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 58) 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

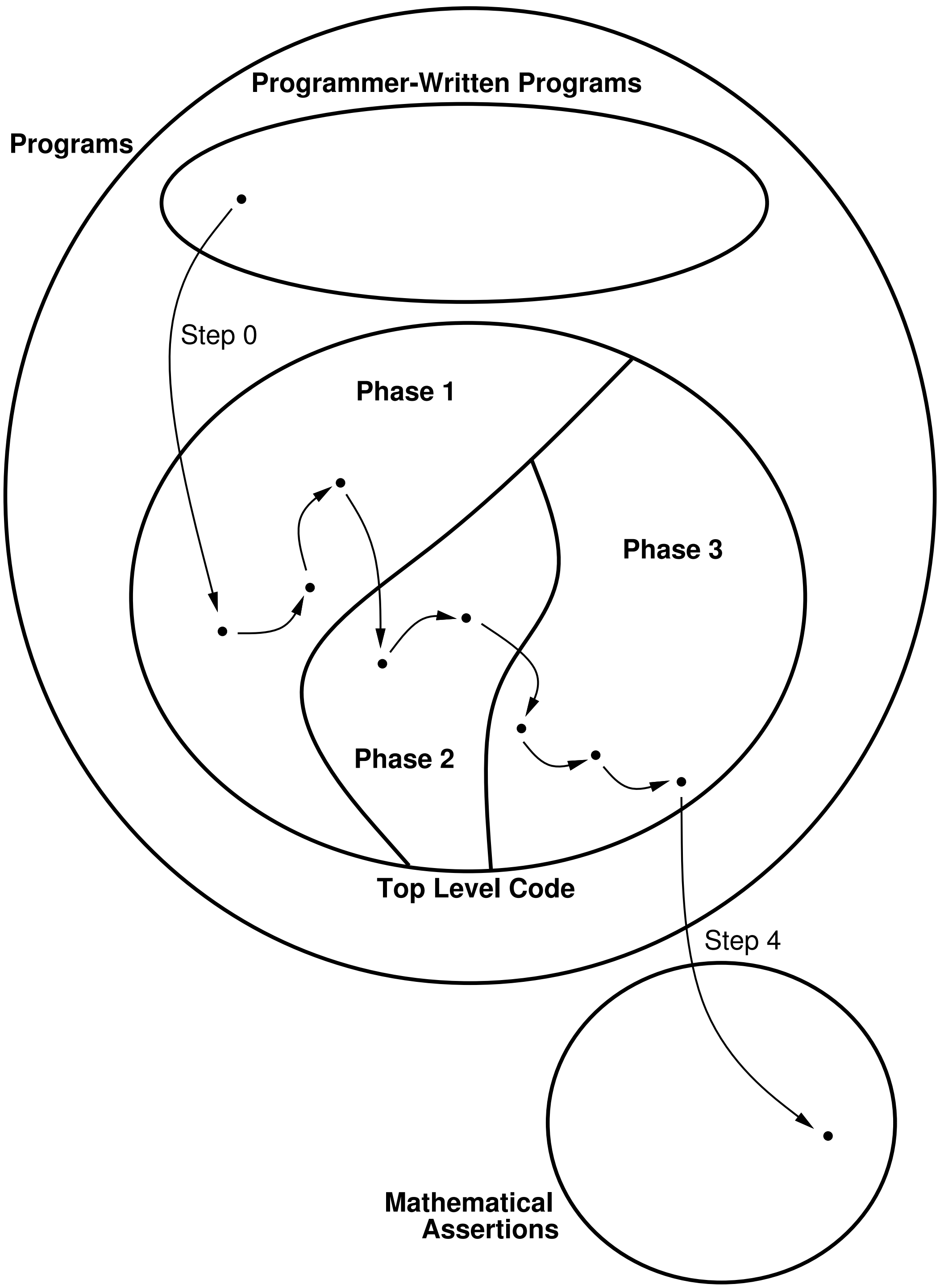

Figure 31: Transforming Programs to Mathematical Assertions in Phases

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 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, $\langle \text{op\_stmt} \rangle$, 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 $\langle \text{cur_assert} \rangle$s. The **assume** and **confirm** statements introduced by indexed method proof rules use $\langle \text{idx_assert} \rangle$s, and those introduced by Krone's rules use $\langle \text{cur_assert} \rangle$s and $\langle \text{old_assert} \rangle$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":

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

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 ($\langle \text{in\_code} \rangle$) is a pattern of nonterminal symbols, cycling repeatedly through $\langle \text{stow\_sec} \rangle$, $\langle \text{ACseq} \rangle$, and $\langle \text{op\_stmt} \rangle$, beginning with $\langle \text{stow\_sec} \rangle$ and concluding with $\langle \text{ACseq} \rangle$. Precise definitions of the proof rules can be conveniently stated in terms of the following portions of $\langle \text{in\_code} \rangle$:

1. a prefix ($\langle \text{cd\_prefix} \rangle$, concludes with $\langle \text{op\_stmt} \rangle$),

2. a suffix ($\langle \text{cd\_suffix} \rangle$, begins and ends with $\langle \text{ACseq} \rangle$), and

3. a kernal ($\langle \text{cd\_kern} \rangle$, begins with $\langle \text{ACseq} \rangle$ and, if there is an $\langle \text{op\_stmt} \rangle$, concludes with an $\langle \text{op\_stmt} \rangle$).

