one:

Lemma 4.9 *Let Prog′ be a well-prepared valid assertive program that (1) satisfies the conditions of Lemma 4.4 and (2) contains no calls to relationally specified external procedures. Then the process of rewriting Prog′ in the math direction using the rules of Chapter III always terminates with a valid mathematical assertion.*

Theorem 2 follows from Definition 4.3, Lemma 4.4, and Lemma 4.9.

Theorem 2 (Relative Completeness) *If the assertion language is expressive, then the proof system composed of the rules of Chapter III is relatively complete over the set of programs that contain no calls to relationally specified external procedures.*

Note that relative completeness is contingent upon the expressiveness of the assertion language.

The rest of this chapter is devoted to proving the lemmas that establish Theorems 1 and 2. Along the way, we need to state and prove auxiliary lemmas that are

important in the proofs of the main lemmas. Because Theorems 1 and 2 form the critical heart of our thesis, we want to be very careful in establishing their truth. By being this careful, we leave very little to the imagination, so we take this opportunity to apologize for the repetitive nature of the proof cases.

4.3 General Auxiliary Lemmas

In this section, we present lemmas that are helpful for proving the lemmas involved in both soundness and relative completeness. We begin with the fact that the proof rules are well-formed.

4.3.1 Proof Rules Are Well-Formed

Recall from Section 3.2 that each of the rules that operates within phase 1, 2, or 3 is defined as being applicable, in the math direction, to some top level code, say $top\_lev\_code_{\mathcal{P}}$, only if there is some instantiation, say Inst, such that $top\_lev\_code_{\mathcal{P}} = \text{Inst}(\mathcal{P})$ and both $\text{Inst}(\mathcal{P})$ and $\text{Inst}(\mathcal{M})$ are syntactically correct top level code. We are thus assured, by definition, that the result of any rule application is (syntactically correct) top level code. We would do well, however, to reserve judgment on the usefulness of the rules until we were at least convinced that, whenever an instantiation, say Inst, of $\mathcal{P}$ (i.e., $\text{Inst}(\mathcal{P})$) is top level code, $\text{Inst}(\mathcal{M})$ is *also* top level code. Moreover, it is important to know that there exist many instantiations of $\mathcal{P}$ that produce top level code. Concerning application of the rules in the opposite direction—the program direction—we would want to be convinced of the same facts where the roles of $\mathcal{P}$ and

$\mathcal{M}$ are interchanged. This collection of facts gives us confidence that the rules are generally applicable, are well-formed. These facts are stated in the following lemmas.

Lemma 4.10 *For each of the rules that operates within phase 1, 2, or 3, whenever an instantiation, say Inst, of $\mathcal{P}$ (i.e., Inst($\mathcal{P}$)) is top level code, Inst($\mathcal{M}$) is also top level code.*

Lemma 4.11 *For each of the rules that operates within phase 1, 2, or 3, whenever an instantiation, say Inst, of $\mathcal{M}$ (i.e., Inst($\mathcal{M}$)) is top level code, Inst($\mathcal{P}$) is also top level code.*

Lemma 4.12 *For each of the rules that operates within phase 1, 2, or 3, there exist infinitely many instantiations of $\mathcal{P}$ that produce top level code, and there exist infinitely many instantiations of $\mathcal{M}$ that produce top level code.*

Complete proofs of these lemmas require examining each of the eleven rules that operates within phase 1, 2, or 3. We provide here a sketch showing how to perform this examination, leaving the complete examination to the skeptical reader. Note that top level code is rewritten from a (possibly empty) finite sequence of nonterminal symbols that occur in a repeated waltz rhythm of three: stow section, then assume-confirm sequence, followed by a guarded block (see Figure 32). This observation will be used frequently. For example, we can show that if there is at least one instantiation of a rule's $\mathcal{P}$ that produces top level code, there are infinitely many such instantiations. Each rule involves either *prec_top_lev_code* or *top_lev_code*. These symbols can be instantiated to any top level code. Now top level code can simply be any arbitrary

number of **assume** statements because it can be rewritten from exactly one triple of nonterminals; the stow section and guarded block can be empty; and the assume-confirm sequence can be any arbitrary number of **assume** statements. Because there is no fixed limit on the number of **assume** statements there may be in such a sequence, there are infinitely many ways to instantiate either *prec_top_lev_code* or *top_lev_code*.

As a prototypical example, we argue that there is at least one way to instantiate $\mathcal{P}$ of the rule for procedure call (see Figure 46) to top level code. This can be done if the part that lies strictly between *prec_top_lev_code* and *fol_top_lev_code* can be instantiated to top level code. Superficially, that is easy because **alter all stow**(i) is a stow section; $ACseq_0$ is an assume-confirm sequence; and the **whenever** statement is a guarded block. We must show additionally that the part within the **whenever** statement can be rewritten from ⟨cd_suffix⟩—that it is a code suffix. Recall from Section 3.1 that a code suffix is a portion of internal code, and that internal code is a pattern of nonterminal symbols, cycling repeatedly through ⟨stow_sec⟩, ⟨ACseq⟩, and ⟨op_stmt⟩, beginning with ⟨stow_sec⟩ and concluding with ⟨ACseq⟩. A code suffix begins and ends with ⟨ACseq⟩. Furthermore, a code suffix must be rewritten according to the restricted grammar productions of Figure 35. Stow sections consist of one **stow** statement, and assume-confirm sequences contain zero or more **confirm** statements.

The code suffix within our **whenever** statement begins with an empty assume-confirm sequence. The operational statement (⟨op_stmt⟩) is a procedure call, and the stow section is **stow**(j). This completes a whole cycle; so, if we instantiate *cd_suffix*

to any code suffix, we have instantiated the part within the **whenever** statement to a code suffix. Finally, we observe that an instantiation meeting the above conditions can be found that also satisfies the non-context-free syntactic restrictions by organizing the indexes of **stow** statements to be everywhere increasing, and by arranging that references to index i occur only after **stow**(i). Therefore, there is at least one way to instantiate $\mathcal{P}$ to top level code. Then, by the result of the previous paragraph, there are infinitely many ways to instantiate $\mathcal{P}$ to top level code. We have justified Lemma 4.12 for $\mathcal{P}$ of the procedure call rule.

Does Lemma 4.10 hold for the case of the procedure call rule? Let Inst be an instantiation of $\mathcal{P}$, and suppose Inst($\mathcal{P}$) is top level code. We are to show that Inst($\mathcal{M}$) is top level code. The statement "**confirm** (Br_Cd) $\Rightarrow$ (pre[x $\rightsquigarrow$ ac_i, y $\rightsquigarrow$ ad_i, z $\rightsquigarrow$ z_i])" extends the assume-confirm sequence started by $ACseq_0$. Because Inst($\mathcal{P}$) is top level code, if an indexed variable of Br_Cd has index h, then **stow**(h) occurs earlier in the program than this new **confirm** statement. All variables in pre[x $\rightsquigarrow$ ac_i, y $\rightsquigarrow$ ad_i, z $\rightsquigarrow$ z_i] are indexed by i, and **stow**(i) precedes the **confirm** statement. Therefore, this **confirm** statement satisfies non-context-free syntactic restriction number 2 (see p. 93). This **confirm** statement is followed immediately by an empty guarded block. A stow section, **alter all stow**(j), comes next, and is succeeded by an empty assume-confirm sequence. The guarded block that follows is a **whenever** statement.

We must be sure that the statement sequence within this **whenever** statement is a code suffix. It is; we can take the initial **assume** statement to be the first statement of the assume-confirm sequence with which *cd_suffix* begins. The variables of this **assume** statement are all indexed by either i or j. So this **assume** statement satisfies non-context-free syntactic restriction number 2 because the statements **stow**(i) and **stow**(j) appear earlier in the program. As the syntax requires, the **whenever** statement is followed by some top level code: *fol_top_lev_code*. Therefore, Lemma 4.10 holds for the case of the procedure call rule.

A similar argument shows Lemma 4.11 to hold for the case of the procedure call rule. These lemmas can be shown to hold for the cases of each of the other ten rules that operate in phases 1, 2, and 3.

We come now to the question whether the rule that governs Step 0 of Figure 31—the bridge rule—is well-formed. The following lemma addresses the most interesting aspect of this question.

Lemma 4.13 *Let $\mathcal{P}$ and $\mathcal{M}$ be as they are defined in the bridge rule. Let Inst be an instantiation of $\mathcal{P}$ such that* p_body *is a procedure body (i.e., can be rewritten from* $\langle$*p_body*$\rangle$*). Then there exists an instantiation Inst$'$ such that Inst$'(\mathcal{P})$ = Inst$(\mathcal{P})$ and the indexes produced by the relation Stows_added can be chosen so that Inst$'(\mathcal{M})$ is top level code.*

Proof. Given this lemma's hypotheses, we show that there exists an instantiation Inst$'$ such that Inst$'(\mathcal{P})$ = Inst$(\mathcal{P})$ and the indexes produced by the relation Stows_added can be chosen so that Inst$'(\mathcal{M})$ is top level code. The stow section it (Inst$'(\mathcal{M})$) begins

with is **alter all stow**(i). The assume-confirm sequence is empty, and it concludes with a non-empty guarded block, a **whenever** statement. We must be sure that the statement sequence within this **whenever** statement is a code suffix. We can take the **assume** statement to be the first statement of the assume-confirm sequence with which Stows_added(*p_body*) begins. Stows_added(*p_body*) needs to be—and can be taken to be—a code kernal (⟨cd_kern⟩). Thus, it concludes with an operational statement (⟨op_stmt⟩). Therefore, it can be followed by a stow section (**stow**(j)) and an assume-confirm sequence (a **confirm** statement) to complete the code suffix. Note that non-context-free syntactic restriction number 2 is satisfied because (1) all the variables in the **assume** statement are indexed by i and **stow**(i) appears earlier in the program and (2) all the variables in the **confirm** statement are indexed by either i or j and both **stow**(i) and **stow**(j) appear earlier in the program. The indexes produced by the relation Stows_added need to be chosen so that non-context-free syntactic restriction number 1 is satisfied. This can be done if j and i are such that $j - i$ is at least as large as the number of operational statements in *p_body*. So we let $\text{Inst}' = \text{Inst}$ except for the instantiations of i and j. We assure that $\text{Inst}'(j) - \text{Inst}'(i)$ is at least as large as the number of operational statements in *p_body*. Finally, we chose the indexes produced by the relation Stows_added to be an increasing sequence of integers strictly between $\text{Inst}'(i)$ and $\text{Inst}'(j)$. □

Finally, we consider the question whether the rule that governs Step 4 of Figure 31—the rule of inference bridging predicate logic and the indexed method—is well-formed. By definition, H is a syntactically correct assertion in context C if and

only if **confirm** H is a syntactically correct **confirm** statement in context C. The one remaining question is whether **confirm** H is syntactically correct top level code. It is. It begins with an empty stow section, which is followed by an assume-confirm sequence that contains exactly one **confirm** statement. It concludes with an empty guarded block. We are now justified in believing the rules of the indexed method to be well-formed.

4.3.2 Factoring Statement Sequences

The applicability of the proof rules is defined with respect to the syntax of top level code. However, the meaning of any assertive program—in particular, the meaning of any top level code—is defined with respect to the syntax of Section 2.1. According to this syntax, each assertive program—including top level code—is simply a sequence of statements. Some of these statements contain other statement sequences, and so on. It is this simpler syntax that is relevant to arguments about preserving invalidity in the math or program directions because these arguments are based on the meanings—the semantics—of the original and resulting assertive programs. In these arguments, we will be analyzing the semantics of assertive programs as concatenations of two or more statement sequences. Thus, it is important that we know, as the following lemma states, that the interpretation of the concatenation of two sequences of statements is the interpretation of the second sequence in the environment resulting from interpreting the first sequence.

Lemma 4.14 (Factoring)

$$\mathcal{I}(SS1\ SS2)(env) = \mathcal{I}(SS2)(\mathcal{I}(SS1)(env)) \tag{4.4}$$

This lemma is easily established from equation 2.23 by induction on the length of SS1.

4.3.3 Shallow Lemmas

The lemmas in this section are shallow in that their proofs are easy; they are not far removed from the definitions. They are, however, convenient for the proofs in Sections 4.4 and 4.5. The first of these lemmas follows easily from equation 2.26 by induction on the length of SS.

Lemma 4.15 (CF Is a Stuck State)

$$AE(env) = CF \Rightarrow AE(\mathcal{I}(SS)(env)) = CF \tag{4.5}$$

Similarly, Lemma 4.16 follows from equation 2.25 by induction on the length of SS.

Lemma 4.16 (VT Is a Stuck State)

$$AE(env) = VT \Rightarrow AE(\mathcal{I}(SS)(env)) = VT \tag{4.6}$$

Suppose S1 is a **loop while** statement and env_{NL} is some neutral environment. Then it is possible for the function $\mathcal{I}(\text{S1})$, which is obtained via the minimum fixed point operator, to be undefined at env_{NL}. Then, of course, the composition of any function f with $\mathcal{I}(\text{S1})$ is also undefined at env_{NL}. That is to say, if $\mathcal{I}(\text{S1})(\text{env}_{\text{NL}})$ is undefined, then $f(\mathcal{I}(\text{S1})(\text{env}_{\text{NL}}))$ is undefined. Lemma 4.17 depends on this fact.

Lemma 4.17 (Never from Undefined to CF)

$$AE(\mathcal{I}(SS2)(\mathcal{I}(SS1)(env))) = CF \Rightarrow \mathcal{I}(SS1)(env) \text{ is defined.} \tag{4.7}$$

Proof. The assertion that AE($\mathcal{I}$(SS2)($\mathcal{I}$(SS1)(env))) = CF implies that $\mathcal{I}$(SS2)($\mathcal{I}$(SS1)(env)) is defined. Suppose, by way of contradiction, that $\mathcal{I}$(SS1)(env) is undefined. Then, because it is a composition of functions, $\mathcal{I}$(SS2)($\mathcal{I}$(SS1)(env)) would be undefined. □

Just as a later environment being categorically false means that all earlier environments must have been defined (Lemma 4.17), we state in Lemma 4.18 that a later environment being categorically false means that no earlier environment was vacuously true.

Lemma 4.18 (Never from VT to CF)

$$AE(\mathcal{I}(SS)(env)) = CF \Rightarrow AE(env) \neq VT \tag{4.8}$$

Proof. Suppose AE($\mathcal{I}$(SS)(env)) = CF. Then AE($\mathcal{I}$(SS)(env)) ≠ VT. By the contrapositive of Lemma 4.16, we have AE(env) ≠ VT, the desired conclusion. □

Lemma 4.19 states that if a later environment is categorically false but the earlier one was not, then the earlier environment was neutral.

Lemma 4.19 (From Non-CF to CF Means From NL to CF)

$$(AE(env) \neq CF \wedge AE(\mathcal{I}(SS)(env)) = CF) \Rightarrow AE(env) = NL \tag{4.9}$$

Proof. Suppose AE($\mathcal{I}$(SS)(env)) = CF. Then, from Lemma 4.17, we have that env is defined. From Lemma 4.18, we have that AE(env) ≠ VT. From the hypothesis we

have that AE(env) $\neq$ CF. Therefore, by elimination, we must have AE(env) = NL.

□

4.3.4 Deeper Lemmas

The proofs of the lemmas in this section are a little more difficult in that most of them involve mathematical induction. Lemma 4.20 tells us that later setups are suffixes of earlier setups.

Lemma 4.20 (Effect on Setup) *If $SPE(\mathcal{I}(SS)(env)) = se$, then there exists a setup se_0 such that $SPE(env) = se_0 \circ se$, where $\circ$ is the symbol for sequence concatenation.*

Proof. Checking the semantic definitions, we observe that executions of all statements other than the **alter all** statement leave the setup unchanged. Execution of the **alter all** statement shortens any nonempty setup by removing the front state. Our proof is by induction on the length of SS. The base case is a length of zero; i.e., SS = ε, the empty sequence of statements. In this case, the lemma is satisfied by choosing se_0 to be the empty sequence of states, Λ. For the induction step, suppose $0 < n$ and that the lemma holds for all SSs whose length is less than n. We suppose that the length of the statement sequence S1 SS2 is n, and prove the lemma for S1 SS2. So we suppose SPE($\mathcal{I}$(S1 SS2)(env)) = se. Then, by equation 2.23, we have SPE($\mathcal{I}$(SS2)($\mathcal{I}$(S1)(env))) = se. By the induction hypothesis, there exists se_1 such that SPE($\mathcal{I}$(S1)(env)) = $se_1 \circ$ se. If S1 is not an **alter all** statement or AE($\mathcal{I}$(S1)(env)) $\neq$ NL, then se_0 satisfies the lemma if we take it to equal se_1. If S1 is

an **alter all** statement and $\mathrm{AE}(\mathcal{I}(\mathrm{S1})(\mathrm{env})) = \mathrm{NL}$, then se_0 satisfies the lemma if we take it to equal $\langle \mathrm{st} \rangle \circ \mathrm{se}_1$, where $\langle \mathrm{st} \rangle$ is a one-element sequence of some state st. □

The next lemma states that subsequent interpretations of Br_Cd are identical to its initial interpretation.

Lemma 4.21 (Interpretations of Br_Cd Are Stable) *Let $\mathcal{R}$ be an intermediate result obtained by applying the proof rules in the math direction to a programmer-written program. Furthermore, let $\mathcal{R}$ have the following form:*

$$\begin{array}{lll} \mathcal{R} & \stackrel{\text{def}}{=} & C\backslash \quad \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textbf{whenever } \mathrm{Br\_Cd} \textbf{ do} \\ \quad \textit{guarded\_code} \\ \textbf{end whenever} \\ \textit{interm\_top\_lev\_code} \\ \textit{fol\_top\_lev\_code} \end{array} \end{array} \tag{4.10}$$

Let env be some environment. Then

$$\mathcal{I}(\mathrm{Br\_Cd})(\mathcal{I}\left(\begin{array}{l} \textbf{whenever } \mathrm{Br\_Cd} \textbf{ do} \\ \quad \textit{guarded\_code} \\ \textbf{end whenever} \\ \textit{interm\_top\_lev\_code} \end{array}\right)(\mathit{env})) = \mathcal{I}(\mathrm{Br\_Cd})(\mathit{env}) \tag{4.11}$$

Proof. There are six non-compound statements in the language: procedure call, **confirm**, **assume**, **remember**, **stow**, and **alter all**. Except for **stow**, interpretation of each non-compound statement leaves the index-state unchanged. By the syntactic restriction that indices of **stow** statements are everywhere increasing and Lemma 4.22, which follows, there is an index i such that (1) if $\mathbf{stow}(h)$ is a statement of *guarded_code* or *interm_top_lev_code*, then $i < h$ and (2) every free variable of Br_Cd is an indexed variable, and every indexed variable ξ_n appearing in Br_Cd is such that $n < i$. Note that interpretation of $\mathbf{stow}(j)$ changes the index-state only

at j. Therefore, for all n such that $n < i$, if S1 is a statement of *guarded_code* or *interm_top_lev_code*, then $\mathrm{ISE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}))(n) = \mathrm{ISE}(\mathrm{env})(n)$. Induction on program structure can be used to show from this fact that, for $n < i$,

$$\mathrm{ISE}(\mathcal{I}\left(\begin{array}{l}\textbf{whenever } \mathrm{Br\_Cd} \textbf{ do}\\ \quad \textit{guarded\_code}\\ \textbf{end whenever}\\ \textit{interm\_top\_lev\_code}\end{array}\right)(\mathrm{env}))(n) = \mathrm{ISE}(\mathrm{env})(n) \tag{4.12}$$

This lemma's conclusion follows immediately. □

Lemma 4.22 (Br_Cds Refer Only to Variables with Earlier Indices) *Let $\mathcal{R}$ be an intermediate result obtained by applying the proof rules in the math direction to a programmer-written program. Furthermore, let $\mathcal{R}$ have the following form:*

$$\begin{array}{rcll}\mathcal{R} & \stackrel{\mathrm{def}}{=} & C\backslash & \textit{prec\_top\_lev\_code}\\ & & & \textbf{alter all}\\ & & & \textbf{stow}(i)\\ & & & \mathit{ACseq}_0\\ & & & \textbf{whenever } \mathrm{Br\_Cd} \textbf{ do}\\ & & & \quad \textit{guarded\_code}\\ & & & \textbf{end whenever}\\ & & & \textit{fol\_top\_lev\_code}\end{array} \tag{4.13}$$

Then every free variable of Br_Cd is an indexed variable, and every indexed variable ξ_h appearing in Br_Cd is such that $h < i$.

Proof. We use induction on the number of rule applications to show that every intermediate result obtained by applying the proof rules in the math direction to a programmer-written program satisfies the lemma's conditions. The base case is when exactly one rule is applied. That would have to be the bridge rule. The result of applying the bridge rule has exactly one **whenever** statement. The Br_Cd of this

statement is "**true**"; it contains no free variables. So, the conditions of the lemma are satisfied vacuously. Therefore, any result of applying the bridge rule satisfies the lemma's conditions.

The induction step is to assume that an intermediate result satisfies the lemma's conditions and, then, to show the result of applying each rule in the math direction satisfies the lemma's conditions. We consider each rule in turn.

By the induction hypothesis, $\mathcal{P}$ of the rule for **assume** satisfies the lemma's conditions. The **whenever** statement of $\mathcal{M}$ is in the same position as the one in $\mathcal{P}$ from which it was derived; it follows a sequence of **assume**s and **confirm**s that, in turn, follows an **stow**(i) statement. Furthermore, the **whenever** statement of $\mathcal{M}$ has the same branch condition, "Br_Cd", as the one in $\mathcal{P}$. Hence, $\mathcal{M}$ of the rule for **assume** satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for procedure call satisfies the lemma's conditions. The **whenever** statement of $\mathcal{M}$ follows an empty sequence of **assume**s and **confirm**s that, in turn, follows an **stow**(j) statement. The **whenever** statement of $\mathcal{M}$ has the same branch condition, "Br_Cd", as the one in $\mathcal{P}$. Furthermore, by the syntactic restriction that the indices in **stow** statements are everywhere increasing, $i < j$. By the transitivity of the less-than relation, we have that every indexed variable ξ_h appearing in Br_Cd is such that $h < j$. Hence, $\mathcal{M}$ of the rule for procedure call satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for selection in the absence of an **else** clause satisfies the lemma's conditions. $\mathcal{M}$ is derived from $\mathcal{P}$ by replacing one

whenever statement with eight statements. Let WE1 stand for the first, and WE2 for the second, of the **whenever** statements. Because $\mathcal{P}$ satisfies the lemma's conditions, we have that every indexed variable ξ_h appearing in Br_Cd is such that $h < i$. We also have $i < j$; hence, $h < j$. Boolean program expressions, in particular *b_p_e*, do not contain indexed variables. So all the variables of $\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i]$ are indexed by i. Therefore, every indexed variable ξ_g appearing in "$(\mathrm{Br\_Cd}) \wedge (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i]))(\mathrm{env}_1)$" is such that $g < j$. Hence, WE1 satisfies the lemma's conditions. Because $i < n$, we have that every indexed variable ξ_h appearing in Br_Cd is such that $h < n$. Hence, WE2 satisfies the lemma's conditions. Therefore, $\mathcal{M}$ of the rule for selection in the absence of an **else** clause satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for selection in the presence of an **else** clause satisfies the lemma's conditions. $\mathcal{M}$ is derived from $\mathcal{P}$ by replacing one **whenever** statement with eleven statements. Let WE1 stand for the first, WE2 for the second, and WE3 for the third, of the **whenever** statements. Because $\mathcal{P}$ satisfies the lemma's conditions, we have that every indexed variable ξ_h appearing in Br_Cd is such that $h < i$. We also have $i < j$; hence, $h < j$. Boolean program expressions, in particular *b_p_e*, do not contain indexed variables. So all the variables of $\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i]$ are indexed by i. Therefore, every indexed variable ξ_g appearing in "$(\mathrm{Br\_Cd}) \wedge (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i]))(\mathrm{env}_1)$" is such that $g < j$. Hence, WE1 satisfies the lemma's conditions. We have $i < l$; hence, $h < l$. Therefore, every indexed variable ξ_g appearing in "$(\mathrm{Br\_Cd}) \wedge \neg(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i]))(\mathrm{env}_1)$" is such that $g < l$. Hence, WE2 satisfies the lemma's conditions. Because $i < n$, we have

that every indexed variable ξ_h appearing in Br_Cd is such that $h < n$. Hence, WE2 satisfies the lemma's conditions. Therefore, $\mathcal{M}$ of the rule for selection in the presence of an **else** clause satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the **loop while** rule satisfies the lemma's conditions. $\mathcal{M}$ is derived from $\mathcal{P}$ by replacing one **whenever** statement with seven statements. Let WE1 stand for the first, and WE2 for the second, of the **whenever** statements. Because $\mathcal{P}$ satisfies the lemma's conditions, we have that every indexed variable ξ_h appearing in Br_Cd is such that $h < i$. We also have $i < j$; hence, $h < j$. Boolean program expressions, in particular *b_p_e*, do not contain indexed variables. So all the variables of $\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i]$ are indexed by i. Therefore, every indexed variable ξ_g appearing in "$(\mathrm{Br\_Cd}) \wedge (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i]))(\mathrm{env}_1)$" is such that $g < j$. Hence, WE1 satisfies the lemma's conditions. Because $i < l$, we have that every indexed variable ξ_h appearing in Br_Cd is such that $h < l$. Hence, WE2 satisfies the lemma's conditions. Therefore, $\mathcal{M}$ of the **loop while** rule satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for **confirm** satisfies the lemma's conditions. The **whenever** statement of $\mathcal{M}$ is in the same position as the one in $\mathcal{P}$ from which it was derived; it follows a sequence of **assume**s and **confirm**s that, in turn, follows an **stow**(i) statement. Furthermore, the **whenever** statement of $\mathcal{M}$ has the same branch condition, "Br_Cd", as the one in $\mathcal{P}$. Hence, $\mathcal{M}$ of the rule for **confirm** satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for empty guarded blocks satisfies the lemma's conditions. Every statement of $\mathcal{M}$ is a statement of $\mathcal{P}$. Hence, $\mathcal{M}$ of the rule for empty guarded blocks satisfies the lemma's conditions.

Due to its additional syntactic restriction, neither $\mathcal{P}$ nor $\mathcal{M}$ of the rule for **alter all** contains any **whenever** statements. Hence, $\mathcal{M}$ of the rule for **alter all** (vacuously) satisfies the lemma's conditions.

One argument serves for the remaining three rules: the rule for consecutive **assume** statements, the **assume-confirm** rule, and the rule for consecutive **confirm** statements. By the induction hypothesis, the rule's $\mathcal{P}$ satisfies the lemma's conditions. Every **whenever** statement of $\mathcal{M}$ is a statement of $\mathcal{P}$. Hence, the rule's $\mathcal{M}$ satisfies the lemma's conditions.

The induction step is now complete. Hence the proof of this lemma (4.22) is finished. □

Consider compound statements, such as selection or iteration statements, that may appear within **whenever** statements. These compound statements contain **stow**(h) statements. When a compound statement is still intact within a **whenever** statement, it has yet to be transformed by the application (in the math direction) of a proof rule. Therefore, no variable that is subscripted by h appears anywhere in the program. We state this fact in Lemma 4.23.

Lemma 4.23 (Internal Indices Are Sealed) *Let $\mathcal{R}$ be an intermediate result obtained by applying the proof rules in the math direction to a programmer-written*

program. Furthermore, let $\mathcal{R}$ *have the following form:*

$$\mathcal{R} \stackrel{\text{def}}{=} C\backslash \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textbf{whenever } \text{Br\_Cd } \textbf{do} \\ \quad \textit{guarded\_code} \\ \textbf{end whenever} \\ \textit{fol\_top\_lev\_code} \end{array} \tag{4.14}$$

Let CST be any compound statement of guarded_code. Let CI be the set of indices h that appear in **stow**(h) *statements within CST. Let S1 be any statement (including statements within compound statements) of prec_top_lev_code, guarded_code, cd_suffix, or fol_top_lev_code. Then, if indexed variable* ξ_h *occurs in S1,* $h \notin CI$.

Proof. We use induction on the number of rule applications to show that every intermediate result obtained by applying the proof rules in the math direction to a programmer-written program satisfies the lemma's conditions. The base case is when exactly one rule is applied. That would have to be the bridge rule. Because *p_body* is programmer-written, it contains no subscripted variables. By the definition of the Stows_added relation, Stows_added(*p_body*) contains no subscripted variables. Hence, the result of applying the bridge rule, $\mathcal{M}$, has subscripted variables only in two statements. These variables are subscripted only by i and j. By the syntactic restriction that the indices in **stow** statements are everywhere increasing, no compound statement of Stows_added(*p_body*) can contain either a **stow**(i) statement or a **stow**(j) statement. Therefore, any result of applying the bridge rule satisfies the lemma's conditions.

The induction step is to assume that an intermediate result satisfies the lemma's conditions and, then, to show the result of applying each rule in the math direction satisfies the lemma's conditions. We consider each rule in turn.

Application of some of the rules introduces no new indexed variables into $\mathcal{M}$. These rules are: the rule for **assume**, the rule for empty guarded blocks, the rule for **alter all**, the rule for consecutive **assume** statements, the **assume-confirm** rule, and the rule for consecutive **confirm** statements. By the induction hypothesis, the rule's $\mathcal{P}$ satisfies the lemma's conditions. Hence, the rule's $\mathcal{M}$ satisfies the lemma's conditions. We consider each of the remaining rules in turn.

By the induction hypothesis, $\mathcal{P}$ of the rule for procedure call satisfies the lemma's conditions. The only indexed variables introduced into $\mathcal{M}$ are indexed by i and j, but neither **stow**(i) nor **stow**(j) is within a compound statement. Hence, $\mathcal{M}$ of the rule for procedure call satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for selection in the absence of an **else** clause satisfies the lemma's conditions. The only indexed variables introduced into $\mathcal{M}$ are indexed by i, j, k, or n, but none of **stow**(i), **stow**(j), **stow**(k), or **stow**(n) is within a compound statement of $\mathcal{M}$ (even though **stow**(j) and **stow**(k) were within a compound statement of $\mathcal{P}$). Hence, $\mathcal{M}$ of the rule for selection in the absence of an **else** clause satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for selection in the presence of an **else** clause satisfies the lemma's conditions. The only indexed variables introduced into $\mathcal{M}$ are indexed by i, j, k, l, m, or n, but none of **stow**(i), **stow**(j), **stow**(k), **stow**(l),

stow(m), or **stow**(n) is within a compound statement of $\mathcal{M}$ (even though **stow**(j), **stow**(k), **stow**(l), and **stow**(m) were within a compound statement of $\mathcal{P}$). Hence, $\mathcal{M}$ of the rule for selection in the presence of an **else** clause satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the **loop while** rule satisfies the lemma's conditions. The only indexed variables introduced into $\mathcal{M}$ are indexed by i, j, k, or l, but none of **stow**(i), **stow**(j), **stow**(k), or **stow**(l) is within a compound statement of $\mathcal{M}$ (even though **stow**(j) and **stow**(k) were within a compound statement of $\mathcal{P}$). Hence, $\mathcal{M}$ of the **loop while** rule satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for **confirm** satisfies the lemma's conditions. The only indexed variable introduced into $\mathcal{M}$ is indexed by i, but **stow**(i) is not within a compound statement. Hence, $\mathcal{M}$ of the rule for **confirm** satisfies the lemma's conditions.

The induction step is now complete. Hence the proof of this lemma (4.23) is finished. □

The old state and the declaration meanings of an environment remain unchanged when a statement that may appear in top level code is interpreted. We state and prove this fact below in Lemma 4.25. Lemma 4.24 states that the interpretation of sequences of such statements also preserves both the old state and the declaration meanings. That is to say, interpretation of top level code preserves both the old state and the declaration meanings.

Lemma 4.24 (Sequences of Statements That Preserve os and d) *Let env be an environment. If, for any environment env′, OSE($\mathcal{I}$(S1)(env′)) = OSE(env′) and DME($\mathcal{I}$(S1)(env′)) = DME(env′) for any statement S1 of SS for which $\mathcal{I}$(S1) is defined at env′, then, if $\mathcal{I}$(SS) is defined at env, OSE($\mathcal{I}$(SS)(env)) = OSE(env) and DME($\mathcal{I}$(SS)(env)) = DME(env).*

Proof. We use induction on the length of SS. The base case is a length of zero; i.e., SS = ε, the empty sequence of statements. Because the interpretation of the empty sequence of statements is the identity function, $\mathcal{I}$(SS)(env) = env. Substitution of equals then yields OSE($\mathcal{I}$(SS)(env)) = OSE(env) and DME($\mathcal{I}$(SS)(env)) = DME(env); we have finished the base case. For the induction step, suppose $0 < n$ and that the lemma holds for all SSs whose length is less than n. We suppose that the length of the statement sequence ST1 SS2 is n, and prove the lemma for ST1 SS2. Let env be an arbitrary environment. Suppose $\mathcal{I}$(ST1 SS2) is defined at env. Further suppose, given any environment env′, that OSE($\mathcal{I}$(S1)(env′)) = OSE(env′) and DME($\mathcal{I}$(S1)(env′)) = DME(env′) for any statement S1 of ST1 SS2 for which $\mathcal{I}$(S1) is defined at env′. Then, given any environment env″, OSE($\mathcal{I}$(S1)(env″)) = OSE(env″) and DME($\mathcal{I}$(S1)(env″)) = DME(env″) for any statement S1 of SS2 for which $\mathcal{I}$(S1) is defined at env″. By equation 2.23, $\mathcal{I}$(ST1) is defined at env, and $\mathcal{I}$(SS2) is defined at $\mathcal{I}$(ST1)(env). Also, by equation 2.23, OSE($\mathcal{I}$(ST1 SS2)(env)) = OSE($\mathcal{I}$(SS2)($\mathcal{I}$(ST1)(env))). Thus, we have OSE($\mathcal{I}$(ST1)(env)) = OSE(env) and DME($\mathcal{I}$(ST1)(env)) = DME(env). Setting env″ = $\mathcal{I}$(ST1)(env), we have OSE($\mathcal{I}$(SS2)($\mathcal{I}$(ST1)(env))) = OSE($\mathcal{I}$(ST1)(env)).

Therefore, the chain of equalities yields OSE($\mathcal{I}$(ST1 SS2)(env)) = OSE(env). Identical reasoning gives DME($\mathcal{I}$(ST1 SS2)(env)) = DME(env), concluding the induction step and the proof. □

Lemma 4.25 (Statements Within Top Level Code Preserve os and d) *Let S1 be any statement that can occur within top level code. (Top level code is defined in Section 3.1 and summarized in Figure 32. The statement S1 may be a statement of the top-level statement sequence, or it may, recursively, be a statement within a compound statement. The restrictions of Section 3.1 assure that S1 is not a* **remember** *statement.) If env is any environment at which $\mathcal{I}$(S1) is defined, then OSE($\mathcal{I}$(S1)(env)) = OSE(env) and DME($\mathcal{I}$(S1)(env)) = DME(env).*

Proof. If AE(env) $\neq$ NL, then $\mathcal{I}$(S1)(env) = env. Substitution of equals gives OSE($\mathcal{I}$(S1)(env)) = OSE(env) and DME($\mathcal{I}$(S1)(env)) = DME(env). Therefore, for the remainder of this proof, we assume AE(env) = NL. We use induction on the structure of statement S1. In the base case, S1 is one of the five non-compound statements: procedure call, **confirm**, **assume**, **stow**, and **alter all**. The definitions of the interpretations of these five statements yield immediately that OSE($\mathcal{I}$(S1)(env)) = OSE(env) and DME($\mathcal{I}$(S1)(env)) = DME(env).

In the induction step, S1 is one of the four compound statements: **if-then**, **if-then-else**, **loop while**, and **whenever**. The induction hypothesis is that the lemma is true for each statement within S1. In the following treatments of each of the four compound statements, we suppose that env is any environment at which $\mathcal{I}$(S1) is defined.

Let S1 be the statement "**if** *b_p_e* **then** SS **end if**".

Case $\mathcal{I}$(b_p_e)(*env*) = **false**. In this case, $\mathcal{I}$(S1)(env) = env. Substitution of equals gives OSE($\mathcal{I}$(S1)(env)) = OSE(env) and DME($\mathcal{I}$(S1)(env)) = DME(env).

Case $\mathcal{I}$(b_p_e)(*env*) = **true**. In this case, $\mathcal{I}$(S1)(env) = $\mathcal{I}$(SS)(env). By the induction hypothesis, if ST1 is a statement of SS, env′ is some environment, and $\mathcal{I}$(ST1) is defined at env′, then OSE($\mathcal{I}$(ST1)(env′)) = OSE(env′) and DME($\mathcal{I}$(ST1)(env′)) = DME(env′). Then, by Lemma 4.24, OSE($\mathcal{I}$(SS)(env)) = OSE(env) and DME($\mathcal{I}$(SS)(env)) = DME(env). Substitution of equals gives OSE($\mathcal{I}$(S1)(env)) = OSE(env) and DME($\mathcal{I}$(S1)(env)) = DME(env).

Let S1 be the statement "**if** *b_p_e* **then** SS1 **else** SS2 **end if**".

Case $\mathcal{I}$(b_p_e)(*env*) = **true**. In this case, $\mathcal{I}$(S1)(env) = $\mathcal{I}$(SS1)(env). By the induction hypothesis, if ST1 is a statement of SS1, env′ is some environment, and $\mathcal{I}$(ST1) is defined at env′, then OSE($\mathcal{I}$(ST1)(env′)) = OSE(env′) and DME($\mathcal{I}$(ST1)(env′)) = DME(env′). Then, by Lemma 4.24, OSE($\mathcal{I}$(SS1)(env)) = OSE(env) and DME($\mathcal{I}$(SS1)(env)) = DME(env). Substitution of equals gives OSE($\mathcal{I}$(S1)(env)) = OSE(env) and DME($\mathcal{I}$(S1)(env)) = DME(env).

Case $\mathcal{I}$(b_p_e)(*env*) = **false**. In this case, $\mathcal{I}$(S1)(env) = $\mathcal{I}$(SS2)(env). By the induction hypothesis, if ST1 is a statement of SS2, env′ is some environment, and $\mathcal{I}$(ST1) is defined at env′, then OSE($\mathcal{I}$(ST1)(env′)) = OSE(env′) and DME($\mathcal{I}$(ST1)(env′)) = DME(env′). Then, by Lemma 4.24, OSE($\mathcal{I}$(SS2)(env)) = OSE(env) and DME($\mathcal{I}$(SS2)(env)) = DME(env). Substitution of equals gives OSE($\mathcal{I}$(S1)(env)) = OSE(env) and DME($\mathcal{I}$(S1)(env)) = DME(env).

Let S1 be the statement "**loop maintaining** Inv **while** *b_p_e* **do** SS **end loop**". By the induction hypothesis, if ST1 is a statement of SS, env′ is some environment, and $\mathcal{I}$(ST1) is defined at env′, then OSE($\mathcal{I}$(ST1)(env′)) = OSE(env′) and DME($\mathcal{I}$(ST1)(env′)) = DME(env′). Then, by Lemma 4.24, OSE($\mathcal{I}$(SS)(env)) = OSE(env) and DME($\mathcal{I}$(SS)(env)) = DME(env). In the first three cases of the definition of Λ_W, OSE(Λ_W(WL)(env)) = OSE(env) and DME(Λ_W(WL)(env)) = DME(env). Because $\mathcal{I}$(S1) is defined at env, MFP(Λ_W) is defined at env. Therefore, because MFP(Λ_W) is a fixed point of Λ_W, OSE(MFP(Λ_W)(env)) = OSE(env) and DME(MFP(Λ_W)(env)) = DME(env). By Lemma 4.26 (to follow), OSE($\mathcal{I}$(S1)(env)) = OSE(env) and DME($\mathcal{I}$(S1)(env)) = DME(env).

Let S1 be the statement "**whenever** Br_Cd **do** SS **end whenever**".

Case $\mathcal{I}$(*Br_Cd*)(*env*) = **false**. In this case, $\mathcal{I}$(S1)(env) = env. Substitution of equals gives OSE($\mathcal{I}$(S1)(env)) = OSE(env) and DME($\mathcal{I}$(S1)(env)) = DME(env).

Case $\mathcal{I}$(*Br_Cd*)(*env*) = **true**. In this case, $\mathcal{I}$(S1)(env) = $\mathcal{I}$(SS)(env). By the induction hypothesis, if ST1 is a statement of SS, env′ is some environment, and $\mathcal{I}$(ST1) is defined at env′, then OSE($\mathcal{I}$(ST1)(env′)) = OSE(env′) and DME($\mathcal{I}$(ST1)(env′)) = DME(env′). Then, by Lemma 4.24, OSE($\mathcal{I}$(SS)(env)) = OSE(env) and DME($\mathcal{I}$(SS)(env)) = DME(env). Substitution of equals gives OSE($\mathcal{I}$(S1)(env)) = OSE(env) and DME($\mathcal{I}$(S1)(env)) = DME(env).

The induction step and the proof are now complete. □

Lemma 4.26 *Let F be a function from environments to environments such that $OSE(F(env)) = OSE(env)$ and $DME(F(env)) = DME(env)$. Then $OSE(Fgt(F(Rem(env)))) = OSE(env)$ and $DME(Fgt(F(Rem(env)))) = DME(env)$.*

Proof. By the definitions of Rem and Fgt, for any environment env′, DME(Rem(env′)) = DME(env′) and DME(Fgt(env′)) = DME(env′). Therefore, DME(Fgt(F(Rem(env)))) = DME(F(Rem(env))) = DME(Rem(env)) = DME(env). Let ξ be an arbitrary current variable name. Let k be an arbitrary natural number. Then $\#^{k+1}\xi$ is an arbitrary old variable name. By definition of Fgt, OSE(Fgt(F(Rem(env))))($\#^{k+1}\xi$) = OSE(F(Rem(env)))($\#\#^{k+1}\xi$) = OSE(F(Rem(env)))($\#^{k+2}\xi$). By the lemma's hypothesis concerning F, OSE(F(Rem(env)))($\#^{k+2}\xi$) = OSE(Rem(env))($\#^{k+2}\xi$). By definition of Rem, OSE(Rem(env))($\#^{k+2}\xi$) = OSE(env)($\#^{k+1}\xi$). The chain of equalities gives us OSE(Fgt(F(Rem(env))))($\#^{k+1}\xi$) = OSE(env)($\#^{k+1}\xi$). Therefore, OSE(Fgt(F(Rem(env)))) = OSE(env). □

Lemma 4.27 follows immediately from Lemmas 4.24 and 4.25.

Lemma 4.27 (Statement Sequences Preserve os and d) *Let env be an environment. Let SS be a sequence of statements, every one of which can occur within top level code. (SS contains no* **remember** *statements.) If $\mathcal{I}(SS)$ is defined at env, then $OSE(\mathcal{I}(SS)(env)) = OSE(env)$ and $DME(\mathcal{I}(SS)(env)) = DME(env)$.*

Definition 4.4 *A statement sequence SS is* unaffected by old state *(or* unaffected by os*) if and only if for any two environments that differ at most in their old states, say*

$$env_1 = [a, cs, os_1, ns, se, d] \tag{4.15}$$

$$env_2 = [a, cs, os_2, ns, se, d], \tag{4.16}$$

1. $\mathcal{I}(SS)(env_1)$ *is defined if and only if* $\mathcal{I}(SS)(env_2)$ *is defined, and*

2. *if* $\mathcal{I}(SS)(env_1)$ *is defined, say*

$$\mathcal{I}(SS)(env_1) = [a', cs', os_1, ns', se', d], \tag{4.17}$$

then

$$\mathcal{I}(SS)(env_2) = [a', cs', os_2, ns', se', d]. \tag{4.18}$$

Lemma 4.28 (Sequences of Statements That Are Unaffected by os) *If each statement S1 of statement sequence SS is unaffected by old state, then SS is unaffected by old state.*

Proof. We use induction on the length of SS. The base case is a length of zero; i.e., SS $= \varepsilon$, the empty sequence of statements. Let env_1 and env_2 be defined as in Definition 4.4. Because the interpretation of the empty sequence of statements is the identity function, both $\mathcal{I}(\text{SS})(\text{env}_1)$ and $\mathcal{I}(\text{SS})(\text{env}_2)$ are defined, $\mathcal{I}(\text{SS})(\text{env}_1) = \text{env}_1$, and $\mathcal{I}(\text{SS})(\text{env}_2) = \text{env}_2$. Because $\mathcal{I}(\text{SS})(\text{env}_2)$ satisfies the condition of Definition 4.4, SS is unaffected by old state. For the induction step, suppose $0 < n$ and that the lemma holds for all SSs whose length is less than n. We suppose that the length of the

statement sequence ST1 SS2 is n, and prove the lemma for ST1 SS2. Let env_1 and env_2 be defined as in Definition 4.4. Because ST1 is not affected by old state, $\mathcal{I}(\text{ST1})(\text{env}_1)$ is defined if and only if $\mathcal{I}(\text{ST1})(\text{env}_2)$ is defined. By equation 2.23 and the composition of functions, if $\mathcal{I}(\text{ST1})(\text{env}_1)$ is not defined, then neither $\mathcal{I}(\text{ST1 SS2})(\text{env}_1)$ nor $\mathcal{I}(\text{ST1 SS2})(\text{env}_2)$ is defined. Suppose $\mathcal{I}(\text{ST1})(\text{env}_1)$ is defined, say

$$\mathcal{I}(\text{ST1})(\text{env}_1) = [\text{a}', \text{cs}', \text{os}_1, \text{ns}', \text{se}', \text{d}]. \tag{4.19}$$

Then

$$\mathcal{I}(\text{ST1})(\text{env}_2) = [\text{a}', \text{cs}', \text{os}_2, \text{ns}', \text{se}', \text{d}]. \tag{4.20}$$

By the induction hypothesis, SS2 is unaffected by old state. Therefore, $\mathcal{I}(\text{SS2})(\mathcal{I}(\text{ST1})(\text{env}_1))$ is defined if and only if $\mathcal{I}(\text{SS2})(\mathcal{I}(\text{ST1})(\text{env}_2))$ is defined. Suppose $\mathcal{I}(\text{SS2})(\mathcal{I}(\text{ST1})(\text{env}_1))$ is defined, say

$$\mathcal{I}(\text{SS2})(\mathcal{I}(\text{ST1})(\text{env}_1)) = [\text{a}'', \text{cs}'', \text{os}_1, \text{ns}'', \text{se}'', \text{d}]. \tag{4.21}$$

Then

$$\mathcal{I}(\text{SS2})(\mathcal{I}(\text{ST1})(\text{env}_2)) = [\text{a}'', \text{cs}'', \text{os}_2, \text{ns}'', \text{se}'', \text{d}]. \tag{4.22}$$

By equation 2.23, ST1 SS2 is unaffected by old state. □

Lemma 4.29 (Top Level Code Is Unaffected by os) *Each statement that can occur within top level code is unaffected by old state.*

Proof. Let S1 be a statement that can occur within top level code. If $\text{AE}(\text{env}) \neq \text{NL}$, then $\mathcal{I}(\text{S1})(\text{env}) = \text{env}$. In this case, S1 satisfies the definition of being unaffected by

old state. Therefore, for the remainder of this proof, we assume AE(env) = NL. We use induction on the structure of statement S1. In the base case, because S1 is not a **remember** statement, S1 is one of the five non-compound statements: procedure call, **confirm**, **assume**, **stow**, and **alter all**. In top level code, **confirm** and **assume** statements do not contain old variables; i.e., none of the statements of top level code have the form **confirm** ⟨old_assert⟩ or **assume** ⟨old_assert⟩. Therefore, examination of the definitions of the interpretations of these five statements reveals that S1 is unaffected by old state.

In the induction step, S1 is one of the four compound statements: **if-then**, **if-then-else**, **loop while**, and **whenever**. The induction hypothesis is that each statement within S1 is unaffected by old state. The arguments for the **if-then-else** and **whenever** statements are similar to the argument for the **if-then** statement. We give here only the arguments for the **if-then** and **loop while** statements. Let env_1 and env_2 be defined as in Definition 4.4.

Let S1 be the statement "**if** *b_p_e* **then** SS **end if**".

Case $\mathcal{I}(\text{b\_p\_e})(env_1) =$ **false**. Because *b_p_e* contains no old variables, $\mathcal{I}(b\_p\_e)(\text{env}_2) =$ **false**. Therefore, both $\mathcal{I}(\text{S1})(\text{env}_1)$ and $\mathcal{I}(\text{S1})(\text{env}_2)$ are defined, $\mathcal{I}(\text{S1})(\text{env}_1) = \text{env}_1$, and $\mathcal{I}(\text{S1})(\text{env}_2) = \text{env}_2$. Because $\mathcal{I}(\text{S1})(\text{env}_2)$ satisfies the condition of Definition 4.4, S1 is unaffected by old state.

Case $\mathcal{I}(\text{b\_p\_e})(env_1) =$ **true**. Because *b_p_e* contains no old variables, $\mathcal{I}(b\_p\_e)(\text{env}_2) =$ **true**. Therefore, $\mathcal{I}(\text{S1})(\text{env}_1) = \mathcal{I}(\text{SS})(\text{env}_1)$ and $\mathcal{I}(\text{S1})(\text{env}_2) =$

$\mathcal{I}(\mathrm{SS})(\mathrm{env}_2)$. By the induction hypothesis and Lemma 4.28, SS is unaffected by old state. By substitution of equals in Definition 4.4, S1 is unaffected by old state.

Let S1 be the statement "**loop maintaining** Inv **while** *b_p_e* **do** SS **end loop**". By the induction hypothesis and Lemma 4.28, SS is unaffected by old state; *b_p_e* contains no old variables. Each old variable in Inv contains exactly one # sign. Let ξ be an arbitrary current variable name. By definition of function Rem, $\mathrm{OSE}(\mathrm{Rem}(\mathrm{env}_1))(\#\xi) = \mathrm{cs}(\xi) = \mathrm{OSE}(\mathrm{Rem}(\mathrm{env}_2))(\#\xi)$. Therefore, if $\mathrm{MFP}(\Lambda_{\mathrm{W}})(\mathrm{Rem}(\mathrm{env}_1))$ is not defined, then $\mathrm{MFP}(\Lambda_{\mathrm{W}})(\mathrm{Rem}(\mathrm{env}_2))$ is not defined. Furthermore, if $\mathrm{MFP}(\Lambda_{\mathrm{W}})(\mathrm{Rem}(\mathrm{env}_1))$ is defined, say

$$\mathrm{MFP}(\Lambda_{\mathrm{W}})(\mathrm{Rem}(\mathrm{env}_1)) = [\mathrm{a}', \mathrm{cs}', \mathrm{os}_1', \mathrm{ns}', \mathrm{se}', \mathrm{d}], \tag{4.23}$$

then

$$\mathrm{MFP}(\Lambda_{\mathrm{W}})(\mathrm{Rem}(\mathrm{env}_2)) = [\mathrm{a}', \mathrm{cs}', \mathrm{os}_2', \mathrm{ns}', \mathrm{se}', \mathrm{d}]. \tag{4.24}$$

Hence,

$$\mathrm{Fgt}(\mathrm{MFP}(\Lambda_{\mathrm{W}})(\mathrm{Rem}(\mathrm{env}_1))) = [\mathrm{a}', \mathrm{cs}', \mathrm{os}_1, \mathrm{ns}', \mathrm{se}', \mathrm{d}] \tag{4.25}$$

$$\mathrm{Fgt}(\mathrm{MFP}(\Lambda_{\mathrm{W}})(\mathrm{Rem}(\mathrm{env}_2))) = [\mathrm{a}', \mathrm{cs}', \mathrm{os}_2, \mathrm{ns}', \mathrm{se}', \mathrm{d}]. \tag{4.26}$$

Therefore, S1 is unaffected by old state.

The induction step and the proof are now complete. □

Lemma 4.30 follows immediately from Lemmas 4.28 and 4.29.

Lemma 4.30 (Statement Sequences Are Unaffected by os) *If SS is a sequence of statements, every one of which can occur within top level code, then SS is unaffected by old state.*

Interpretation of internal code—statement sequences internal to **whenever**, selection, or iteration statements—is unaffected by the index state or the setup. Lemmas 4.31 and 4.32 capture this fact.

Lemma 4.31 *Let S1 be a statement that may appear in internal code. Let* env_1 *and* env_2 *be two environments. If* $\mathcal{I}(S1)$ *is defined at* env_1, $AE(env_1) = AE(env_2)$, $CSE(env_1) = CSE(env_2)$, $OSE(env_1) = OSE(env_2)$, *and* $DME(env_1) = DME(env_2)$, *then* $\mathcal{I}(S1)$ *is defined at* env_2, $AE(\mathcal{I}(S1)(env_1)) = AE(\mathcal{I}(S1)(env_2))$, $CSE(\mathcal{I}(S1)(env_1)) = CSE(\mathcal{I}(S1)(env_2))$, $OSE(\mathcal{I}(S1)(env_1)) = OSE(\mathcal{I}(S1)(env_2))$, *and* $DME(\mathcal{I}(S1)(env_1)) = DME(\mathcal{I}(S1)(env_2))$.

Proof. According to Section 3.1 (especially Figure 34), S1 may be **stow**(⟨nat_num⟩), **confirm** ⟨cur_assert⟩, or an operational statement. If $\mathrm{AE}(\mathrm{env}_1) \neq \mathrm{NL}$, then $\mathrm{AE}(\mathrm{env}_2) \neq \mathrm{NL}$. We would then have $\mathcal{I}(\mathrm{S1})(\mathrm{env}_1) = \mathrm{env}_1$ and $\mathcal{I}(\mathrm{S1})(\mathrm{env}_2) = \mathrm{env}_2$. The lemma follows immediately by substitution of equals.

Therefore, for the remainder of this proof, we assume $\mathrm{AE}(\mathrm{env}_1) = \mathrm{NL}$. (Hence, $\mathrm{AE}(\mathrm{env}_2) = \mathrm{NL}$.) We use induction on the structure of statement S1. In the base case, S1 is one of the three non-compound statements that can occur in internal code: procedure call, **confirm**, and **stow**.

Let S1 be a procedure call. By the semantics of procedure call, $\mathrm{OSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{OSE}(\mathrm{env}_1) = \mathrm{OSE}(\mathrm{env}_2) = \mathrm{OSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$ and $\mathrm{DME}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{DME}(\mathrm{env}_1) = \mathrm{DME}(\mathrm{env}_2) = \mathrm{DME}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$. Again, by the semantics of procedure call, because $\mathrm{CSE}(\mathrm{env}_1) = \mathrm{CSE}(\mathrm{env}_2)$ and $\mathrm{DME}(\mathrm{env}_1) =$

$\mathrm{DME}(\mathrm{env}_2)$, $\mathrm{AE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{AE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$ and $\mathrm{CSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{CSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$. Hence, the lemma holds for procedure call.

Let S1 be **confirm** Q. Because Q is a current assertion—an assertion that contains only current variables—$\mathrm{CSE}(\mathrm{env}_1) = \mathrm{CSE}(\mathrm{env}_2)$ implies $\mathcal{I}(\mathrm{Q})(\mathrm{env}_1) = \mathcal{I}(\mathrm{Q})(\mathrm{env}_2)$. Therefore, by the semantics of keyConfirm Q, $\mathrm{AE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{AE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$. Again, by the semantics of keyConfirm Q, $\mathrm{CSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{CSE}(\mathrm{env}_1) = \mathrm{CSE}(\mathrm{env}_2) = \mathrm{CSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$, $\mathrm{OSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{OSE}(\mathrm{env}_1) = \mathrm{OSE}(\mathrm{env}_2) = \mathrm{OSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$, and $\mathrm{DME}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{DME}(\mathrm{env}_1) = \mathrm{DME}(\mathrm{env}_2) = \mathrm{DME}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$. Hence, the lemma holds for **confirm** Q.

Let S1 be **stow**(i). By the semantics of **stow**(i), $\mathrm{AE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{AE}(\mathrm{env}_1) = \mathrm{AE}(\mathrm{env}_2) = \mathrm{AE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$, $\mathrm{CSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{CSE}(\mathrm{env}_1) = \mathrm{CSE}(\mathrm{env}_2) = \mathrm{CSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$, $\mathrm{OSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{OSE}(\mathrm{env}_1) = \mathrm{OSE}(\mathrm{env}_2) = \mathrm{OSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$, and $\mathrm{DME}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{DME}(\mathrm{env}_1) = \mathrm{DME}(\mathrm{env}_2) = \mathrm{DME}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$. Hence, the lemma holds for **stow**(i).

In the induction step, S1 is one of the three compound statements that can occur in internal code: **if-then**, **if-then-else**, and **loop while**. The induction hypothesis is that the lemma is true for each statement within S1.

Let S1 be one of the two selection statements. Because all of b_p_e's variables are current variables and $\mathrm{CSE}(\mathrm{env}_1) = \mathrm{CSE}(\mathrm{env}_2)$, $\mathcal{I}(b\_p\_e)(\mathrm{env}_1) = \mathcal{I}(b\_p\_e)(\mathrm{env}_2)$. So, without loss of generality, there is a statement sequence SS′ such that $\mathcal{I}(\mathrm{S1})(\mathrm{env}_1) = \mathcal{I}(\mathrm{SS}')(\mathrm{env}_1)$ and $\mathcal{I}(\mathrm{S1})(\mathrm{env}_2) = \mathcal{I}(\mathrm{SS}')(\mathrm{env}_2)$. SS′ is either the empty statement sequence or a sequence of statements within S1. If it is the empty sequence, the

lemma follows immediately from the semantics of the empty sequence and substitution of equals. If it is a sequence of statements within S1, the lemma follows from the induction hypothesis and a simple induction on the length of SS′. Hence, the lemma holds if S1 is a selection statement.

Let S1 be the statement "**loop maintaining** Inv **while** *b_p_e* **do** SS **end loop**". Because $\mathcal{I}$(S1) is defined at env_1, MFP(Λ_W) is defined at Rem(env_1). Therefore, if env is some environment resulting from zero or more iterations of the loop beginning in environment Rem(env_1), then $\mathcal{I}$(SS) is defined at env. By the induction hypothesis and a simple induction on the length of SS, if env_3 and env_4 are two environments such that $\mathcal{I}$(SS) is defined at env_3, AE(env_3) = AE(env_4), CSE(env_3) = CSE(env_4), OSE(env_3) = OSE(env_4), and DME(env_3) = DME(env_4), then $\mathcal{I}$(SS) is defined at env_4, AE($\mathcal{I}$(SS)(env_3)) = AE($\mathcal{I}$(SS)(env_4)), CSE($\mathcal{I}$(SS)(env_3)) = CSE($\mathcal{I}$(SS)(env_4)), OSE($\mathcal{I}$(SS)(env_3)) = OSE($\mathcal{I}$(SS)(env_4)), and DME($\mathcal{I}$(SS)(env_3)) = DME($\mathcal{I}$(SS)(env_4)). Under the same assumptions about env_3 and env_4, because the variables in Inv are only old variables and current variables and the variables in *b_p_e* are only current variables, $\mathcal{I}$(Inv)(env_3) = $\mathcal{I}$(Inv)(env_4) and $\mathcal{I}$(*b_p_e*)(env_3) = $\mathcal{I}$(*b_p_e*)(env_4). Now AE(Rem(env_1)) = AE(Rem(env_2)), CSE(Rem(env_1)) = CSE(Rem(env_2)), OSE(Rem(env_1)) = OSE(Rem(env_2)), and DME(Rem(env_1)) = DME(Rem(env_2)). Therefore, MFP(Λ_W) is defined at Rem(env_2), AE(MFP(Λ_W)(Rem(env_1))) = AE(MFP(Λ_W)(Rem(env_2))), CSE(MFP(Λ_W)(Rem(env_1))) = CSE(MFP(Λ_W)(Rem(env_2))), OSE(MFP(Λ_W)(Rem(env_1))) = OSE(MFP(Λ_W)(Rem(env_2))), and

$\mathrm{DME}(\mathrm{MFP}(\Lambda_{\mathrm{W}})(\mathrm{Rem}(\mathrm{env}_1))) = \mathrm{DME}(\mathrm{MFP}(\Lambda_{\mathrm{W}})(\mathrm{Rem}(\mathrm{env}_2)))$. By the definition of the function Fgt, $\mathcal{I}(\mathrm{S1})$ is defined at env_2, $\mathrm{AE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{AE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$, $\mathrm{CSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{CSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$, $\mathrm{OSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{OSE}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$, and $\mathrm{DME}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_1)) = \mathrm{DME}(\mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$. Hence, the lemma holds if S1 is a **loop-while** statement. The induction step and the proof are now finished. □

Lemma 4.32 (Interpretation of Internal Code) *Let SS be internal code. Let* env_1 *and* env_2 *be two environments. If* $\mathcal{I}(SS)$ *is defined at* env_1, $AE(env_1) = AE(env_2)$, $CSE(env_1) = CSE(env_2)$, $OSE(env_1) = OSE(env_2)$, *and* $DME(env_1) = DME(env_2)$, *then* $\mathcal{I}(SS)$ *is defined at* env_2, $AE(\mathcal{I}(SS)(env_1)) = AE(\mathcal{I}(SS)(env_2))$, $CSE(\mathcal{I}(SS)(env_1)) = CSE(\mathcal{I}(SS)(env_2))$, $OSE(\mathcal{I}(SS)(env_1)) = OSE(\mathcal{I}(SS)(env_2))$, *and* $DME(\mathcal{I}(SS)(env_1)) = DME(\mathcal{I}(SS)(env_2))$.

Proof. This lemma follows easily from Lemma 4.31 by induction on the length of SS. □

4.3.5 Negative-Branch-Condition Independence

The negative-branch-condition independence lemma is key for our ability to establish that each rule preserves invalidity in the math direction. The interpretations of the original and the result of a rule application in a given environment do not always result in equal environments. Recall that the rules are constructed not to preserve semantics but merely to preserve invalidity. The negative-branch-condition independence lemma states that interpretation of a particular section of code beginning in environments that are "nearly" equal results in environments that are "nearly" equal. The definition

of Equal_except_at establishes what it means for two environments to be "nearly" equal.

Definition 4.5 *Equal_except_at is a predicate.*

$$\mathit{Equal\_except\_at} : (\wp(\mathit{Integers}) \times \mathit{Environments} \times \mathit{Environments}) \rightarrow \mathit{Boolean} \quad (4.27)$$

Equal_except_at(A, env_1, env_2) *if and only if all of the following hold:*

$$AE(env_1) = AE(env_2) \quad (4.28)$$

$$CSE(env_1) = CSE(env_2) \quad (4.29)$$

$$h \notin A \Rightarrow ISE(env_1)(h) = ISE(env_2)(h) \quad (4.30)$$

$$SPE(env_1) = SPE(env_2) \quad (4.31)$$

$$OSE(env_1) = OSE(env_2) \quad (4.32)$$

$$DME(env_1) = DME(env_2) \quad (4.33)$$

Furthermore, if $\mathcal{I}(SS)$ *is undefined at* env_1 *and* $\mathcal{I}(SS)$ *is undefined at* env_2*, then* *Equal_except_at*$(A, \mathcal{I}(SS)(env_1), \mathcal{I}(SS)(env_2))$ *where SS is a statement sequence.*

Two environments are "nearly" equal if (1) all their components except possibly their index-states are equal and (2) their index-states agree everywhere except on a certain set of integers. In all uses we make of this definition, the set of integers, A, is finite. The interpretations of a statement sequence in two environments are "nearly" equal if (1) both interpretations are defined and "nearly" equal or (2) both interpretations are undefined.

If the interpretations of a statement in two environments that are nearly equal are always nearly equal as well, we say that the statement is *uncritical* of that notion of near equality. Lemma 4.33 states that the interpretations of a statement sequence in two environments that are nearly equal remain nearly equal, assuming each statement of the sequence is uncritical of that notion of near equality.

Lemma 4.33 (Sequences of Uncritical Statements) *Let SN be a set of integers and SS a statement sequence. Let* env_1 *and* env_2 *be two environments. If Equal_except_at*(SN, env_1, env_2) *implies Equal_except_at*$(SN, \mathcal{I}(S1)(env_1), \mathcal{I}(S1)(env_2))$ *for any statement S1 of SS, then Equal_except_at*$(SN, \mathcal{I}(SS)(env_3), \mathcal{I}(SS)(env_4))$ *for any two environments* env_3 *and* env_4 *such that Equal_except_at*(SN, env_3, env_4).

Proof. We use induction on the length of SS. The base case is a length of zero; i.e., $\mathrm{SS} = \varepsilon$, the empty sequence of statements. Because the interpretation of the empty sequence of statements is the identity function, $\mathcal{I}(\mathrm{SS})(\mathrm{env}) = \mathrm{env}$. So, because $\mathrm{Equal\_except\_at}(\mathrm{SN}, \mathrm{env}_3, \mathrm{env}_4)$, $\mathrm{Equal\_except\_at}(\mathrm{SN}, \mathcal{I}(\mathrm{SS})(\mathrm{env}_3), \mathcal{I}(\mathrm{SS})(\mathrm{env}_4))$. For the induction step, suppose $0 < n$ and that the lemma holds for all SSs whose length is less than n. We suppose that the length of the statement sequence ST1 SS2 is n, and prove the lemma for ST1 SS2. Suppose that $\mathrm{Equal\_except\_at}(\mathrm{SN}, \mathrm{env}_1, \mathrm{env}_2)$ implies $\mathrm{Equal\_except\_at}(\mathrm{SN}, \mathcal{I}(\mathrm{S1})(\mathrm{env}_1), \mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$ for any statement S1 of ST1 SS2. Then $\mathrm{Equal\_except\_at}(\mathrm{SN}, \mathrm{env}_1, \mathrm{env}_2)$ implies $\mathrm{Equal\_except\_at}(\mathrm{SN}, \mathcal{I}(\mathrm{S1})(\mathrm{env}_1), \mathcal{I}(\mathrm{S1})(\mathrm{env}_2))$ for any statement S1 of SS2. $\mathrm{Equal\_except\_at}(\mathrm{SN}, \mathrm{env}_1, \mathrm{env}_2)$ also implies

Equal_except_at(SN, $\mathcal{I}$(ST1)(env$_1$), $\mathcal{I}$(ST1)(env$_2$)). By the induction hypothesis, Equal_except_at(SN, $\mathcal{I}$(SS2)(env$_3$), $\mathcal{I}$(SS2)(env$_4$)) for any two environments env$_3$ and env$_4$ such that Equal_except_at(SN, env$_3$, env$_4$). This last relation is satisfied if we let env$_3$ = $\mathcal{I}$(ST1)(env$_1$) and env$_4$ = $\mathcal{I}$(ST1)(env$_2$). So, Equal_except_at(SN, $\mathcal{I}$(SS2)($\mathcal{I}$(ST1)(env$_1$)), $\mathcal{I}$(SS2)($\mathcal{I}$(ST1)(env$_2$))) for any two environments env$_1$ and env$_2$ such that Equal_except_at(SN, env$_1$, env$_2$). By the definition of the interpretation of a sequence of statements, we have Equal_except_at(SN, $\mathcal{I}$(ST1 SS2)(env$_1$), $\mathcal{I}$(ST1 SS2)(env$_2$)) for any two environments env$_1$ and env$_2$ such that Equal_except_at(SN, env$_1$, env$_2$). By a change of two variables, that is Equal_except_at(SN, $\mathcal{I}$(ST1 SS2)(env$_3$), $\mathcal{I}$(ST1 SS2)(env$_4$)) for any two environments env$_3$ and env$_4$ such that Equal_except_at(SN, env$_3$, env$_4$). The induction step and the proof are now complete. □

Lemma 4.34 states that a statement is uncritical of a certain notion of near equality if the statement has no variables indexed by the integers where the environments differ.

Lemma 4.34 (Uninvolved Indices) *Let SN be a set of integers. Let S1 be a statement that contains no variables indexed by an integer in SN. That is to say, if ξ_h is an indexed variable that occurs in S1, then $h \notin SN$. Let $SB \subseteq SN$. Let env$_1$ and env$_2$ be two environments. If Equal_except_at(SB, env$_1$, env$_2$), then Equal_except_at(SB, $\mathcal{I}$(S1)(env$_1$), $\mathcal{I}$(S1)(env$_2$)).*

Proof. If AE(env$_1$) $\neq$ NL, then, because Equal_except_at(SB, env$_1$, env$_2$), AE(env$_2$) $\neq$ NL. We would then have $\mathcal{I}$(S1)(env$_1$) = env$_1$ and $\mathcal{I}$(S1)(env$_2$) =

env_2. Then, by substitution of equals, if Equal_except_at(SB, env_1, env_2), then Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)).

Therefore, for the remainder of this proof, we assume AE(env_1) = NL. We use induction on the structure of statement S1. In the base case, S1 is one of the six non-compound statements: procedure call, **confirm**, **assume**, **remember**, **stow**, and **alter all**. We assume the lemma's hypotheses to prove Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)).

Let S1 be a procedure call. Not Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)) only if env_1 and env_2 differ in their assert status, their current state, or their declaration meanings. But they do not differ in those places because Equal_except_at(SB, env_1, env_2). Hence, Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)).

Let S1 be a **confirm** or **assume** statement. Not Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)) only if env_1 and env_2 differ in their assert status, their current state, their old state, or their index state at an integer h such that, for some ξ, ξ_h is a variable of S1. However, if h is such that, for some ξ, ξ_h is a variable of S1, then $h \notin$ SN. Hence $h \notin$ SB. Because Equal_except_at(SB, env_1, env_2), env_1 and env_2 do not differ in their assert status, their current state, their old state, or their index state at h. Therefore, Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)).

Let S1 be a **remember** statement. Not Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)) only if env_1 and env_2 differ

in their assert status, their current state, or their old state. But they do not differ in those places because Equal_except_at(SB, env_1, env_2). Hence, Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)).

Let S1 be a **stow** statement. Not Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)) only if env_1 and env_2 differ in their assert status, their current state, or their index state at some integer $i \notin$ SB. But they do not differ in those places because Equal_except_at(SB, env_1, env_2). Hence, Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)).

Let S1 be an **alter all** statement. Not Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)) only if env_1 and env_2 differ in their assert status or their setup. But they do not differ in those places because Equal_except_at(SB, env_1, env_2). Hence, Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)).

In the induction step, S1 is one of the four compound statements: **if-then**, **if-then-else**, **loop while**, and **whenever**. The induction hypothesis is that the lemma is true for each statement within S1. We assume the lemma's hypotheses to prove Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)).

Let S1 be the statement "**if** *b_p_e* **then** SS **end if**". There are no indexed variables in *b_p_e*. Hence, because Equal_except_at(SB, env_1, env_2), $\mathcal{I}$(*b_p_e*)(env_1) = $\mathcal{I}$(*b_p_e*)(env_2).

Case $\mathcal{I}$(b_p_e)(env_1) = **false**. Then $\mathcal{I}$(*b_p_e*)(env_2) = **false**. Hence, we have $\mathcal{I}$(S1)(env_1) = env_1 and $\mathcal{I}$(S1)(env_2) = env_2. Therefore,

Equal_except_at(SB, $\mathcal{I}$(S1)(env$_1$), $\mathcal{I}$(S1)(env$_2$)).

Case $\mathcal{I}$(b_p_e)(env_1) = **true**. Then $\mathcal{I}(b\_p\_e)$(env$_2$) = **true**. Hence, we have $\mathcal{I}$(S1)(env$_1$) = $\mathcal{I}$(SS)(env$_1$) and $\mathcal{I}$(S1)(env$_2$) = $\mathcal{I}$(SS)(env$_2$). By the induction hypothesis, if ST1 is a statement of SS, then Equal_except_at(SB, $\mathcal{I}$(ST1)(env$_1$), $\mathcal{I}$(ST1)(env$_2$)). By Lemma 4.33, Equal_except_at(SB, $\mathcal{I}$(SS)(env$_1$), $\mathcal{I}$(SS)(env$_2$)). Therefore, Equal_except_at(SB, $\mathcal{I}$(S1)(env$_1$), $\mathcal{I}$(S1)(env$_2$)).

Let S1 be the statement "**if** *b_p_e* **then** SS1 **else** SS2 **end if**". There are no indexed variables in *b_p_e*. Hence, because Equal_except_at(SB, env$_1$, env$_2$), $\mathcal{I}(b\_p\_e)$(env$_1$) = $\mathcal{I}(b\_p\_e)$(env$_2$).

Case $\mathcal{I}$(b_p_e)(env_1) = **true**. Then $\mathcal{I}(b\_p\_e)$(env$_2$) = **true**. Hence, we have $\mathcal{I}$(S1)(env$_1$) = $\mathcal{I}$(SS1)(env$_1$) and $\mathcal{I}$(S1)(env$_2$) = $\mathcal{I}$(SS1)(env$_2$). By the induction hypothesis, if ST1 is a statement of SS1, then Equal_except_at(SB, $\mathcal{I}$(ST1)(env$_1$), $\mathcal{I}$(ST1)(env$_2$)). By Lemma 4.33, Equal_except_at(SB, $\mathcal{I}$(SS1)(env$_1$), $\mathcal{I}$(SS1)(env$_2$)). Therefore, Equal_except_at(SB, $\mathcal{I}$(S1)(env$_1$), $\mathcal{I}$(S1)(env$_2$)).

Case $\mathcal{I}$(b_p_e)(env_1) = **false**. Then $\mathcal{I}(b\_p\_e)$(env$_2$) = **false**. Hence, we have $\mathcal{I}$(S1)(env$_1$) = $\mathcal{I}$(SS2)(env$_1$) and $\mathcal{I}$(S1)(env$_2$) = $\mathcal{I}$(SS2)(env$_2$). By the induction hypothesis, if ST1 is a statement of SS2, then Equal_except_at(SB, $\mathcal{I}$(ST1)(env$_1$), $\mathcal{I}$(ST1)(env$_2$)). By Lemma 4.33, Equal_except_at(SB, $\mathcal{I}$(SS2)(env$_1$), $\mathcal{I}$(SS2)(env$_2$)). Therefore, Equal_except_at(SB, $\mathcal{I}$(S1)(env$_1$), $\mathcal{I}$(S1)(env$_2$)).

Let S1 be the statement "**loop maintaining** Inv **while** *b_p_e* **do** SS **end loop**". Let env_3 and env_4 be any two environments such that Equal_except_at(SB, env_3, env_4). By the definitions of the functions Rem and Fgt, Equal_except_at(SB, Rem(env_3), Rem(env_4)) and Equal_except_at(SB, Fgt(env_3), Fgt(env_4)). Therefore, we have shown Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)) and are done, when we show Equal_except_at(SB, MFP(Λ_W)(env_3), MFP(Λ_W)(env_4)). Neither Inv nor *b_p_e* contain indexed variables. Therefore, Equal_except_at(SB, $\mathcal{I}$(Inv)(env_3), $\mathcal{I}$(Inv)(env_4)) and Equal_except_at(SB, $\mathcal{I}$(*b_p_e*)(env_3), $\mathcal{I}$(*b_p_e*)(env_4)). By the induction hypothesis and Lemma 4.33, Equal_except_at(SB, $\mathcal{I}$(SS)(env_3), $\mathcal{I}$(SS)(env_4)). Because MFP(Λ_W) is a fixed point of Λ_W, MFP(Λ_W) is defined at env_3 if and only if MFP(Λ_W) is defined at env_4, and Equal_except_at(SB, MFP(Λ_W)(env_3), MFP(Λ_W)(env_4)); we are finished with the **loop while** statement.

Let S1 be the statement "**whenever** Br_Cd **do** SS **end whenever**". If ξ_h is an indexed variable that occurs in Br_Cd, then $h \notin$ SN. Hence, because Equal_except_at(SB, env_1, env_2) and SB $\subseteq$ SN, $\mathcal{I}$(Br_Cd)(env_1) = $\mathcal{I}$(Br_Cd)(env_2).

Case $\mathcal{I}$(*Br_Cd*)(*env*$_1$) = **false**. Then $\mathcal{I}$(Br_Cd)(env_2) = **false**. Hence, we have $\mathcal{I}$(S1)(env_1) = env_1 and $\mathcal{I}$(S1)(env_2) = env_2. Therefore, Equal_except_at(SB, $\mathcal{I}$(S1)(env_1), $\mathcal{I}$(S1)(env_2)).

Case $\mathcal{I}$(*Br_Cd*)(*env*$_1$) = **true**. Then $\mathcal{I}$(Br_Cd)(env_2) = **true**. Hence, we have $\mathcal{I}$(S1)(env_1) = $\mathcal{I}$(SS)(env_1) and $\mathcal{I}$(S1)(env_2) = $\mathcal{I}$(SS)(env_2). By the induction hypothesis, if ST1 is a statement

of SS, then Equal_except_at(SB, $\mathcal{I}$(ST1)(env$_1$), $\mathcal{I}$(ST1)(env$_2$)). By Lemma 4.33, Equal_except_at(SB, $\mathcal{I}$(SS)(env$_1$), $\mathcal{I}$(SS)(env$_2$)). Therefore, Equal_except_at(SB, $\mathcal{I}$(S1)(env$_1$), $\mathcal{I}$(S1)(env$_2$)).

The induction step and the proof are now complete. □

The negative-branch-condition independence lemma is a consequence of Lemma 4.35, which we are able to prove by induction on the number of rule applications.

Lemma 4.35 *Let $\mathcal{R}$ be an intermediate result obtained by applying the proof rules in the math direction to a programmer-written program. Furthermore, let $\mathcal{R}$ have the following form:*

$$\mathcal{R} \stackrel{\text{def}}{=} C\backslash \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textbf{whenever } \text{Br\_Cd } \textbf{do} \\ \quad \textit{guarded\_code} \\ \textbf{end whenever} \\ \textit{fol\_top\_lev\_code} \end{array} \tag{4.34}$$

Let GI be the set of indices h that appear in **stow**(h) *statements anywhere in guarded_code. Let $GS \subseteq GI$. Let env_1 and env_2 be two environments. Let S1 be any statement of fol_top_lev_code. If Equal_except_at(GS, env_1, env_2) and $\neg\mathcal{I}(\text{Br\_Cd})(env_1)$, then Equal_except_at$(GS, \mathcal{I}(S1)(env_1), \mathcal{I}(S1)(env_2))$. If $h \in GI$ and ξ_h occurs in S1, then each of the following is true. S1 is not within a* **whenever** *statement. S1 is either an* **assume** *statement or a* **confirm** *statement.*

Proof. We use induction on the number of rule applications to show that every intermediate result obtained by applying the proof rules in the math direction to a

programmer-written program satisfies the lemma's conditions. The base case is when exactly one rule is applied. That would have to be the bridge rule. The result of applying the bridge rule is a three-statement program; the last statement is the only **whenever** statement. In this case, then, *fol_top_lev_code* is empty. Therefore, the lemma's statement S1 does not exist. So, the conditions of the lemma are satisfied vacuously. Therefore, any result of applying the bridge rule satisfies the lemma's conditions.