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

$$\begin{aligned}
\langle\text{in_code}\rangle &::= \langle\text{cd_prefix}\rangle\langle\text{stow_sec}\rangle\langle\text{ACseq}\rangle && (3.5)\\
\langle\text{cd_prefix}\rangle &::= \{\langle\text{stow_sec}\rangle\langle\text{ACseq}\rangle\langle\text{op_stmt}\rangle\} && (3.6)\\
\langle\text{cd_suffix}\rangle &::= \{\langle\text{ACseq}\rangle\langle\text{op_stmt}\rangle\langle\text{stow_sec}\rangle\}\langle\text{ACseq}\rangle && (3.7)\\
\langle\text{cd_kern}\rangle &::= \langle\text{ACseq}\rangle[\langle\text{op_stmt}\rangle\{\langle\text{stow_sec}\rangle\langle\text{ACseq}\rangle\langle\text{op_stmt}\rangle\}] && (3.8)
\end{aligned}$$

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

$$\begin{aligned}
\langle\text{gd_blk}\rangle &::= \varepsilon \mid \textbf{whenever } \langle\text{idx_assert}\rangle \textbf{ do} && (3.9)\\
&\qquad\qquad \langle\text{cd_suffix}\rangle \\
&\qquad\quad \textbf{end whenever} \\
\langle\text{top_lev_code}\rangle &::= \{\langle\text{stow_sec}\rangle\langle\text{ACseq}\rangle\langle\text{gd_blk}\rangle\} && (3.10)
\end{aligned}$$

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

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

$$\langle\text{p\_body}\rangle ::= \langle\text{cd\_kern}\rangle \qquad (3.11)$$

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

$$
\begin{array}{rcl}
\langle \text{top\_lev\_code} \rangle & ::= & \{\langle \text{stow\_sec} \rangle \langle \text{ACseq} \rangle \langle \text{gd\_blk} \rangle\} \\
\langle \text{gd\_blk} \rangle & ::= & \varepsilon \mid \textbf{whenever } \langle \text{idx\_assert} \rangle \textbf{ do} \\
 & & \qquad \langle \text{cd\_suffix} \rangle \\
 & & \quad \textbf{end whenever} \\
\langle \text{p\_body} \rangle & ::= & \langle \text{cd\_kern} \rangle \\
\langle \text{cd\_kern} \rangle & ::= & \langle \text{ACseq} \rangle [\langle \text{op\_stmt} \rangle \{\langle \text{stow\_sec} \rangle \langle \text{ACseq} \rangle \langle \text{op\_stmt} \rangle\}] \\
\langle \text{cd\_suffix} \rangle & ::= & \{\langle \text{ACseq} \rangle \langle \text{op\_stmt} \rangle \langle \text{stow\_sec} \rangle\} \langle \text{ACseq} \rangle \\
\langle \text{cd\_prefix} \rangle & ::= & \{\langle \text{stow\_sec} \rangle \langle \text{ACseq} \rangle \langle \text{op\_stmt} \rangle\} \\
\langle \text{in\_code} \rangle & ::= & \langle \text{cd\_prefix} \rangle \langle \text{stow\_sec} \rangle \langle \text{ACseq} \rangle \\
\langle \text{stow\_sec} \rangle & ::= & \varepsilon \mid \textbf{stow}(\langle \text{nat\_num} \rangle) \mid \textbf{alter all} \\
 & & \qquad \qquad \qquad \qquad \textbf{stow}(\langle \text{nat\_num} \rangle) \\
\langle \text{ACseq} \rangle & ::= & \{\textbf{assume } \langle \text{assert} \rangle \mid \textbf{confirm } \langle \text{assert} \rangle\} \\
\langle \text{assert} \rangle & ::= & \langle \text{cur\_assert} \rangle \mid \langle \text{old\_assert} \rangle \mid \langle \text{idx\_assert} \rangle \\
\langle \text{op\_stmt} \rangle & ::= & \langle \text{call} \rangle \mid \langle \text{selec} \rangle \mid \langle \text{iter} \rangle \\
\langle \text{selec} \rangle & ::= & \textbf{if } \langle \text{b\_p\_e} \rangle \textbf{ then} \\
 & & \quad \langle \text{in\_code} \rangle \\
 & & [\textbf{else} \\
 & & \quad \langle \text{in\_code} \rangle] \\
 & & \textbf{end if} \\
\langle \text{iter} \rangle & ::= & \textbf{loop} \\
 & & \quad \textbf{maintaining } \langle \text{old\_assert} \rangle \\
 & & \quad \textbf{while } \langle \text{b\_p\_e} \rangle \textbf{ do} \\
 & & \qquad \langle \text{in\_code} \rangle \\
 & & \textbf{end loop} \\
\langle \text{call} \rangle & ::= & \langle \text{p\_nm} \rangle (\langle \text{cur\_var\_list} \rangle)
\end{array}
$$