The induction step is to assume that an intermediate result satisfies the lemma's conditions and, then, to show the result of applying each rule in the math direction satisfies the lemma's conditions. The **whenever** statement of the lemma could be chosen to occur in *prec_top_lev_code* of the rule; it could be chosen to be one of the **whenever** statements explicitly mentioned in the rule; or it could be chosen to occur in *fol_top_lev_code* of the rule. In the case that the **whenever** statement of the lemma occurs in *fol_top_lev_code* of the rule, the induction hypothesis ensures that the result of the rule's application also satisfies the lemma's conditions—because the rule makes no changes to *fol_top_lev_code*. Therefore, as we examine each rule in turn, we need to consider just two cases:

1. the **whenever** statement of the lemma occurs in *prec_top_lev_code* of the rule, and

2. the **whenever** statement of the lemma is one of the **whenever** statements explicitly mentioned in the rule.

In the sequel, we refer to these cases by number. Now we examine each rule in turn.

By the induction hypothesis, $\mathcal{P}$ of the rule for **assume** satisfies the lemma's conditions. $\mathcal{M}$ is derived from $\mathcal{P}$ by replacing one **whenever** statement with an **assume** statement and a **whenever** statement. Let ST1 stand for the **assume** statement: **assume** (Br_Cd) $\Rightarrow$ (H). Let ST2 stand for the **whenever** statement.

Case 1. We must show that both ST1 and ST2 satisfy the conditions the lemma places on S1. The statement "**assume** H" of $\mathcal{P}$ is within a **whenever** statement; therefore, by the induction hypothesis, if ξ_h occurs in H, then $h \notin$ GI. Because it appears as the condition of a **whenever** statement, and not in an **assume** or **confirm** statement, by the induction hypothesis, if ξ_h occurs in Br_Cd, then $h \notin$ GI. Therefore, if ξ_h occurs in ST1, then $h \notin$ GI. Suppose env_1 and env_2 satisfy the lemma's assumptions. By Lemma 4.34, Equal_except_at(GS, $\mathcal{I}$(ST1)(env_1), $\mathcal{I}$(ST1)(env_2)). Hence, the lemma's conditions are satisfied for ST1. Because ST2 is derived from a **whenever** statement of $\mathcal{P}$ by removing one **assume** statement, by the induction hypothesis, if ξ_h occurs in ST2, then $h \notin$ GI. By Lemma 4.34, Equal_except_at(GS, $\mathcal{I}$(ST2)(env_1), $\mathcal{I}$(ST2)(env_2)). Hence, the lemma's conditions are satisfied for ST2. Therefore, all the lemma's conditions are satisfied for this case.

Case 2. The set of indices that appear in **stow**s in ST2 is the same as the set for the **whenever** statement of $\mathcal{P}$ from which it was derived. So, by the induction hypothesis, all the lemma's conditions are satisfied for this case. Hence, $\mathcal{M}$ of the rule for **assume** satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for procedure call satisfies the lemma's conditions. $\mathcal{M}$ is derived from $\mathcal{P}$ by replacing one **whenever** statement with a **confirm** statement, an **alter all** statement, a **stow**(j) statement, and a **whenever** statement. Let ST1 stand for the **confirm** statement: **confirm** (Br_Cd) $\Rightarrow$ (pre[x $\rightsquigarrow$ ac_i, y $\rightsquigarrow$ ad_i, z $\rightsquigarrow$ z_i]). Let ST2 stand for the **whenever** statement.

Case 1. We must show that ST1, **alter all**, **stow**(j), and ST2 satisfy the conditions the lemma places on S1. By reasons given in the preceding paragraph, if ξ_h occurs in Br_Cd, then $h \notin$ GI. The preconditions and postconditions of procedures do not contain indexed variables. In particular, pre contains no indexed variables. Hence, all variables in pre[x $\rightsquigarrow$ ac_i, y $\rightsquigarrow$ ad_i, z $\rightsquigarrow$ z_i] are indexed by i. Because **stow**(i) is not within a **whenever** statement of *prec_top_lev_code*, $i \notin$ GI. Therefore, if ξ_h occurs in ST1, then $h \notin$ GI. ST2 is derived from a **whenever** statement of $\mathcal{P}$ by replacing a procedure call and a **stow**(j) statement with an **assume** statement all of whose variables are indexed by either i or j. Because **stow**(j) is not within a **whenever** statement of *prec_top_lev_code*, $j \notin$ GI. Because $\{i, j\} \cap \mathrm{GI} = \emptyset$, by the induction hypothesis, if ξ_h occurs in ST2, then $h \notin$ GI. Neither **alter all** nor **stow**(j) contains any indexed variables. Let S1 be any of ST1, **alter all**, **stow**(j), or ST2. Then, if ξ_h occurs in S1, then $h \notin$ GI. By Lemma 4.34, Equal_except_at(GS, $\mathcal{I}$(S1)(env$_1$), $\mathcal{I}$(S1)(env$_2$)). Therefore, all the lemma's conditions are satisfied for this case.

Case 2. The set of indices that appear in **stow**s in ST2 is a subset of the set for the **whenever** statement of $\mathcal{P}$ from which it was derived. So, by the induction hypothe-

sis and the definition of subset, all the lemma's conditions are satisfied for this case. Hence, $\mathcal{M}$ of the rule for procedure call satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for selection in the absence of an **else** clause satisfies the lemma's conditions. $\mathcal{M}$ is derived from $\mathcal{P}$ by replacing one **whenever** statement with eight statements. Let WE1 stand for the first, and WE2 for the second, of the **whenever** statements. Let SU1 stand for the first, and SU2 for the second, of the **assume** statements.

Case 1. We must show that all eight statements satisfy the conditions the lemma places on S1. We do so by showing that if S1 is one of these eight statements, then, if ξ_h occurs in S1, then $h \notin$ GI. By Lemma 4.34, Equal_except_at(GS, $\mathcal{I}$(S1)(env$_1$), $\mathcal{I}$(S1)(env$_2$)). Therefore, all the lemma's conditions would be satisfied for this case. Let S1 be one of **stow**(j), **stow**(n), or **alter all**. Then, because S1 contains no indexed variables, we have, vacuously, that, if ξ_h occurs in S1, then $h \notin$ GI. Because it appears as the condition of a **whenever** statement, and not in an **assume** or **confirm** statement, by the induction hypothesis, if ξ_h occurs in Br_Cd, then $h \notin$ GI. Boolean program expressions, in particular *b_p_e*, do not contain indexed variables. So all the variables of MExp(*b_p_e*)[y $\rightsquigarrow$ y$_i$] are indexed by i. Because none of **stow**(i), **stow**(j), **stow**(k), or **stow**(n) occur in *prec_top_lev_code*, $\{i, j, k, n\} \cap$ GI $= \emptyset$. All variables of the statement "**assume** x$_j$ = x$_i$" are indexed by i or j. All other parts of WE1 are derived from statements within a selection statement, which, in turn, is within a **whenever** statement of $\mathcal{P}$. For these reasons and the induction hypothesis, if ξ_h occurs in WE1, then $h \notin$ GI. WE2 is derived

from a **whenever** statement of $\mathcal{P}$ by removing a selection statement and a **stow**(n) statement. So, by the induction hypothesis, if ξ_h occurs in WE2, then $h \notin$ GI. Let S1 be one of SU1 or SU2. Then, because SU1 and SU2 are composed of Br_Cd, MExp(b_p_e)[y $\rightsquigarrow$ y_i], and, respectively, $x_n = x_k$ and $x_n = x_i$, if ξ_h occurs in S1, then $h \notin$ GI. We have checked all eight statements. Therefore, all the lemma's conditions are satisfied for this case.

Case 2. We divide this case into two subcases: (a) WE1 is the **whenever** statement of the lemma and (b) WE2 is the **whenever** statement of the lemma.

Case 2(a). The set, GW1, of indices that appear in **stow**s in WE1 is a subset of the set for the **whenever** statement of $\mathcal{P}$ that contains the selection statement from which WE1 was derived. So, by the induction hypothesis and the definition of subset, each statement of *fol_top_lev_code* satisfies the lemma's conditions for WE1. By Lemma 4.22, if ξ_h occurs in Br_Cd, then $h \notin$ GW1. By Lemma 4.23, if ξ_h occurs in *cd_suffix*, then $h \notin$ GW1. Hence, if ξ_h occurs in WE2, then $h \notin$ GW1. By Lemma 4.34, Equal_except_at(GS, $\mathcal{I}$(WE2)(env$_1$), $\mathcal{I}$(WE2)(env$_2$)), where GS $\subseteq$ GW1. Therefore, WE2 satisfies the lemma's conditions for WE1. Because they have no indexed variables, **alter all** and **stow**(n) both satisfy the lemma's conditions for WE1. If ξ_h occurs in SU2, then $h \notin$ GW1. So, SU2 satisfies the lemma's conditions for WE1. We are left to consider SU1. Note that $k \in$ GW1, and there are values of ξ (current variable names) such that ξ_k occurs in SU1. We must show, therefore, that SU1 is not within a **whenever**

statement; it is not. We must also show that SU1 is either an **assume** statement or a **confirm** statement; it is an **assume** statement. Finally, suppose GS $\subseteq$ GW1; Equal_except_at(GS, env_1, env_2); and $\neg\mathcal{I}((\text{Br\_Cd}) \wedge (\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i]))(\text{env}_1)$. We must show Equal_except_at(GS, $\mathcal{I}(\text{SU1})(\text{env}_1), \mathcal{I}(\text{SU1})(\text{env}_2)$). We noted above that if ξ_h occurs in Br_Cd, then $h \notin$ GW1. So, because $i \notin$ GW1, if ξ_h occurs in (Br_Cd) $\wedge$ (MExp(b_p_e)[y $\rightsquigarrow$ y_i]), then $h \notin$ GW1. Therefore, $\mathcal{I}((\text{Br\_Cd}) \wedge (\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i]))(\text{env}_2) = \mathcal{I}((\text{Br\_Cd}) \wedge (\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i]))(\text{env}_1) =$ **false**. So, $\mathcal{I}(\text{SU1})(\text{env}_1) = \text{env}_1$ and $\mathcal{I}(\text{SU1})(\text{env}_2) = \text{env}_2$. Therefore, Equal_except_at(GS, $\mathcal{I}(\text{SU1})(\text{env}_1), \mathcal{I}(\text{SU1})(\text{env}_2)$), and we are done.

Case 2(b). The set, GW2, of indices that appear in **stow**s in WE2 (i.e., the **stow**s of *cd_suffix*) is a subset of the set for the **whenever** statement of $\mathcal{P}$ from which WE2 was derived. So, by the induction hypothesis and the definition of subset, each statement of *fol_top_lev_code* satisfies the lemma's conditions for WE2.

Hence, $\mathcal{M}$ of the rule for selection in the absence of an **else** clause satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for selection in the presence of an **else** clause satisfies the lemma's conditions. $\mathcal{M}$ is derived from $\mathcal{P}$ by replacing one **whenever** statement with eleven statements. Let WE1 stand for the first, WE2 for the second, and WE3 for the third, of the **whenever** statements. Let SU1 stand for the first, and SU2 for the second, of the **assume** statements.

Case 1. We must show that all eleven statements satisfy the conditions the lemma places on S1. We do so by showing that if S1 is one of these

eleven statements, then, if ξ_h occurs in S1, then $h \notin$ GI. By Lemma 4.34, Equal_except_at(GS, $\mathcal{I}$(S1)(env$_1$), $\mathcal{I}$(S1)(env$_2$)). Therefore, all the lemma's conditions would be satisfied for this case. Let S1 be one of **stow**(j), **stow**(l), **stow**(n), or **alter all**. Then, because S1 contains no indexed variables, we have, vacuously, that, if ξ_h occurs in S1, then $h \notin$ GI. Because it appears as the condition of a **whenever** statement, and not in an **assume** or **confirm** statement, by the induction hypothesis, if ξ_h occurs in Br_Cd, then $h \notin$ GI. Boolean program expressions, in particular *b_p_e*, do not contain indexed variables. So all the variables of MExp(*b_p_e*)[y $\rightsquigarrow$ y$_i$] are indexed by i. Because none of **stow**(i), **stow**(j), **stow**(k), **stow**(l), **stow**(m), or **stow**(n) occur in *prec_top_lev_code*, $\{i, j, k, l, m, n\} \cap$ GI $= \emptyset$. All variables of the statement "**assume** x$_j$ = x$_i$" are indexed by i or j. All other parts of WE1 are derived from statements within a selection statement, which, in turn, is within a **whenever** statement of $\mathcal{P}$. For these reasons and the induction hypothesis, if ξ_h occurs in WE1, then $h \notin$ GI. All variables of the statement "**assume** x$_l$ = x$_i$" are indexed by i or l. All other parts of WE2 are derived from statements within a selection statement, which, in turn, is within a **whenever** statement of $\mathcal{P}$. For these reasons and the induction hypothesis, if ξ_h occurs in WE2, then $h \notin$ GI. WE3 is derived from a **whenever** statement of $\mathcal{P}$ by removing a selection statement and a **stow**(n) statement. So, by the induction hypothesis, if ξ_h occurs in WE3, then $h \notin$ GI. Let S1 be one of SU1 or SU2. Then, because SU1 and SU2 are composed of Br_Cd, MExp(*b_p_e*)[y $\rightsquigarrow$ y$_i$], and, respectively, x$_n$ = x$_k$ and x$_n$ = x$_m$, if ξ_h occurs in S1, then $h \notin$ GI. We have checked all eleven statements. Therefore, all the lemma's

conditions are satisfied for this case.

Case 2. We divide this case into three subcases: (a) WE1 is the **whenever** statement of the lemma; (b) WE2 is the **whenever** statement of the lemma; and (c) WE3 is the **whenever** statement of the lemma.

Case 2(a). The set, GW1, of indices that appear in **stow**s in WE1 is a subset of the set for the **whenever** statement of $\mathcal{P}$ that contains the selection statement from which WE1 was derived. So, by the induction hypothesis and the definition of subset, each statement of *fol_top_lev_code* satisfies the lemma's conditions for WE1. By Lemma 4.22, if ξ_h occurs in Br_Cd, then $h \notin$ GW1. By Lemma 4.23, if ξ_h occurs in *cd_suffix*, then $h \notin$ GW1. Hence, if ξ_h occurs in WE3, then $h \notin$ GW1. By Lemma 4.34, Equal_except_at(GS, $\mathcal{I}$(WE3)(env$_1$), $\mathcal{I}$(WE3)(env$_2$)), where GS $\subseteq$ GW1. Therefore, WE3 satisfies the lemma's conditions for WE1. If ξ_h occurs in (Br_Cd) $\wedge \neg$(MExp(b_p_e)[y $\rightsquigarrow$ y$_i$]), then $h \notin$ GW1. By the syntactic restriction that indexes in **stow**s are everywhere increasing and Lemma 4.23, if ξ_h occurs within WE2, then $h \notin$ GW1. Hence, if ξ_h occurs in WE2, then $h \notin$ GW1. By Lemma 4.34, Equal_except_at(GS, $\mathcal{I}$(WE2)(env$_1$), $\mathcal{I}$(WE2)(env$_2$)), where GS $\subseteq$ GW1. Therefore, WE2 satisfies the lemma's conditions for WE1. Because they have no indexed variables, **alter all**, **stow**(l), and **stow**(n) all satisfy the lemma's conditions for WE1. If ξ_h occurs in SU2, then $h \notin$ GW1. So, SU2 satisfies the lemma's conditions for WE1. We are left to consider SU1. Note that $k \in$ GW1, and there are values of ξ (current variable names) such that ξ_k occurs in SU1. We must show, therefore, that SU1 is not within a **whenever**

statement; it is not. We must also show that SU1 is either an **assume** statement or a **confirm** statement; it is an **assume** statement. Finally, suppose GS $\subseteq$ GW1; Equal_except_at(GS, env_1, env_2); and $\neg\mathcal{I}((\text{Br\_Cd}) \wedge (\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i]))(\text{env}_1)$. We must show Equal_except_at(GS, $\mathcal{I}(\text{SU1})(\text{env}_1), \mathcal{I}(\text{SU1})(\text{env}_2)$). We noted above that if ξ_h occurs in Br_Cd, then $h \notin$ GW1. So, because $i \notin$ GW1, if ξ_h occurs in (Br_Cd) $\wedge$ (MExp(b_p_e)[y $\rightsquigarrow$ y_i]), then $h \notin$ GW1. Therefore, $\mathcal{I}((\text{Br\_Cd}) \wedge (\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i]))(\text{env}_2) = \mathcal{I}((\text{Br\_Cd}) \wedge (\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i]))(\text{env}_1) =$ **false**. So, $\mathcal{I}(\text{SU1})(\text{env}_1) = \text{env}_1$ and $\mathcal{I}(\text{SU1})(\text{env}_2) = \text{env}_2$. Therefore, Equal_except_at(GS, $\mathcal{I}(\text{SU1})(\text{env}_1), \mathcal{I}(\text{SU1})(\text{env}_2)$), and we are done.

Case 2(b). The set, GW2, of indices that appear in **stow**s in WE2 is a subset of the set for the **whenever** statement of $\mathcal{P}$ that contains the selection statement from which WE2 was derived. So, by the induction hypothesis and the definition of subset, each statement of *fol_top_lev_code* satisfies the lemma's conditions for WE2. By Lemma 4.22, if ξ_h occurs in Br_Cd, then $h \notin$ GW2. By Lemma 4.23, if ξ_h occurs in *cd_suffix*, then $h \notin$ GW2. Hence, if ξ_h occurs in WE3, then $h \notin$ GW2. By Lemma 4.34, Equal_except_at(GS, $\mathcal{I}(\text{WE3})(\text{env}_1), \mathcal{I}(\text{WE3})(\text{env}_2)$), where GS $\subseteq$ GW2. Therefore, WE3 satisfies the lemma's conditions for WE2. Because they have no indexed variables, **alter all** and **stow**(n) both satisfy the lemma's conditions for WE2. If ξ_h occurs in SU1, then $h \notin$ GW2. So, SU1 satisfies the lemma's conditions for WE2. We are left to consider SU2. Note that $m \in$ GW2, and there are values of ξ (current variable names) such that ξ_m occurs in SU2. We must show, therefore, that SU2 is not within a **whenever**

statement; it is not. We must also show that SU2 is either an **assume** statement or a **confirm** statement; it is an **assume** statement. Finally, suppose GS $\subseteq$ GW2; Equal_except_at(GS, env_1, env_2); and $\neg\mathcal{I}((\text{Br\_Cd}) \wedge \neg(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i]))(\text{env}_1)$. We must show Equal_except_at(GS, $\mathcal{I}(\text{SU2})(\text{env}_1), \mathcal{I}(\text{SU2})(\text{env}_2)$). We noted above that if ξ_h occurs in Br_Cd, then $h \notin$ GW2. So, because $i \notin$ GW2, if ξ_h occurs in (Br_Cd) $\wedge \neg(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i])$, then $h \notin$ GW2. Therefore, $\mathcal{I}((\text{Br\_Cd}) \wedge \neg(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i]))(\text{env}_2) = \mathcal{I}((\text{Br\_Cd}) \wedge \neg(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i]))(\text{env}_1) =$ **false**. So, $\mathcal{I}(\text{SU2})(\text{env}_1) = \text{env}_1$ and $\mathcal{I}(\text{SU2})(\text{env}_2) = \text{env}_2$. Therefore, Equal_except_at(GS, $\mathcal{I}(\text{SU2})(\text{env}_1), \mathcal{I}(\text{SU2})(\text{env}_2)$), and we are done.

Case 2(c). The set, GW3, of indices that appear in **stow**s in WE3 (i.e., the **stow**s of *cd_suffix*) is a subset of the set for the **whenever** statement of $\mathcal{P}$ from which WE3 was derived. So, by the induction hypothesis and the definition of subset, each statement of *fol_top_lev_code* satisfies the lemma's conditions for WE3.

Hence, $\mathcal{M}$ of the rule for selection in the presence of an **else** clause satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the **loop while** rule satisfies the lemma's conditions. $\mathcal{M}$ is derived from $\mathcal{P}$ by replacing one **whenever** statement with seven statements. Let WE1 stand for the first, and WE2 for the second, of the **whenever** statements.

Case 1. We must show that all seven statements satisfy the conditions the lemma places on S1. We do so by showing that if S1 is one of these seven statements, then, if ξ_h occurs in S1, then $h \notin$ GI. By Lemma 4.34,

Equal_except_at(GS, $\mathcal{I}$(S1)(env$_1$), $\mathcal{I}$(S1)(env$_2$)). Therefore, all the lemma's conditions would be satisfied for this case. Let S1 be one of **stow**(j), **stow**(l), or **alter all**. Then, because S1 contains no indexed variables, we have, vacuously, that, if ξ_h occurs in S1, then $h \notin$ GI. Because it appears as the condition of a **whenever** statement, and not in an **assume** or **confirm** statement, by the induction hypothesis, if ξ_h occurs in Br_Cd, then $h \notin$ GI. Boolean program expressions, in particular *b_p_e*, do not contain indexed variables. So all the variables of MExp(*b_p_e*)[y $\rightsquigarrow$ y$_i$] are indexed by i. Because none of **stow**(i), **stow**(j), **stow**(k), or **stow**(l) occur in *prec_top_lev_code*, $\{i, j, k, l\} \cap$ GI $= \emptyset$. All variables of the statement "**assume** (MExp(*b_p_e*)[y $\rightsquigarrow$ y$_j$]) $\wedge$ (Inv[x $\rightsquigarrow$ x$_j$, #x $\rightsquigarrow$ x$_i$])" are indexed by i or j. All variables of the statement "**confirm** Inv[x $\rightsquigarrow$ x$_k$, #x $\rightsquigarrow$ x$_i$]" are indexed by i or k. All other parts of WE1 are derived from statements within a **loop while** statement, which, in turn, is within a **whenever** statement of $\mathcal{P}$. For these reasons and the induction hypothesis, if ξ_h occurs in WE1, then $h \notin$ GI. WE2 is derived from a **whenever** statement of $\mathcal{P}$ by removing a **loop while** statement and a **stow**(l) statement, and inserting an **assume** statement. All variables of the statement "**assume** ($\neg$(MExp(*b_p_e*)[y $\rightsquigarrow$ y$_l$])) $\wedge$ (Inv[x $\rightsquigarrow$ x$_l$, #x $\rightsquigarrow$ x$_i$])" are indexed by i or l. By this fact and the induction hypothesis, if ξ_h occurs in WE2, then $h \notin$ GI. Because if ξ_h occurs in Br_Cd, then $h \notin$ GI, and because all variables of the expression "(Inv[x $\rightsquigarrow$ x$_i$, #x $\rightsquigarrow$ x$_i$])" are indexed by i, if ξ_h occurs in "**confirm** (Br_Cd) $\Rightarrow$ (Inv[x $\rightsquigarrow$ x$_i$, #x $\rightsquigarrow$ x$_i$])", then $h \notin$ GI. We have checked all seven statements. Therefore, all the lemma's conditions are satisfied for this case.

Case 2. We divide this case into two subcases: (a) WE1 is the **whenever** statement of the lemma and (b) WE2 is the **whenever** statement of the lemma.

Case 2(a). The set, GW1, of indices that appear in **stow**s in WE1 is a subset of the set for the **whenever** statement of $\mathcal{P}$ that contains the **loop while** statement from which WE1 was derived. So, by the induction hypothesis and the definition of subset, each statement of *fol_top_lev_code* satisfies the lemma's conditions for WE1. By Lemma 4.22, if ξ_h occurs in Br_Cd, then $h \notin$ GW1. By Lemma 4.23, if ξ_h occurs in *cd_suffix*, then $h \notin$ GW1. $\{i, l\} \cap \text{GW1} = \emptyset$. Hence, if ξ_h occurs in WE2, then $h \notin$ GW1. By Lemma 4.34, Equal_except_at(GS, $\mathcal{I}$(WE2)(env$_1$), $\mathcal{I}$(WE2)(env$_2$)), where GS $\subseteq$ GW1. Therefore, WE2 satisfies the lemma's conditions for WE1. Because they have no indexed variables, **alter all** and **stow**(l) both satisfy the lemma's conditions for WE1. So, we have finished this case.

Case 2(b). The set, GW2, of indices that appear in **stow**s in WE2 (i.e., the **stow**s of *cd_suffix*) is a subset of the set for the **whenever** statement of $\mathcal{P}$ from which WE2 was derived. So, by the induction hypothesis and the definition of subset, each statement of *fol_top_lev_code* satisfies the lemma's conditions for WE2.

Hence, $\mathcal{M}$ of the **loop while** rule satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for **confirm** satisfies the lemma's conditions. $\mathcal{M}$ is derived from $\mathcal{P}$ by replacing one **whenever** statement with a **confirm** statement and a **whenever** statement. Let ST1 stand for the **confirm** statement: **confirm** (Br_Cd) $\Rightarrow$ (H[x $\rightsquigarrow$ x$_i$]). Let ST2 stand for the **whenever** statement.

Case 1. We must show that both ST1 and ST2 satisfy the conditions the lemma

places on S1. The statement "**confirm** $H\lfloor \mathrm{x} \rfloor$" of $\mathcal{P}$ is within a **whenever** statement; therefore, by the induction hypothesis, if ξ_h occurs in H, then $h \notin \mathrm{GI}$. Every free current variable name in H is replaced—in $H[\mathrm{x} \rightsquigarrow \mathrm{x}_i]$—by a variable subscripted with i. By the syntactic restriction that the indices appearing in **stow**s are everywhere increasing, $i \notin \mathrm{GI}$. Because it appears as the condition of a **whenever** statement, and not in an **assume** or **confirm** statement, by the induction hypothesis, if ξ_h occurs in Br_Cd, then $h \notin \mathrm{GI}$. Therefore, if ξ_h occurs in ST1, then $h \notin \mathrm{GI}$. Suppose env_1 and env_2 satisfy the lemma's assumptions. By Lemma 4.34, $\mathrm{Equal\_except\_at}(\mathrm{GS}, \mathcal{I}(\mathrm{ST1})(\mathrm{env}_1), \mathcal{I}(\mathrm{ST1})(\mathrm{env}_2))$. Hence, the lemma's conditions are satisfied for ST1. Because ST2 is derived from a **whenever** statement of $\mathcal{P}$ by removing one **confirm** statement, by the induction hypothesis, if ξ_h occurs in ST2, then $h \notin \mathrm{GI}$. By Lemma 4.34, $\mathrm{Equal\_except\_at}(\mathrm{GS}, \mathcal{I}(\mathrm{ST2})(\mathrm{env}_1), \mathcal{I}(\mathrm{ST2})(\mathrm{env}_2))$. Hence, the lemma's conditions are satisfied for ST2. Therefore, all the lemma's conditions are satisfied for this case.

Case 2. The set of indices that appear in **stow**s in ST2 is the same as the set for the **whenever** statement of $\mathcal{P}$ from which it was derived. So, by the induction hypothesis, all the lemma's conditions are satisfied for this case.

Hence, $\mathcal{M}$ of the rule for **confirm** satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for empty guarded blocks satisfies the lemma's conditions.

Case 1. Every statement of $\mathcal{M}$ is a statement of $\mathcal{P}$. Therefore, by the induction hypothesis, all the lemma's conditions are satisfied for this case.

Case 2. Because there are no **whenever** statements explicitly mentioned in $\mathcal{M}$, all the lemma's conditions are satisfied (vacuously) for this case.

Hence, $\mathcal{M}$ of the rule for empty guarded blocks satisfies the lemma's conditions.

Due to its additional syntactic restriction, neither $\mathcal{P}$ nor $\mathcal{M}$ of the rule for **alter all** contains any **whenever** statements. Hence, $\mathcal{M}$ of the rule for **alter all** (vacuously) satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for consecutive **assume** statements satisfies the lemma's conditions.

Case 1. Given the lemma's hypotheses, we have Equal_except_at(GS, $\mathcal{I}$(**assume** H_1)(env$_1$), $\mathcal{I}$(**assume** H_1)(env$_2$)) and Equal_except_at(GS, $\mathcal{I}$(**assume** H_2)(env$_1$), $\mathcal{I}$(**assume** H_2)(env$_2$)). If AE(env$_1$) $\neq$ NL, then Equal_except_at(GS, $\mathcal{I}$(**assume** $(H_1) \wedge (H_2)$)(env$_1$), $\mathcal{I}$(**assume** $(H_1) \wedge (H_2)$)(env$_2$)). So, we suppose AE(env$_1$) = NL. If AE($\mathcal{I}$(**assume** H_1)(env$_1$)) $\neq$ NL, then $\mathcal{I}(H_1)$(env$_1$) = $\mathcal{I}(H_1)$(env$_2$) = **false**. In this case, $\mathcal{I}$(**assume** $(H_1) \wedge (H_2)$)(env$_1$) equals env$_1$ everywhere except at the assert status where AE($\mathcal{I}$(**assume** $(H_1) \wedge (H_2)$)(env$_1$)) = VT. The same is true for env$_2$. Hence, Equal_except_at(GS, $\mathcal{I}$(**assume** $(H_1) \wedge (H_2)$)(env$_1$), $\mathcal{I}$(**assume** $(H_1) \wedge (H_2)$)(env$_2$)). So, we suppose AE($\mathcal{I}$(**assume** H_1)(env$_1$)) = NL. Then $\mathcal{I}(H_1)$(env$_1$) = $\mathcal{I}(H_1)$(env$_2$) = **true**. Hence, $\mathcal{I}$(**assume** $(H_1) \wedge (H_2)$)(env$_1$) = $\mathcal{I}$(**assume** H_2)(env$_1$) and $\mathcal{I}$(**assume** $(H_1) \wedge (H_2)$)(env$_2$) = $\mathcal{I}$(**assume** H_2)(env$_2$). Therefore, Equal_except_at(GS, $\mathcal{I}$(**assume** $(H_1) \wedge (H_2)$)(env$_1$), $\mathcal{I}$(**assume** $(H_1) \wedge (H_2)$)(env$_2$)). The statement "**assume** $(H_1) \wedge (H_2)$" is an **assume** statement. By the induction

hypothesis, it is not within a **whenever** statement. Therefore, all the lemma's conditions are satisfied for this case.

Case 2. Because there are no **whenever** statements explicitly mentioned in $\mathcal{M}$, all the lemma's conditions are satisfied (vacuously) for this case.

Hence, $\mathcal{M}$ of the rule for consecutive **assume** statements satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the **assume-confirm** rule satisfies the lemma's conditions.

Case 1. Given the lemma's hypotheses, we have Equal_except_at(GS, $\mathcal{I}$(**assume** H_1)(env$_1$), $\mathcal{I}$(**assume** H_1)(env$_2$)) and Equal_except_at(GS, $\mathcal{I}$(**confirm** H_2)(env$_1$), $\mathcal{I}$(**confirm** H_2)(env$_2$)). If AE(env$_1$) $\neq$ NL, then Equal_except_at(GS, $\mathcal{I}$(**confirm** $(H_1) \Rightarrow (H_2)$)(env$_1$), $\mathcal{I}$(**confirm** $(H_1) \Rightarrow (H_2)$)(env$_2$)). So, we suppose AE(env$_1$) = NL. If AE($\mathcal{I}$(**assume** H_1)(env$_1$)) $\neq$ NL, then $\mathcal{I}(H_1)$(env$_1$) = $\mathcal{I}(H_1)$(env$_2$) = **false**. In this case, $\mathcal{I}$(**confirm** $(H_1) \Rightarrow (H_2)$)(env$_1$) = env$_1$ and $\mathcal{I}$(**confirm** $(H_1) \Rightarrow (H_2)$)(env$_2$) = env$_2$. Hence, Equal_except_at(GS, $\mathcal{I}$(**confirm** $(H_1) \Rightarrow (H_2)$)(env$_1$), $\mathcal{I}$(**confirm** $(H_1) \Rightarrow (H_2)$)(env$_2$)). So, we suppose AE($\mathcal{I}$(**assume** H_1)(env$_1$)) = NL. Then $\mathcal{I}(H_1)$(env$_1$) = $\mathcal{I}(H_1)$(env$_2$) = **true**. Hence, $\mathcal{I}$(**confirm** $(H_1) \Rightarrow (H_2)$)(env$_1$) = $\mathcal{I}$(**confirm** H_2)(env$_1$) and $\mathcal{I}$(**confirm** $(H_1) \Rightarrow (H_2)$)(env$_2$) = $\mathcal{I}$(**confirm** H_2)(env$_2$). Therefore, Equal_except_at(GS, $\mathcal{I}$(**confirm** $(H_1) \Rightarrow (H_2)$)(env$_1$), $\mathcal{I}$(**confirm** $(H_1) \Rightarrow (H_2)$)(env$_2$)). The statement "**confirm** $(H_1) \Rightarrow (H_2)$" is a **confirm** statement. By the induction hypothesis, it is not within a **whenever** statement. Therefore, all the

lemma's conditions are satisfied for this case.

Case 2. Because there are no **whenever** statements explicitly mentioned in $\mathcal{M}$, all the lemma's conditions are satisfied (vacuously) for this case.

Hence, $\mathcal{M}$ of the **assume-confirm** rule satisfies the lemma's conditions.

By the induction hypothesis, $\mathcal{P}$ of the rule for consecutive **confirm** statements satisfies the lemma's conditions.

Case 1. Given the lemma's hypotheses, we have Equal_except_at(GS, $\mathcal{I}$(**confirm** H_1)(env$_1$), $\mathcal{I}$(**confirm** H_1)(env$_2$)) and Equal_except_at(GS, $\mathcal{I}$(**confirm** H_2)(env$_1$), $\mathcal{I}$(**confirm** H_2)(env$_2$)). If AE(env$_1$) $\neq$ NL, then Equal_except_at(GS, $\mathcal{I}$(**confirm** $(H_1) \wedge (H_2)$)(env$_1$), $\mathcal{I}$(**confirm** $(H_1) \wedge (H_2)$)(env$_2$)). So, we suppose AE(env$_1$) = NL. If AE($\mathcal{I}$(**confirm** H_1)(env$_1$)) $\neq$ NL, then $\mathcal{I}(H_1)$(env$_1$) = $\mathcal{I}(H_1)$(env$_2$) = **false**. In this case, $\mathcal{I}$(**confirm** $(H_1) \wedge (H_2)$)(env$_1$) equals env$_1$ everywhere except at the assert status where AE($\mathcal{I}$(**confirm** $(H_1) \wedge (H_2)$)(env$_1$)) = CF. The same is true for env$_2$. Hence, Equal_except_at(GS, $\mathcal{I}$(**confirm** $(H_1) \wedge (H_2)$)(env$_1$), $\mathcal{I}$(**confirm** $(H_1) \wedge (H_2)$)(env$_2$)). So, we suppose AE($\mathcal{I}$(**confirm** H_1)(env$_1$)) = NL. Then $\mathcal{I}(H_1)$(env$_1$) = $\mathcal{I}(H_1)$(env$_2$) = **true**. Hence, $\mathcal{I}$(**confirm** $(H_1) \wedge (H_2)$)(env$_1$) = $\mathcal{I}$(**confirm** H_2)(env$_1$) and $\mathcal{I}$(**confirm** $(H_1) \wedge (H_2)$)(env$_2$) = $\mathcal{I}$(**confirm** H_2)(env$_2$). Therefore, Equal_except_at(GS, $\mathcal{I}$(**confirm** $(H_1) \wedge (H_2)$)(env$_1$), $\mathcal{I}$(**confirm** $(H_1) \wedge (H_2)$)(env$_2$)). The statement "**confirm** $(H_1) \wedge (H_2)$" is a **confirm** statement. By the induction hypothesis, it is not within a **whenever** statement. Therefore, all the lemma's conditions are satisfied for this case.

Case 2. Because there are no **whenever** statements explicitly mentioned in $\mathcal{M}$, all the lemma's conditions are satisfied (vacuously) for this case.

Hence, $\mathcal{M}$ of the rule for consecutive **confirm** statements satisfies the lemma's conditions.

The induction step is now complete. Hence the proof of this lemma (4.35) is finished. □

Now we are ready to state and prove the negative-branch-condition independence lemma.

Lemma 4.36 (Negative-Branch-Condition Independence) *Let $\mathcal{R}$ be an intermediate result obtained by applying the proof rules in the math direction to a programmer-written program. Furthermore, let $\mathcal{R}$ have the following form:*

$$\mathcal{R} \stackrel{\text{def}}{=} C\backslash \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textbf{whenever } \text{Br\_Cd } \textbf{do} \\ \quad \textit{guarded\_code} \\ \textbf{end whenever} \\ \textit{fol\_top\_lev\_code} \end{array} \tag{4.35}$$

Let GI be the set of indices h that appear in **stow**(h) *statements anywhere in guarded_code. Let $GS \subseteq GI$. Let env_1 and env_2 be two environments. If Equal_except_at(GS, env_1, env_2) and $\neg\mathcal{I}(\text{Br\_Cd})(env_1)$, then Equal_except_at$(GS, \mathcal{I}(fol_top_lev_code)(env_1), \mathcal{I}(fol_top_lev_code)(env_2))$.*

Proof. Given the hypotheses of this lemma, we must show $\text{Equal\_except\_at}(\text{GS}, \mathcal{I}(fol\_top\_lev\_code)(\text{env}_1), \mathcal{I}(fol\_top\_lev\_code)(\text{env}_2))$. If it were always true that *fol_top_lev_code* contained no variables indexed by a member

of the set GI, an appeal to Lemmas 4.34 and 4.33 would complete our proof. But *fol_top_lev_code* can contain variables indexed by members of the set GI. However, we will show that the different states of those indices can have no effect on the interpretation of *fol_top_lev_code*. The reason is that any occurrence in *fol_top_lev_code* of a variable indexed by a member of GI must be in an assertion in such a way that it is "guarded" by Br_Cd. For example, occurrences in *fol_top_lev_code* of a variable indexed by a member of GI are frequently in the *consequent* of an **assume** statement at the top level of *fol_top_lev_code* that has the form "**assume** (Br_Cd) $\Rightarrow$ (*consequent*)". This lemma (4.36) follows from Lemmas 4.35 and 4.33.

□

4.3.6 Internal-Index Independence

The internal-index independence lemma is key for our ability to establish invalidity preservation for the rules that handle compound statements such as **loop while** and selection. In contrast with the negative-branch-condition independence lemma, the internal-index independence lemma does not require the hypothesis that Br_Cd evaluates as false. The other important difference is that, here, we permit differences only on the set of indices h that appear in **stow**(h) statements within the compound statement.

Lemma 4.37 (Internal-Index Independence) *Let $\mathcal{R}$ be an intermediate result obtained by applying the proof rules to a programmer-written program toward the goal of obtaining an assertion. Furthermore, let $\mathcal{R}$ have the following form:*

$$\mathcal{R} \stackrel{\text{def}}{=} C\backslash \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textbf{whenever } \text{Br\_Cd } \textbf{do} \\ \quad \textit{compound\_stmt} \\ \quad \textbf{stow}(n) \\ \quad \textit{cd\_suffix} \\ \textbf{end whenever} \\ \textit{fol\_top\_lev\_code} \end{array} \tag{4.36}$$

Let CI be the set of indices h that appear in **stow**(h) *statements within compound_stmt. Let env_1 and env_2 be two environments. If Equal_except_at(CI, env_1, env_2), then Equal_except_at$(CI, \mathcal{I}(cd_suffix)(env_1), \mathcal{I}(cd_suffix)(env_2))$. Furthermore, Equal_except_at$(CI, \mathcal{I}(fol_top_lev_code)(env_1), \mathcal{I}(fol_top_lev_code)(env_2))$.*

Proof. Let S1 be any statement of *cd_suffix* or *fol_top_lev_code*. By Lemma 4.23, if indexed variable ξ_h occurs in S1, $h \notin$ CI. By Lemma 4.34, Equal_except_at$(\text{CI}, \mathcal{I}(\text{S1})(\text{env}_1), \mathcal{I}(\text{S1})(\text{env}_2))$. By Lemma 4.33, Equal_except_at$(\text{CI}, \mathcal{I}(cd_suffix)(\text{env}_1), \mathcal{I}(cd_suffix)(\text{env}_2))$ and Equal_except_at$(\text{CI}, \mathcal{I}(fol_top_lev_code)(\text{env}_1), \mathcal{I}(fol_top_lev_code)(\text{env}_2))$. □

4.4 Soundness Lemmas

In this section we establish Lemma 4.3 by proving a series of lemmas, one for each proof rule of Chapter III. Each lemma states that the rule preserves invalidity in the math direction. Skepticism may be greatest about the **if-then-else** statement;

therefore, we present its lemma and proof first. The proof is more complicated than the others. Readers wishing a gentler introduction are invited to skip ahead to some shorter proofs before returning here.

Lemma 4.38 *The rule for selection in the presence of an* **else** *clause preserves invalidity in the math direction.*

Proof. We assume that $\mathcal{P}$ is invalid and show that $\mathcal{M}$ is invalid. Let $\mathrm{env}_I^{\mathcal{P}}$ be a witness to $\mathcal{P}$'s invalidity, and let $\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\mathrm{env}_I^{\mathcal{P}})$; i.e., $\mathrm{AE}(\mathrm{env}_I^{\mathcal{P}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$. We will show the existence of an environment, $\mathrm{env}_I^{\mathcal{M}}$, such that $\mathrm{AE}(\mathrm{env}_I^{\mathcal{M}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$, where $\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\mathrm{env}_I^{\mathcal{M}})$. The proof is organized by cases. First we make the following two definitions.

$$\mathrm{env}_i^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\,ACseq_0)(\mathrm{env}_I^{\mathcal{P}}) \tag{4.37}$$

$$\mathrm{env}_i^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\,ACseq_0)(\mathrm{env}_I^{\mathcal{M}}) \tag{4.38}$$

We will be defining other environments as well. Figure 77 shows the positions of the environments defined for $\mathcal{P}$, and figure 78 shows the positions of the environments defined for $\mathcal{M}$.

Case $AE(env_i^{\mathcal{P}}) = CF$. In this case, we take $\mathrm{env}_I^{\mathcal{M}} = \mathrm{env}_I^{\mathcal{P}}$. Then $\mathrm{AE}(\mathrm{env}_i^{\mathcal{M}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$.

Case $AE(env_i^{\mathcal{P}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_i^{\mathcal{P}}) = \mathrm{NL}$. We may, for this case, write $\mathrm{env}_i^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}]$, where $\mathrm{ns}(i) = \mathrm{cs}_i$. With the help of Lemmas 4.20 and 4.27, we may also write $\mathrm{env}_I^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. As shown in Figure 77, we let $\mathrm{env}_{\mathrm{ew}}^{\mathcal{P}} \stackrel{\mathrm{def}}{=}$

$\mathcal{P} \stackrel{\text{def}}{=} C\backslash$ $\quad \text{env}_I^{\mathcal{P}}$

prec_top_lev_code
alter all
stow(i)
$ACseq_0$
$\quad \text{env}_i^{\mathcal{P}}$
whenever Br_Cd **do**
  if b_p_e **then**
    stow(j)
    cd_kern_1
    stow(k)
    $ACseq_1$
  else
    stow(l)
    cd_kern_2
    stow(m)
    $ACseq_2$
  end if
  stow(n)
  $\quad \text{env}_n^{\mathcal{P}}$
  cd_suffix
end whenever
$\quad \text{env}_{\text{ew}}^{\mathcal{P}}$
fol_top_lev_code
$\quad \text{env}_F^{\mathcal{P}}$

Figure 77: Environments Defined for $\mathcal{P}$ of the Rule for Selection in the Presence of an **else** Clause

$\mathcal{M} \stackrel{\text{def}}{=} C\backslash$ $\quad \text{env}_I^{\mathcal{M}}$

prec_top_lev_code
alter all
stow(i)
$ACseq_0$
$\quad \text{env}_i^{\mathcal{M}}$

alter all
stow(j)
$\quad \text{env}_j^{\mathcal{M}}$

whenever (Br_Cd) $\wedge$ (MExp(b_p_e)[y $\rightsquigarrow$ y_i]) **do**
  assume $\text{x}_j = \text{x}_i$
  cd_kern_1
  stow(k)
  $ACseq_1$
end whenever
alter all
stow(l)
$\quad \text{env}_l^{\mathcal{M}}$

whenever (Br_Cd) $\wedge \neg$(MExp(b_p_e)[y $\rightsquigarrow$ y_i]) **do**
  assume $\text{x}_l = \text{x}_i$
  cd_kern_2
  stow(m)
  $ACseq_2$
end whenever
alter all
stow(n)
$\quad \text{env}_n^{\mathcal{M}}$

assume ((Br_Cd) $\wedge$ (MExp(b_p_e)[y $\rightsquigarrow$ y_i])) $\Rightarrow$ ($\text{x}_n = \text{x}_k$)
assume ((Br_Cd) $\wedge \neg$(MExp(b_p_e)[y $\rightsquigarrow$ y_i])) $\Rightarrow$ ($\text{x}_n = \text{x}_m$)
whenever Br_Cd **do**
  cd_suffix
end whenever
$\quad \text{env}_{\text{ew}}^{\mathcal{M}}$

fol_top_lev_code
$\quad \text{env}_F^{\mathcal{M}}$

Figure 78: Environments Defined for $\mathcal{M}$ of the Rule for Selection in the Presence of an **else** Clause

$\mathcal{I}$(**whenever** Br_Cd **do** ... **end whenever**)(env$_i^{\mathcal{P}}$).

Case $\neg\mathcal{I}(Br\_Cd)(env_i^{\mathcal{P}})$. (We are still inside case AE(env$_i^{\mathcal{P}}$) $\neq$ CF, i.e., AE(env$_i^{\mathcal{P}}$) = NL.) In this case, env$_{\rm ew}^{\mathcal{P}}$ = env$_i^{\mathcal{P}}$, and we take env$_I^{\mathcal{M}}$ = [NL, cs$_I$, ns$_I$, se$_I$ $\circ$ $\langle$cs$_i$, cs$_i$, cs$_i\rangle$ $\circ$ se, os, d]. Then env$_i^{\mathcal{M}}$ = [NL, cs$_i$, ns, $\langle$cs$_i$, cs$_i$, cs$_i\rangle$ $\circ$ se, os, d]. As shown in Figure 78, we let env$_{\rm ew}^{\mathcal{M}}$ $\stackrel{\rm def}{=}$ $\mathcal{I}$(**alter all stow**(j) ... **whenever** Br_Cd **do** *cd_suffix* **end whenever**)(env$_i^{\mathcal{M}}$). By Lemma 4.22, the value of $\mathcal{I}$(Br_Cd)(env) depends only on ISE(env) for any environment env. So, because ISE(env$_i^{\mathcal{M}}$) = ISE(env$_i^{\mathcal{P}}$), we have $\neg\mathcal{I}$(Br_Cd)(env$_i^{\mathcal{M}}$). Therefore, we have another equality for env$_{\rm ew}^{\mathcal{M}}$, namely,

$$\text{env}_{\rm ew}^{\mathcal{M}} = \mathcal{I}\left(\begin{array}{l}\textbf{alter all stow}(j)\\ \textbf{alter all stow}(l)\\ \textbf{alter all stow}(n)\end{array}\right)(\text{env}_i^{\mathcal{M}}). \tag{4.39}$$

Therefore, env$_{\rm ew}^{\mathcal{M}}$ = [NL, cs$_i$, ns$'$, se, os, d] where

$$\text{ns}'(h) = \begin{cases}\text{ns}(h) & \text{if } h \notin \{j,l,n\}\\ \text{cs}_i & \text{if } h \in \{j,l,n\}\end{cases}. \tag{4.40}$$

So, Equal_except_at($\{j,l,n\}$, env$_{\rm ew}^{\mathcal{P}}$, env$_{\rm ew}^{\mathcal{M}}$). The set $\{j,l,n\}$ is a subset of the set of indices h that appear in **stow**(h) statements anywhere within the **whenever** statement explicitly shown in the schema $\mathcal{P}$. In this case, by Lemma 4.36, Equal_except_at($\{j,l,n\}$, $\mathcal{I}$(*fol_top_lev_code*)(env$_{\rm ew}^{\mathcal{P}}$), $\mathcal{I}$(*fol_top_lev_code*)(env$_{\rm ew}^{\mathcal{M}}$)). Hence, Equal_except_at($\{j,l,n\}$, env$_F^{\mathcal{P}}$, env$_F^{\mathcal{M}}$). Therefore, AE(env$_F^{\mathcal{M}}$) = AE(env$_F^{\mathcal{P}}$) = CF.

Case $\mathcal{I}(Br\_Cd)(env_i^{\mathcal{P}})$. (We are still inside case AE(env$_i^{\mathcal{P}}$) $\neq$ CF, i.e., AE(env$_i^{\mathcal{P}}$) = NL.) In this case, env$_{\rm ew}^{\mathcal{P}}$ = $\mathcal{I}$(**if** *b_p_e* **then** ... **end if stow**(n) *cd_suffix*)(env$_i^{\mathcal{P}}$). Let env$_n^{\mathcal{P}}$ $\stackrel{\rm def}{=}$ $\mathcal{I}$(**if** *b_p_e* **then** ... **end if stow**(n))(env$_i^{\mathcal{P}}$). In this case,

we take $\mathrm{env}_I^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \langle \mathrm{cs}_i, \mathrm{cs}_i, \mathrm{CSE}(\mathrm{env}_n^{\mathcal{P}})\rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. Then $\mathrm{env}_i^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}, \langle \mathrm{cs}_i, \mathrm{cs}_i, \mathrm{CSE}(\mathrm{env}_n^{\mathcal{P}})\rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. We define $\mathrm{env}_j^{\mathcal{M}}$, $\mathrm{env}_l^{\mathcal{M}}$, and $\mathrm{env}_n^{\mathcal{M}}$ as follows.

$$\mathrm{env}_j^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{alter all stow}(j))(\mathrm{env}_i^{\mathcal{M}}) \tag{4.41}$$

$$\mathrm{env}_l^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}\left(\begin{array}{l}\textbf{whenever}\\ (\mathrm{Br\_Cd}) \wedge (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])\ \textbf{do}\\ \ldots\\ \textbf{end whenever alter all stow}(l)\end{array}\right)(\mathrm{env}_j^{\mathcal{M}}) \tag{4.42}$$

$$\mathrm{env}_n^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}\left(\begin{array}{l}\textbf{whenever}\\ (\mathrm{Br\_Cd}) \wedge \neg(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])\ \textbf{do}\\ \ldots\\ \textbf{end whenever alter all stow}(n)\end{array}\right)(\mathrm{env}_l^{\mathcal{M}}) \tag{4.43}$$

Case $\mathcal{I}(\mathrm{b\_p\_e})(env_i^{\mathcal{P}})$. In this case, $\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])(\mathrm{env}_i^{\mathcal{P}})$. In each of the following environments that we defined above, namely, $\mathrm{env}_i^{\mathcal{M}}$, $\mathrm{env}_j^{\mathcal{M}}$, $\mathrm{env}_l^{\mathcal{M}}$, and $\mathrm{env}_n^{\mathcal{M}}$, we have, for $h \in \{i, j, l, n\}$, $\mathrm{ISE}(\mathrm{env}_h^{\mathcal{M}})(i) = \mathrm{ns}(i)$. Therefore, we also have, for $h \in \{i, j, l, n\}$, $\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])(\mathrm{env}_h^{\mathcal{M}})$. According to the semantics of **if-then-else**, $\mathrm{env}_n^{\mathcal{P}} = \mathcal{I}(\textbf{stow}(j)\ cd\_kern_1\ \textbf{stow}(k)\ ACseq_1\ \textbf{stow}(n))(\mathrm{env}_i^{\mathcal{P}})$. According to the semantics of **whenever**, $\mathrm{env}_l^{\mathcal{M}} = \mathcal{I}(\textbf{assume}\ \mathrm{x}_j = \mathrm{x}_i\ cd\_kern_1\ \textbf{stow}(k)\ ACseq_1\ \textbf{alter all stow}(l))(\mathrm{env}_j^{\mathcal{M}})$. Because $\mathrm{ISE}(\mathrm{env}_j^{\mathcal{M}})(j) = \mathrm{cs}_i = \mathrm{ISE}(\mathrm{env}_i^{\mathcal{M}})(i) = \mathrm{ISE}(\mathrm{env}_j^{\mathcal{M}})(i)$, $\mathrm{env}_j^{\mathcal{M}} = \mathcal{I}(\textbf{assume}\ \mathrm{x}_j = \mathrm{x}_i)(\mathrm{env}_j^{\mathcal{M}})$, so $\mathrm{env}_l^{\mathcal{M}} = \mathcal{I}(cd\_kern_1\ \textbf{stow}(k)\ ACseq_1\ \textbf{alter all stow}(l))(\mathrm{env}_j^{\mathcal{M}})$. Because $\mathrm{CSE}(\mathrm{env}_j^{\mathcal{M}}) = \mathrm{cs}_i = \mathrm{CSE}(\mathrm{env}_i^{\mathcal{P}})$, by Lemma 4.32, $\mathrm{AE}(\mathrm{env}_l^{\mathcal{M}}) = \mathrm{AE}(\mathrm{env}_n^{\mathcal{P}})$, $\mathrm{CSE}(\mathrm{env}_l^{\mathcal{M}}) = \mathrm{CSE}(\mathrm{env}_n^{\mathcal{P}})$, $\mathrm{OSE}(\mathrm{env}_l^{\mathcal{M}}) = \mathrm{OSE}(\mathrm{env}_n^{\mathcal{P}})$, and $\mathrm{DME}(\mathrm{env}_l^{\mathcal{M}}) = \mathrm{DME}(\mathrm{env}_n^{\mathcal{P}})$.

Case $AE(env_n^{\mathcal{P}}) = CF$. Then, by Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{AE}(\mathrm{env}_l^{\mathcal{M}}) = \mathrm{AE}(\mathrm{env}_n^{\mathcal{P}}) = \mathrm{CF}$.

Case $AE(env_n^{\mathcal{P}}) \neq CF$. By Lemma 4.19, $\mathrm{AE}(\mathrm{env}_n^{\mathcal{P}}) = \mathrm{NL}$. Hence, $\mathrm{AE}(\mathrm{env}_l^{\mathcal{M}}) = \mathrm{NL}$.

In this case (i.e., $\mathcal{I}(b\_p\_e)(\mathrm{env}_i^{\mathcal{P}})$), according to the semantics of **whenever**, $\mathrm{env}_n^{\mathcal{M}} = \mathcal{I}(\textbf{alter all stow}(n))(\mathrm{env}_l^{\mathcal{M}})$. Therefore, $\mathrm{AE}(\mathrm{env}_n^{\mathcal{M}}) = \mathrm{NL}$ and, due to the setup, $\mathrm{CSE}(\mathrm{env}_n^{\mathcal{M}}) = \mathrm{CSE}(\mathrm{env}_n^{\mathcal{P}})$. Furthermore,

$$\begin{aligned} \mathrm{ISE}(\mathrm{env}_n^{\mathcal{M}})(k) &= \mathrm{ISE}(\mathrm{env}_l^{\mathcal{M}})(k) && (4.44)\\ &= \mathrm{ISE}(\mathrm{env}_n^{\mathcal{P}})(k) && (4.45)\\ &= \mathrm{CSE}(\mathrm{env}_n^{\mathcal{P}}) && (4.46) \end{aligned}$$

and

$$\begin{aligned} \mathrm{ISE}(\mathrm{env}_n^{\mathcal{M}})(n) &= \mathrm{CSE}(\mathrm{env}_n^{\mathcal{M}}) && (4.47)\\ &= \mathrm{CSE}(\mathrm{env}_n^{\mathcal{P}}) && (4.48)\\ &= \mathrm{ISE}(\mathrm{env}_n^{\mathcal{M}})(k). && (4.49) \end{aligned}$$

We now have the following equalities for $\mathrm{env}_F^{\mathcal{P}}$ and $\mathrm{env}_F^{\mathcal{M}}$.

$$\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\textit{cd\_suffix fol\_top\_lev\_code})(\mathrm{env}_n^{\mathcal{P}}) \quad (4.50)$$

$$\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}\left(\begin{array}{l} \textbf{assume } ((\mathrm{Br\_Cd}) \wedge \\ \quad (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])) \Rightarrow (\mathrm{x}_n = \mathrm{x}_k) \\ \textbf{assume } ((\mathrm{Br\_Cd}) \wedge \\ \quad \neg(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])) \Rightarrow (\mathrm{x}_n = \mathrm{x}_m) \\ \textbf{whenever } \mathrm{Br\_Cd} \textbf{ do} \\ \quad \textit{cd\_suffix} \\ \textbf{end whenever} \\ \textit{fol\_top\_lev\_code} \end{array}\right)(\mathrm{env}_n^{\mathcal{M}}) \quad (4.51)$$

Because $\mathrm{ISE}(\mathrm{env}_n^{\mathcal{M}})(k) = \mathrm{ISE}(\mathrm{env}_n^{\mathcal{M}})(n)$, $\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])(\mathrm{env}_n^{\mathcal{M}})$, and $\mathcal{I}(\mathrm{Br\_Cd})(\mathrm{env}_n^{\mathcal{M}})$, we have

$$\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\textit{cd\_suffix fol\_top\_lev\_code})(\mathrm{env}_n^{\mathcal{M}}). \quad (4.52)$$

The only place $\text{env}_n^{\mathcal{M}}$ and $\text{env}_n^{\mathcal{P}}$ differ is at $\text{ISE}(\text{env}_n^{\mathcal{M}})(l)$. That is to say, $\text{Equal\_except\_at}(\{l\}, \text{env}_n^{\mathcal{P}}, \text{env}_n^{\mathcal{M}})$. By two applications of Lemma 4.37, $\text{Equal\_except\_at}(\{l\}, \text{env}_F^{\mathcal{P}}, \text{env}_F^{\mathcal{M}})$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$.

Case $\neg\mathcal{I}(\text{b\_p\_e})(env_i^{\mathcal{P}})$. In this case, $\neg\mathcal{I}(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i])(\text{env}_i^{\mathcal{P}})$. We also have, for $h \in \{i, j, l, n\}$, $\neg\mathcal{I}(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i])(\text{env}_h^{\mathcal{M}})$. According to the semantics of **if-then-else**, $\text{env}_n^{\mathcal{P}} = \mathcal{I}(\textbf{stow}(l)\ cd\_kern_2\ \textbf{stow}(m)\ ACseq_2\ \textbf{stow}(n))(\text{env}_i^{\mathcal{P}})$. According to the semantics of **whenever**, $\text{env}_l^{\mathcal{M}} = \mathcal{I}(\textbf{alter all stow}(l))(\text{env}_j^{\mathcal{M}})$. So $\text{AE}(\text{env}_l^{\mathcal{M}}) = \text{AE}(\text{env}_j^{\mathcal{M}}) = \text{AE}(\text{env}_i^{\mathcal{M}}) = \text{NL}$, and $\text{CSE}(\text{env}_l^{\mathcal{M}}) = \text{cs}_i = \text{CSE}(\text{env}_i^{\mathcal{P}})$. According to the semantics of **whenever**, $\text{env}_n^{\mathcal{M}} = \mathcal{I}(\textbf{assume}\ \text{x}_l = \text{x}_i\ cd\_kern_2\ \textbf{stow}(m)\ ACseq_2\ \textbf{alter all stow}(n))(\text{env}_l^{\mathcal{M}})$. Because $\text{ISE}(\text{env}_l^{\mathcal{M}})(l) = \text{cs}_i = \text{ISE}(\text{env}_i^{\mathcal{M}})(i) = \text{ISE}(\text{env}_l^{\mathcal{M}})(i)$, $\text{env}_l^{\mathcal{M}} = \mathcal{I}(\textbf{assume}\ \text{x}_l = \text{x}_i)(\text{env}_l^{\mathcal{M}})$, so $\text{env}_n^{\mathcal{M}} = \mathcal{I}(cd\_kern_2\ \textbf{stow}(m)\ ACseq_2\ \textbf{alter all stow}(n))(\text{env}_l^{\mathcal{M}})$. Because $\text{CSE}(\text{env}_l^{\mathcal{M}}) = \text{cs}_i = \text{CSE}(\text{env}_i^{\mathcal{P}})$, by Lemma 4.32, $\text{AE}(\text{env}_n^{\mathcal{M}}) = \text{AE}(\text{env}_n^{\mathcal{P}})$, $\text{OSE}(\text{env}_n^{\mathcal{M}}) = \text{OSE}(\text{env}_n^{\mathcal{P}})$, and $\text{DME}(\text{env}_n^{\mathcal{M}}) = \text{DME}(\text{env}_n^{\mathcal{P}})$.

Case $AE(env_n^{\mathcal{P}}) = CF$. Then, by Lemma 4.15, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{AE}(\text{env}_n^{\mathcal{M}}) = \text{AE}(\text{env}_n^{\mathcal{P}}) = \text{CF}$.

Case $AE(env_n^{\mathcal{P}}) \neq CF$. By Lemma 4.19, $\text{AE}(\text{env}_n^{\mathcal{P}}) = \text{NL}$. Hence, $\text{AE}(\text{env}_l^{\mathcal{M}}) = \text{NL}$. Furthermore, due to the setup, $\text{CSE}(\text{env}_n^{\mathcal{M}}) = \text{CSE}(\text{env}_n^{\mathcal{P}})$. We have the following sequence of equations.

$$\begin{aligned} \text{ISE}(\text{env}_n^{\mathcal{M}})(m) &= \text{CSE}(\text{env}_n^{\mathcal{P}}) && (4.53)\\ &= \text{CSE}(\text{env}_n^{\mathcal{M}}) && (4.54)\\ &= \text{ISE}(\text{env}_n^{\mathcal{M}})(n) && (4.55) \end{aligned}$$

Equations 4.50 and 4.51 still hold for $\text{env}_F^{\mathcal{P}}$ and $\text{env}_F^{\mathcal{M}}$. Because $\text{ISE}(\text{env}_n^{\mathcal{M}})(m) = \text{ISE}(\text{env}_n^{\mathcal{M}})(n)$, $\mathcal{I}(\neg(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i]))(\text{env}_n^{\mathcal{M}})$, and $\mathcal{I}(\text{Br\_Cd})(\text{env}_n^{\mathcal{M}})$, we have

$$\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textit{cd\_suffix fol\_top\_lev\_code})(\text{env}_n^{\mathcal{M}}). \tag{4.56}$$

The only place $\text{env}_n^{\mathcal{M}}$ and $\text{env}_n^{\mathcal{P}}$ differ is at $\text{ISE}(\text{env}_n^{\mathcal{M}})(j)$. That is to say, $\text{Equal\_except\_at}(\{j\}, \text{env}_n^{\mathcal{P}}, \text{env}_n^{\mathcal{M}})$. By two applications of Lemma 4.37, $\text{Equal\_except\_at}(\{j\}, \text{env}_F^{\mathcal{P}}, \text{env}_F^{\mathcal{M}})$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. □

Lemma 4.39 *The rule for selection in the absence of an* **else** *clause preserves invalidity in the math direction.*

Proof. The proof is similar to the proof that the rule for selection in the presence of an **else** clause preserves invalidity in the math direction. The important difference is in the case in which $\neg\mathcal{I}(b\_p\_e)(\text{env}_i^{\mathcal{P}})$ (within the cases $\text{AE}(\text{env}_i^{\mathcal{P}}) \neq \text{CF}$ and $\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{P}})$). We supply here only the argument for this case. The environments defined in the complete proof are similar to those defined for the rule for selection in the presence of an **else** clause. Figure 79 shows the positions of the environments defined for $\mathcal{P}$, and figure 80 shows the positions of the environments defined for $\mathcal{M}$. Because $\mathcal{M}$ of the rule for selection in the absence of an **else** clause introduces two—and not three—**alter all** statements, we insert two—and not three—states in the setup. We take $\text{env}_I^{\mathcal{M}} = [\text{NL}, \text{cs}_I, \text{ns}_I, \text{se}_I \circ \langle \text{cs}_i, \text{CSE}(\text{env}_n^{\mathcal{P}})\rangle \circ \text{se}, \text{os}, \text{d}]$. Then $\text{env}_i^{\mathcal{M}} = [\text{NL}, \text{cs}_i, \text{ns}, \langle \text{cs}_i, \text{CSE}(\text{env}_n^{\mathcal{P}})\rangle \circ \text{se}, \text{os}, \text{d}]$.

Case $\neg\mathcal{I}(\text{b\_p\_e})(\textit{env}_i^{\mathcal{P}})$. In this case, $\neg\mathcal{I}(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i])(\text{env}_i^{\mathcal{P}})$. We also have, for $h \in \{i, j, n\}$, $\neg\mathcal{I}(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i])(\text{env}_h^{\mathcal{M}})$. According to the semantics of

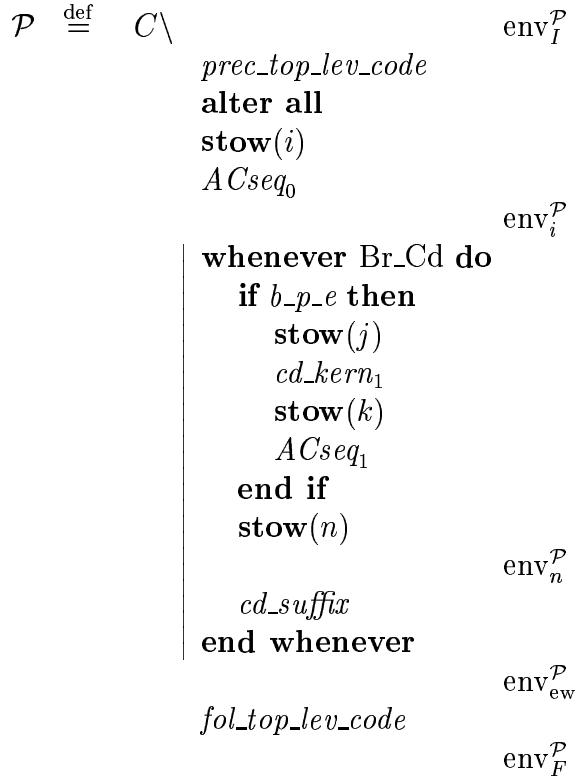

Figure 79: Environments Defined for $\mathcal{P}$ of the Rule for Selection in the Absence of an **else** Clause

$\mathcal{M} \stackrel{\text{def}}{=} C\backslash$ $\quad \text{env}_I^{\mathcal{M}}$

prec_top_lev_code
alter all
stow(i)
$ACseq_0$
$\quad \text{env}_i^{\mathcal{M}}$

| **alter all**
| **stow**(j)
| $\quad \text{env}_j^{\mathcal{M}}$
| **whenever** (Br_Cd) $\wedge$ (MExp(b_p_e)[y $\rightsquigarrow$ y_i]) **do**
| **assume** $\text{x}_j = \text{x}_i$
| cd_kern_1
| **stow**(k)
| $ACseq_1$
| **end whenever**
| **alter all**
| **stow**(n)
| $\quad \text{env}_n^{\mathcal{M}}$
| **assume** ((Br_Cd) $\wedge$ (MExp(b_p_e)[y $\rightsquigarrow$ y_i])) $\Rightarrow$ ($\text{x}_n = \text{x}_k$)
| **assume** ((Br_Cd) $\wedge \neg$(MExp(b_p_e)[y $\rightsquigarrow$ y_i])) $\Rightarrow$ ($\text{x}_n = \text{x}_i$)
| **whenever** Br_Cd **do**
| *cd_suffix*
| **end whenever**
| $\quad \text{env}_{\text{ew}}^{\mathcal{M}}$

fol_top_lev_code
$\quad \text{env}_F^{\mathcal{M}}$

Figure 80: Environments Defined for $\mathcal{M}$ of the Rule for Selection in the Absence of an **else** Clause

if-then, $\text{env}_n^{\mathcal{P}} = \mathcal{I}(\textbf{stow}(n))(\text{env}_i^{\mathcal{P}})$. Therefore, Equal_except_at$(\{n\}, \text{env}_n^{\mathcal{P}}, \text{env}_i^{\mathcal{P}})$. In particular, $\text{AE}(\text{env}_n^{\mathcal{P}}) = \text{AE}(env_i^{\mathcal{P}}) = \text{NL}$. According to the semantics of **whenever**, $\text{env}_n^{\mathcal{M}} = \mathcal{I}(\textbf{alter all stow}(n))(\text{env}_j^{\mathcal{M}})$. Therefore, $\text{env}_n^{\mathcal{M}} = [\text{NL}, \text{CSE}(\text{env}_n^{\mathcal{P}}), \text{ns}_n^{\mathcal{M}}, \text{se}, \text{os}, \text{d}]$ where

$$\text{ns}_n^{\mathcal{M}}(h) = \begin{cases} \text{ns}(h) & \text{if } h \notin \{j, n\} \\ \text{cs}_i & \text{if } h = j \\ \text{CSE}(\text{env}_n^{\mathcal{P}}) & \text{if } h = n \end{cases} . \tag{4.57}$$

Hence, Equal_except_at$(\{j\}, \text{env}_n^{\mathcal{P}}, \text{env}_n^{\mathcal{M}})$. We have the following sequence of equations.

$$\begin{aligned} \text{ISE}(\text{env}_n^{\mathcal{M}})(i) &= \text{ISE}(\text{env}_n^{\mathcal{P}})(i) && (4.58) \\ &= \text{CSE}(\text{env}_i^{\mathcal{P}}) && (4.59) \\ &= \text{CSE}(\text{env}_n^{\mathcal{P}}) && (4.60) \\ &= \text{CSE}(\text{env}_n^{\mathcal{M}}) && (4.61) \\ &= \text{ISE}(\text{env}_n^{\mathcal{M}})(n) && (4.62) \end{aligned}$$

Recall that $\text{env}_F^{\mathcal{P}} = \mathcal{I}(cd\_suffix\ fol\_top\_lev\_code)(\text{env}_n^{\mathcal{P}})$. Because $\text{ISE}(\text{env}_n^{\mathcal{M}})(i) = \text{ISE}(\text{env}_n^{\mathcal{M}})(n)$, $\mathcal{I}(\neg(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i]))(\text{env}_n^{\mathcal{M}})$, and $\mathcal{I}(\text{Br\_Cd})(\text{env}_n^{\mathcal{M}})$, we have

$$\text{env}_F^{\mathcal{M}} = \mathcal{I}(cd\_suffix\ fol\_top\_lev\_code)(\text{env}_n^{\mathcal{M}}). \tag{4.63}$$

Recalling that Equal_except_at$(\{j\}, \text{env}_n^{\mathcal{P}}, \text{env}_n^{\mathcal{M}})$, by two applications of Lemma 4.37, Equal_except_at$(\{j\}, \text{env}_F^{\mathcal{P}}, \text{env}_F^{\mathcal{M}})$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. □

Lemma 4.40 *The* **loop while** *rule preserves invalidity in the math direction.*

Proof. We assume that $\mathcal{P}$ is invalid and show that $\mathcal{M}$ is invalid. Let $\mathrm{env}_I^{\mathcal{P}}$ be a witness to $\mathcal{P}$'s invalidity, and let $\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\mathrm{env}_I^{\mathcal{P}})$; i.e., $\mathrm{AE}(\mathrm{env}_I^{\mathcal{P}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$. We will show the existence of an environment, $\mathrm{env}_I^{\mathcal{M}}$, such that $\mathrm{AE}(\mathrm{env}_I^{\mathcal{M}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$, where $\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\mathrm{env}_I^{\mathcal{M}})$. The proof is organized by cases. First we make the following two definitions.

$$\mathrm{env}_i^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\mathit{ACseq}_0)(\mathrm{env}_I^{\mathcal{P}}) \tag{4.64}$$

$$\mathrm{env}_i^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\mathit{ACseq}_0)(\mathrm{env}_I^{\mathcal{M}}) \tag{4.65}$$

We will be defining other environments as well. Figure 81 shows the positions of the environments defined for $\mathcal{P}$, and figure 82 shows the positions of the environments defined for $\mathcal{M}$.

Case $AE(env_i^{\mathcal{P}}) = CF$. In this case, we take $\mathrm{env}_I^{\mathcal{M}} = \mathrm{env}_I^{\mathcal{P}}$. Then $\mathrm{AE}(\mathrm{env}_i^{\mathcal{M}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$.

Case $AE(env_i^{\mathcal{P}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_i^{\mathcal{P}}) = \mathrm{NL}$. We may, for this case, write $\mathrm{env}_i^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}]$, where $\mathrm{ns}(i) = \mathrm{cs}_i$. With the help of Lemmas 4.20 and 4.27, we may also write $\mathrm{env}_I^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. As shown in Figure 77, we let $\mathrm{env}_{\mathrm{ew}}^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{whenever}\ \mathrm{Br\_Cd}\ \textbf{do}\ \ldots\ \textbf{end whenever})(\mathrm{env}_i^{\mathcal{P}})$.

Case $\neg\mathcal{I}(Br\_Cd)(env_i^{\mathcal{P}})$. This case is similar to the corresponding case in the proof of Lemma 4.38, and we do not repeat it here.

Case $\mathcal{I}(Br\_Cd)(env_i^{\mathcal{P}})$. (We are still inside case $\mathrm{AE}(\mathrm{env}_i^{\mathcal{P}}) \neq \mathrm{CF}$, i.e., $\mathrm{AE}(\mathrm{env}_i^{\mathcal{P}}) =$

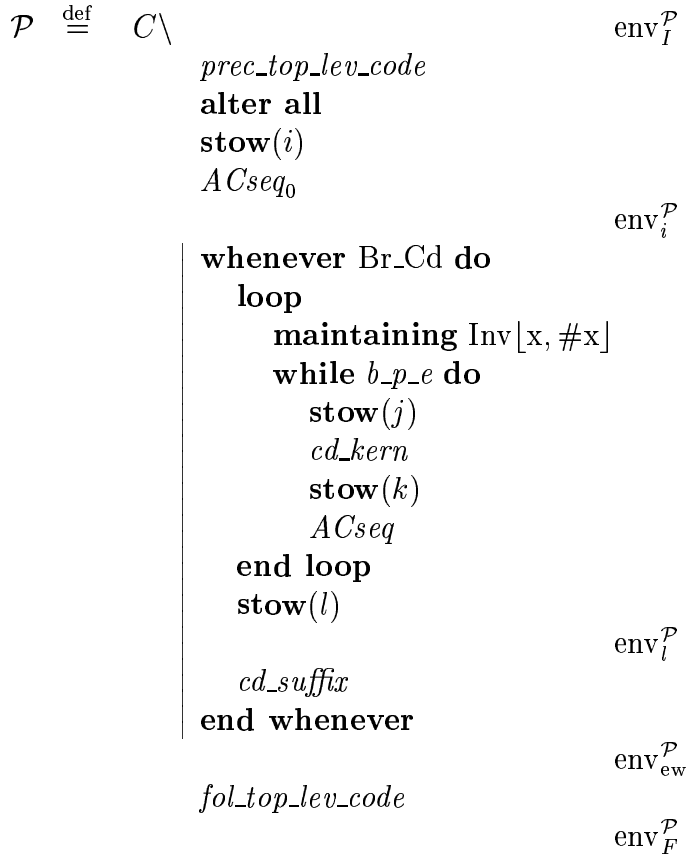

Figure 81: Environments Defined for $\mathcal{P}$ of the **loop while** Rule

$\mathcal{M} \stackrel{\text{def}}{=} C\backslash$ $\quad \text{env}_I^{\mathcal{M}}$

prec_top_lev_code
alter all
stow(i)
ACseq$_0$
$\quad \text{env}_i^{\mathcal{M}}$
confirm (Br_Cd) $\Rightarrow$ (Inv[x $\rightsquigarrow$ x$_i$, #x $\rightsquigarrow$ x$_i$])
alter all
stow(j)
$\quad \text{env}_j^{\mathcal{M}}$
whenever (Br_Cd) $\wedge$ (MExp(b_p_e)[y $\rightsquigarrow$ y$_i$]) **do**
  assume (MExp(b_p_e)[y $\rightsquigarrow$ y$_j$]) $\wedge$ (Inv[x $\rightsquigarrow$ x$_j$, #x $\rightsquigarrow$ x$_i$])
  cd_kern
  stow(k)
  ACseq
  confirm Inv[x $\rightsquigarrow$ x$_k$, #x $\rightsquigarrow$ x$_i$]
end whenever
alter all
stow(l)
$\quad \text{env}_l^{\mathcal{M}}$
whenever Br_Cd **do**
  assume ($\neg$(MExp(b_p_e)[y $\rightsquigarrow$ y$_l$])) $\wedge$ (Inv[x $\rightsquigarrow$ x$_l$, #x $\rightsquigarrow$ x$_i$])
  cd_suffix
end whenever
$\quad \text{env}_{\text{ew}}^{\mathcal{M}}$
fol_top_lev_code
$\quad \text{env}_F^{\mathcal{M}}$

Figure 82: Environments Defined for $\mathcal{M}$ of the **loop while** Rule

NL.) In this case, $\mathrm{env}_{\mathrm{ew}}^{\mathcal{P}} = \mathcal{I}(\textbf{loop}\ \ldots\ \textbf{end loop stow}(l)\ \mathit{cd\_suffix})(\mathrm{env}_i^{\mathcal{P}})$. Let $\mathrm{env}_l^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{loop}\ \ldots\ \textbf{end loop stow}(l))(\mathrm{env}_i^{\mathcal{P}})$. Hence, $\mathrm{env}_{\mathrm{ew}}^{\mathcal{P}} = \mathcal{I}(\mathit{cd\_suffix})(\mathrm{env}_l^{\mathcal{P}})$.

Case $AE(env_l^{\mathcal{P}}) = CF$.