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

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

Figure 33: Grammar Productions Restricted for ⟨p_body⟩

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

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

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 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.

$$\langle \text{stow\_sec} \rangle \ ::= \ \textbf{stow}(\langle \text{nat\_num} \rangle) \tag{3.16}$$
$$\begin{aligned} \langle \text{ACseq} \rangle \ ::= \ \{&\textbf{assume} \ \langle \text{idx\_assert} \rangle \mid \textbf{confirm} \ \langle \text{cur\_assert} \rangle \\ &\mid \textbf{confirm} \ \langle \text{idx\_assert} \rangle\} \end{aligned} \tag{3.17}$$

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

$$\langle \text{stow\_sec} \rangle \ ::= \ \varepsilon \mid \textbf{alter all} \quad (3.18)$$
$$\textbf{stow}(\langle \text{nat\_num} \rangle)$$
$$\langle \text{ACseq} \rangle \ ::= \ \{\textbf{assume} \ \langle \text{idx\_assert} \rangle \mid \textbf{confirm} \ \langle \text{idx\_assert} \rangle\} \quad (3.19)$$

Figure 36: Grammar Productions Restricted for ⟨top_lev_code⟩

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

$$\textit{top\_lev\_code}_{\mathcal{P}} = \text{Inst}(\mathcal{P}) \quad (3.20)$$
$$\textit{top\_lev\_code}_{\mathcal{M}} = \text{Inst}(\mathcal{M}) \quad (3.21)$$

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

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, $\textit{cd\_kern}_1$ 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.

[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

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 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 "$(\#q = q) \wedge (q = \Lambda \Leftrightarrow \text{empty})$"
procedure Make_two (x: Integer)
ensures "$x = 2$"
procedure Make_three (x: Integer)
ensures "$x = 3$"
$\}$

Figure 38: Example Context

$C' \backslash$ **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 $C'' \backslash$ *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 69. 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

$$\begin{aligned} \mathit{code}_{\mathcal{P}} &= \mathrm{Inst}(\mathcal{P}) && (3.22) \\ \mathit{top\_lev\_code}_{\mathcal{M}} &= \mathrm{Inst}(\mathcal{M}) && (3.23) \end{aligned}$$

and $\mathit{top\_lev\_code}_{\mathcal{M}}$ and $\mathit{code}_{\mathcal{P}}$ are both syntactically correct, then, in the math direction, $\mathit{top\_lev\_code}_{\mathcal{M}}$ may be derived from $\mathit{code}_{\mathcal{P}}$, and, in the program direction, $\mathit{code}_{\mathcal{P}}$ may be derived from $\mathit{top\_lev\_code}_{\mathcal{M}}$. The syntax of $\mathit{code}_{\mathcal{P}}$ 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 $\mathit{top\_lev\_code}_{\mathcal{M}}$ is that of Section 3.1.

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 68). 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 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⟩,

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

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

Figure 40: Equations Defining the Bridge Rule

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 "$\lfloor$" and "$\rfloor$" means, in the example of "post$\lfloor \#\mathrm{x}, \mathrm{x} \rfloor$", 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 "[", "$\leadsto$", and "]" has important syntactic implications. It means, for example, that "post$[\#\mathrm{x} \leadsto \mathrm{x}_i, \mathrm{x} \leadsto \mathrm{x}_j]$" 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 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.