Case $\neg\mathcal{I}(Inv[x \rightsquigarrow x_i, \#x \rightsquigarrow x_i])(env_i^{\mathcal{P}})$. In this case we take $\mathrm{env}_I^{\mathcal{M}} = \mathrm{env}_I^{\mathcal{P}}$ so that $\mathrm{env}_i^{\mathcal{M}} = \mathrm{env}_i^{\mathcal{P}}$. Then we have $\neg\mathcal{I}((\mathrm{Br\_Cd}) \Rightarrow (\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_i, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i]))(\mathrm{env}_i^{\mathcal{M}})$. Therefore, $\mathrm{AE}(\mathcal{I}(\textbf{confirm}\ (\mathrm{Br\_Cd}) \Rightarrow (\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_i, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i]))(\mathrm{env}_i^{\mathcal{M}})) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$.

Case $\mathcal{I}(Inv[x \rightsquigarrow x_i, \#x \rightsquigarrow x_i])(env_i^{\mathcal{P}})$. (We are still inside case $\mathrm{AE}(\mathrm{env}_l^{\mathcal{P}}) = \mathrm{CF}$.) Therefore, there exists an environment $\mathrm{env}_e^{\mathcal{P}} \stackrel{\mathrm{def}}{=} [\mathrm{NL}, \mathrm{cs}_e, \mathrm{ns}_e, \mathrm{se}, \mathrm{os}_e, \mathrm{d}]$ where

$$\mathrm{ns}_e(h) = \mathrm{ns}(h) \text{ if } h < j \text{ or } k < h \tag{4.66}$$

$$\mathrm{os}_e(\psi) = \begin{cases} \mathrm{cs}_i(\xi) & \text{if } \psi = \#\xi \\ \mathrm{os}(\#^{k+1}\xi) & \text{if } \psi = \#^{k+2}\xi \end{cases} \tag{4.67}$$

that is the result of executing the loop zero or more iterations beginning in environment $\mathrm{env}_i^{\mathcal{P}}$ such that $\mathcal{I}(b\_p\_e)(\mathrm{env}_e^{\mathcal{P}})$, $\mathcal{I}(\mathrm{Inv}\lfloor \mathrm{x}, \#\mathrm{x} \rfloor)(\mathrm{env}_e^{\mathcal{P}})$, and execution of the loop's body beginning in environment $\mathrm{env}_e^{\mathcal{P}}$ produces either a categorically false environment or a neutral environment in which the loop invariant is interpreted to be false. That is to say, $\mathrm{AE}(\mathcal{I}(\textbf{stow}(j)\ \mathit{cd\_kern}\ \textbf{stow}(k)\ \mathit{ACseq})(\mathrm{env}_e^{\mathcal{P}})) = \mathrm{CF}$ or $\neg\mathcal{I}(\mathrm{Inv}\lfloor \mathrm{x}, \#\mathrm{x} \rfloor)(\mathcal{I}(\textbf{stow}(j)\ \mathit{cd\_kern}\ \textbf{stow}(k)\ \mathit{ACseq})(\mathrm{env}_e^{\mathcal{P}}))$. We take $\mathrm{env}_I^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \langle \mathrm{cs}_e \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. Then $\mathrm{env}_i^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}, \langle \mathrm{cs}_e \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. Hence, $\mathcal{I}((\mathrm{Br\_Cd}) \Rightarrow (\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_i, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i]))(\mathrm{env}_i^{\mathcal{M}})$. Therefore, $\mathrm{env}_j^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_e, \mathrm{ns}_j, \mathrm{se}, \mathrm{os}, \mathrm{d}]$ where

$$\mathrm{ns}_j(h) = \begin{cases} \mathrm{ns}(h) & \text{if } h \neq j \\ \mathrm{cs}_e & \text{if } h = j \end{cases} . \tag{4.68}$$

Hence,

$$\mathrm{ns}_j(h) = \mathrm{ns}(h) \text{ if } h < j \text{ or } k < h. \tag{4.69}$$

All components except the index state and the old state of $\mathrm{env}_j^{\mathcal{M}}$ and $\mathrm{env}_e^{\mathcal{P}}$ are equal, and $\mathrm{ns}_j(h) = \mathrm{ns}_e(h)$ if $h < j$ or $k < h$. Due to the properties of $\mathrm{env}_e^{\mathcal{P}}$, and because the value of the old state has no effect on the interpretation of $\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_j, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i]$, *cd_kern* **stow**$(k)$ *ACseq*, and $\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_k, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i]$, we know the following to be true. $\mathcal{I}(\mathrm{Br\_Cd})(\mathrm{env}_j^{\mathcal{M}})$. $\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])(\mathrm{env}_j^{\mathcal{M}})$. $\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_j])(\mathrm{env}_j^{\mathcal{M}})$. $\mathcal{I}(\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_j, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i])(\mathrm{env}_j^{\mathcal{M}})$. $\mathcal{I}(\textbf{whenever } (\mathrm{Br\_Cd}) \wedge (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i]) \textbf{ do } \ldots \textbf{ end whenever})(\mathrm{env}_j^{\mathcal{M}}) = \mathcal{I}(\textbf{assume } (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_j]) \wedge (\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_j, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i])\ cd\_kern\ \textbf{stow}(k)\ ACseq\ \textbf{confirm } \mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_k, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i])(\mathrm{env}_j^{\mathcal{M}})$. $\mathrm{AE}(\mathcal{I}(\textbf{assume } (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_j]) \wedge (\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_j, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i])\ cd\_kern\ \textbf{stow}(k)\ ACseq)(\mathrm{env}_j^{\mathcal{M}})) = \mathrm{CF}$ or $\neg\mathcal{I}(\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_k, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i])(\mathcal{I}(\textbf{assume } (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_j]) \wedge (\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_j, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i])\ cd\_kern\ \textbf{stow}(k)\ ACseq)(\mathrm{env}_j^{\mathcal{M}}))$. In either case, $\mathrm{AE}(\mathcal{I}(\textbf{assume } (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_j]) \wedge (\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_j, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i])\ cd\_kern\ \textbf{stow}(k)\ ACseq\ \textbf{confirm } \mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_k, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i])(\mathrm{env}_j^{\mathcal{M}})) = \mathrm{CF}$. Therefore, $\mathrm{AE}(\mathcal{I}(\textbf{whenever } (\mathrm{Br\_Cd}) \wedge (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i]) \textbf{ do } \ldots \textbf{ end whenever})(\mathrm{env}_j^{\mathcal{M}})) = \mathrm{CF}$. Hence, by Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$.

Case $AE(env_l^{\mathcal{P}}) \neq CF$. In this case, by Lemmas 4.17 and 4.19, we have $\mathrm{env}_l^{\mathcal{P}}$ defined and $\mathrm{AE}(\mathrm{env}_l^{\mathcal{P}}) = \mathrm{NL}$. We may write $\mathrm{env}_l^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_l, \mathrm{ns}_l^{\mathcal{P}}, \mathrm{se}, \mathrm{os}, \mathrm{d}]$ where

$$\mathrm{ns}_l^{\mathcal{P}}(h) \;=\; \mathrm{ns}(h) \text{ if } h < j \text{ or } l < h \tag{4.70}$$

$$\mathrm{ns}_l^{\mathcal{P}}(l) \;=\; \mathrm{cs}_l. \tag{4.71}$$

We take $\mathrm{env}_I^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \langle \mathrm{cs}_i, \mathrm{cs}_l \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. Then $\mathrm{env}_i^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}, \langle \mathrm{cs}_i, \mathrm{cs}_l \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. Because the loop terminated in a neutral environment when interpreted in $\mathrm{env}_i^{\mathcal{P}}$, its invariant was satisfied at the beginning of each iteration. In particular it was satisfied at the beginning of the first iteration. Therefore, $\mathcal{I}(\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_i, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i])(\mathrm{env}_i^{\mathcal{M}})$. Hence, $\mathrm{env}_j^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}_j^{\mathcal{M}}, \langle \mathrm{cs}_l \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$ where $\mathrm{ns}_j^{\mathcal{M}}(h) = \mathrm{ns}(h)$ if $h \neq j$.

Case loop had at least one iteration when interpreted in $\mathrm{env}_i^{\mathcal{P}}$. In this case, $\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])(\mathrm{env}_j^{\mathcal{M}})$. By the semantics of **whenever**, $\mathrm{env}_l^{\mathcal{M}} = \mathcal{I}(\textbf{assume}\ (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_j]) \wedge (\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_j, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i])\ cd\_kern\ \textbf{stow}(k)\ ACseq\ \textbf{confirm}\ \mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_k, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i]\ \textbf{alter all stow}(l))(\mathrm{env}_j^{\mathcal{M}})$. Because the loop terminated in a neutral environment when interpreted in $\mathrm{env}_i^{\mathcal{P}}$, $\mathrm{env}_l^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_l, \mathrm{ns}_l', \mathrm{se}, \mathrm{os}, \mathrm{d}]$ where $\mathrm{ns}_l'(h) = \mathrm{ns}(h)$ if $h < j$ or $l < h$.

Case loop had zero iterations when interpreted in $\mathrm{env}_i^{\mathcal{P}}$. In this case, $\neg\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])(\mathrm{env}_j^{\mathcal{M}})$. By the semantics of **whenever**, $\mathrm{env}_l^{\mathcal{M}} = \mathcal{I}(\textbf{alter all stow}(l))(\mathrm{env}_j^{\mathcal{M}})$. Therefore, $\mathrm{env}_l^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_l, \mathrm{ns}_l'', \mathrm{se}, \mathrm{os}, \mathrm{d}]$ where $\mathrm{ns}_l''(h) = \mathrm{ns}(h)$ if $h < j$ or $l < h$.

We now return to the argument for the case $\mathrm{AE}(\mathrm{env}_l^{\mathcal{P}}) \neq \mathrm{CF}$. Whether the loop had zero or more iterations when interpreted in $\mathrm{env}_i^{\mathcal{P}}$, $\mathrm{env}_l^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_l, \mathrm{ns}_l^{\mathcal{M}}, \mathrm{se}, \mathrm{os}, \mathrm{d}]$ where

$$\mathrm{ns}_l^{\mathcal{M}}(h) \;=\; \mathrm{ns}(h) \text{ if } h < j \text{ or } l < h \tag{4.72}$$

$$\mathrm{ns}_l^{\mathcal{M}}(l) \;=\; \mathrm{cs}_l. \tag{4.73}$$

Because $\mathrm{env}^{\mathcal{P}}_{\mathrm{ew}} = \mathcal{I}(\textit{cd\_suffix})(\mathrm{env}^{\mathcal{P}}_l)$ and $\mathrm{env}^{\mathcal{P}}_F = \mathcal{I}(\textit{fol\_top\_lev\_code})(\mathrm{env}^{\mathcal{P}}_{\mathrm{ew}})$, $\mathrm{env}^{\mathcal{P}}_F = \mathcal{I}(\textit{cd\_suffix fol\_top\_lev\_code})(\mathrm{env}^{\mathcal{P}}_l)$. Because the loop terminated in a neutral environment when interpreted in $\mathrm{env}^{\mathcal{P}}_i$, $\mathcal{I}((\neg(\mathrm{MExp}(\textit{b\_p\_e})[\mathrm{y} \rightsquigarrow \mathrm{y}_l])) \wedge (\mathrm{Inv}[\mathrm{x} \rightsquigarrow \mathrm{x}_l, \#\mathrm{x} \rightsquigarrow \mathrm{x}_i]))(\mathrm{env}^{\mathcal{P}}_l)$. Therefore, $\mathrm{env}^{\mathcal{M}}_F = \mathcal{I}(\textit{cd\_suffix fol\_top\_lev\_code})(\mathrm{env}^{\mathcal{M}}_l)$. Let $A \stackrel{\mathrm{def}}{=} \{h | j \leq h \leq k\}$. Because $\mathrm{Equal\_except\_at}(A, \mathrm{env}^{\mathcal{P}}_l, \mathrm{env}^{\mathcal{M}}_l)$, two applications of Lemma 4.37 give us $\mathrm{Equal\_except\_at}(A, \mathrm{env}^{\mathcal{P}}_F, \mathrm{env}^{\mathcal{M}}_F)$. Therefore, $\mathrm{AE}(\mathrm{env}^{\mathcal{M}}_F) = \mathrm{CF}$.

□

Lemma 4.41 *The bridge rule preserves invalidity in the math direction.*

Proof. We assume that $\mathcal{P}$ is invalid and show that $\mathcal{M}$ is invalid. Let $\mathrm{env}^{\mathcal{P}}_I$ be a witness to $\mathcal{P}$'s invalidity, and let $\mathrm{env}^{\mathcal{P}}_F = \mathcal{I}(\mathcal{P})(\mathrm{env}^{\mathcal{P}}_I)$; i.e., $\mathrm{AE}(\mathrm{env}^{\mathcal{P}}_I) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}^{\mathcal{P}}_F) = \mathrm{CF}$. We will show the existence of an environment, $\mathrm{env}^{\mathcal{M}}_I$, such that $\mathrm{AE}(\mathrm{env}^{\mathcal{M}}_I) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}^{\mathcal{M}}_F) = \mathrm{CF}$, where $\mathrm{env}^{\mathcal{M}}_F = \mathcal{I}(\mathcal{M})(\mathrm{env}^{\mathcal{M}}_I)$. We may write $\mathrm{env}^{\mathcal{P}}_I = [\mathrm{NL}, \mathrm{cs}, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}]$. We take $\mathrm{env}^{\mathcal{M}}_I = [\mathrm{NL}, \mathrm{cs}, \mathrm{ns}, \langle \mathrm{cs} \rangle \circ \mathrm{se}, \mathrm{os}', \mathrm{d}]$ where

$$\mathrm{os}'(\psi) = \begin{cases} \mathrm{cs}(\xi) & \text{if } \psi = \#\xi \\ \mathrm{os}(\#^{k+1}\xi) & \text{if } \psi = \#^{k+2}\xi \end{cases} . \tag{4.74}$$

By the semantics of **whenever**,

$$\mathrm{env}^{\mathcal{M}}_F = \mathcal{I}\left(\begin{array}{l} \textbf{alter all} \\ \textbf{stow}(i) \\ \textbf{assume } \mathrm{pre}[\mathrm{x} \rightsquigarrow \mathrm{x}_i] \wedge \textbf{is\_initial}(\mathrm{z}_i) \\ \mathrm{Stows\_added}(\textit{p\_body}) \\ \textbf{stow}(j) \\ \textbf{confirm } \mathrm{post}[\#\mathrm{x} \rightsquigarrow \mathrm{x}_i, \mathrm{x} \rightsquigarrow \mathrm{x}_j] \end{array}\right)(\mathrm{env}^{\mathcal{M}}_I). \tag{4.75}$$

Let

$$\mathrm{env}^{\mathcal{M}}_j \stackrel{\mathrm{def}}{=} \mathcal{I}\left(\begin{array}{l} \textbf{alter all} \\ \textbf{stow}(i) \\ \textbf{assume } \mathrm{pre}[\mathrm{x} \rightsquigarrow \mathrm{x}_i] \wedge \textbf{is\_initial}(\mathrm{z}_i) \\ \mathrm{Stows\_added}(\textit{p\_body}) \\ \textbf{stow}(j) \end{array}\right)(\mathrm{env}^{\mathcal{M}}_I). \tag{4.76}$$

By Lemma 4.16, $\mathcal{I}(\text{pre}\lfloor \text{x} \rfloor \wedge \textbf{is\_initial}(\text{z}))(\mathcal{I}(\textbf{remember})(\text{env}_I^{\mathcal{P}}))$. Therefore, because $\text{ISE}(\mathcal{I}(\textbf{alter all stow}(i))(\text{env}_I^{\mathcal{M}}))(i) = \text{cs}$, $\mathcal{I}(\text{pre}[\text{x} \rightsquigarrow \text{x}_i] \wedge \textbf{is\_initial}(\text{z}_i))(\mathcal{I}(\textbf{alter all stow}(i))(\text{env}_I^{\mathcal{M}}))$. Note also that $\text{CSE}(\mathcal{I}(\textbf{alter all stow}(i))(\text{env}_I^{\mathcal{M}})) = \text{cs}$.

Case $AE(\mathcal{I}(\textbf{remember assume } pre\lfloor x \rfloor \wedge \textbf{is\_initial}(z)\ \text{p\_body})(env_I^{\mathcal{P}})) = CF$. Because *p_body* contains no indexed variables and the interpretation of any **stow** statement changes at most the index state of an environment, $\text{AE}(\text{env}_j^{\mathcal{M}}) = \text{CF}$. By Lemma 4.15, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$.

Case $AE(\mathcal{I}(\textbf{remember assume } pre\lfloor x \rfloor \wedge \textbf{is\_initial}(z)\ \text{p\_body})(env_I^{\mathcal{P}})) \neq CF$. In this case, by Lemma 4.19, we have $\text{AE}(\mathcal{I}(\textbf{remember assume } \text{pre}\lfloor \text{x} \rfloor \wedge \textbf{is\_initial}(\text{z})\ p\_body)(\text{env}_I^{\mathcal{P}})) = \text{NL}$. Note that $\text{ISE}(\text{env}_j^{\mathcal{M}})(i) = \text{cs}$. Because *p_body* contains no indexed variables and the interpretation of any **stow** statement changes at most the index state of an environment, $\text{CSE}(\text{env}_j^{\mathcal{M}}) = \text{CSE}(\mathcal{I}(\textbf{remember assume } \text{pre}\lfloor \text{x} \rfloor \wedge \textbf{is\_initial}(\text{z})\ p\_body)(\text{env}_I^{\mathcal{P}}))$. Note also that $\text{ISE}(\text{env}_j^{\mathcal{M}})(j) = \text{CSE}(\text{env}_j^{\mathcal{M}})$. In this case, it must be that $\neg\mathcal{I}(\text{post}\lfloor \#\text{x}, \text{x} \rfloor)(\mathcal{I}(\textbf{remember assume } \text{pre}\lfloor \text{x} \rfloor \wedge \textbf{is\_initial}(\text{z})\ p\_body)(\text{env}_I^{\mathcal{P}}))$. Because $\text{OSE}(\mathcal{I}(\textbf{remember assume } \text{pre}\lfloor \text{x} \rfloor \wedge \textbf{is\_initial}(\text{z})\ p\_body)(\text{env}_I^{\mathcal{P}}))(\#\xi) = \text{cs}(\xi)$, $\neg\mathcal{I}(\text{post}[\#\text{x} \rightsquigarrow \text{x}_i, \text{x} \rightsquigarrow \text{x}_j])(\text{env}_j^{\mathcal{M}})$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. □

Lemma 4.42 *The rule for procedure call preserves invalidity in the math direction.*

Proof. We assume that $\mathcal{P}$ is invalid and show that $\mathcal{M}$ is invalid. Let $\text{env}_I^{\mathcal{P}}$ be a witness to $\mathcal{P}$'s invalidity, and let $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\text{env}_I^{\mathcal{P}})$; i.e., $\text{AE}(\text{env}_I^{\mathcal{P}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$. We will show the existence of an environment, $\text{env}_I^{\mathcal{M}}$, such that

$\mathrm{AE}(\mathrm{env}_I^{\mathcal{M}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$, where $\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\mathrm{env}_I^{\mathcal{M}})$. The proof is organized by cases. First we make the following two definitions.

$$\mathrm{env}_i^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\mathit{ACseq}_0)(\mathrm{env}_I^{\mathcal{P}}) \tag{4.77}$$

$$\mathrm{env}_i^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\mathit{ACseq}_0)(\mathrm{env}_I^{\mathcal{M}}) \tag{4.78}$$

We will be defining other environments as well. Figure 83 shows the positions of the environments defined for $\mathcal{P}$ and for $\mathcal{M}$.

Case $AE(env_i^{\mathcal{P}}) = CF$. In this case, we take $\mathrm{env}_I^{\mathcal{M}} = \mathrm{env}_I^{\mathcal{P}}$. Then $\mathrm{AE}(\mathrm{env}_i^{\mathcal{M}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$.

Case $AE(env_i^{\mathcal{P}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_i^{\mathcal{P}}) = \mathrm{NL}$. We may, for this case, write $\mathrm{env}_i^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}]$, where $\mathrm{ns}(i) = \mathrm{cs}_i$. With the help of Lemmas 4.20 and 4.27, we may also write $\mathrm{env}_I^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. As shown in Figure 83, we let $\mathrm{env}_{\mathrm{ew}}^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{whenever}\ \mathrm{Br\_Cd}\ \textbf{do}\ \ldots\ \textbf{end whenever})(\mathrm{env}_i^{\mathcal{P}})$.

Case $\neg\mathcal{I}(Br\_Cd)(env_i^{\mathcal{P}})$. (We are still inside case $\mathrm{AE}(\mathrm{env}_i^{\mathcal{P}}) \neq \mathrm{CF}$, i.e., $\mathrm{AE}(\mathrm{env}_i^{\mathcal{P}}) = \mathrm{NL}$.) In this case, $\mathrm{env}_{\mathrm{ew}}^{\mathcal{P}} = \mathrm{env}_i^{\mathcal{P}}$, and we take $\mathrm{env}_I^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \langle \mathrm{cs}_i \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. Then $\mathrm{env}_i^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}, \langle \mathrm{cs}_i \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. Figure 83 shows our definition of $\mathrm{env}_{\mathrm{ew}}^{\mathcal{M}}$ with respect to $\mathrm{env}_i^{\mathcal{M}}$. By Lemma 4.22, the value of $\mathcal{I}(\mathrm{Br\_Cd})(\mathrm{env})$ depends only on $\mathrm{ISE}(\mathrm{env})$ for any environment env. So, because $\mathrm{ISE}(\mathrm{env}_i^{\mathcal{M}}) = \mathrm{ISE}(\mathrm{env}_i^{\mathcal{P}})$, we have $\neg\mathcal{I}(\mathrm{Br\_Cd})(\mathrm{env}_i^{\mathcal{M}})$. Therefore, we have another equality for $\mathrm{env}_{\mathrm{ew}}^{\mathcal{M}}$, namely,

$$\mathrm{env}_{\mathrm{ew}}^{\mathcal{M}} = \mathcal{I}(\textbf{alter all stow}(j))(\mathrm{env}_i^{\mathcal{M}}). \tag{4.79}$$

$\mathcal{P} \stackrel{\text{def}}{=} C\backslash$ $\quad \text{env}_I^{\mathcal{P}}$

prec_top_lev_code
alter all
stow(i)
$ACseq_0$
$\quad \text{env}_i^{\mathcal{P}}$
whenever Br_Cd **do**
  P_nm(ac, ad)
  stow(j)
$\quad \text{env}_j^{\mathcal{P}}$
  cd_suffix
end whenever
$\quad \text{env}_{\text{ew}}^{\mathcal{P}}$
fol_top_lev_code
$\quad \text{env}_F^{\mathcal{P}}$

$\mathcal{M} \stackrel{\text{def}}{=} C\backslash$ $\quad \text{env}_I^{\mathcal{M}}$

prec_top_lev_code
alter all
stow(i)
$ACseq_0$
$\quad \text{env}_i^{\mathcal{M}}$
confirm (Br_Cd) $\Rightarrow$ (pre[x $\rightsquigarrow$ ac$_i$, y $\rightsquigarrow$ ad$_i$, z $\rightsquigarrow$ z$_i$])
alter all
stow(j)
$\quad \text{env}_j^{\mathcal{M}}$
whenever Br_Cd **do**
  assume b$_j$ = b$_i$ $\wedge$ (post[#x $\rightsquigarrow$ ac$_i$, x $\rightsquigarrow$ ac$_j$,
    #y $\rightsquigarrow$ ad$_i$, y $\rightsquigarrow$ ad$_j$,
    #z $\rightsquigarrow$ z$_i$, z $\rightsquigarrow$ z$_j$])
  cd_suffix
end whenever
$\quad \text{env}_{\text{ew}}^{\mathcal{M}}$
fol_top_lev_code
$\quad \text{env}_F^{\mathcal{M}}$

Figure 83: Environments Defined for $\mathcal{P}$ and $\mathcal{M}$ of the Rule for Procedure Call

Therefore, $\mathrm{env}_{\mathrm{ew}}^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}', \mathrm{se}, \mathrm{os}, \mathrm{d}]$ where

$$\mathrm{ns}'(h) = \begin{cases} \mathrm{ns}(h) & \text{if } h \neq j \\ \mathrm{cs}_i & \text{if } h = j \end{cases} . \tag{4.80}$$

So, $\mathrm{Equal\_except\_at}(\{j\}, \mathrm{env}_{\mathrm{ew}}^{\mathcal{P}}, \mathrm{env}_{\mathrm{ew}}^{\mathcal{M}})$. The set $\{j\}$ is a subset of the set of indices h that appear in **stow**(h) statements anywhere within the **whenever** statement explicitly shown in the schema $\mathcal{P}$. In this case, by Lemma 4.36, $\mathrm{Equal\_except\_at}(\{j\}, \mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}_{\mathrm{ew}}^{\mathcal{P}}), \mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}_{\mathrm{ew}}^{\mathcal{M}}))$. Hence, $\mathrm{Equal\_except\_at}(\{j\}, \mathrm{env}_F^{\mathcal{P}}, \mathrm{env}_F^{\mathcal{M}})$. Therefore, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $\mathcal{I}(\mathit{Br\_Cd})(\mathit{env}_i^{\mathcal{P}})$. (We are still inside case $\mathrm{AE}(\mathrm{env}_i^{\mathcal{P}}) \neq \mathrm{CF}$, i.e., $\mathrm{AE}(\mathrm{env}_i^{\mathcal{P}}) = \mathrm{NL}$.) In this case, $\mathrm{env}_{\mathrm{ew}}^{\mathcal{P}} = \mathcal{I}(\mathrm{P\_nm}(\mathrm{ac}, \mathrm{ad})\ \mathbf{stow}(j)\ \mathit{cd\_suffix})(\mathrm{env}_i^{\mathcal{P}})$, and $\mathrm{env}_{\mathrm{ew}}^{\mathcal{M}} = \mathcal{I}(\mathbf{assume}\ \mathrm{b}_j = \mathrm{b}_i \wedge (\mathrm{post}[\#\mathrm{x} \rightsquigarrow \mathrm{ac}_i, \mathrm{x} \rightsquigarrow \mathrm{ac}_j, \#\mathrm{y} \rightsquigarrow \mathrm{ad}_i, \mathrm{y} \rightsquigarrow \mathrm{ad}_j, \#\mathrm{z} \rightsquigarrow \mathrm{z}_i, \mathrm{z} \rightsquigarrow \mathrm{z}_j])\ \mathit{cd\_suffix})(\mathrm{env}_j^{\mathcal{M}})$. Let $\mathrm{env}_j^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathrm{P\_nm}(\mathrm{ac}, \mathrm{ad})\ \mathbf{stow}(j))(\mathrm{env}_i^{\mathcal{P}})$. Let $\mathrm{cs}_j \stackrel{\mathrm{def}}{=} \mathrm{CSE}(\mathrm{env}_j^{\mathcal{P}})$. In this case, we take $\mathrm{env}_I^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \langle \mathrm{cs}_j \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. Then $\mathrm{env}_i^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}, \langle \mathrm{cs}_j \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. Because $\mathcal{P}$ is syntactically correct, the declaration-meaning d of environment $\mathrm{env}_I^{\mathcal{P}}$ contains $\mathrm{d}(\mathrm{P\_nm})$. Let $\mathrm{d}(\mathrm{P\_nm}) \stackrel{\mathrm{def}}{=} [\mathrm{dp}, \mathrm{ep}, \mathrm{pf}, \mathrm{sf}]$. Because $\mathrm{AE}(\mathrm{env}_I^{\mathcal{P}}) = \mathrm{NL}$, we have that dp is the same as pre, ep is the same as post, and $\mathrm{d}(\mathrm{P\_nm})$ is conformal.

Case $\mathit{AE}(\mathit{env}_j^{\mathcal{P}}) = \mathit{CF}$. Because $\mathrm{d}(\mathrm{P\_nm})$ is conformal, if $\mathrm{dp}(\mathrm{cs}_i(\mathrm{ac}), \mathrm{cs}_i(\mathrm{ad}), \mathrm{cs}_i(\mathrm{z}))$, then $\mathrm{sf}(\mathrm{cs}_i(\mathrm{ac}), \mathrm{cs}_i(\mathrm{ad}), \mathrm{cs}_i(\mathrm{z})) \neq \mathrm{CF}$. Therefore, $\neg\mathrm{dp}(\mathrm{cs}_i(\mathrm{ac}), \mathrm{cs}_i(\mathrm{ad}), \mathrm{cs}_i(\mathrm{z}))$. Hence, $\neg\mathcal{I}(\mathrm{pre}[\mathrm{x} \rightsquigarrow \mathrm{ac}_i, \mathrm{y} \rightsquigarrow \mathrm{ad}_i, \mathrm{z} \rightsquigarrow \mathrm{z}_i])(\mathrm{env}_i^{\mathcal{M}})$. Therefore $\mathrm{AE}(\mathcal{I}(\mathbf{confirm}\ (\mathrm{Br\_Cd}) \Rightarrow (\mathrm{pre}[\mathrm{x} \rightsquigarrow \mathrm{ac}_i, \mathrm{y} \rightsquigarrow \mathrm{ad}_i, \mathrm{z} \rightsquigarrow \mathrm{z}_i]))(\mathrm{env}_i^{\mathcal{M}})) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$.

Case $\mathit{AE}(\mathit{env}_j^{\mathcal{P}}) \neq \mathit{CF}$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_j^{\mathcal{P}}) = \mathrm{NL}$. Therefore, $\mathrm{dp}(\mathrm{cs}_i(\mathrm{ac}), \mathrm{cs}_i(\mathrm{ad}), \mathrm{cs}_i(\mathrm{z}))$, and $\mathrm{sf}(\mathrm{cs}_i(\mathrm{ac}), \mathrm{cs}_i(\mathrm{ad}), \mathrm{cs}_i(\mathrm{z})) = \mathrm{NL}$. Hence,

$\mathcal{I}(\text{pre}[\text{x} \rightsquigarrow \text{ac}_i, \text{y} \rightsquigarrow \text{ad}_i, \text{z} \rightsquigarrow \text{z}_i])(\text{env}_i^{\mathcal{M}})$. Therefore, $\text{env}_j^{\mathcal{M}} = [\text{NL}, \text{cs}_j, \text{ns}_j, \text{se}, \text{os}, \text{d}]$ where

$$\text{ns}_j(h) = \begin{cases} \text{ns}(h) & \text{if } h \neq j \\ \text{cs}_j & \text{if } h = j \end{cases} . \tag{4.81}$$

Hence, $\text{env}_j^{\mathcal{M}} = \text{env}_j^{\mathcal{P}}$. Because d(P_nm) is conformal, $\text{ep}(\text{cs}_i(\text{ac}), \text{cs}_i(\text{ad}), \text{cs}_i(\text{z}), \text{cs}_j(\text{ac}), \text{cs}_j(\text{ad}), \text{cs}_j(\text{z}))$. Furthermore, if $\xi \notin \{\text{ac}, \text{ad}, \text{z}\}$, then $\text{cs}_j(\xi) = \text{cs}_i(\xi)$. Because $\text{ns}_j(j) = \text{cs}_j$ and $\text{ns}_j(i) = \text{cs}_i$, $\mathcal{I}(\textbf{assume } \text{b}_j = \text{b}_i \wedge (\text{post}[\#\text{x} \rightsquigarrow \text{ac}_i, \text{x} \rightsquigarrow \text{ac}_j, \#\text{y} \rightsquigarrow \text{ad}_i, \text{y} \rightsquigarrow \text{ad}_j, \#\text{z} \rightsquigarrow \text{z}_i, \text{z} \rightsquigarrow \text{z}_j]))(\text{env}_j^{\mathcal{M}}) = \text{env}_j^{\mathcal{M}}$. Therefore, $\text{env}_{\text{ew}}^{\mathcal{M}} = \text{env}_{\text{ew}}^{\mathcal{P}}$, and $\text{env}_F^{\mathcal{M}} = \text{env}_F^{\mathcal{P}}$. Hence, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. □

Lemma 4.43 *The rule for* **assume** *preserves invalidity in the math direction.*

Proof. We assume that $\mathcal{P}$ is invalid and show that $\mathcal{M}$ is invalid. Let $\text{env}_I^{\mathcal{P}}$ be a witness to $\mathcal{P}$'s invalidity, and let $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\text{env}_I^{\mathcal{P}})$; i.e., $\text{AE}(\text{env}_I^{\mathcal{P}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$. We will show the existence of an environment, $\text{env}_I^{\mathcal{M}}$, such that $\text{AE}(\text{env}_I^{\mathcal{M}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$, where $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\text{env}_I^{\mathcal{M}})$. The proof is organized by cases. In each case we take $\text{env}_I^{\mathcal{M}} = \text{env}_I^{\mathcal{P}}$. First we make the following two definitions.

$$\text{env}_i^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code } \textbf{alter all stow}(i) ACseq_0)(\text{env}_I^{\mathcal{P}}) \tag{4.82}$$

$$\text{env}_i^{\mathcal{M}} \stackrel{\text{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code } \textbf{alter all stow}(i) ACseq_0)(\text{env}_I^{\mathcal{M}}) \tag{4.83}$$

Consequently, $\text{env}_i^{\mathcal{M}} = \text{env}_i^{\mathcal{P}}$.

Case $AE(env_i^{\mathcal{P}}) = CF$. In this case, $\text{AE}(\text{env}_i^{\mathcal{M}}) = \text{CF}$. By Lemma 4.15, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$.

Case $AE(env_i^{\mathcal{P}}) \neq CF$. In this case, by Lemma 4.19, we have $\text{AE}(\text{env}_i^{\mathcal{P}}) = \text{NL}$.

Case $\neg\mathcal{I}(Br\_Cd)(env_i^{\mathcal{P}})$. In this case, $\neg\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{M}})$. By the semantics of **whenever**, $\text{env}_F^{\mathcal{P}} = \mathcal{I}(fol\_top\_lev\_code)(\text{env}_i^{\mathcal{P}})$ and $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{assume}\ (\text{Br\_Cd}) \Rightarrow (H)\ fol\_top\_lev\_code)(\text{env}_i^{\mathcal{M}})$. Because $\neg\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{M}})$, $\mathcal{I}((\text{Br\_Cd}) \Rightarrow (H))(\text{env}_i^{\mathcal{M}})$. Therefore, $\mathcal{I}(\textbf{assume}\ (\text{Br\_Cd}) \Rightarrow (H))(\text{env}_i^{\mathcal{M}}) = \text{env}_i^{\mathcal{M}}$. By equation 2.23, $\text{env}_F^{\mathcal{M}} = \mathcal{I}(fol\_top\_lev\_code)(\text{env}_i^{\mathcal{M}})$. Hence, $\text{env}_F^{\mathcal{M}} = \text{env}_F^{\mathcal{P}}$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$.

Case $\mathcal{I}(Br\_Cd)(env_i^{\mathcal{P}})$. In this case, $\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{M}})$. By the semantics of **whenever**, $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\textbf{assume}\ H\ cd\_suffix\ fol\_top\_lev\_code)(\text{env}_i^{\mathcal{P}})$ and $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{assume}\ (\text{Br\_Cd}) \Rightarrow (H)\ cd\_suffix\ fol\_top\_lev\_code)(\text{env}_i^{\mathcal{M}})$. Because $\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{M}})$, $\mathcal{I}(\textbf{assume}\ (\text{Br\_Cd}) \Rightarrow (H))(\text{env}_i^{\mathcal{M}}) = \mathcal{I}(\textbf{assume}\ H)(\text{env}_i^{\mathcal{M}})$. By two applications of equation 2.23, $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{assume}\ H\ cd\_suffix\ fol\_top\_lev\_code)(\text{env}_i^{\mathcal{M}})$. Hence, $\text{env}_F^{\mathcal{M}} = \text{env}_F^{\mathcal{P}}$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. □

Lemma 4.44 *The rule for* **confirm** *preserves invalidity in the math direction.*

Proof. We assume that $\mathcal{P}$ is invalid and show that $\mathcal{M}$ is invalid. Let $\text{env}_I^{\mathcal{P}}$ be a witness to $\mathcal{P}$'s invalidity, and let $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\text{env}_I^{\mathcal{P}})$; i.e., $\text{AE}(\text{env}_I^{\mathcal{P}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$. We will show the existence of an environment, $\text{env}_I^{\mathcal{M}}$, such that $\text{AE}(\text{env}_I^{\mathcal{M}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$, where $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\text{env}_I^{\mathcal{M}})$. The proof is organized by cases. In each case we take $\text{env}_I^{\mathcal{M}} = \text{env}_I^{\mathcal{P}}$. First we make the following two definitions.

$$\text{env}_i^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(prec\_top\_lev\_code\ \textbf{alter all stow}(i)\,ACseq_0)(\text{env}_I^{\mathcal{P}}) \tag{4.84}$$

$$\text{env}_i^{\mathcal{M}} \stackrel{\text{def}}{=} \mathcal{I}(prec\_top\_lev\_code\ \textbf{alter all stow}(i)\,ACseq_0)(\text{env}_I^{\mathcal{M}}) \tag{4.85}$$

Consequently, $\text{env}_i^{\mathcal{M}} = \text{env}_i^{\mathcal{P}}$. Furthermore, $\text{ISE}(\text{env}_i^{\mathcal{M}})(i) = \text{CSE}(\text{env}_i^{\mathcal{M}}) = \text{CSE}(\text{env}_i^{\mathcal{P}})$.

Case $AE(env_i^{\mathcal{P}}) = CF$. In this case, $\text{AE}(\text{env}_i^{\mathcal{M}}) = \text{CF}$. By Lemma 4.15, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$.

Case $AE(env_i^{\mathcal{P}}) \neq CF$. In this case, by Lemma 4.19, we have $\text{AE}(\text{env}_i^{\mathcal{P}}) = \text{NL}$.

Case $\neg\mathcal{I}(Br\_Cd)(env_i^{\mathcal{P}})$. In this case, $\neg\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{M}})$. By the semantics of **whenever**, $\text{env}_F^{\mathcal{P}} = \mathcal{I}(fol\_top\_lev\_code)(\text{env}_i^{\mathcal{P}})$ and $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{confirm}\ (\text{Br\_Cd}) \Rightarrow (H[\text{x} \rightsquigarrow \text{x}_i])\ fol\_top\_lev\_code)(\text{env}_i^{\mathcal{M}})$. Because $\neg\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{M}})$, $\mathcal{I}((\text{Br\_Cd}) \Rightarrow (H[\text{x} \rightsquigarrow \text{x}_i]))(\text{env}_i^{\mathcal{M}})$. Therefore, $\mathcal{I}(\textbf{confirm}\ (\text{Br\_Cd}) \Rightarrow (H)[\text{x} \rightsquigarrow \text{x}_i])(\text{env}_i^{\mathcal{M}}) = \text{env}_i^{\mathcal{M}}$. By equation 2.23, $\text{env}_F^{\mathcal{M}} = \mathcal{I}(fol\_top\_lev\_code)(\text{env}_i^{\mathcal{M}})$. Hence, $\text{env}_F^{\mathcal{M}} = \text{env}_F^{\mathcal{P}}$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$.