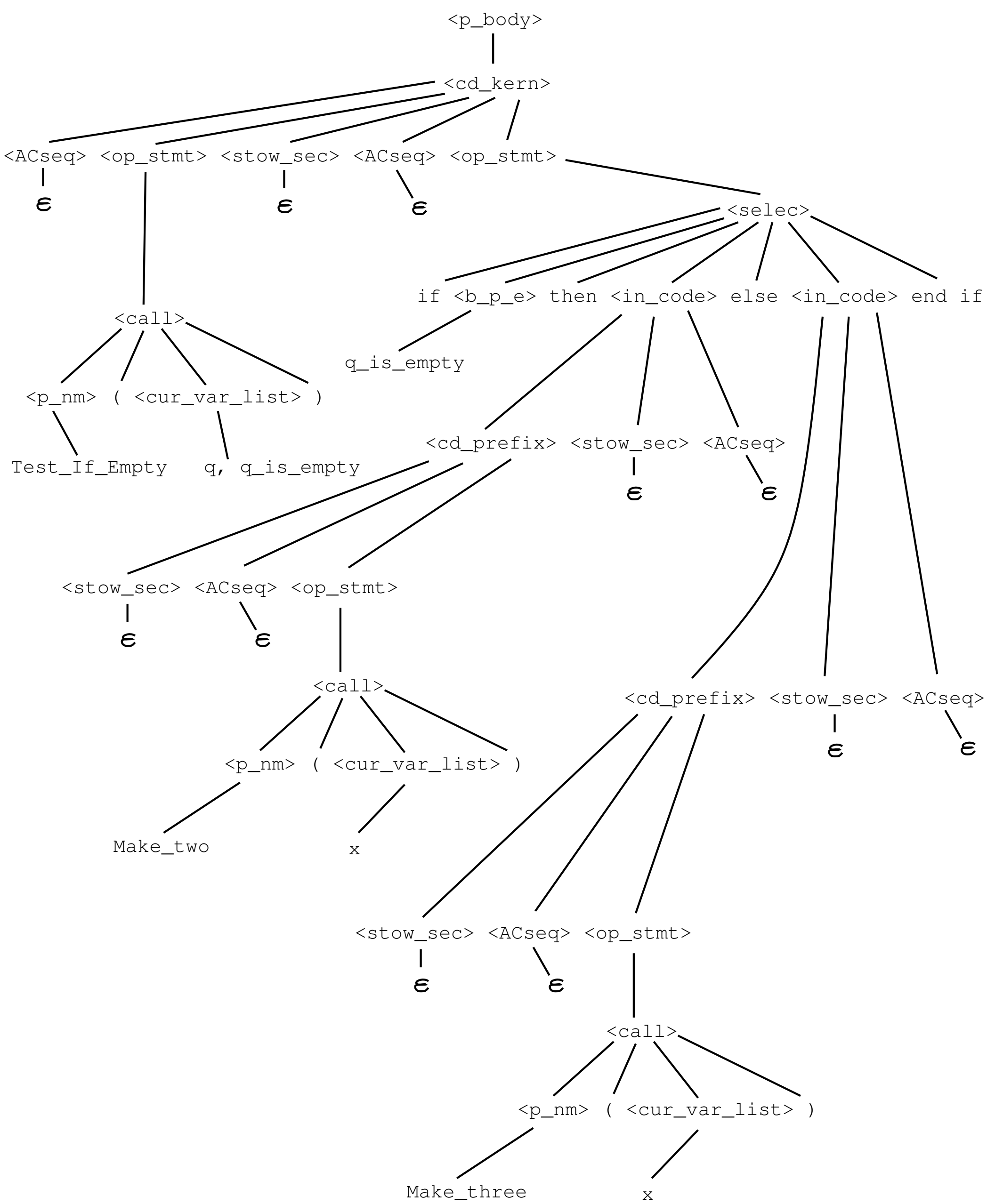

Figure 41: Grammatic Derivation of Body of Change_X

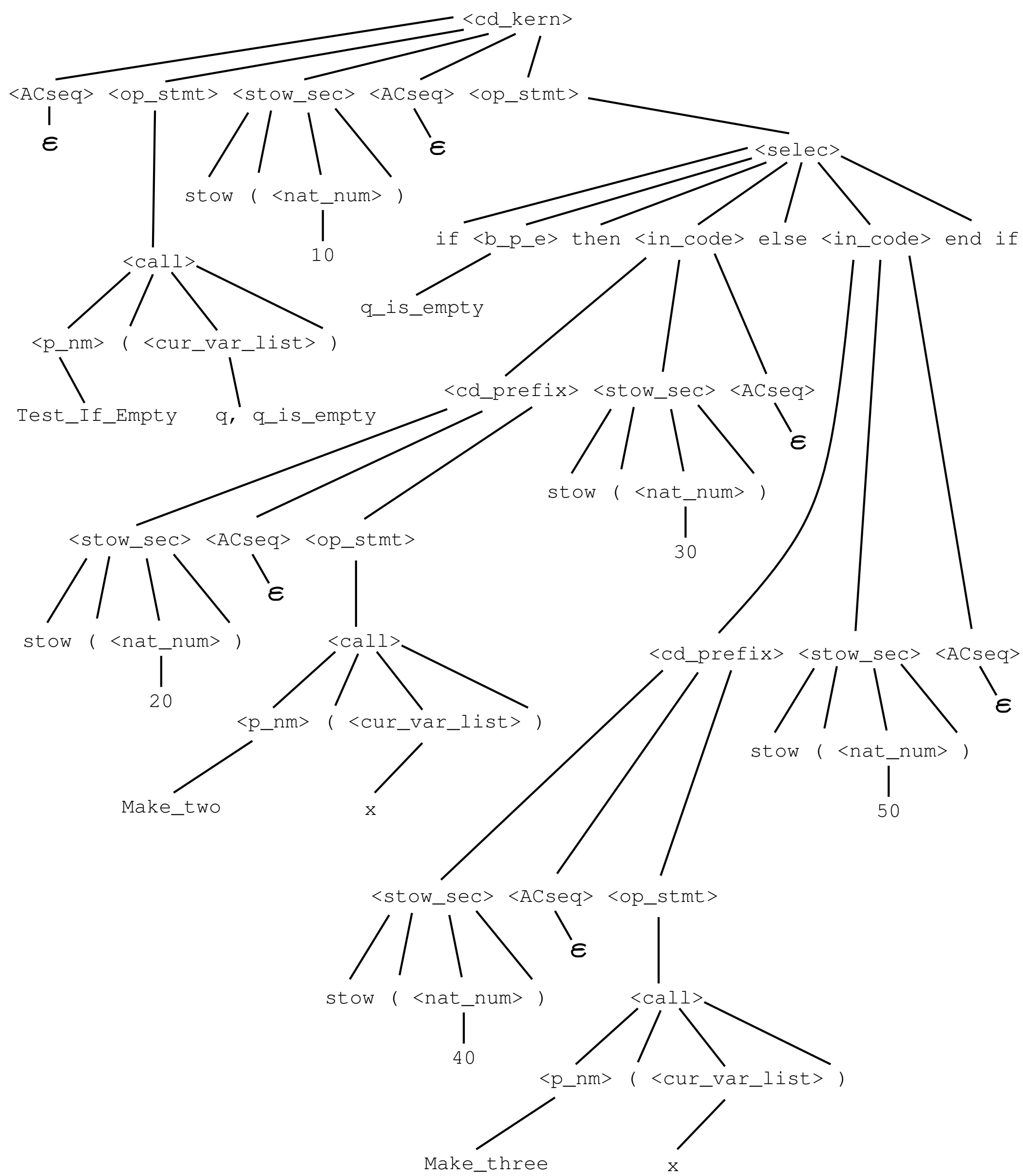

Figure 42: Grammatic Derivation of Stows_Added(Body of Change_X)

$C' \backslash$ **alter all**
stow(0)
whenever true do
assume true $\wedge$ **is_initial**(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 $(q_{60} = q_0) \wedge (q_{60} = \Lambda \Rightarrow x_{60} = 2) \wedge (q_{60} \neq \Lambda \Rightarrow x_{60} = 3)$
end whenever

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

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 58.

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 46) 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** $\text{post}[\#\text{x} \rightsquigarrow \text{x}_i, \text{x} \rightsquigarrow \text{x}_j]$ has the same effect on the assert status in its environment as **confirm** $\text{post}\lfloor\#\text{x}, \text{x}\rfloor$ 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.

$$
\begin{aligned}
\mathcal{P} \stackrel{\text{def}}{=} C\backslash\ & \textit{prec\_top\_lev\_code} \\
& \textbf{alter all} \\
& \textbf{stow}(i) \\
& ACseq_0 \\
& \textbf{whenever } \text{Br\_Cd } \textbf{do} \\
& \quad \textbf{assume } H \\
& \quad \textit{cd\_suffix} \\
& \textbf{end whenever} \\