Case $\mathcal{I}(Br\_Cd)(env_i^{\mathcal{P}})$. In this case, $\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{M}})$. By the semantics of **whenever**, $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\textbf{confirm}\ H\lfloor\text{x}\rfloor\ cd\_suffix\ fol\_top\_lev\_code)(\text{env}_i^{\mathcal{P}})$ and $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{confirm}\ (\text{Br\_Cd}) \Rightarrow (H[\text{x} \rightsquigarrow \text{x}_i])\ cd\_suffix\ fol\_top\_lev\_code)(\text{env}_i^{\mathcal{M}})$. Because $\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{M}})$, $\mathcal{I}(\textbf{confirm}\ (\text{Br\_Cd}) \Rightarrow (H[\text{x} \rightsquigarrow \text{x}_i]))(\text{env}_i^{\mathcal{M}}) = \mathcal{I}(\textbf{confirm}\ H[\text{x} \rightsquigarrow \text{x}_i])(\text{env}_i^{\mathcal{M}})$. By two applications of equation 2.23, $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{confirm}\ H[\text{x} \rightsquigarrow \text{x}_i]\ cd\_suffix\ fol\_top\_lev\_code)(\text{env}_i^{\mathcal{M}})$. Because $\text{ISE}(\text{env}_i^{\mathcal{M}})(i) = \text{CSE}(\text{env}_i^{\mathcal{P}})$, $\mathcal{I}(\textbf{confirm}\ H[\text{x} \rightsquigarrow \text{x}_i])(\text{env}_i^{\mathcal{M}}) = \mathcal{I}(\textbf{confirm}\ H\lfloor\text{x}\rfloor)(\text{env}_i^{\mathcal{P}})$. By two applications of equation 2.23, $\text{env}_F^{\mathcal{M}} = \text{env}_F^{\mathcal{P}}$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. □

Lemma 4.45 *The rule for empty guarded blocks preserves invalidity in the math direction.*

Proof. We assume that $\mathcal{P}$ is invalid and show that $\mathcal{M}$ is invalid. Let $\text{env}_I^{\mathcal{P}}$ be a witness to $\mathcal{P}$'s invalidity, and let $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\text{env}_I^{\mathcal{P}})$; i.e., $\text{AE}(\text{env}_I^{\mathcal{P}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$. We will show the existence of an environment, $\text{env}_I^{\mathcal{M}}$, such that $\text{AE}(\text{env}_I^{\mathcal{M}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$, where $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\text{env}_I^{\mathcal{M}})$. We take $\text{env}_I^{\mathcal{M}} = \text{env}_I^{\mathcal{P}}$ and make the following two definitions.

$$\text{env}_i^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\,ACseq_0)(\text{env}_I^{\mathcal{P}}) \tag{4.86}$$

$$\text{env}_i^{\mathcal{M}} \stackrel{\text{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\,ACseq_0)(\text{env}_I^{\mathcal{M}}) \tag{4.87}$$

Consequently, $\text{env}_i^{\mathcal{M}} = \text{env}_i^{\mathcal{P}}$.

Case $AE(env_i^{\mathcal{P}}) = CF$. In this case, $\text{AE}(\text{env}_i^{\mathcal{M}}) = \text{CF}$. By Lemma 4.15, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$.

Case $AE(env_i^{\mathcal{P}}) \neq CF$. In this case, by Lemma 4.19, we have $\text{AE}(\text{env}_i^{\mathcal{P}}) = \text{NL}$. Whether $\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{P}})$ or $\neg\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{P}})$, $\mathcal{I}(\textbf{whenever}\ \text{Br\_Cd}\ \textbf{do end whenever})(\text{env}_i^{\mathcal{P}}) = \mathcal{I}(\varepsilon)(\text{env}_i^{\mathcal{P}}) = \text{env}_i^{\mathcal{P}}$. By equation 2.23, $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\textit{fol\_top\_lev\_code})(\text{env}_i^{\mathcal{P}})$. Because $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textit{fol\_top\_lev\_code})(\text{env}_i^{\mathcal{M}})$, $\text{env}_F^{\mathcal{M}} = \text{env}_F^{\mathcal{P}}$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. □

Lemma 4.46 *The rule for* **alter all** *preserves invalidity in the math direction.*

Proof. We assume that $\mathcal{P}$ is invalid and show that $\mathcal{M}$ is invalid. Let $\text{env}_I^{\mathcal{P}}$ be a witness to $\mathcal{P}$'s invalidity, and let $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\text{env}_I^{\mathcal{P}})$; i.e., $\text{AE}(\text{env}_I^{\mathcal{P}}) = \text{NL}$ and

$\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$. We will show the existence of an environment, $\mathrm{env}_I^{\mathcal{M}}$, such that $\mathrm{AE}(\mathrm{env}_I^{\mathcal{M}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$, where $\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\mathrm{env}_I^{\mathcal{M}})$. The proof is organized by cases. First we make the following four definitions.

$$\mathrm{env}_1^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code})(\mathrm{env}_I^{\mathcal{P}}) \tag{4.88}$$

$$\mathrm{env}_a^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{alter all})(\mathrm{env}_1^{\mathcal{P}}) \tag{4.89}$$

$$\mathrm{env}_s^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{stow}(i))(\mathrm{env}_a^{\mathcal{P}}) \tag{4.90}$$

$$\mathrm{env}_1^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code})(\mathrm{env}_I^{\mathcal{M}}) \tag{4.91}$$

Consequently,

$$\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}_s^{\mathcal{P}}) \tag{4.92}$$

$$\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}_1^{\mathcal{M}}). \tag{4.93}$$

Case $AE(env_1^{\mathcal{P}}) = CF$. In this case, we take $\mathrm{env}_I^{\mathcal{M}} = \mathrm{env}_I^{\mathcal{P}}$. Hence, $\mathrm{env}_1^{\mathcal{M}} = \mathrm{env}_1^{\mathcal{P}}$. Therefore, $\mathrm{AE}(\mathrm{env}_1^{\mathcal{M}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$.

Case $AE(env_1^{\mathcal{P}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_1^{\mathcal{P}}) = \mathrm{NL}$. We may, for this case, write $\mathrm{env}_1^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_1, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}]$. With the help of Lemmas 4.20 and 4.27, we may also write $\mathrm{env}_I^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. In this case, we take $\mathrm{env}_I^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}'_I, \mathrm{se}_I \circ \mathrm{tail}(\mathrm{se}), \mathrm{os}, \mathrm{d}]$, where

$$\mathrm{ns}'_I(h) = \begin{cases} \mathrm{ns}_I(h) & \text{if } h \neq i \\ \mathrm{first}(\mathrm{se}) & \text{if } h = i \end{cases} . \tag{4.94}$$

By Lemma 4.18, $\mathrm{AE}(\mathrm{env}_a^{\mathcal{P}}) \neq \mathrm{VT}$. Then, by the semantics of **alter all**, $\mathrm{AE}(\mathrm{env}_a^{\mathcal{P}}) = \mathrm{NL}$, $\mathrm{se} \neq \varepsilon$, and $\mathrm{env}_a^{\mathcal{P}} = [\mathrm{NL}, \mathrm{first}(\mathrm{se}), \mathrm{ns}, \mathrm{tail}(\mathrm{se}), \mathrm{os}, \mathrm{d}]$. By the semantics of **stow**(i),

$\mathrm{env}_s^{\mathcal{P}} = [\mathrm{NL}, \mathrm{first}(\mathrm{se}), \mathrm{ns}', \mathrm{tail}(\mathrm{se}), \mathrm{os}, \mathrm{d}]$, where

$$\mathrm{ns}'(h) = \begin{cases} \mathrm{ns}(h) & \text{if } h \neq i \\ \mathrm{first}(\mathrm{se}) & \text{if } h = i \end{cases} . \tag{4.95}$$

By the syntax restriction that references to index i occur only after **stow**(i) (page 93), no statement of *prec_top_lev_code* contains a reference to any variable indexed with i. Because ns_I' differs from ns_I only at i, $\mathrm{env}_1^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_1, \mathrm{ns}'', \mathrm{tail}(\mathrm{se}), \mathrm{os}, \mathrm{d}]$, where

$$\mathrm{ns}''(h) = \begin{cases} \mathrm{ns}(h) & \text{if } h \neq i \\ \mathrm{ns}_I'(h) & \text{if } h = i \end{cases} . \tag{4.96}$$

Note that $\mathrm{ns}'' = \mathrm{ns}'$. Therefore, $\mathrm{env}_1^{\mathcal{M}}$ differs from $\mathrm{env}_s^{\mathcal{P}}$ only in the current state. Due to the additional syntactic restriction of the rule for **alter all**, *fol_top_lev_code* contains only **alter all**, **stow**, **assume**, and **confirm** statements. The **assume** and **confirm** statements refer only to indexed variables. Hence, **stow** is the only one of these statements whose semantics is affected by the current state; however, every one of these **stow** statements is immediately preceded by an **alter all** statement. Therefore, $\mathrm{CSE}(\mathrm{env}_1^{\mathcal{M}})$ and $\mathrm{CSE}(\mathrm{env}_s^{\mathcal{P}})$ have no effect on the values, respectively, of $\mathcal{I}(\textit{fol\_top\_lev\_code})(\mathrm{env}_1^{\mathcal{M}})$ and $\mathcal{I}(\textit{fol\_top\_lev\_code})(\mathrm{env}_s^{\mathcal{P}})$. Therefore, $\mathrm{env}_F^{\mathcal{M}}$ and $\mathrm{env}_F^{\mathcal{P}}$ differ at most in the current state. In particular, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$. □

Lemma 4.47 *The rule for consecutive* **assume** *statements preserves invalidity in the math direction.*

Proof. We assume that $\mathcal{P}$ is invalid and show that $\mathcal{M}$ is invalid. Let $\mathrm{env}_I^{\mathcal{P}}$ be a witness to $\mathcal{P}$'s invalidity, and let $\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\mathrm{env}_I^{\mathcal{P}})$; i.e., $\mathrm{AE}(\mathrm{env}_I^{\mathcal{P}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$. We will show the existence of an environment, $\mathrm{env}_I^{\mathcal{M}}$, such that

$\mathrm{AE}(\mathrm{env}_I^{\mathcal{M}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$, where $\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\mathrm{env}_I^{\mathcal{M}})$. The proof is organized by cases. In each case we take $\mathrm{env}_I^{\mathcal{M}} = \mathrm{env}_I^{\mathcal{P}}$. First we make the following five definitions.

$$\mathrm{env}_1^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code})(\mathrm{env}_I^{\mathcal{P}}) \tag{4.97}$$

$$\mathrm{env}_2^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{assume } H_1)(\mathrm{env}_1^{\mathcal{P}}) \tag{4.98}$$

$$\mathrm{env}_b^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{assume } H_2)(\mathrm{env}_2^{\mathcal{P}}) \tag{4.99}$$

$$\mathrm{env}_1^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code})(\mathrm{env}_I^{\mathcal{M}}) \tag{4.100}$$

$$\mathrm{env}_b^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{assume } (H_1) \wedge (H_2))(\mathrm{env}_1^{\mathcal{M}}) \tag{4.101}$$

Consequently,

$$\mathrm{env}_1^{\mathcal{M}} = \mathrm{env}_1^{\mathcal{P}} \tag{4.102}$$

$$\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}_b^{\mathcal{P}}) \tag{4.103}$$

$$\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}_b^{\mathcal{M}}). \tag{4.104}$$

Case $AE(env_1^{\mathcal{P}}) = CF$. Therefore, $\mathrm{AE}(\mathrm{env}_1^{\mathcal{M}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$.

Case $AE(env_1^{\mathcal{P}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_1^{\mathcal{P}}) = \mathrm{NL}$. By substitution of equals, $\mathrm{AE}(\mathrm{env}_1^{\mathcal{M}}) = \mathrm{NL}$. By Lemma 4.18 and the semantics of the **assume** statement, $\mathrm{AE}(\mathrm{env}_2^{\mathcal{P}}) = \mathrm{NL}$. Therefore, $\mathrm{env}_2^{\mathcal{P}} = \mathrm{env}_1^{\mathcal{P}}$. Similarly, $\mathrm{env}_b^{\mathcal{P}} = \mathrm{env}_2^{\mathcal{P}} = \mathrm{env}_1^{\mathcal{P}}$. Hence, $\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{P}})$ and $\mathcal{I}(H_2)(\mathrm{env}_1^{\mathcal{P}})$. Therefore, $\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{M}})$ and $\mathcal{I}(H_2)(\mathrm{env}_1^{\mathcal{M}})$. Hence, $\mathcal{I}((H_1) \wedge (H_2))(\mathrm{env}_1^{\mathcal{M}})$. Therefore, by the semantics of the **assume** statement, $\mathrm{env}_b^{\mathcal{M}} = \mathrm{env}_1^{\mathcal{M}}$. Hence, $\mathrm{env}_b^{\mathcal{M}} = \mathrm{env}_b^{\mathcal{P}}$. Therefore, $\mathrm{env}_F^{\mathcal{M}} = \mathrm{env}_F^{\mathcal{P}}$. Hence, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$. □

Lemma 4.48 *The* **assume-confirm** *rule preserves invalidity in the math direction.*

Proof. We assume that $\mathcal{P}$ is invalid and show that $\mathcal{M}$ is invalid. Let $\text{env}_I^{\mathcal{P}}$ be a witness to $\mathcal{P}$'s invalidity, and let $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\text{env}_I^{\mathcal{P}})$; i.e., $\text{AE}(\text{env}_I^{\mathcal{P}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$. We will show the existence of an environment, $\text{env}_I^{\mathcal{M}}$, such that $\text{AE}(\text{env}_I^{\mathcal{M}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$, where $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\text{env}_I^{\mathcal{M}})$. The proof is organized by cases. In each case we take $\text{env}_I^{\mathcal{M}} = \text{env}_I^{\mathcal{P}}$. First we make the following three definitions.

$$\text{env}_1^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\mathit{top\_lev\_code})(\text{env}_I^{\mathcal{P}}) \tag{4.105}$$

$$\text{env}_2^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\textbf{assume } H_1)(\text{env}_1^{\mathcal{P}}) \tag{4.106}$$

$$\text{env}_1^{\mathcal{M}} \stackrel{\text{def}}{=} \mathcal{I}(\mathit{top\_lev\_code})(\text{env}_I^{\mathcal{M}}) \tag{4.107}$$

Consequently,

$$\text{env}_1^{\mathcal{M}} = \text{env}_1^{\mathcal{P}} \tag{4.108}$$

$$\text{env}_F^{\mathcal{P}} = \mathcal{I}(\textbf{confirm } H_2)(\text{env}_2^{\mathcal{P}}) \tag{4.109}$$

$$\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{confirm } (H_1) \Rightarrow (H_2))(\text{env}_1^{\mathcal{M}}). \tag{4.110}$$

Case $AE(env_1^{\mathcal{P}}) = CF$. Therefore, $\text{AE}(\text{env}_1^{\mathcal{M}}) = \text{CF}$. By Lemma 4.15, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$.

Case $AE(env_1^{\mathcal{P}}) \neq CF$. In this case, by Lemma 4.19, we have $\text{AE}(\text{env}_1^{\mathcal{P}}) = \text{NL}$. By substitution of equals, $\text{AE}(\text{env}_1^{\mathcal{M}}) = \text{NL}$. By Lemma 4.18 and the semantics of the **assume** statement, $\text{AE}(\text{env}_2^{\mathcal{P}}) = \text{NL}$. Therefore, $\text{env}_2^{\mathcal{P}} = \text{env}_1^{\mathcal{P}}$. Hence, $\mathcal{I}(H_1)(\text{env}_1^{\mathcal{P}})$.

By substitution of equals, $\mathcal{I}(H_1)(\text{env}_1^{\mathcal{M}})$. By the semantics of the **confirm** statement, we have $\neg\mathcal{I}(H_2)(\text{env}_2^{\mathcal{P}})$. By substitution of equals, $\neg\mathcal{I}(H_2)(\text{env}_1^{\mathcal{M}})$. Therefore, $\neg\mathcal{I}((H_1) \Rightarrow (H_2))(\text{env}_1^{\mathcal{M}})$. By the semantics of the **confirm** statement, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. □

Lemma 4.49 *The rule for consecutive* **confirm** *statements preserves invalidity in the math direction.*

Proof. We assume that $\mathcal{P}$ is invalid and show that $\mathcal{M}$ is invalid. Let $\text{env}_I^{\mathcal{P}}$ be a witness to $\mathcal{P}$'s invalidity, and let $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\text{env}_I^{\mathcal{P}})$; i.e., $\text{AE}(\text{env}_I^{\mathcal{P}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$. We will show the existence of an environment, $\text{env}_I^{\mathcal{M}}$, such that $\text{AE}(\text{env}_I^{\mathcal{M}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$, where $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\text{env}_I^{\mathcal{M}})$. The proof is organized by cases. In each case we take $\text{env}_I^{\mathcal{M}} = \text{env}_I^{\mathcal{P}}$. First we make the following five definitions.

$$\text{env}_1^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code})(\text{env}_I^{\mathcal{P}}) \tag{4.111}$$

$$\text{env}_2^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\textbf{confirm } H_1)(\text{env}_1^{\mathcal{P}}) \tag{4.112}$$

$$\text{env}_b^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\textbf{confirm } H_2)(\text{env}_2^{\mathcal{P}}) \tag{4.113}$$

$$\text{env}_1^{\mathcal{M}} \stackrel{\text{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code})(\text{env}_I^{\mathcal{M}}) \tag{4.114}$$

$$\text{env}_b^{\mathcal{M}} \stackrel{\text{def}}{=} \mathcal{I}(\textbf{confirm } (H_1) \wedge (H_2))(\text{env}_1^{\mathcal{M}}) \tag{4.115}$$

Consequently,

$$\text{env}_1^{\mathcal{M}} = \text{env}_1^{\mathcal{P}} \tag{4.116}$$

$$\text{env}_F^{\mathcal{P}} = \mathcal{I}(\textit{fol\_top\_lev\_code})(\text{env}_b^{\mathcal{P}}) \tag{4.117}$$

$$\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textit{fol\_top\_lev\_code})(\text{env}_b^{\mathcal{M}}). \tag{4.118}$$

Case $AE(env_1^{\mathcal{P}}) = CF$. Therefore, $\mathrm{AE}(\mathrm{env}_1^{\mathcal{M}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$.

Case $AE(env_1^{\mathcal{P}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_1^{\mathcal{P}}) = \mathrm{NL}$. By substitution of equals, $\mathrm{AE}(\mathrm{env}_1^{\mathcal{M}}) = \mathrm{NL}$.

Case $AE(env_2^{\mathcal{P}}) = CF$. By the semantics of the **confirm** statement, $\neg\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{P}})$. By substitution of equals, $\neg\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{M}})$. By the semantics of the **confirm** statement, $\mathrm{AE}(\mathrm{env}_b^{\mathcal{M}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$.

Case $AE(env_2^{\mathcal{P}}) \neq CF$. Hence, $\mathrm{AE}(\mathrm{env}_2^{\mathcal{P}}) = \mathrm{NL}$. By the semantics of the **confirm** statement, $\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{P}})$ and $\mathrm{env}_2^{\mathcal{P}} = \mathrm{env}_1^{\mathcal{P}}$. By substitution of equals, $\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{M}})$.

Case $AE(env_b^{\mathcal{P}}) = CF$. By the semantics of the **confirm** statement, $\neg\mathcal{I}(H_2)(\mathrm{env}_2^{\mathcal{P}})$. By substitution of equals, $\neg\mathcal{I}(H_2)(\mathrm{env}_1^{\mathcal{M}})$. By the semantics of the **confirm** statement, $\mathrm{AE}(\mathrm{env}_b^{\mathcal{M}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$.

Case $AE(env_b^{\mathcal{P}}) \neq CF$. Hence, $\mathrm{AE}(\mathrm{env}_b^{\mathcal{P}}) = \mathrm{NL}$. By the semantics of the **confirm** statement, $\mathcal{I}(H_2)(\mathrm{env}_2^{\mathcal{P}})$ and $\mathrm{env}_b^{\mathcal{P}} = \mathrm{env}_2^{\mathcal{P}} = \mathrm{env}_1^{\mathcal{P}} = \mathrm{env}_1^{\mathcal{M}}$. By substitution of equals, $\mathcal{I}(H_2)(\mathrm{env}_1^{\mathcal{M}})$. Recall that $\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{M}})$. Therefore, $\mathcal{I}((H_1) \wedge (H_2))(\mathrm{env}_1^{\mathcal{M}})$. By the semantics of the **confirm** statement, $\mathrm{env}_b^{\mathcal{M}} = \mathrm{env}_1^{\mathcal{M}}$. Therefore, $\mathrm{env}_b^{\mathcal{M}} = \mathrm{env}_b^{\mathcal{P}}$ and $\mathrm{env}_F^{\mathcal{M}} = \mathrm{env}_F^{\mathcal{P}}$. Hence, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$. □

Lemma 4.50 *The rule of inference bridging predicate logic and the indexed method preserves invalidity in the math direction.*

Proof. We suppose that **confirm** H is an invalid program. That is to say, there exists a neutral environment env such that $\mathrm{AE}(\textbf{confirm } H) = \mathrm{CF}$. By the semantics of the **confirm** statement, the component states of env make an assignment of values to the

variables in H that renders H false in some model of context C's logic. Consequently, the mathematical assertion H is false (i.e., invalid) in context C. □

4.5 Relative Completeness Lemmas

4.5.1 Related Valid Program Differing in Assertions Only

Our first obligation regarding relative completeness is to establish Lemma 4.4.

Proof of Lemma 4.4. Because the lemma assumes the assertion language to be expressive, for each internal procedure there exists an assertion expressing the post relation corresponding to that procedure's precondition and its body. By the same assumption, for each loop there exists an assertion expressing its tightest loop invariant. Therefore, Prog′ of the lemma exists. It suffices to show that Prog′ is valid.

Consider procedure P_nm with precondition pre and postcondition (in Prog) post. In Prog′ P_nm has postcondition post′. By the definition of post′ and because Prog is valid, interpretation in Prog′—in a neutral environment—of P_nm's declaration does not produce a categorically false environment. Interpretation of any call to P_nm is the same in Prog and Prog′ because the only difference between the declaration meanings of P_nm in the two programs is in the effect predicate, which plays no role in the interpretation of a procedure call.

Consider a loop with invariant (in Prog) Inv. In Prog′ this loop has invariant Inv′. Because Inv′ is by definition an invariant for the loop, and because Prog is valid, interpretation of the loop in Prog′—in a neutral environment—does not produce a

categorically false environment. Therefore, the interpretations of the two loops in identical environments produce identical environments.

The interpretations of all other statements are unaffected by the replacements made to produce Prog$'$. Therefore, Prog$'$ is also valid. □

4.5.2 System of Rewrite Rules Is Terminating

If we fix the direction of application to one orientation, say the math direction, then the proof rules of Chapter III can be regarded as rewrite rules. We establish that this system of rewrite rules is terminating by proving Lemma 4.5.

Proof of Lemma 4.5. A well-prepared assertive program is an assertive program that conforms to the syntax of either (1) top level code or (2) $\mathcal{P}$ defined in the bridge rule (Figure 40). These two cases are mutually exclusive. The bridge rule is always applicable (in the math direction) in the latter case—never in the former case. Let us refer to an an assertive program that conforms to the syntax of top level code as a *top level program.* Application of the bridge rule in the math direction always produces a top level program. It remains to show that the process of rewriting any top level program in the math direction using the remaining rules of Chapter III always terminates with a mathematical assertion.

We define a total function from the set of top level programs into the natural numbers. This function, which we call Meas, gives a measure on the size of assertive programs. Then we show that, for each rule of Chapter III (other than the bridge rule), Meas($\mathcal{M}$) < Meas($\mathcal{P}$). Therefore, the process of rewriting assertive programs in the math direction must terminate in finitely many steps.

Table 2: Multipliers for Function Meas

multiplier	*statement*
1	**whenever**
1	**alter all**
1	**stow**
1	**assume** at top level
1	**confirm** at top level
2	**assume** inside **whenever**
2	**confirm** inside **whenever**
5	procedure call
11	**loop**
12	**if**

Meas's value for a given assertive program is defined by (1) counting the number of occurrences of each statement in the program, recursively, including statements within statements; (2) multiplying the count for each statement by the corresponding number in Table 2; and (3) summing the resulting ten products.

The rule for **assume**'s $\mathcal{M}$ (Figure 44) has one fewer **assume** statement inside a **whenever** statement and one more **assume** statement at the top level than its $\mathcal{P}$. Therefore, $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 2 \cdot 1 + 1 \cdot 1$; i.e., $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 1$. Hence, $\text{Meas}(\mathcal{M}) < \text{Meas}(\mathcal{P})$. As we examine the remaining rules, we are comparing the rule's $\mathcal{M}$ with its $\mathcal{P}$.

The rule for procedure call's $\mathcal{M}$ (Figure 46) has one fewer procedure call, one more **confirm** statement at the top level, one more **alter all** statement, and one more **assume** statement inside a **whenever** statement. Therefore, $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 5 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 2 \cdot 1$; i.e., $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 1$. Hence, $\text{Meas}(\mathcal{M}) < \text{Meas}(\mathcal{P})$.

$\mathcal{M}$ of the rule for selection in the absence of an **else** clause (Figure 48) has one fewer **if** statement, two more **alter all** statements, one more **whenever** statement, one more **assume** statement inside a **whenever** statement, and two more **assume** statements at the top level. Therefore, $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 12 \cdot 1 + 1 \cdot 2 + 1 \cdot 1 + 2 \cdot 1 + 1 \cdot 2$; i.e., $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 5$. Hence, $\text{Meas}(\mathcal{M}) < \text{Meas}(\mathcal{P})$.

$\mathcal{M}$ of the rule for selection in the presence of an **else** clause (Figures 49 and 50) has one fewer **if** statement, three more **alter all** statements, two more **whenever** statements, two more **assume** statements inside a **whenever** statement, and two more **assume** statements at the top level. Therefore, $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 12 \cdot 1 + 1 \cdot 3 + 1 \cdot 2 + 2 \cdot 2 + 1 \cdot 2$; i.e., $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 1$. Hence, $\text{Meas}(\mathcal{M}) < \text{Meas}(\mathcal{P})$.

The **loop while** rule's $\mathcal{M}$ (Figures 53 and 54) has one fewer **loop** statement, one more **confirm** statement at the top level, two more **alter all** statements, one more **whenever** statement, two more **assume** statements inside a **whenever** statement, and one more **confirm** statement inside a **whenever** statement. Therefore, $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 11 \cdot 1 + 1 \cdot 1 + 1 \cdot 2 + 1 \cdot 1 + 2 \cdot 2 + 2 \cdot 1$; i.e., $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 1$. Hence, $\text{Meas}(\mathcal{M}) < \text{Meas}(\mathcal{P})$.

The rule for **confirm**'s $\mathcal{M}$ (Figure 55) has one fewer **confirm** statement inside a **whenever** statement and one more **confirm** statement at the top level. Therefore, $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 2 \cdot 1 + 1 \cdot 1$; i.e., $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 1$. Hence, $\text{Meas}(\mathcal{M}) < \text{Meas}(\mathcal{P})$.

$\mathcal{M}$ of the rule for empty guarded blocks (Figure 59) has one fewer **whenever** statement. Therefore, $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 1 \cdot 1$. Hence, $\text{Meas}(\mathcal{M}) < \text{Meas}(\mathcal{P})$.

The rule for **alter all**'s $\mathcal{M}$ (Figure 62) has one fewer **alter all** statement, and one fewer **stow** statement. Therefore, $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 1 \cdot 1 - 1 \cdot 1$. Hence, $\text{Meas}(\mathcal{M}) < \text{Meas}(\mathcal{P})$.

$\mathcal{M}$ of the rule for consecutive **assume** statements (Figure 64) has one fewer **assume** statement at the top level. Therefore, $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 1 \cdot 1$. Hence, $\text{Meas}(\mathcal{M}) < \text{Meas}(\mathcal{P})$.

The **assume-confirm** rule's $\mathcal{M}$ (Figure 66) has one fewer **assume** statement at the top level. Therefore, $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 1 \cdot 1$. Hence, $\text{Meas}(\mathcal{M}) < \text{Meas}(\mathcal{P})$.

$\mathcal{M}$ of the rule for consecutive **confirm** statements (Figure 68) has one fewer **confirm** statement at the top level. Therefore, $\text{Meas}(\mathcal{M}) = \text{Meas}(\mathcal{P}) - 1 \cdot 1$. Hence, $\text{Meas}(\mathcal{M}) < \text{Meas}(\mathcal{P})$.

We have shown that, for each rule of Chapter III (other than the bridge rule), $\text{Meas}(\mathcal{M}) < \text{Meas}(\mathcal{P})$. Therefore, the process of rewriting assertive programs in the math direction must terminate in finitely many steps. We must show, finally, that the result with which the process terminates is a single **confirm** statement so that the rule of inference bridging predicate logic and the indexed method can be applied to it to produce a mathematical assertion.

There is always at least one **confirm** statement in the result of applying a rule in the math direction because (1) the process begins with the bridge rule's $\mathcal{P}$ whose last statement is a **confirm** statement and (2) only one rule, the rule for consecutive **confirm** statements, removes a **confirm** statement when applied in the math direction, and it leaves at least one **confirm** statement in the result. If a top level program

contains a nonempty **whenever** statement, then at least one of the following rules can be applied to it in the math direction: the rule for **assume**, the procedure call rule, a rule for selection, the **loop while** rule, or the rule for **confirm**. The rule for empty guarded blocks can be applied to a top level program if it contains an empty **whenever** statement. If a top level program contains no **whenever** statements but has at least one **alter all** statement, the rule for **alter all** is applicable. If it contains consecutive **assume** statements, there is an (appropriately named!) applicable rule. If the last statement is a **confirm** statement and the penultimate statement is an **assume** statement, the **assume-confirm** rule can be applied. If the program contains consecutive **confirm** statements, there is an applicable rule. If none of the preceding conditions holds for the program, then the program consists of exactly one **confirm** statement. Therefore, the result with which the process terminates is a single **confirm** statement. Hence, the process of rewriting any well-prepared assertive program in the math direction using the rules of Chapter III always terminates with a mathematical assertion. □

4.5.3 Preservation of Invalidity in the Program Direction

At this point we establish Lemma 4.8 by proving a series of lemmas, one for each proof rule of Chapter III, excluding the procedure call rule and the **loop while** rule. Each lemma states that the rule preserves invalidity in the program direction. Skepticism may be greatest about the **if-then-else** statement; therefore, we present its lemma and proof first. The proof is more complicated than the others. Readers wishing a

gentler introduction are invited to skip ahead to some shorter proofs before returning here.

Lemma 4.51 *The rule for selection in the presence of an* **else** *clause preserves invalidity in the program direction.*

Proof. We assume that $\mathcal{M}$ is invalid and show that $\mathcal{P}$ is invalid. Let $\mathrm{env}_I^{\mathcal{M}}$ be a witness to $\mathcal{M}$'s invalidity, and let $\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\mathrm{env}_I^{\mathcal{M}})$; i.e., $\mathrm{AE}(\mathrm{env}_I^{\mathcal{M}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$. We will show the existence of an environment, $\mathrm{env}_I^{\mathcal{P}}$, such that $\mathrm{AE}(\mathrm{env}_I^{\mathcal{P}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$, where $\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\mathrm{env}_I^{\mathcal{P}})$. The proof is organized by cases. First we make the following two definitions.

$$\mathrm{env}_i^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\,ACseq_0)(\mathrm{env}_I^{\mathcal{M}}) \qquad (4.119)$$

$$\mathrm{env}_i^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\,ACseq_0)(\mathrm{env}_I^{\mathcal{P}}) \qquad (4.120)$$

We will be defining other environments as well. Figure 77 shows the positions of the environments defined for $\mathcal{P}$, and figure 78 shows the positions of the environments defined for $\mathcal{M}$.

Case $AE(env_i^{\mathcal{M}}) = CF$. In this case, we take $\mathrm{env}_I^{\mathcal{P}} = \mathrm{env}_I^{\mathcal{M}}$. Then $\mathrm{AE}(\mathrm{env}_i^{\mathcal{P}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $AE(env_i^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_i^{\mathcal{M}}) = \mathrm{NL}$. We may, for this case, write $\mathrm{env}_i^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}, \langle \mathrm{cs}_j, \mathrm{cs}_l, \mathrm{cs}_n \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$, where $\mathrm{ns}(i) = \mathrm{cs}_i$. With the help of Lemmas 4.20 and 4.27, we may also write $\mathrm{env}_I^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \langle \mathrm{cs}_j, \mathrm{cs}_l, \mathrm{cs}_n \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. As shown in Figure 78, we let $\mathrm{env}_{\mathrm{ew}}^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{alter all stow}(j)\ \ldots\ \textbf{whenever}\ \mathrm{Br\_Cd}\ \textbf{do}\ \textit{cd\_suffix}\ \textbf{end whenever})(\mathrm{env}_i^{\mathcal{M}})$.

Similarly, as shown in Figure 77, we let $\mathrm{env}^{\mathcal{P}}_{\mathrm{ew}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{whenever}\ \mathrm{Br\_Cd}\ \textbf{do}\ \ldots\ \textbf{end whenever})(\mathrm{env}^{\mathcal{P}}_i)$. In this case, we take $\mathrm{env}^{\mathcal{P}}_I = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}^{\mathcal{P}}_I, \mathrm{se}_I \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$ where

$$\mathrm{ns}^{\mathcal{P}}_I(h) = \begin{cases} \mathrm{ns}_I(h) & \text{if } h \notin \{j, l, n\} \\ \mathrm{cs}_j & \text{if } h = j \\ \mathrm{cs}_l & \text{if } h = l \\ \mathrm{cs}_n & \text{if } h = n \end{cases} . \tag{4.121}$$

Then $\mathrm{env}^{\mathcal{P}}_i = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}^{\mathcal{P}}_i, \mathrm{se}, \mathrm{os}, \mathrm{d}]$ where

$$\mathrm{ns}^{\mathcal{P}}_i(h) = \begin{cases} \mathrm{ns}(h) & \text{if } h \notin \{j, l, n\} \\ \mathrm{cs}_j & \text{if } h = j \\ \mathrm{cs}_l & \text{if } h = l \\ \mathrm{cs}_n & \text{if } h = n \end{cases} . \tag{4.122}$$

Case $\neg\mathcal{I}(\mathit{Br\_Cd})(\mathit{env}^{\mathcal{M}}_i)$. (We are still inside case $\mathrm{AE}(\mathrm{env}^{\mathcal{M}}_i) \neq \mathrm{CF}$, i.e., $\mathrm{AE}(\mathrm{env}^{\mathcal{M}}_i) = \mathrm{NL}$.) Hence (by Lemma 4.22), $\neg\mathcal{I}(\mathrm{Br\_Cd})(\mathrm{env}^{\mathcal{P}}_i)$. Therefore, by the semantics of the **whenever** statement, $\mathrm{env}^{\mathcal{P}}_{\mathrm{ew}} = \mathrm{env}^{\mathcal{P}}_i$. Similarly, by Lemma 4.22 and the semantics of the **whenever** statement,

$$\mathrm{env}^{\mathcal{M}}_{\mathrm{ew}} = \mathcal{I}\left(\begin{array}{l} \textbf{alter all stow}(j) \\ \textbf{alter all stow}(l) \\ \textbf{alter all stow}(n) \end{array}\right)(\mathrm{env}^{\mathcal{M}}_i). \tag{4.123}$$

Therefore, $\mathrm{env}^{\mathcal{M}}_{\mathrm{ew}} = [\mathrm{NL}, \mathrm{cs}_n, \mathrm{ns}^{\mathcal{M}}_i, \mathrm{se}, \mathrm{os}, \mathrm{d}]$ where $\mathrm{ns}^{\mathcal{M}}_i = \mathrm{ns}^{\mathcal{P}}_i$. Because $\mathrm{AE}(\mathrm{env}^{\mathcal{M}}_{\mathrm{ew}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}^{\mathcal{M}}_F) = \mathrm{CF}$, *fol_top_lev_code* is not empty. The first statement of *fol_top_lev_code* is **alter all** because $\mathcal{M}$ is in phase 1 (as described in Figure 31). Because $\mathrm{env}^{\mathcal{P}}_{\mathrm{ew}}$ differs from $\mathrm{env}^{\mathcal{M}}_{\mathrm{ew}}$ only in the current state (of particular importance is the fact that $\mathrm{SPE}(\mathrm{env}^{\mathcal{P}}_{\mathrm{ew}}) = \mathrm{SPE}(\mathrm{env}^{\mathcal{M}}_{\mathrm{ew}})$), $\mathcal{I}(\textbf{alter all})(\mathrm{env}^{\mathcal{P}}_{\mathrm{ew}}) = \mathcal{I}(\textbf{alter all})(\mathrm{env}^{\mathcal{M}}_{\mathrm{ew}})$. Hence, $\mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}^{\mathcal{P}}_{\mathrm{ew}}) = \mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}^{\mathcal{M}}_{\mathrm{ew}})$; i.e., $\mathrm{env}^{\mathcal{P}}_F = \mathrm{env}^{\mathcal{M}}_F$. Therefore, $\mathrm{AE}(\mathrm{env}^{\mathcal{P}}_F) = \mathrm{CF}$.

Case $\mathcal{I}(Br\_Cd)(env_i^{\mathcal{M}})$. (We are still inside case $\mathrm{AE}(\mathrm{env}_i^{\mathcal{M}}) \neq \mathrm{CF}$, i.e., $\mathrm{AE}(\mathrm{env}_i^{\mathcal{M}}) = \mathrm{NL}$.) Hence (by Lemma 4.22), $\mathcal{I}(\mathrm{Br\_Cd})(\mathrm{env}_i^{\mathcal{P}})$. Let $\mathrm{env}_n^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{if } b\_p\_e \textbf{ then } \ldots \textbf{ end if } \textbf{stow}(n))(\mathrm{env}_i^{\mathcal{P}})$. Therefore, $\mathrm{env}_{\mathrm{ew}}^{\mathcal{P}} = \mathcal{I}(cd\_suffix)(\mathrm{env}_n^{\mathcal{P}})$.

Case $\mathcal{I}(\mathrm{b\_p\_e})(env_i^{\mathcal{M}})$. In each of the environments $\mathrm{env}_i^{\mathcal{M}}$, $\mathrm{env}_j^{\mathcal{M}}$, $\mathrm{env}_l^{\mathcal{M}}$, and $\mathrm{env}_n^{\mathcal{M}}$ (shown in Figure 78), we have, for $h \in \{i, j, l, n\}$, $\mathrm{ISE}(\mathrm{env}_h^{\mathcal{M}})(i) = \mathrm{ns}(i)$. Therefore, we also have, for $h \in \{i, j, l, n\}$, $\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])(\mathrm{env}_h^{\mathcal{M}})$. Because $\mathrm{ISE}(\mathrm{env}_h^{\mathcal{P}})(i) = \mathrm{ns}(i)$, $\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])(\mathrm{env}_i^{\mathcal{P}})$. According to the semantics of **if-then-else**, $\mathrm{env}_n^{\mathcal{P}} = \mathcal{I}(\textbf{stow}(j)\ cd\_kern_1\ \textbf{stow}(k)\ ACseq_1\ \textbf{stow}(n))(\mathrm{env}_i^{\mathcal{P}})$. According to the semantics of **whenever**, $\mathrm{env}_n^{\mathcal{M}} = \mathcal{I}(\textbf{alter all stow}(j)\ \textbf{assume}\ \mathrm{x}_j = \mathrm{x}_i\ cd\_kern_1\ \textbf{stow}(k)\ ACseq_1\ \textbf{alter all stow}(l)\ \textbf{alter all stow}(n))(\mathrm{env}_i^{\mathcal{M}})$.

Case $AE(env_n^{\mathcal{M}}) = CF$. By Lemma 4.16, $\mathrm{AE}(\mathcal{I}(\textbf{alter all stow}(j)\ \textbf{assume}\ \mathrm{x}_j = \mathrm{x}_i)(\mathrm{env}_i^{\mathcal{M}})) \neq \mathrm{VT}$. Therefore, $\mathrm{cs}_j = \mathrm{ns}(i) = \mathrm{cs}_i$. In this case, we must have $\mathrm{AE}(\mathcal{I}(\textbf{assume}\ \mathrm{x}_j = \mathrm{x}_i\ cd\_kern_1\ \textbf{stow}(k)\ ACseq_1)(\mathrm{env}_j^{\mathcal{M}})) = \mathrm{CF}$. Hence, we also have $\mathrm{AE}(\mathrm{env}_n^{\mathcal{P}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $AE(env_n^{\mathcal{M}}) \neq CF$. By Lemma 4.19, $\mathrm{AE}(\mathrm{env}_n^{\mathcal{M}}) = \mathrm{NL}$. By Lemma 4.16, $\mathrm{AE}(\mathcal{I}(\textbf{assume}\ ((\mathrm{Br\_Cd}) \wedge (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])) \Rightarrow (\mathrm{x}_n = \mathrm{x}_k))(\mathrm{env}_n^{\mathcal{M}})) \neq \mathrm{VT}$. Therefore, $\mathrm{CSE}(\mathrm{env}_n^{\mathcal{M}}) = \mathrm{cs}_n = \mathrm{CSE}(\mathcal{I}(\textbf{assume}\ \mathrm{x}_j = \mathrm{x}_i\ cd\_kern_1\ \textbf{stow}(k)\ ACseq_1)(\mathrm{env}_j^{\mathcal{M}})) = \mathrm{CSE}(\mathrm{env}_n^{\mathcal{P}})$. Hence, $\mathrm{Equal\_except\_at}(\{l\}, \mathrm{env}_n^{\mathcal{P}}, \mathrm{env}_n^{\mathcal{M}})$. By two applications of Lemma 4.37, $\mathrm{Equal\_except\_at}(\{l\}, \mathrm{env}_F^{\mathcal{P}}, \mathrm{env}_F^{\mathcal{M}})$. Therefore, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $\neg\mathcal{I}(\mathrm{b\_p\_e})(env_i^{\mathcal{M}})$. In this case, we have, for $h \in \{i, j, l, n\}$, $\neg\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])(\mathrm{env}_h^{\mathcal{M}})$. We also have $\neg\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow$

$y_i]$)($env_i^{\mathcal{P}}$). According to the semantics of **if-then-else**, $env_n^{\mathcal{P}}$ = $\mathcal{I}$(**stow**(l) *cd_kern*$_2$ **stow**(m) *ACseq*$_2$ **stow**(n))($env_i^{\mathcal{P}}$). According to the semantics of **whenever**, $env_l^{\mathcal{M}} = \mathcal{I}$(**alter all stow**($j$) **alter all stow**($l$))($env_i^{\mathcal{M}}$) and $env_n^{\mathcal{M}} = \mathcal{I}$(**assume** $x_l = x_i$ *cd_kern*$_2$ **stow**(m) *ACseq*$_2$ **alter all stow**(n))($env_l^{\mathcal{M}}$).

Case AE($env_n^{\mathcal{M}}$) = *CF.* By Lemma 4.16, AE($\mathcal{I}$(**assume** $x_l = x_i$)($env_l^{\mathcal{M}}$)) $\neq$ VT. Therefore, $cs_l = ns(i) = cs_i$. In this case, we must have AE($\mathcal{I}$(**assume** $x_l = x_i$ *cd_kern*$_2$ **stow**(m) *ACseq*$_2$)($env_l^{\mathcal{M}}$)) = CF. Hence, we also have AE($env_n^{\mathcal{P}}$) = CF. By Lemma 4.15, AE($env_F^{\mathcal{P}}$) = CF.

Case AE($env_n^{\mathcal{M}}$) $\neq$ *CF.* By Lemma 4.19, AE($env_n^{\mathcal{M}}$) = NL. By Lemma 4.16, AE($\mathcal{I}$(**assume** ((Br_Cd) $\wedge$ $\neg$(MExp(b_p_e)[y $\rightsquigarrow$ y_i])) $\Rightarrow$ ($x_n = x_m$))($env_n^{\mathcal{M}}$)) $\neq$ VT. Therefore, CSE($env_n^{\mathcal{M}}$) = cs_n = CSE($\mathcal{I}$(**assume** $x_l = x_i$ *cd_kern*$_2$ **stow**(m) *ACseq*$_2$)($env_l^{\mathcal{M}}$)) = CSE($env_n^{\mathcal{P}}$). Hence, Equal_except_at($\{j\}, env_n^{\mathcal{P}}, env_n^{\mathcal{M}}$). By two applications of Lemma 4.37, Equal_except_at($\{j\}, env_F^{\mathcal{P}}, env_F^{\mathcal{M}}$). Therefore, AE($env_F^{\mathcal{P}}$) = CF. □

Lemma 4.52 *The rule for selection in the absence of an* **else** *clause preserves invalidity in the program direction.*

Proof. The proof is similar to the proof that the rule for selection in the presence of an **else** clause preserves invalidity in the program direction. The important difference is in the case in which $\neg\mathcal{I}(b\_p\_e)(env_i^{\mathcal{M}})$ (within the cases AE($env_i^{\mathcal{M}}$) $\neq$ CF and $\mathcal{I}$(Br_Cd)($env_i^{\mathcal{M}}$)). We supply here only the argument for this case. The environments defined in the complete proof are similar to those defined for the rule for selection in the presence of an **else** clause. Figure 79 shows the positions of the environments

defined for $\mathcal{P}$, and figure 80 shows the positions of the environments defined for $\mathcal{M}$. In the case that $\mathrm{AE}(\mathrm{env}_i^{\mathcal{M}}) \neq \mathrm{CF}$, because $\mathcal{M}$ of the rule for selection in the absence of an **else** clause introduces two—and not three—**alter all** statements, we may write $\mathrm{env}_i^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}, \langle \mathrm{cs}_j, \mathrm{cs}_n \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$, where $\mathrm{ns}(i) = \mathrm{cs}_i$. With the help of Lemmas 4.20 and 4.27, we may also write $\mathrm{env}_I^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \langle \mathrm{cs}_j, \mathrm{cs}_n \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. In this case, we take $\mathrm{env}_I^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I^{\mathcal{P}}, \mathrm{se}_I \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$ where

$$\mathrm{ns}_I^{\mathcal{P}}(h) = \begin{cases} \mathrm{ns}_I(h) & \text{if } h \notin \{j, n\} \\ \mathrm{cs}_j & \text{if } h = j \\ \mathrm{cs}_n & \text{if } h = n \end{cases} . \tag{4.124}$$

Then $\mathrm{env}_i^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}_i^{\mathcal{P}}, \mathrm{se}, \mathrm{os}, \mathrm{d}]$ where

$$\mathrm{ns}_i^{\mathcal{P}}(h) = \begin{cases} \mathrm{ns}(h) & \text{if } h \notin \{j, n\} \\ \mathrm{cs}_j & \text{if } h = j \\ \mathrm{cs}_n & \text{if } h = n \end{cases} . \tag{4.125}$$

Case $\neg\mathcal{I}(\mathrm{b\_p\_e})(env_i^{\mathcal{M}})$. In this case, we have, for $h \in \{i, j, n\}$, $\neg\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])(\mathrm{env}_h^{\mathcal{M}})$. We also have $\neg\mathcal{I}(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])(\mathrm{env}_i^{\mathcal{P}})$. According to the semantics of **if-then**, $\mathrm{env}_n^{\mathcal{P}} = \mathcal{I}(\mathbf{stow}(n))(\mathrm{env}_i^{\mathcal{P}})$. According to the semantics of **whenever**, $\mathrm{env}_n^{\mathcal{M}} = \mathcal{I}(\mathbf{alter\ all\ stow}(j)\ \mathbf{alter\ all\ stow}(n))(\mathrm{env}_i^{\mathcal{M}})$. Because $\mathrm{AE}(\mathrm{env}_i^{\mathcal{M}}) \neq \mathrm{CF}$, $\mathrm{AE}(\mathrm{env}_n^{\mathcal{M}}) \neq \mathrm{CF}$. By Lemma 4.19, $\mathrm{AE}(\mathrm{env}_n^{\mathcal{M}}) = \mathrm{NL}$. By Lemma 4.16, $\mathrm{AE}(\mathcal{I}(\mathbf{assume}\ ((\mathrm{Br\_Cd}) \wedge \neg(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])) \Rightarrow (\mathrm{x}_n = \mathrm{x}_i))(\mathrm{env}_n^{\mathcal{M}})) \neq \mathrm{VT}$. Therefore, $\mathrm{CSE}(\mathrm{env}_n^{\mathcal{M}}) = \mathrm{ns}(i) = \mathrm{cs}_i = \mathrm{CSE}(\mathrm{env}_n^{\mathcal{P}})$. Hence, $\mathrm{Equal\_except\_at}(\{j\}, \mathrm{env}_n^{\mathcal{P}}, \mathrm{env}_n^{\mathcal{M}})$. By two applications of Lemma 4.37, $\mathrm{Equal\_except\_at}(\{j\}, \mathrm{env}_F^{\mathcal{P}}, \mathrm{env}_F^{\mathcal{M}})$. Therefore, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$. □

Lemma 4.53 *The bridge rule preserves invalidity in the program direction.*

Proof. We assume that $\mathcal{M}$ is invalid and show that $\mathcal{P}$ is invalid. Let $\mathrm{env}_I^{\mathcal{M}}$ be a witness to $\mathcal{M}$'s invalidity, and let $\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\mathrm{env}_I^{\mathcal{M}})$; i.e., $\mathrm{AE}(\mathrm{env}_I^{\mathcal{M}}) = \mathrm{NL}$ and

$\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$. We will show the existence of an environment, $\mathrm{env}_I^{\mathcal{P}}$, such that $\mathrm{AE}(\mathrm{env}_I^{\mathcal{P}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$, where $\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\mathrm{env}_I^{\mathcal{P}})$. We may write $\mathrm{env}_I^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}', \mathrm{ns}, \langle \mathrm{cs} \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. We take $\mathrm{env}_I^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}]$. By the semantics of **whenever**,

$$\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}\left(\begin{array}{l}\textbf{alter all}\\ \textbf{stow}(i)\\ \textbf{assume}\ \mathrm{pre}[\mathrm{x} \rightsquigarrow \mathrm{x}_i] \wedge \textbf{is\_initial}(\mathrm{z}_i)\\ \mathrm{Stows\_added}(p\_body)\\ \textbf{stow}(j)\\ \textbf{confirm}\ \mathrm{post}[\#\mathrm{x} \rightsquigarrow \mathrm{x}_i, \mathrm{x} \rightsquigarrow \mathrm{x}_j]\end{array}\right)(\mathrm{env}_I^{\mathcal{M}}). \tag{4.126}$$

Let

$$\mathrm{env}_j^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}\left(\begin{array}{l}\textbf{alter all}\\ \textbf{stow}(i)\\ \textbf{assume}\ \mathrm{pre}[\mathrm{x} \rightsquigarrow \mathrm{x}_i] \wedge \textbf{is\_initial}(\mathrm{z}_i)\\ \mathrm{Stows\_added}(p\_body)\\ \textbf{stow}(j)\end{array}\right)(\mathrm{env}_I^{\mathcal{M}}). \tag{4.127}$$

By Lemma 4.16, $\mathcal{I}(\mathrm{pre}[\mathrm{x} \rightsquigarrow \mathrm{x}_i] \wedge \textbf{is\_initial}(\mathrm{z}_i))(\mathcal{I}(\textbf{alter all}\ \textbf{stow}(i))(\mathrm{env}_I^{\mathcal{M}}))$. Therefore, because $\mathrm{cs} = \mathrm{ISE}(\mathcal{I}(\textbf{alter all}\ \textbf{stow}(i))(\mathrm{env}_I^{\mathcal{M}}))(i)$, $\mathcal{I}(\mathrm{pre}\lfloor \mathrm{x} \rfloor \wedge \textbf{is\_initial}(\mathrm{z}))(\mathcal{I}(\textbf{remember})(\mathrm{env}_I^{\mathcal{P}}))$. Note also that $\mathrm{cs} = \mathrm{CSE}(\mathcal{I}(\textbf{alter all}\ \textbf{stow}(i))(\mathrm{env}_I^{\mathcal{M}}))$.

Case $AE(env_j^{\mathcal{M}}) = CF$. Because *p_body* and $\mathrm{Stows\_added}(p\_body)$ are unaffected by old state (Lemma 4.30), *p_body* contains no indexed variables, and the interpretation of any **stow** statement changes at most the index state of an environment, $\mathrm{AE}(\mathcal{I}(\textbf{remember assume}\ \mathrm{pre}\lfloor \mathrm{x} \rfloor \wedge \textbf{is\_initial}(\mathrm{z})\ p\_body)(\mathrm{env}_I^{\mathcal{P}})) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $AE(env_j^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_j^{\mathcal{M}}) = \mathrm{NL}$. Note that $\mathrm{ISE}(\mathrm{env}_j^{\mathcal{M}})(i) = \mathrm{cs}$. Because *p_body* and $\mathrm{Stows\_added}(p\_body)$ are unaffected by old state (Lemma 4.30), *p_body* contains no indexed variables,

and the interpretation of any **stow** statement changes at most the index state of an environment, $\mathrm{CSE}(\mathrm{env}_j^{\mathcal{M}}) = \mathrm{CSE}(\mathcal{I}(\textbf{remember assume } \mathrm{pre}\lfloor \mathrm{x} \rfloor \wedge \textbf{is\_initial}(\mathrm{z})\ \ p\_body)(\mathrm{env}_I^{\mathcal{P}}))$. Note also that $\mathrm{ISE}(\mathrm{env}_j^{\mathcal{M}})(j) = \mathrm{CSE}(\mathrm{env}_j^{\mathcal{M}})$. In this case, it must be that $\neg\mathcal{I}(\mathrm{post}[\#\mathrm{x} \rightsquigarrow \mathrm{x}_i, \mathrm{x} \rightsquigarrow \mathrm{x}_j])(\mathrm{env}_j^{\mathcal{M}})$. Because $\mathrm{OSE}(\mathcal{I}(\textbf{remember assume } \mathrm{pre}\lfloor \mathrm{x} \rfloor \wedge \textbf{is\_initial}(\mathrm{z})\ \ p\_body)(\mathrm{env}_I^{\mathcal{P}}))(\#\xi) = \mathrm{cs}(\xi)$, $\neg\mathcal{I}(\mathrm{post}\lfloor \#\mathrm{x}, \mathrm{x} \rfloor)(\mathcal{I}(\textbf{remember assume } \mathrm{pre}\lfloor \mathrm{x} \rfloor \wedge \textbf{is\_initial}(\mathrm{z})\ \ p\_body)(\mathrm{env}_I^{\mathcal{P}}))$. Therefore, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$. □

Lemma 4.54 *The rule for* **assume** *preserves invalidity in the program direction.*

Proof. We assume that $\mathcal{M}$ is invalid and show that $\mathcal{P}$ is invalid. Let $\mathrm{env}_I^{\mathcal{M}}$ be a witness to $\mathcal{M}$'s invalidity, and let $\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\mathrm{env}_I^{\mathcal{M}})$; i.e., $\mathrm{AE}(\mathrm{env}_I^{\mathcal{M}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$. We will show the existence of an environment, $\mathrm{env}_I^{\mathcal{P}}$, such that $\mathrm{AE}(\mathrm{env}_I^{\mathcal{P}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$, where $\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\mathrm{env}_I^{\mathcal{P}})$. The proof is organized by cases. In each case we take $\mathrm{env}_I^{\mathcal{P}} = \mathrm{env}_I^{\mathcal{M}}$. First we make the following two definitions.

$$\mathrm{env}_i^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(prec\_top\_lev\_code\ \textbf{alter all stow}(i)\,ACseq_0)(\mathrm{env}_I^{\mathcal{M}}) \qquad (4.128)$$

$$\mathrm{env}_i^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(prec\_top\_lev\_code\ \textbf{alter all stow}(i)\,ACseq_0)(\mathrm{env}_I^{\mathcal{P}}) \qquad (4.129)$$

Consequently, $\mathrm{env}_i^{\mathcal{P}} = \mathrm{env}_i^{\mathcal{M}}$.

Case $AE(env_i^{\mathcal{M}}) = CF$. In this case, $\mathrm{AE}(\mathrm{env}_i^{\mathcal{P}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $AE(env_i^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_i^{\mathcal{M}}) = \mathrm{NL}$.

Case $\neg\mathcal{I}(Br\_Cd)(env_i^{\mathcal{M}})$. In this case, $\neg\mathcal{I}(\mathrm{Br\_Cd})(\mathrm{env}_i^{\mathcal{P}})$. By the semantics of

whenever, $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{assume}\ (\text{Br\_Cd}) \Rightarrow (H)\ \textit{fol\_top\_lev\_code})(\text{env}_i^{\mathcal{M}})$ and $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\textit{fol\_top\_lev\_code})(\text{env}_i^{\mathcal{P}})$. Because $\neg\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{M}})$, $\mathcal{I}((\text{Br\_Cd}) \Rightarrow (H))(\text{env}_i^{\mathcal{M}})$. Therefore, $\mathcal{I}(\textbf{assume}\ (\text{Br\_Cd}) \Rightarrow (H))(\text{env}_i^{\mathcal{M}}) = \text{env}_i^{\mathcal{M}}$. By equation 2.23, $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textit{fol\_top\_lev\_code})(\text{env}_i^{\mathcal{M}})$. Hence, $\text{env}_F^{\mathcal{P}} = \text{env}_F^{\mathcal{M}}$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$.

Case $\mathcal{I}(\textit{Br\_Cd})(\textit{env}_i^{\mathcal{M}})$. In this case, $\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{P}})$. By the semantics of **whenever**, $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\textbf{assume}\ H\ \textit{cd\_suffix}\ \textit{fol\_top\_lev\_code})(\text{env}_i^{\mathcal{P}})$ and $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{assume}\ (\text{Br\_Cd}) \Rightarrow (H)\ \textit{cd\_suffix}\ \textit{fol\_top\_lev\_code})(\text{env}_i^{\mathcal{M}})$. Because $\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{M}})$, $\mathcal{I}(\textbf{assume}\ (\text{Br\_Cd}) \Rightarrow (H))(\text{env}_i^{\mathcal{M}}) = \mathcal{I}(\textbf{assume}\ H)(\text{env}_i^{\mathcal{M}})$. By two applications of equation 2.23, $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{assume}\ H\ \textit{cd\_suffix}\ \textit{fol\_top\_lev\_code})(\text{env}_i^{\mathcal{M}})$. Hence, $\text{env}_F^{\mathcal{P}} = \text{env}_F^{\mathcal{M}}$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$. □

Lemma 4.55 *The rule for* **confirm** *preserves invalidity in the program direction.*

Proof. We assume that $\mathcal{M}$ is invalid and show that $\mathcal{P}$ is invalid. Let $\text{env}_I^{\mathcal{M}}$ be a witness to $\mathcal{M}$'s invalidity, and let $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\text{env}_I^{\mathcal{M}})$; i.e., $\text{AE}(\text{env}_I^{\mathcal{M}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. We will show the existence of an environment, $\text{env}_I^{\mathcal{P}}$, such that $\text{AE}(\text{env}_I^{\mathcal{P}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$, where $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\text{env}_I^{\mathcal{P}})$. The proof is organized by cases. In each case we take $\text{env}_I^{\mathcal{P}} = \text{env}_I^{\mathcal{M}}$. First we make the following two definitions.

$$\text{env}_i^{\mathcal{M}} \stackrel{\text{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\,ACseq_0)(\text{env}_I^{\mathcal{M}}) \quad (4.130)$$

$$\text{env}_i^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\,ACseq_0)(\text{env}_I^{\mathcal{P}}) \quad (4.131)$$

Consequently, $\text{env}_i^{\mathcal{P}} = \text{env}_i^{\mathcal{M}}$. Furthermore, $\text{ISE}(\text{env}_i^{\mathcal{M}})(i) = \text{CSE}(\text{env}_i^{\mathcal{M}}) = \text{CSE}(\text{env}_i^{\mathcal{P}})$.

Case $AE(env_i^{\mathcal{M}}) = CF$. In this case, $\text{AE}(\text{env}_i^{\mathcal{P}}) = \text{CF}$. By Lemma 4.15, $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$.

Case $AE(env_i^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\text{AE}(\text{env}_i^{\mathcal{M}}) = \text{NL}$.

Case $\neg\mathcal{I}(Br\_Cd)(env_i^{\mathcal{M}})$. In this case, $\neg\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{P}})$. By the semantics of **whenever**, $\text{env}_F^{\mathcal{P}} = \mathcal{I}(fol\_top\_lev\_code)(\text{env}_i^{\mathcal{P}})$ and $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{confirm}\ (\text{Br\_Cd}) \Rightarrow (H[\text{x} \rightsquigarrow \text{x}_i])\ fol\_top\_lev\_code)(\text{env}_i^{\mathcal{M}})$. Because $\neg\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{M}})$, $\mathcal{I}((\text{Br\_Cd}) \Rightarrow (H[\text{x} \rightsquigarrow \text{x}_i]))(\text{env}_i^{\mathcal{M}})$. Therefore, $\mathcal{I}(\textbf{confirm}\ (\text{Br\_Cd}) \Rightarrow (H)[\text{x} \rightsquigarrow \text{x}_i])(\text{env}_i^{\mathcal{M}}) = \text{env}_i^{\mathcal{M}}$. By equation 2.23, $\text{env}_F^{\mathcal{M}} = \mathcal{I}(fol\_top\_lev\_code)(\text{env}_i^{\mathcal{M}})$. Hence, $\text{env}_F^{\mathcal{P}} = \text{env}_F^{\mathcal{M}}$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$.

Case $\mathcal{I}(Br\_Cd)(env_i^{\mathcal{M}})$. In this case, $\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{P}})$. By the semantics of **whenever**, $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\textbf{confirm}\ H\lfloor\text{x}\rfloor\ cd\_suffix\ fol\_top\_lev\_code)(\text{env}_i^{\mathcal{P}})$ and $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{confirm}\ (\text{Br\_Cd}) \Rightarrow (H[\text{x} \rightsquigarrow \text{x}_i])\ cd\_suffix\ fol\_top\_lev\_code)(\text{env}_i^{\mathcal{M}})$. Because $\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{M}})$, $\mathcal{I}(\textbf{confirm}\ (\text{Br\_Cd}) \Rightarrow (H[\text{x} \rightsquigarrow \text{x}_i]))(\text{env}_i^{\mathcal{M}}) = \mathcal{I}(\textbf{confirm}\ H[\text{x} \rightsquigarrow \text{x}_i])(\text{env}_i^{\mathcal{M}})$. By two applications of equation 2.23, $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{confirm}\ H[\text{x} \rightsquigarrow \text{x}_i]\ cd\_suffix\ fol\_top\_lev\_code)(\text{env}_i^{\mathcal{M}})$. Because $\text{ISE}(\text{env}_i^{\mathcal{M}})(i) = \text{CSE}(\text{env}_i^{\mathcal{P}})$, $\mathcal{I}(\textbf{confirm}\ H[\text{x} \rightsquigarrow \text{x}_i])(\text{env}_i^{\mathcal{M}}) = \mathcal{I}(\textbf{confirm}\ H\lfloor\text{x}\rfloor)(\text{env}_i^{\mathcal{P}})$. By two applications of equation 2.23, $\text{env}_F^{\mathcal{P}} = \text{env}_F^{\mathcal{M}}$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$. □

Lemma 4.56 *The rule for empty guarded blocks preserves invalidity in the program direction.*

Proof. We assume that $\mathcal{M}$ is invalid and show that $\mathcal{P}$ is invalid. Let $\text{env}_I^{\mathcal{M}}$ be

a witness to $\mathcal{M}$'s invalidity, and let $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\text{env}_I^{\mathcal{M}})$; i.e., $\text{AE}(\text{env}_I^{\mathcal{M}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. We will show the existence of an environment, $\text{env}_I^{\mathcal{P}}$, such that $\text{AE}(\text{env}_I^{\mathcal{P}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$, where $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\text{env}_I^{\mathcal{P}})$. We take $\text{env}_I^{\mathcal{P}} = \text{env}_I^{\mathcal{M}}$ and make the following two definitions.

$$\text{env}_i^{\mathcal{M}} \stackrel{\text{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\mathit{ACseq}_0)(\text{env}_I^{\mathcal{M}}) \quad (4.132)$$

$$\text{env}_i^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\mathit{ACseq}_0)(\text{env}_I^{\mathcal{P}}) \quad (4.133)$$

Consequently, $\text{env}_i^{\mathcal{P}} = \text{env}_i^{\mathcal{M}}$.

Case $AE(env_i^{\mathcal{M}}) = CF$. In this case, $\text{AE}(\text{env}_i^{\mathcal{P}}) = \text{CF}$. By Lemma 4.15, $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$.

Case $AE(env_i^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\text{AE}(\text{env}_i^{\mathcal{M}}) = \text{NL}$. Whether $\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{P}})$ or $\neg\mathcal{I}(\text{Br\_Cd})(\text{env}_i^{\mathcal{P}})$, $\mathcal{I}(\textbf{whenever}\ \text{Br\_Cd}\ \textbf{do end whenever})(\text{env}_i^{\mathcal{P}}) = \mathcal{I}(\varepsilon)(\text{env}_i^{\mathcal{P}}) = \text{env}_i^{\mathcal{P}}$. By equation 2.23, $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathit{fol\_top\_lev\_code})(\text{env}_i^{\mathcal{P}})$. Because $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\mathit{fol\_top\_lev\_code})(\text{env}_i^{\mathcal{M}})$, $\text{env}_F^{\mathcal{M}} = \text{env}_F^{\mathcal{P}}$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$. □

Lemma 4.57 *The rule for* **alter all** *preserves invalidity in the program direction.*

Proof. We assume that $\mathcal{M}$ is invalid and show that $\mathcal{P}$ is invalid. Let $\text{env}_I^{\mathcal{M}}$ be a witness to $\mathcal{M}$'s invalidity, and let $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\text{env}_I^{\mathcal{M}})$; i.e., $\text{AE}(\text{env}_I^{\mathcal{M}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. We will show the existence of an environment, $\text{env}_I^{\mathcal{P}}$, such that $\text{AE}(\text{env}_I^{\mathcal{P}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$, where $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\text{env}_I^{\mathcal{P}})$. The proof is organized by cases. First we make the following four definitions.

$$\text{env}_1^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code})(\text{env}_I^{\mathcal{P}}) \quad (4.134)$$

$$\mathrm{env}_a^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{alter all})(\mathrm{env}_1^{\mathcal{P}}) \tag{4.135}$$

$$\mathrm{env}_s^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{stow}(i))(\mathrm{env}_a^{\mathcal{P}}) \tag{4.136}$$

$$\mathrm{env}_1^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code})(\mathrm{env}_I^{\mathcal{M}}) \tag{4.137}$$

Consequently,

$$\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}_s^{\mathcal{P}}) \tag{4.138}$$

$$\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}_1^{\mathcal{M}}). \tag{4.139}$$

Case $AE(env_1^{\mathcal{M}}) = CF$. In this case, we take $\mathrm{env}_I^{\mathcal{P}} = \mathrm{env}_I^{\mathcal{M}}$. Hence, $\mathrm{env}_1^{\mathcal{P}} = \mathrm{env}_1^{\mathcal{M}}$. Therefore, $\mathrm{AE}(\mathrm{env}_1^{\mathcal{P}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $AE(env_1^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_1^{\mathcal{M}}) = \mathrm{NL}$. We may, for this case, write $\mathrm{env}_1^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_1, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}]$. With the help of Lemmas 4.20 and 4.27, we may also write $\mathrm{env}_I^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. In this case, we take $\mathrm{env}_I^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \langle \mathrm{ns}(i) \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. Then $\mathrm{env}_1^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_1, \mathrm{ns}, \langle \mathrm{ns}(i) \rangle \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. By the semantics of **alter all**, $\mathrm{env}_a^{\mathcal{P}} = [\mathrm{NL}, \mathrm{ns}(i), \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}]$. By the semantics of **stow**(i), $\mathrm{env}_s^{\mathcal{P}} = \mathrm{env}_a^{\mathcal{P}}$. Therefore, $\mathrm{env}_s^{\mathcal{P}}$ differs from $\mathrm{env}_1^{\mathcal{M}}$ only in the current state. Due to the additional syntactic restriction of the rule for **alter all**, *fol_top_lev_code* contains only **alter all**, **stow**, **assume**, and **confirm** statements. The **assume** and **confirm** statements refer only to indexed variables. Hence, **stow** is the only one of these statements whose semantics is affected by the current state; however, every one of these **stow** statements is immediately preceded by an **alter all** statement. Therefore, $\mathrm{CSE}(\mathrm{env}_1^{\mathcal{M}})$ and $\mathrm{CSE}(\mathrm{env}_s^{\mathcal{P}})$ have no effect on the values, respectively, of

$\mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}_1^{\mathcal{M}})$ and $\mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}_s^{\mathcal{P}})$. Therefore, $\mathrm{env}_F^{\mathcal{M}}$ and $\mathrm{env}_F^{\mathcal{P}}$ differ at most in the current state. In particular, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$. □

Lemma 4.58 *The rule for consecutive* **assume** *statements preserves invalidity in the program direction.*

Proof. We assume that $\mathcal{M}$ is invalid and show that $\mathcal{P}$ is invalid. Let $\mathrm{env}_I^{\mathcal{M}}$ be a witness to $\mathcal{M}$'s invalidity, and let $\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\mathrm{env}_I^{\mathcal{M}})$; i.e., $\mathrm{AE}(\mathrm{env}_I^{\mathcal{M}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$. We will show the existence of an environment, $\mathrm{env}_I^{\mathcal{P}}$, such that $\mathrm{AE}(\mathrm{env}_I^{\mathcal{P}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$, where $\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\mathrm{env}_I^{\mathcal{P}})$. The proof is organized by cases. In each case we take $\mathrm{env}_I^{\mathcal{P}} = \mathrm{env}_I^{\mathcal{M}}$. First we make the following five definitions.

$$\mathrm{env}_1^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code})(\mathrm{env}_I^{\mathcal{P}}) \tag{4.140}$$

$$\mathrm{env}_2^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{assume } H_1)(\mathrm{env}_1^{\mathcal{P}}) \tag{4.141}$$

$$\mathrm{env}_b^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{assume } H_2)(\mathrm{env}_2^{\mathcal{P}}) \tag{4.142}$$

$$\mathrm{env}_1^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code})(\mathrm{env}_I^{\mathcal{M}}) \tag{4.143}$$

$$\mathrm{env}_b^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{assume } (H_1) \wedge (H_2))(\mathrm{env}_1^{\mathcal{M}}) \tag{4.144}$$

Consequently,

$$\mathrm{env}_1^{\mathcal{P}} = \mathrm{env}_1^{\mathcal{M}} \tag{4.145}$$

$$\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}_b^{\mathcal{P}}) \tag{4.146}$$

$$\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathit{fol\_top\_lev\_code})(\mathrm{env}_b^{\mathcal{M}}). \tag{4.147}$$

Case $AE(env_1^{\mathcal{M}}) = CF$. Therefore, $\mathrm{AE}(\mathrm{env}_1^{\mathcal{P}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $AE(env_1^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_1^{\mathcal{M}}) = \mathrm{NL}$. By substitution of equals, $\mathrm{AE}(\mathrm{env}_1^{\mathcal{P}}) = \mathrm{NL}$. By Lemma 4.18 and the semantics of the **assume** statement, $\mathrm{AE}(\mathrm{env}_b^{\mathcal{M}}) = \mathrm{NL}$. Therefore, $\mathcal{I}((H_1) \wedge (H_2))(\mathrm{env}_1^{\mathcal{M}})$ and $\mathrm{env}_b^{\mathcal{M}} = \mathrm{env}_1^{\mathcal{M}}$. Hence, $\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{M}})$ and $\mathcal{I}(H_2)(\mathrm{env}_1^{\mathcal{M}})$. Therefore, $\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{P}})$. Hence, $\mathrm{env}_2^{\mathcal{P}} = \mathrm{env}_1^{\mathcal{P}}$. Therefore, $\mathcal{I}(H_2)(\mathrm{env}_2^{\mathcal{P}})$. Hence, $\mathrm{env}_b^{\mathcal{P}} = \mathrm{env}_2^{\mathcal{P}} = \mathrm{env}_1^{\mathcal{P}}$. Therefore, $\mathrm{env}_b^{\mathcal{P}} = \mathrm{env}_b^{\mathcal{M}}$. Hence, $\mathrm{env}_F^{\mathcal{P}} = \mathrm{env}_F^{\mathcal{M}}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$. □

Lemma 4.59 *The* **assume-confirm** *rule preserves invalidity in the program direction.*

Proof. We assume that $\mathcal{M}$ is invalid and show that $\mathcal{P}$ is invalid. Let $\mathrm{env}_I^{\mathcal{M}}$ be a witness to $\mathcal{M}$'s invalidity, and let $\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\mathrm{env}_I^{\mathcal{M}})$; i.e., $\mathrm{AE}(\mathrm{env}_I^{\mathcal{M}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$. We will show the existence of an environment, $\mathrm{env}_I^{\mathcal{P}}$, such that $\mathrm{AE}(\mathrm{env}_I^{\mathcal{P}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$, where $\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\mathrm{env}_I^{\mathcal{P}})$. The proof is organized by cases. In each case we take $\mathrm{env}_I^{\mathcal{P}} = \mathrm{env}_I^{\mathcal{M}}$. First we make the following three definitions.

$$\mathrm{env}_1^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(top\_lev\_code)(\mathrm{env}_I^{\mathcal{P}}) \tag{4.148}$$

$$\mathrm{env}_2^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{assume } H_1)(\mathrm{env}_1^{\mathcal{P}}) \tag{4.149}$$

$$\mathrm{env}_1^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(top\_lev\_code)(\mathrm{env}_I^{\mathcal{M}}) \tag{4.150}$$

Consequently,

$$\mathrm{env}_1^{\mathcal{M}} = \mathrm{env}_1^{\mathcal{P}} \tag{4.151}$$

$$\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\textbf{confirm } H_2)(\mathrm{env}_2^{\mathcal{P}}) \tag{4.152}$$

$$\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\textbf{confirm } (H_1) \Rightarrow (H_2))(\mathrm{env}_1^{\mathcal{M}}). \tag{4.153}$$

Case $AE(env_1^{\mathcal{M}}) = CF$. Therefore, $\mathrm{AE}(\mathrm{env}_1^{\mathcal{P}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $AE(env_1^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_1^{\mathcal{M}}) = \mathrm{NL}$. By substitution of equals, $\mathrm{AE}(\mathrm{env}_1^{\mathcal{P}}) = \mathrm{NL}$. By the semantics of the **confirm** statement, $\neg\mathcal{I}((H_1) \Rightarrow (H_2))(\mathrm{env}_1^{\mathcal{M}})$. By the definition of implication, $\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{M}})$ and $\neg\mathcal{I}(H_2)(\mathrm{env}_1^{\mathcal{M}})$. By substitution of equals, $\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{P}})$. By the semantics of the **assume** statement, $\mathrm{env}_2^{\mathcal{P}} = \mathrm{env}_1^{\mathcal{P}}$. By substitution of equals, $\neg\mathcal{I}(H_2)(\mathrm{env}_2^{\mathcal{P}})$. By the semantics of the **confirm** statement, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$. □

Lemma 4.60 *The rule for consecutive* **confirm** *statements preserves invalidity in the program direction.*

Proof. We assume that $\mathcal{M}$ is invalid and show that $\mathcal{P}$ is invalid. Let $\mathrm{env}_I^{\mathcal{M}}$ be a witness to $\mathcal{M}$'s invalidity, and let $\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\mathrm{env}_I^{\mathcal{M}})$; i.e., $\mathrm{AE}(\mathrm{env}_I^{\mathcal{M}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{M}}) = \mathrm{CF}$. We will show the existence of an environment, $\mathrm{env}_I^{\mathcal{P}}$, such that $\mathrm{AE}(\mathrm{env}_I^{\mathcal{P}}) = \mathrm{NL}$ and $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$, where $\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\mathrm{env}_I^{\mathcal{P}})$. The proof is organized by cases. In each case we take $\mathrm{env}_I^{\mathcal{P}} = \mathrm{env}_I^{\mathcal{M}}$. First we make the following five definitions.

$$\mathrm{env}_1^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\mathit{prec\_top\_lev\_code})(\mathrm{env}_I^{\mathcal{P}}) \tag{4.154}$$

$$\mathrm{env}_2^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{confirm } H_1)(\mathrm{env}_1^{\mathcal{P}}) \tag{4.155}$$

$$\mathrm{env}_b^{\mathcal{P}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{confirm } H_2)(\mathrm{env}_2^{\mathcal{P}}) \tag{4.156}$$

$$\mathrm{env}_1^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(prec\_top\_lev\_code)(\mathrm{env}_I^{\mathcal{M}}) \quad (4.157)$$

$$\mathrm{env}_b^{\mathcal{M}} \stackrel{\mathrm{def}}{=} \mathcal{I}(\textbf{confirm}\ (H_1) \wedge (H_2))(\mathrm{env}_1^{\mathcal{M}}) \quad (4.158)$$

Consequently,

$$\mathrm{env}_1^{\mathcal{M}} = \mathrm{env}_1^{\mathcal{P}} \quad (4.159)$$

$$\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(fol\_top\_lev\_code)(\mathrm{env}_b^{\mathcal{P}}) \quad (4.160)$$

$$\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(fol\_top\_lev\_code)(\mathrm{env}_b^{\mathcal{M}}). \quad (4.161)$$

Case $AE(env_1^{\mathcal{M}}) = CF$. Therefore, $\mathrm{AE}(\mathrm{env}_1^{\mathcal{P}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $AE(env_1^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_1^{\mathcal{M}}) = \mathrm{NL}$. By substitution of equals, $\mathrm{AE}(\mathrm{env}_1^{\mathcal{P}}) = \mathrm{NL}$.

Case $AE(env_b^{\mathcal{M}}) = CF$. By the semantics of the **confirm** statement, $\neg\mathcal{I}((H_1) \wedge (H_2))(\mathrm{env}_1^{\mathcal{M}})$.