& \textit{fol\_top\_lev\_code}
\end{aligned}
\tag{3.26}
$$

$$
\begin{aligned}
\mathcal{M} \stackrel{\text{def}}{=} C\backslash\ & \textit{prec\_top\_lev\_code} \\
& \textbf{alter all} \\
& \textbf{stow}(i) \\
& ACseq_0 \\
& \textbf{assume } (\text{Br\_Cd}) \Rightarrow (H) \\
& \textbf{whenever } \text{Br\_Cd } \textbf{do} \\
& \quad \textit{cd\_suffix} \\
& \textbf{end whenever} \\
& \textit{fol\_top\_lev\_code}
\end{aligned}
\tag{3.27}
$$

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

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.

$C' \backslash$ **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**

That is, we need to construct a neutral environment env′ such that execution of $\mathcal{M}$ 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 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

$$
\begin{array}{llll}
\mathcal{P} & \stackrel{\text{def}}{=} & C\backslash & \textit{prec\_top\_lev\_code} \\
 & & & \textbf{alter all} \\
 & & & \textbf{stow}(i) \\
 & & & ACseq_0 \\
 & & & \textbf{whenever } \mathrm{Br\_Cd} \textbf{ do} \\
 & & & \quad \mathrm{P\_nm}(\mathrm{ac}, \mathrm{ad}) \\
 & & & \quad \textbf{stow}(j) \\
 & & & \quad \textit{cd\_suffix} \\
 & & & \textbf{end whenever} \\
 & & & \textit{fol\_top\_lev\_code}
\end{array}
\tag{3.28}
$$

$$
\begin{array}{llll}
\mathcal{M} & \stackrel{\text{def}}{=} & C\backslash & \textit{prec\_top\_lev\_code} \\
 & & & \textbf{alter all} \\
 & & & \textbf{stow}(i) \\
 & & & ACseq_0 \\
 & & & \textbf{confirm } (\mathrm{Br\_Cd}) \Rightarrow (\mathrm{pre}[\mathrm{x} \rightsquigarrow \mathrm{ac}_i, \mathrm{y} \rightsquigarrow \mathrm{ad}_i, \mathrm{z} \rightsquigarrow \mathrm{z}_i]) \\
 & & & \textbf{alter all} \\
 & & & \textbf{stow}(j) \\
 & & & \textbf{whenever } \mathrm{Br\_Cd} \textbf{ do} \\
 & & & \quad \textbf{assume } \mathrm{b}_j = \mathrm{b}_i \wedge (\mathrm{post}[\#\mathrm{x} \rightsquigarrow \mathrm{ac}_i, \mathrm{x} \rightsquigarrow \mathrm{ac}_j, \\
 & & & \qquad\qquad\qquad\qquad\quad \#\mathrm{y} \rightsquigarrow \mathrm{ad}_i, \mathrm{y} \rightsquigarrow \mathrm{ad}_j, \\
 & & & \qquad\qquad\qquad\qquad\quad \#\mathrm{z} \rightsquigarrow \mathrm{z}_i, \mathrm{z} \rightsquigarrow \mathrm{z}_j]) \\
 & & & \quad \textit{cd\_suffix} \\
 & & & \textbf{end whenever} \\
 & & & \textit{fol\_top\_lev\_code}
\end{array}
\tag{3.29}
$$

Additional Syntactic Restriction:

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

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**(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

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

$\mathcal{P} \stackrel{\text{def}}{=} C\backslash$ *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 $(\mathrm{Br\_Cd}) \wedge (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])$ **do**
  assume $\mathrm{x}_j = \mathrm{x}_i$
  cd_kern
  stow(k)
  ACseq
end whenever
alter all
stow(n)
assume $((\mathrm{Br\_Cd}) \wedge (\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])) \Rightarrow (\mathrm{x}_n = \mathrm{x}_k)$
assume $((\mathrm{Br\_Cd}) \wedge \neg(\mathrm{MExp}(b\_p\_e)[\mathrm{y} \rightsquigarrow \mathrm{y}_i])) \Rightarrow (\mathrm{x}_n = \mathrm{x}_i)$
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

$$
\mathcal{P} \stackrel{\text{def}}{=} C\backslash \quad \begin{array}{l}
\textit{prec\_top\_lev\_code} \\
\textbf{alter all} \\
\textbf{stow}(i) \\
ACseq_0 \\
\left| \begin{array}{l}
\textbf{whenever } \mathrm{Br\_Cd} \textbf{ do} \\
\quad \textbf{if } \textit{b\_p\_e} \textbf{ then} \\
\quad\quad \textbf{stow}(j) \\
\quad\quad \textit{cd\_kern}_1 \\
\quad\quad \textbf{stow}(k) \\
\quad\quad ACseq_1 \\
\quad \textbf{else} \\
\quad\quad \textbf{stow}(l) \\
\quad\quad \textit{cd\_kern}_2 \\
\quad\quad \textbf{stow}(m) \\
\quad\quad ACseq_2 \\
\quad \textbf{end if} \\
\quad \textbf{stow}(n) \\
\quad \textit{cd\_suffix} \\
\textbf{end whenever}
\end{array} \right. \\
\textit{fol\_top\_lev\_code}
\end{array} \tag{3.32}
$$

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

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

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**) $\Rightarrow (x_{10} = x_0 \wedge ((q_0 = q_{10}) \wedge (q_{10} = \Lambda \Leftrightarrow \text{q\_is\_empty}_{10})))$
  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**(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**) $\wedge$ (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**) $\wedge \neg$(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**) $\wedge$ (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**) $\wedge \neg$(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

construct env′ to be a witness to $\mathcal{M}$'s invalidity. Because $\mathcal{M}$ has three additional **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 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.

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

$$
\begin{array}{lll}
\mathcal{P} & \stackrel{\mathrm{def}}{=} & C\backslash \; \begin{array}[t]{l}
\textit{prec\_top\_lev\_code} \\
\textbf{alter all} \\
\textbf{stow}(i) \\
\textit{ACseq}_0 \\
\textbf{whenever } \mathrm{Br\_Cd} \textbf{ do} \\
\quad \textbf{loop} \\
\quad\quad \textbf{maintaining } \mathrm{Inv}\lfloor \mathrm{x}, \#\mathrm{x} \rfloor \\
\quad\quad \textbf{while } \textit{b\_p\_e} \textbf{ do} \\
\quad\quad\quad \textbf{stow}(j) \\
\quad\quad\quad \textit{cd\_kern} \\
\quad\quad\quad \textbf{stow}(k) \\
\quad\quad\quad \textit{ACseq} \\
\quad \textbf{end loop} \\
\quad \textbf{stow}(l) \\
\quad \textit{cd\_suffix} \\
\textbf{end whenever} \\
\textit{fol\_top\_lev\_code}
\end{array}
\end{array}
\tag{3.34}
$$

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

$$\mathcal{M} \stackrel{\text{def}}{=} C\backslash \quad \textit{prec\_top\_lev\_code} \tag{3.35}$$

alter all
stow(i)
$ACseq_0$
confirm (Br_Cd) $\Rightarrow$ (Inv[x $\rightsquigarrow$ x_i, #x $\rightsquigarrow$ x_i])
alter all
stow(j)
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)
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
fol_top_lev_code