Case $\neg\mathcal{I}(H_1)(env_1^{\mathcal{M}})$. In this case, $\mathrm{AE}(\mathrm{env}_2^{\mathcal{P}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $\mathcal{I}(H_1)(env_1^{\mathcal{M}})$. By substitution of equals, $\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{P}})$. By the semantics of the **confirm** statement, $\mathrm{env}_2^{\mathcal{P}} = \mathrm{env}_1^{\mathcal{P}}$. Because $\neg\mathcal{I}((H_1) \wedge (H_2))(\mathrm{env}_1^{\mathcal{M}})$, $\neg\mathcal{I}(H_2)(\mathrm{env}_1^{\mathcal{M}})$. By substitution of equals, $\neg\mathcal{I}(H_2)(\mathrm{env}_2^{\mathcal{P}})$. Therefore, $\mathrm{AE}(\mathrm{env}_b^{\mathcal{P}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $AE(env_b^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_b^{\mathcal{M}}) = \mathrm{NL}$. Therefore, $\mathcal{I}((H_1) \wedge (H_2))(\mathrm{env}_1^{\mathcal{M}})$. Hence, $\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{M}})$ and $\mathcal{I}(H_2)(\mathrm{env}_1^{\mathcal{M}})$. Therefore, $\mathcal{I}(H_1)(\mathrm{env}_1^{\mathcal{P}})$. Hence, $\mathrm{env}_2^{\mathcal{P}} = \mathrm{env}_1^{\mathcal{P}}$. By substitution of equals, $\mathcal{I}(H_2)(\mathrm{env}_2^{\mathcal{P}})$.

Therefore, $\text{env}_b^{\mathcal{P}} = \text{env}_2^{\mathcal{P}} = \text{env}_1^{\mathcal{P}} = \text{env}_b^{\mathcal{M}}$. By substitution of equals, $\text{env}_F^{\mathcal{P}} = \text{env}_F^{\mathcal{M}}$. Hence, $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$. □

Lemma 4.61 *The rule of inference bridging predicate logic and the indexed method preserves invalidity in the program direction.*

Proof. We suppose that H is invalid (i.e., false) in some model of context C's logic. That is to say, there is an assignment of values to the variables in H that renders H false in that model. Let env be a neutral environment whose component states make the same assignment of values to the variables in H. Then $\text{AE}(\textbf{confirm } H) = \text{CF}$. Therefore, **confirm** H is an invalid program. □

4.5.4 The loop while Rule

We provide here a proof that an application of the **loop while** rule that involves a loop whose tightest loop invariant is expressed in the **maintaining** clause preserves invalidity in the program direction.

Proof of Lemma 4.7. We assume that $\mathcal{M}$ is invalid and show that $\mathcal{P}$ is invalid. Let $\text{env}_I^{\mathcal{M}}$ be a witness to $\mathcal{M}$'s invalidity, and let $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\text{env}_I^{\mathcal{M}})$; i.e., $\text{AE}(\text{env}_I^{\mathcal{M}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. We will show the existence of an environment, $\text{env}_I^{\mathcal{P}}$, such that $\text{AE}(\text{env}_I^{\mathcal{P}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$, where $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\text{env}_I^{\mathcal{P}})$. The proof is organized by cases. First we make the following two definitions.

$$\text{env}_i^{\mathcal{M}} \stackrel{\text{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code } \textbf{alter all stow}(i) ACseq_0)(\text{env}_I^{\mathcal{M}}) \quad (4.162)$$

$$\text{env}_i^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code } \textbf{alter all stow}(i) ACseq_0)(\text{env}_I^{\mathcal{P}}) \quad (4.163)$$

We will be defining other environments as well. Figure 81 shows the positions of the environments defined for $\mathcal{P}$, and figure 82 shows the positions of the environments defined for $\mathcal{M}$.

Case $AE(env_i^{\mathcal{M}}) = CF$. In this case, we take $\text{env}_I^{\mathcal{P}} = \text{env}_I^{\mathcal{M}}$. Then $\text{AE}(\text{env}_i^{\mathcal{P}}) = \text{CF}$. By Lemma 4.15, $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$.

Case $AE(env_i^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\text{AE}(\text{env}_i^{\mathcal{M}}) = \text{NL}$. We may, for this case, write $\text{env}_i^{\mathcal{M}} = [\text{NL}, \text{cs}_i, \text{ns}, \langle \text{cs}_j, \text{cs}_l \rangle \circ \text{se}, \text{os}, \text{d}]$, where $\text{ns}(i) = \text{cs}_i$. With the help of Lemmas 4.20 and 4.27, we may also write $\text{env}_I^{\mathcal{M}} = [\text{NL}, \text{cs}_I, \text{ns}_I, \text{se}_I \circ \langle \text{cs}_j, \text{cs}_l \rangle \circ \text{se}, \text{os}, \text{d}]$. As shown in Figure 82, we let $\text{env}_{\text{ew}}^{\mathcal{M}} \stackrel{\text{def}}{=} \mathcal{I}(\textbf{confirm}\ (\text{Br\_Cd}) \Rightarrow (\text{Inv}[\text{x} \rightsquigarrow \text{x}_i, \#\text{x} \rightsquigarrow \text{x}_i])\ \textbf{alter all stow}(j)\ \ldots\ \textbf{whenever}\ \text{Br\_Cd}\ \textbf{do assume}\ (\neg(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_l])) \wedge (\text{Inv}[\text{x} \rightsquigarrow \text{x}_l, \#\text{x} \rightsquigarrow \text{x}_i])\ cd\_suffix\ \textbf{end whenever})(\text{env}_i^{\mathcal{M}})$. Similarly, as shown in Figure 81, we let $\text{env}_{\text{ew}}^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\textbf{whenever}\ \text{Br\_Cd}\ \textbf{do}\ \ldots\ \textbf{end whenever})(\text{env}_i^{\mathcal{P}})$.

Case $\neg\mathcal{I}(Br\_Cd)(env_i^{\mathcal{M}})$. This case is similar to the corresponding case in the proof of Lemma 4.51, and we do not repeat it here.

Case $\mathcal{I}(Br\_Cd)(env_i^{\mathcal{M}})$. (We are still inside case $\text{AE}(\text{env}_i^{\mathcal{M}}) \neq \text{CF}$, i.e., $\text{AE}(\text{env}_i^{\mathcal{M}}) = \text{NL}$.) We take $\text{env}_I^{\mathcal{P}} = \text{env}_I^{\mathcal{M}}$. Therefore, $\text{env}_i^{\mathcal{P}} = \text{env}_i^{\mathcal{M}}$.

Case $AE(env_j^{\mathcal{M}}) = CF$. In this case, we must have $\neg\mathcal{I}(\text{Inv}[\text{x} \rightsquigarrow \text{x}_i, \#\text{x} \rightsquigarrow \text{x}_i]))(\text{env}_i^{\mathcal{M}})$. By the semantics of **whenever**, the loop is interpreted in environment $\text{env}_i^{\mathcal{P}}$. Because $\neg\mathcal{I}(\text{Inv}\lfloor \text{x}, \#\text{x} \rfloor)(\text{Rem}(\text{env}_i^{\mathcal{P}}))$, the interpretation of the loop in environment $\text{env}_i^{\mathcal{P}}$ is a categorically false environment. By Lemma 4.15, $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$.

Case $AE(env_j^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\text{AE}(\text{env}_j^{\mathcal{M}}) = \text{NL}$.

Case $AE(env_l^{\mathcal{M}}) = CF$. The following statements are true in this case. $\mathcal{I}(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_i])(\text{env}_j^{\mathcal{M}})$. $\mathcal{I}((\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_j]) \wedge (\text{Inv}[\text{x} \rightsquigarrow \text{x}_j, \#\text{x} \rightsquigarrow \text{x}_i]))(\text{env}_j^{\mathcal{M}})$. $\text{AE}(\mathcal{I}(cd\_kern\ \textbf{stow}(k)\ ACseq\ \textbf{confirm}\ \text{Inv}[\text{x} \rightsquigarrow \text{x}_k, \#\text{x} \rightsquigarrow \text{x}_i])(\text{env}_j^{\mathcal{M}})) = \text{CF}$. Because Inv is the tightest loop invariant for the loop, $\mathcal{I}(\text{Inv}[\text{x} \rightsquigarrow \text{x}_j, \#\text{x} \rightsquigarrow \text{x}_i])(\text{env}_j^{\mathcal{M}})$, and the loop and its body are unaffected by old state (Lemma 4.30), cs_j must be the current state resulting from zero or more iterations of the loop when it is interpreted in environment $\text{env}_i^{\mathcal{P}}$ [4, p. 87]. Hence, execution of the loop's body beginning in environment $\text{env}_i^{\mathcal{P}}$ produces either a categorically false environment or a neutral environment in which the loop invariant is interpreted to be false. Therefore, $\text{AE}(\text{env}_l^{\mathcal{P}}) = \text{CF}$. By Lemma 4.15, $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$.

Case $AE(env_l^{\mathcal{M}}) \neq CF$. (We are still inside case $\text{AE}(\text{env}_j^{\mathcal{M}}) \neq \text{CF}$, i.e., $\text{AE}(\text{env}_j^{\mathcal{M}}) = \text{NL}$.) In this case, by Lemma 4.19, we have $\text{AE}(\text{env}_l^{\mathcal{M}}) = \text{NL}$. By Lemma 4.18, $\mathcal{I}((\neg(\text{MExp}(b\_p\_e)[\text{y} \rightsquigarrow \text{y}_l])) \wedge (\text{Inv}[\text{x} \rightsquigarrow \text{x}_l, \#\text{x} \rightsquigarrow \text{x}_i]))(\text{env}_l^{\mathcal{M}})$. Because Inv is the tightest loop invariant for the loop and the loop and its body are unaffected by old state (Lemma 4.30), cs_l is the current state of the first environment resulting from zero or more iterations of the loop when it is interpreted in environment $\text{env}_i^{\mathcal{P}}$ in which the Boolean program expression b_p_e is interpreted to be false [4, p. 87]. In other words, cs_l must be the current state of environment $\text{env}_l^{\mathcal{P}}$. Let $A \stackrel{\text{def}}{=} \{h | j \leq h \leq k\}$. Therefore, $\text{Equal\_except\_at}(A, \text{env}_l^{\mathcal{P}}, \text{env}_l^{\mathcal{M}})$. Note that $\text{env}_F^{\mathcal{M}} = \mathcal{I}(cd\_suffix\ fol\_top\_lev\_code)(\text{env}_l^{\mathcal{M}})$ and $\text{env}_F^{\mathcal{P}} = \mathcal{I}(cd\_suffix\ fol\_top\_lev\_code)(\text{env}_l^{\mathcal{P}})$. Two applications of Lemma 4.37 give us $\text{Equal\_except\_at}(A, \text{env}_F^{\mathcal{P}}, \text{env}_F^{\mathcal{M}})$. Therefore, $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. □

4.5.5 The Procedure Call Rule

We provide here a proof that an application of the procedure call rule that involves a call either to (1) a functionally specified external procedure or (2) to an internal procedure, whose postcondition is the post relation corresponding to the procedure's precondition and its body, preserves invalidity in the program direction.

Proof of Lemma 4.6. We assume that $\mathcal{M}$ is invalid and show that $\mathcal{P}$ is invalid. Let $\text{env}_I^{\mathcal{M}}$ be a witness to $\mathcal{M}$'s invalidity, and let $\text{env}_F^{\mathcal{M}} = \mathcal{I}(\mathcal{M})(\text{env}_I^{\mathcal{M}})$; i.e., $\text{AE}(\text{env}_I^{\mathcal{M}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{M}}) = \text{CF}$. We will show the existence of an environment, $\text{env}_I^{\mathcal{P}}$, such that $\text{AE}(\text{env}_I^{\mathcal{P}}) = \text{NL}$ and $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$, where $\text{env}_F^{\mathcal{P}} = \mathcal{I}(\mathcal{P})(\text{env}_I^{\mathcal{P}})$. The proof is organized by cases. First we make the following two definitions.

$$\text{env}_i^{\mathcal{M}} \stackrel{\text{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\,ACseq_0)(\text{env}_I^{\mathcal{M}}) \qquad (4.164)$$

$$\text{env}_i^{\mathcal{P}} \stackrel{\text{def}}{=} \mathcal{I}(\textit{prec\_top\_lev\_code}\ \textbf{alter all stow}(i)\,ACseq_0)(\text{env}_I^{\mathcal{P}}) \qquad (4.165)$$

We will be defining other environments as well. Figure 83 shows the positions of the environments defined for $\mathcal{P}$ and for $\mathcal{M}$.

Case $AE(env_i^{\mathcal{M}}) = CF$. In this case, we take $\text{env}_I^{\mathcal{P}} = \text{env}_I^{\mathcal{M}}$. Then $\text{AE}(\text{env}_i^{\mathcal{P}}) = \text{CF}$. By Lemma 4.15, $\text{AE}(\text{env}_F^{\mathcal{P}}) = \text{CF}$.

Case $AE(env_i^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\text{AE}(\text{env}_i^{\mathcal{M}}) = \text{NL}$. We may, for this case, write $\text{env}_i^{\mathcal{M}} = [\text{NL}, \text{cs}_i, \text{ns}, \langle \text{cs}_j \rangle \circ \text{se}, \text{os}, \text{d}]$, where $\text{ns}(i) = \text{cs}_i$. With the help of Lemmas 4.20 and 4.27, we may also write $\text{env}_I^{\mathcal{M}} = [\text{NL}, \text{cs}_I, \text{ns}_I, \text{se}_I \circ \langle \text{cs}_j \rangle \circ \text{se}, \text{os}, \text{d}]$.

Case $\neg\mathcal{I}(Br\_Cd)(env_i^{\mathcal{M}})$. This case is similar to the corresponding case in the proof

of Lemma 4.51, and we do not repeat it here.

Case $\mathcal{I}(Br\_Cd)(env_i^{\mathcal{M}})$. (We are still inside case $\mathrm{AE}(\mathrm{env}_i^{\mathcal{M}}) \neq \mathrm{CF}$, i.e., $\mathrm{AE}(\mathrm{env}_i^{\mathcal{M}}) = \mathrm{NL}$.) By definition of the applicability of the procedure call rule, the declaration-meaning d of environment $\mathrm{env}_I^{\mathcal{M}}$ contains d(P_nm). Let $\mathrm{d(P\_nm)} \stackrel{\mathrm{def}}{=} [\mathrm{dp}, \mathrm{ep}, \mathrm{pf}, \mathrm{sf}]$. Because $\mathrm{AE}(\mathrm{env}_I^{\mathcal{M}}) = \mathrm{NL}$, we have that dp is the same as pre, ep is the same as post, and d(P_nm) is conformal.

Case $AE(env_j^{\mathcal{M}}) = CF$. We take $\mathrm{env}_I^{\mathcal{P}} = \mathrm{env}_I^{\mathcal{M}}$. Therefore, $\mathrm{env}_i^{\mathcal{P}} = \mathrm{env}_i^{\mathcal{M}}$. In this case, $\neg\mathcal{I}(\mathrm{pre}[\mathrm{x} \rightsquigarrow \mathrm{ac}_i, \mathrm{y} \rightsquigarrow \mathrm{ad}_i, \mathrm{z} \rightsquigarrow \mathrm{z}_i])(\mathrm{env}_i^{\mathcal{M}})$. Therefore, $\neg\mathrm{dp}(\mathrm{cs}_i(\mathrm{ac}), \mathrm{cs}_i(\mathrm{ad}), \mathrm{cs}_i(\mathrm{z}))$. Hence, $\mathrm{AE}(\mathcal{I}(\mathrm{P\_nm}(\mathrm{ac}, \mathrm{ad}))(\mathrm{env}_i^{\mathcal{P}})) = \mathrm{CF}$. Therefore, $\mathrm{AE}(\mathrm{env}_j^{\mathcal{P}}) = \mathrm{CF}$. By Lemma 4.15, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

Case $AE(env_j^{\mathcal{M}}) \neq CF$. In this case, by Lemma 4.19, we have $\mathrm{AE}(\mathrm{env}_j^{\mathcal{M}}) = \mathrm{NL}$. We take $\mathrm{env}_I^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \mathrm{se}, \mathrm{os}, \mathrm{d}]$. Then $\mathrm{env}_i^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_i, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}]$. Because $\mathrm{AE}(\mathrm{env}_j^{\mathcal{M}}) = \mathrm{NL}$, $\mathcal{I}(\mathrm{pre}[\mathrm{x} \rightsquigarrow \mathrm{ac}_i, \mathrm{y} \rightsquigarrow \mathrm{ad}_i, \mathrm{z} \rightsquigarrow \mathrm{z}_i])(\mathrm{env}_i^{\mathcal{M}})$. Hence, $\mathrm{dp}(\mathrm{cs}_i(\mathrm{ac}), \mathrm{cs}_i(\mathrm{ad}), \mathrm{cs}_i(\mathrm{z}))$. By Lemma 4.16, $\mathcal{I}(\mathrm{b}_j = \mathrm{b}_i \wedge (\mathrm{post}[\#\mathrm{x} \rightsquigarrow \mathrm{ac}_i, \mathrm{x} \rightsquigarrow \mathrm{ac}_j, \#\mathrm{y} \rightsquigarrow \mathrm{ad}_i, \mathrm{y} \rightsquigarrow \mathrm{ad}_j, \#\mathrm{z} \rightsquigarrow \mathrm{z}_i, \mathrm{z} \rightsquigarrow \mathrm{z}_j]))(\mathrm{env}_j^{\mathcal{M}})$. Hence, $\mathrm{ep}(\mathrm{cs}_i(\mathrm{ac}), \mathrm{cs}_i(\mathrm{ad}), \mathrm{cs}_i(\mathrm{z}), \mathrm{cs}_j(\mathrm{ac}), \mathrm{cs}_j(\mathrm{ad}), \mathrm{cs}_j(\mathrm{z}))$. Let $[w_1, w_2, w_3] \stackrel{\mathrm{def}}{=} \mathrm{pf}(\mathrm{cs}_i(\mathrm{ac}), \mathrm{cs}_i(\mathrm{ad}), \mathrm{cs}_i(\mathrm{z}))$. Because P_nm is either (1) a functionally specified external procedure or (2) to an internal procedure, whose postcondition is the post relation corresponding to the procedure's precondition and its body, $[w_1, w_2, w_3] = [\mathrm{cs}_j(\mathrm{ac}), \mathrm{cs}_j(\mathrm{ad}), \mathrm{cs}_j(\mathrm{z})]$. Therefore, $\mathrm{CSE}(\mathrm{env}_j^{\mathcal{P}}) = \mathrm{cs}_j$, and, furthermore, $\mathrm{env}_j^{\mathcal{P}} = \mathrm{env}_j^{\mathcal{M}}$. Because $\mathrm{env}_F^{\mathcal{P}} = \mathcal{I}(cd\_suffix\ fol\_top\_lev\_code)(\mathrm{env}_j^{\mathcal{P}})$ and $\mathrm{env}_F^{\mathcal{M}} = \mathcal{I}(cd\_suffix\ fol\_top\_lev\_code)(\mathrm{env}_j^{\mathcal{M}})$, $\mathrm{env}_F^{\mathcal{P}} = \mathrm{env}_F^{\mathcal{M}}$. Hence, $\mathrm{AE}(\mathrm{env}_F^{\mathcal{P}}) = \mathrm{CF}$.

□

4.6 Relaxing a Simplifying Assumption

We made a simplifying assumption in formulating the proof rules of Chapter III. The presentation of the rules is shorter than it otherwise would have been because we assumed that only the first statement within a **whenever** statement would be replaced by application of a proof rule in the math direction. However, the more flexible method informally described in Chapter I would be better represented by rules that permitted any statement within a **whenever** statement to be replaced. We argue here that, if we relax this simplifying assumption by adding rules that permit any statement within a **whenever** statement to be replaced, all the lemmas and theorems of this chapter would still hold true. That is to say, we made our simplifying assumption without loss of generality.

Figure 84 shows an additional rule for procedure call that permits any call within a **whenever** statement to be replaced—as long as that call is not the first statement within a **whenever** statement. One would use the existing procedure call rule from Chapter III to replace a call that is the first statement within a **whenever** statement. We can add three other rules to our proof system to handle operational statements that are not first within a **whenever** statement—one for the **loop while** statement and two for the two forms of the selection statement.

The proofs that invalidity is preserved in the two directions for these four new rules will contain the same arguments as the corresponding proofs for the existing

$$\mathcal{P} \stackrel{\text{def}}{=} C\backslash \quad \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textit{stow\_sec} \\ \textit{ACseq}_0 \\ \mid \textbf{whenever}\ \text{Br\_Cd}\ \textbf{do} \\ \quad \textit{cd\_kern} \\ \quad \textbf{stow}(i) \\ \quad \textit{ACseq}_1 \\ \quad \text{P\_nm}(\text{ac}, \text{ad}) \\ \quad \textbf{stow}(j) \\ \quad \textit{cd\_suffix} \\ \mid \textbf{end whenever} \\ \textit{fol\_top\_lev\_code} \end{array} \tag{4.166}$$

$$\mathcal{M} \stackrel{\text{def}}{=} C\backslash \quad \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textit{stow\_sec} \\ \textit{ACseq}_0 \\ \mid \textbf{whenever}\ \text{Br\_Cd}\ \textbf{do} \\ \quad \textit{cd\_kern} \\ \quad \textbf{stow}(i) \\ \quad \textit{ACseq}_1 \\ \mid \textbf{end whenever} \\ \mid \textbf{confirm}\ (\text{Br\_Cd}) \Rightarrow (\text{pre}[\text{x} \rightsquigarrow \text{ac}_i, \text{y} \rightsquigarrow \text{ad}_i, \text{z} \rightsquigarrow \text{z}_i]) \\ \mid \textbf{alter all} \\ \mid \textbf{stow}(j) \\ \mid \textbf{assume}\ (\text{Br\_Cd}) \Rightarrow (\text{b}_j = \text{b}_i \wedge (\text{post}[\#\text{x} \rightsquigarrow \text{ac}_i, \text{x} \rightsquigarrow \text{ac}_j, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \#\text{y} \rightsquigarrow \text{ad}_i, \text{y} \rightsquigarrow \text{ad}_j, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \#\text{z} \rightsquigarrow \text{z}_i, \text{z} \rightsquigarrow \text{z}_j])) \\ \mid \textbf{whenever}\ \text{Br\_Cd}\ \textbf{do} \\ \quad \textit{cd\_suffix} \\ \mid \textbf{end whenever} \\ \textit{fol\_top\_lev\_code} \end{array} \tag{4.167}$$

Additional Syntactic Restriction: The same as in Figure 46.

Figure 84: The Rule for a Procedure Call That Is Not the First Statement Within a **whenever** Statement

"first-statement" rules. The new proofs will also use the following two additional observations. In the case that the branch condition Br_Cd evaluates as false, interpretation of the first **whenever** statement in $\mathcal{M}$ is the same as interpretation of the empty sequence of statements. In the case that the branch condition Br_Cd evaluates as true, interpretation of the first **whenever** statement in $\mathcal{M}$ is the same as interpretation of *cd_kern* **stow**(i) $ACseq_1$.

We can still prove all the other lemmas and theorems for this extended proof system. Three of the lemmas deserve special mention here. In the example of the additional proof rule for procedure call (Figure 84), all variables subscripted with i introduced by this rule appear after the **whenever** statement containing the **stow**(i) statement; importantly, they also appear only in **confirm** and **assume** statements in the consequents of assertions whose antecedents are the branch condition (Br_Cd) of the **whenever** statement. This fact means that we can prove Lemma 4.35 and, consequently, the negative-branch-condition independence lemma (4.36). Because the additional rules introduce an extra **whenever** statement, the multipliers for function Meas (Table 2) used in the proof of Lemma 4.5 in Section 4.5.2 need to be changed. The multipliers for procedure call, **loop**, and **if** need to be increased by one to six, twelve, and thirteen, respectively. Because we can still prove all lemmas and theorems for this extended proof system, our simplifying assumption is without loss of generality. The indexed method can process operational statements in any order.

4.7 Summary

In this chapter we have supplied the detailed proofs for Theorems 1 and 2. These two theorems validate the second and third points of our three-part thesis. They establish that the indexed method for reasoning about program behavior is both sound and relatively complete.

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.

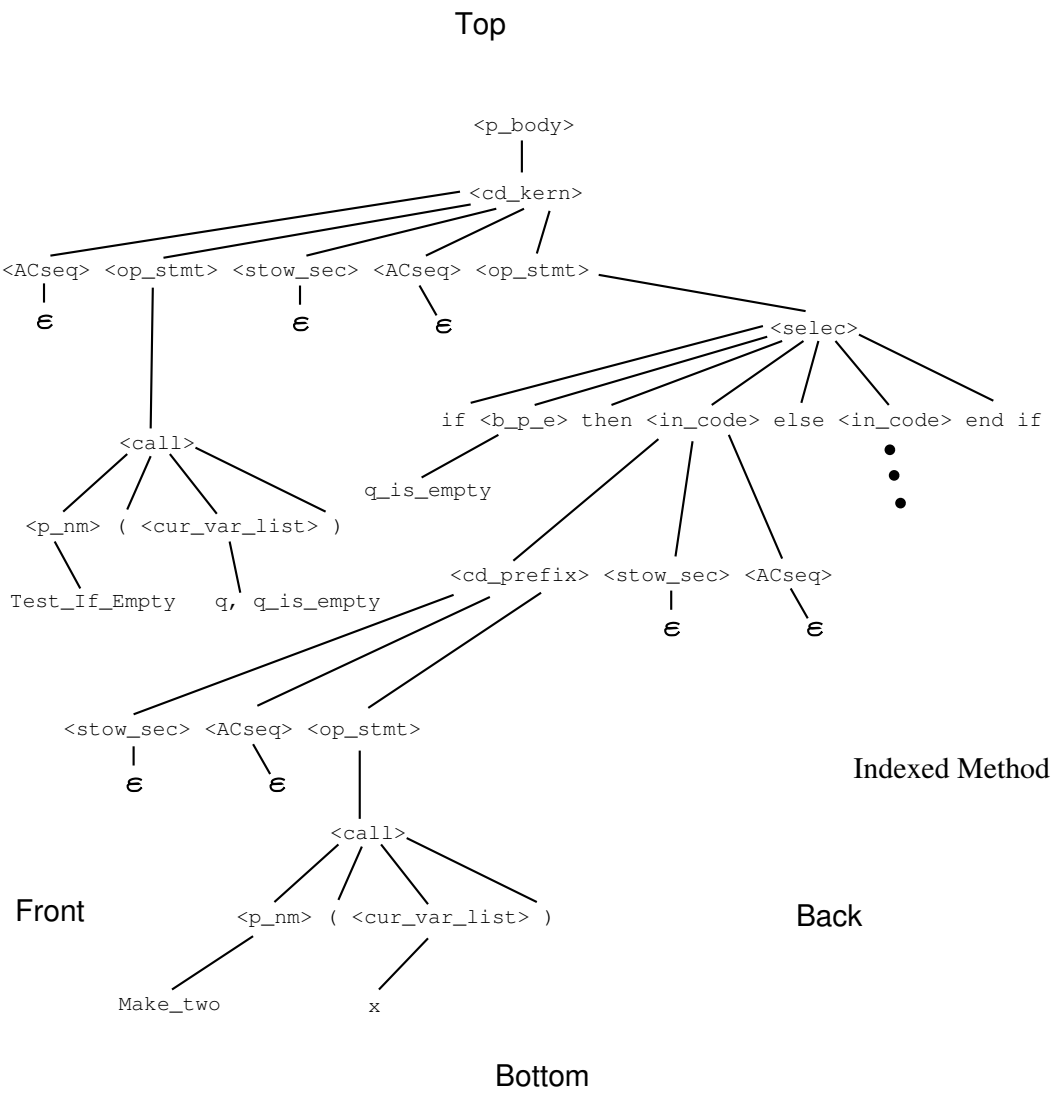

Figure 85: Action Directions Differ Among the Three Methods

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 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 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.

⟨iter⟩ ::= ⟨while⟩ | ⟨loop⟩ (5.1)

⟨loop⟩ ::= **loop** (5.2)
maintaining ⟨old_assert⟩
[⟨in_code⟩]
exit when ⟨b_p_e⟩
{⟨in_code⟩
exit when ⟨b_p_e⟩}
[⟨in_code⟩]
end loop

⟨while⟩ ::= **loop** (5.3)
maintaining ⟨old_assert⟩
while ⟨b_p_e⟩ **do**
⟨in_code⟩
end loop

Figure 86: Redefinition of ⟨iter⟩

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⟩

$$
\begin{array}{lll}
\mathcal{P} & \stackrel{\text{def}}{=} & C\backslash \\
 & & \quad \textit{prec\_top\_lev\_code} \\
 & & \quad \textbf{alter all} \\
 & & \quad \textbf{stow}(i) \\
 & & \quad \textit{ACseq}_0 \\
 & & \quad \textbf{whenever } \text{Br\_Cd } \textbf{do} \\
 & & \quad\quad \textbf{loop} \\
 & & \quad\quad\quad \textbf{maintaining } \text{Inv}\lfloor \text{x}, \#\text{x} \rfloor \\
 & & \quad\quad\quad\quad \textbf{stow}(j_1) \\
 & & \quad\quad\quad\quad \textit{cd\_kern}_1 \quad \textbf{stow}(k_1) \quad \textit{ACseq}_1 \\
 & & \quad\quad\quad \textbf{exit when } \textit{b\_p\_e}_1 \\
 & & \quad\quad\quad\quad \textbf{stow}(j_2) \\
 & & \quad\quad\quad\quad \textit{cd\_kern}_2 \quad \textbf{stow}(k_2) \quad \textit{ACseq}_2 \\
 & & \quad\quad\quad \textbf{exit when } \textit{b\_p\_e}_2 \\
 & & \quad\quad\quad\quad \textbf{stow}(j_3) \\
 & & \quad\quad\quad\quad \textit{cd\_kern}_3 \quad \textbf{stow}(k_3) \quad \textit{ACseq}_3 \\
 & & \quad\quad \textbf{end loop} \\
 & & \quad\quad \textbf{stow}(l) \\
 & & \quad\quad \textit{cd\_suffix} \\
 & & \quad \textbf{end whenever} \\
 & & \quad \textit{fol\_top\_lev\_code}
\end{array}
\tag{5.4}
$$

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

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 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**"

$$
\begin{array}{lll}
\mathcal{M} & \stackrel{\text{def}}{=} & C\backslash \quad \textit{prec\_top\_lev\_code} \\
 & & \textbf{alter all} \\
 & & \textbf{stow}(i) \\
 & & \textit{ACseq}_0 \\
 & & \quad \textbf{confirm } (\text{Br\_Cd}) \Rightarrow (\text{Inv}[\text{x} \rightsquigarrow \text{x}_i, \#\text{x} \rightsquigarrow \text{x}_i]) \\
 & & \quad \textbf{alter all} \quad \textbf{stow}(j_1) \\
 & & \quad \textbf{whenever } \text{Br\_Cd} \textbf{ do} \\
 & & \quad\quad \textbf{assume } \text{Inv}[\text{x} \rightsquigarrow \text{x}_{j_1}, \#\text{x} \rightsquigarrow \text{x}_i] \\
 & & \quad\quad \textit{cd\_kern}_1 \quad \textbf{stow}(k_1) \quad \textit{ACseq}_1 \\
 & & \quad \textbf{end whenever} \\
 & & \quad \textbf{alter all} \quad \textbf{stow}(j_2) \\
 & & \quad \textbf{whenever } (\text{Br\_Cd}) \wedge (\neg\text{MExp}(b\_p\_e_1[\text{y} \rightsquigarrow \text{y}_{k_1}])) \textbf{ do} \\
 & & \quad\quad \textbf{assume } \text{x}_{j_2} = \text{x}_{k_1} \\
 & & \quad\quad \textit{cd\_kern}_2 \quad \textbf{stow}(k_2) \quad \textit{ACseq}_2 \\
 & & \quad \textbf{end whenever} \\
 & & \quad \textbf{alter all} \quad \textbf{stow}(j_3) \\
 & & \quad \textbf{whenever } (\text{Br\_Cd}) \wedge (\neg\text{MExp}(b\_p\_e_1[\text{y} \rightsquigarrow \text{y}_{k_1}])) \\
 & & \quad\quad \wedge (\neg\text{MExp}(b\_p\_e_2[\text{y} \rightsquigarrow \text{y}_{k_2}])) \textbf{ do} \\
 & & \quad\quad \textbf{assume } \text{x}_{j_3} = \text{x}_{k_2} \\
 & & \quad\quad \textit{cd\_kern}_3 \quad \textbf{stow}(k_3) \quad \textit{ACseq}_3 \\
 & & \quad\quad \textbf{confirm } \text{Inv}[\text{x} \rightsquigarrow \text{x}_{k_3}, \#\text{x} \rightsquigarrow \text{x}_i] \\
 & & \quad \textbf{end whenever} \\
 & & \quad \textbf{alter all} \quad \textbf{stow}(l) \\
 & & \quad \textbf{whenever } \text{Br\_Cd} \textbf{ do} \\
 & & \quad\quad \textbf{assume} \\
 & & \quad\quad\quad ((\text{MExp}(b\_p\_e_1[\text{y} \rightsquigarrow \text{y}_{k_1}])) \vee (\text{MExp}(b\_p\_e_2[\text{y} \rightsquigarrow \text{y}_{k_2}]))) \\
 & & \quad\quad\quad \wedge ((\text{MExp}(b\_p\_e_1[\text{y} \rightsquigarrow \text{y}_{k_1}])) \Rightarrow (\text{x}_l = \text{x}_{k_1})) \\
 & & \quad\quad\quad \wedge (((\neg\text{MExp}(b\_p\_e_1[\text{y} \rightsquigarrow \text{y}_{k_1}])) \\
 & & \quad\quad\quad\quad \wedge (\text{MExp}(b\_p\_e_2[\text{y} \rightsquigarrow \text{y}_{k_2}]))) \Rightarrow (\text{x}_l = \text{x}_{k_2})) \\
 & & \quad\quad \textit{cd\_suffix} \\
 & & \quad \textbf{end whenever} \\
 & & \textit{fol\_top\_lev\_code}
\end{array}
\tag{5.5}
$$

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

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.

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.

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. McGraw-Hill Book Company, 1985.

[2] Alan Baddeley. Working memory. *Science*, 255:556–9, 31 January 1992.

[3] Manfred Broy. Experiences with software specification and verification using LP, the Larch Proof Assistant. Technical Report 93, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, November 1992.

[4] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, February 1978.

[5] Laurence Peter Deutsch. *An Interactive Program Verifier*. PhD thesis, Department of Computer Science, University of California, Berkeley, May 1973.

[6] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[7] Stephen H. Edwards and William F. Ogden. Personal communication, October 1995.

[8] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[9] George W. Ernst, Raymond J. Hookway, James A. Menegay, and William F. Ogden. Semantics of programming languages for modular verification. Technical Report CES-85-04, Computer Engineering and Science Department, Case Western Reserve University, 1985.

[10] George W. Ernst, Raymond J. Hookway, James A. Menegay, and William F. Ogden. Modular verification of Ada generics. *Computer Languages*, 16(3/4):259–280, 1991.

[11] George W. Ernst, Raymond J. Hookway, and William F. Ogden. Modular verification of data abstractions with shared realizations. *IEEE Transactions on Software Engineering*, 20(4):288–307, April 1994.

[12] George W. Ernst, Jainendra K. Navlakha, and William F. Ogden. Verification of programs with Procedure-Type parameters. *Acta Informatica*, 18:149–169, 1982.

[13] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[14] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1991.

[15] Kurt Gödel. On formally undecidable propositions of *Principia Mathematica* and related systems I. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic*, pages 596–616. Harvard University Press, Cambridge, Massachusetts, 1967.

[16] Douglas E. Harms. *The Influence of Software Reuse on Programming Language Design*. PhD thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1990.

[17] Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[19] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[20] C. A. R. Hoare. Proof of correctness of data representation. In David Gries, editor, *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, chapter 19, pages 269–281. Springer-Verlag, 1978.

[21] C. A. R. Hoare and P. E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica*, 3:135–154, 1974.

[22] Joseph E. Hollingsworth. *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada.* PhD thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1992.

[23] James Cornelius King. *A Program Verifier.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1969.

[24] Jürgen Koenemann and Scott P. Robertson. Expert problem solving strategies for program comprehension. In *Human Factors in Computing Systems: CHI-91: Reaching Through Technology: Proceedings of the Annual Conference on Human Factors in Computing Systems*, pages 125–130. Association for Computing Machinery, Addison-Wesley, May 1991.

[25] Joan Krone. *The Role of Verification in Software Reusability.* PhD thesis, The Ohio State University, 1988.

[26] Imre Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery.* Cambridge University Press, 1976. Edited by John Worrall and Elie Zahar.

[27] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993. IEEE Computer Society.

[28] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers: First Workshop*, Human/Computer Interaction, pages 80–98, Norwood, NJ, June 1986. Ablex Publishing.

[29] Donald MacKenzie. The automation of proof: A historical and sociological exploration. *IEEE Annals of the History of Computing*, 17(3):7–29, Fall 1995.

[30] Z. Manna. The correctness of programs. *Journal of Computer and Systems Sciences*, 3(2):119–127, 1969.

[31] Z. Manna. Termination of programs represented as interpreted graphs. In *Joint Computer Conference*, volume 36, pages 83–89. American Federation of Information Processing Societies, Spring 1970.

[32] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming*, volume 1, Deductive Reasoning. Addison-Wesley, Reading, Mass., 1985.

[33] William Douglas Maurer. The modification index method of generating verification conditions. In *Proceedings of the 15th Annual ACM Southeast Regional Conference*, pages 426–440. Association for Computing Machinery, April 1977.

[34] John McCarthy. Towards a mathematical science of computation. In C. M. Popplewell, editor, *Information Processing 1962: Proc. IFIP Congress 62*, pages 21–28, Amsterdam, 1963. North Holland.

[35] R. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman & Hall Ltd., 1976.

[36] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[37] P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

[38] Jainendra K. Navlakha. A syntax directed program verification system. Technical Report ESCI-77-3, Computer Engineering and Science Dept., Case Institute of Technology, Case Western Reserve University, Cleveland, Ohio 44106, February 1978.

[39] Frank G. Pagan. *Formal Specification of Programming Languages: A Panoramic Primer*. Prentice-Hall Software Series. Prentice-Hall, 1981.

[40] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, third edition, 1992.

[41] John M. Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[42] John G. Sanderson. *A Relational Theory of Computing*, volume 82 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1980.

[43] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. *Computers and Automation*, 1972.

[44] Murali Sitaraman and Bruce W. Weide. Special feature: Component-based software using RESOLVE. *Software Engineering Notes*, 19(4):21–67, October 1994.

[45] Muralidharan Sitaraman. *Mechanisms and Methods for Performance Tuning of Reusable Software Components*. PhD thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1990.

[46] R. L. Sites. *Proving That Computer Programs Terminate Cleanly.* PhD thesis, Department of Computer Science, Stanford University, May 1974.

[47] Bruce W. Weide. Course notes for CIS 680: Software reuse through data structures and algorithms. Department of Computer and Information Science, The Ohio State University, January 1989.

[48] Bruce W. Weide. Personal communication, August 1995.

[49] Bruce W. Weide, William F. Ogden, and Stuart H. Zweben. Reusable software components. In Marshall C. Yovits, editor, *Advances in Computers*, volume 33, pages 1–65. Academic Press, 1991.