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

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

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

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

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 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**) $\wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow$
        $(\mathrm{q\_is\_empty}_{40} = \mathrm{q\_is\_empty}_{10} \wedge \mathrm{q}_{40} = \mathrm{q}_{10} \wedge \mathrm{x}_{40} = \mathrm{x}_{10})$
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**(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$_{10})))$
alter all
stow(20)
whenever (**true**) $\wedge$ (q_is_empty$_{10}$) **do**
  assume q_is_empty$_{20}$ = q_is_empty$_{10}$ $\wedge\ q_{20} = q_{10} \wedge x_{20} = x_{10}$
  Make_two(x)
  stow(30)
end whenever
alter all
stow(40)
assume ((**true**) $\wedge\ \neg$(q_is_empty$_{10}$)) $\Rightarrow$
    (q_is_empty$_{40}$ = q_is_empty$_{10}$ $\wedge\ q_{40} = q_{10} \wedge x_{40} = x_{10}$)
confirm ((**true**) $\wedge\ \neg$(q_is_empty$_{10}$)) $\Rightarrow$ (**true**)
alter all
stow(50)
assume ((**true**) $\wedge\ \neg$(q_is_empty$_{10}$)) $\Rightarrow$
    (q_is_empty$_{50}$ = q_is_empty$_{40}$ $\wedge\ q_{50} = q_{40} \wedge x_{50} = 3$)
whenever (**true**) $\wedge\ \neg$(q_is_empty$_{10}$) **do**
end whenever
alter all
stow(60)
assume ((**true**) $\wedge$ (q_is_empty$_{10}$)) $\Rightarrow$
    (q_is_empty$_{60}$ = q_is_empty$_{30}$ $\wedge\ q_{60} = q_{30} \wedge x_{60} = x_{30}$)
assume ((**true**) $\wedge\ \neg$(q_is_empty$_{10}$)) $\Rightarrow$
    (q_is_empty$_{60}$ = q_is_empty$_{50}$ $\wedge\ q_{60} = q_{50} \wedge x_{60} = x_{50}$)
confirm (**true**) $\Rightarrow$ $((q_{60} = q_0) \wedge (q_{60} = \Lambda \Rightarrow x_{60} = 2) \wedge (q_{60} \neq \Lambda \Rightarrow x_{60} = 3))$
whenever true do
end whenever

Figure 58: Second Application of Procedure Call Rule

$$\mathcal{P} \stackrel{\text{def}}{=} C\backslash \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textbf{alter all} \\ \textbf{stow}(i) \\ ACseq_0 \\ \left| \begin{array}{l} \textbf{whenever } \mathrm{Br\_Cd} \textbf{ do} \\ \textbf{end whenever} \end{array} \right. \\ \textit{fol\_top\_lev\_code} \end{array} \tag{3.38}$$

$$\mathcal{M} \stackrel{\text{def}}{=} C\backslash \begin{array}{l} \textit{prec\_top\_lev\_code} \\ \textbf{alter all} \\ \textbf{stow}(i) \\ ACseq_0 \\ \textit{fol\_top\_lev\_code} \end{array} \tag{3.39}$$

Figure 59: Equations Defining the Rule for Empty Guarded Blocks

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 68), 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.

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

$C' \backslash$ **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$_{10})))$
alter all
stow(20)
whenever (**true**) $\wedge$ (q_is_empty$_{10}$) **do**
assume q_is_empty$_{20}$ = q_is_empty$_{10}$ $\wedge$ $q_{20} = q_{10} \wedge x_{20} = x_{10}$
Make_two(x)
stow(30)
end whenever
alter all
stow(40)
assume ((**true**) $\wedge$ $\neg$(q_is_empty$_{10}$)) $\Rightarrow$
(q_is_empty$_{40}$ = q_is_empty$_{10}$ $\wedge$ $q_{40} = q_{10} \wedge x_{40} = x_{10}$)
confirm ((**true**) $\wedge$ $\neg$(q_is_empty$_{10}$)) $\Rightarrow$ (**true**)
alter all
stow(50)
assume ((**true**) $\wedge$ $\neg$(q_is_empty$_{10}$)) $\Rightarrow$
(q_is_empty$_{50}$ = q_is_empty$_{40}$ $\wedge$ $q_{50} = q_{40} \wedge x_{50} = 3$)
whenever (**true**) $\wedge$ $\neg$(q_is_empty$_{10}$) **do**
end whenever
alter all
stow(60)
assume ((**true**) $\wedge$ (q_is_empty$_{10}$)) $\Rightarrow$
(q_is_empty$_{60}$ = q_is_empty$_{30}$ $\wedge$ $q_{60} = q_{30} \wedge x_{60} = x_{30}$)
assume ((**true**) $\wedge$ $\neg$(q_is_empty$_{10}$)) $\Rightarrow$
(q_is_empty$_{60}$ = q_is_empty$_{50}$ $\wedge$ $q_{60} = q_{50} \wedge x_{60} = x_{50}$)
confirm (**true**) $\Rightarrow$ $((q_{60} = q_0) \wedge (q_{60} = \Lambda \Rightarrow x_{60} = 2) \wedge (q_{60} \neq \Lambda \Rightarrow x_{60} = 3))$

Figure 60: First Application of Rule for Empty Guarded Blocks

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

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 $\langle$gd_blk$\rangle$ 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**

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

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

$$
\begin{aligned}
\mathcal{P} \quad = \quad C\backslash \quad & \mathit{prec\_top\_lev\_code} && (3.42)\\
& \textbf{assume } H_1 \\
& \textbf{assume } H_2 \\
& \mathit{fol\_top\_lev\_code} \\
\mathcal{M} \quad = \quad C\backslash \quad & \mathit{prec\_top\_lev\_code} && (3.43)\\
& \textbf{assume } (H_1) \wedge (H_2) \\
& \mathit{fol\_top\_lev\_code}
\end{aligned}
$$

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

$$
\begin{aligned}
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\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})) \\
& \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\qquad (\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\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})) \\
& \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\qquad (\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\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})) \\
& \qquad \wedge (((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
& \qquad\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}))) \\
& \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{aligned}
$$

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

$$\begin{aligned} \mathcal{P} &= C\backslash \quad \textit{top\_lev\_code} && (3.44) \\ & \qquad\quad\ \ \textbf{assume } H_1 \\ & \qquad\quad\ \ \textbf{confirm } H_2 \\ \mathcal{M} &= C\backslash \quad \textit{top\_lev\_code} && (3.45) \\ & \qquad\quad\ \ \textbf{confirm } (H_1) \Rightarrow (H_2) \end{aligned}$$

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

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.

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

$$
\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\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})) \\
\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\qquad (\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\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})) \\
\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\qquad (\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\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})) \\
\qquad\qquad \wedge\ (((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
\qquad\qquad\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 \\
\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

$$
\begin{array}{rcl}
\mathcal{P} & = & C\backslash\ \ \textit{prec\_top\_lev\_code} \\
 & & \qquad \textbf{confirm}\ H_1 \\
 & & \qquad \textbf{confirm}\ H_2 \\
 & & \qquad \textit{fol\_top\_lev\_code}
\end{array}
\tag{3.46}
$$

$$
\begin{array}{rcl}
\mathcal{M} & = & C\backslash\ \ \textit{prec\_top\_lev\_code} \\
 & & \qquad \textbf{confirm}\ (H_1) \wedge (H_2) \\
 & & \qquad \textit{fol\_top\_lev\_code}
\end{array}
\tag{3.47}
$$

Figure 68: Equations Defining the Rule for Consecutive **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}))\\
\qquad\qquad \wedge\ (((((\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 69: An Application of the Rule for Consecutive **confirm** Statements

$$
\begin{array}{l}
C' \backslash\ \textbf{confirm}\ ((\textbf{true}) \Rightarrow (\textbf{true} \wedge \textbf{is\_initial}(\mathrm{q\_is\_empty}_0))) \Rightarrow \\
\qquad\qquad (((\textbf{true}) \Rightarrow (\textbf{true})) \\
\qquad\qquad \wedge ((((\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\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 \\
\qquad\qquad ((((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow (\textbf{true})) \\
\qquad\qquad \wedge (((((\textbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
\qquad\qquad\qquad (\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\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 \\
\qquad\qquad ((((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow (\textbf{true})) \\
\qquad\qquad \wedge (((((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
\qquad\qquad\qquad (\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\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})) \\
\qquad\qquad \wedge (((\textbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow \\
\qquad\qquad\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 \\
\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 70: Five Rule Applications

the **assume-confirm** rule and the rule for consecutive **confirm** statements, and a 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.

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

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.

$$
\begin{aligned}
&C' \backslash\ ((\mathbf{true}) \Rightarrow (\mathbf{true} \wedge \mathbf{is\_initial}(\mathrm{q\_is\_empty}_0))) \Rightarrow \\
&\quad (((\mathbf{true}) \Rightarrow (\mathbf{true})) \\
&\quad \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})))) \\
&\quad \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 \\
&\quad ((((\mathbf{true}) \wedge (\mathrm{q\_is\_empty}_{10})) \Rightarrow (\mathbf{true})) \\
&\quad \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)) \\
&\quad \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 \\
&\quad ((((\mathbf{true}) \wedge \neg(\mathrm{q\_is\_empty}_{10})) \Rightarrow (\mathbf{true})) \\
&\quad \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)) \\
&\quad \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})) \\
&\quad \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 \\
&\quad ((\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'

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.

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″\ alter all
      stow(100)
      assume q_100 = Λ ⇔ q_is_empty_100
      whenever true do
         loop
               maintaining q = Λ ⇔ 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 = Λ
      end whenever

where C″ = { y: Item
             procedure Dequeue (q: Queue
                                x: Item)
                       requires "q ≠ Λ"
                       ensures "#q = ⟨x⟩ ∗ q"} ∪ C′.
```

Figure 72: Example: Iteration

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