A COMPUTER SYSTEM FOR CHECKING PROOFS

Scott D. Johnson

Ph.D. Thesis

January 1981

TR 80-444

Department of Computer Science
Cornell University
Ithaca, New York, 14853

A COMPUTER SYSTEM FOR CHECKING PROOFS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment for the Degree of

Doctor of Philosophy

by

Scott David Johnson

January 1981

BIOGRAPHICAL SKETCH

Scott David Johnson was born on October 16, 1953 in Joliet, Illinois. He graduated from Joliet Township High School, West Campus, in 1971. He received the Bachelor of Science degree in computer science from the Illinois Institute of Technology in 1975, graduating with high honors. In 1979 he received the degree of Master of Science in computer science from Cornell University. During the 1979-80 academic year, he was an instructor at Cornell University. He has accepted a position with the the Western Development Laboratories of the Ford Aerospace Corporation.

ACKNOWLEDGMENTS

"And I want to thank the Academy, my family, Harry who fed my dog during rehearsals ..." I once thought the long-winded speeches those people give were rather contrived. But faced with the task of writing this section, I think I know just how they feel.

First of all, I must thank my advisor Bob Constable. If it were not for all the carrots and sticks (mostly carrots) he has used on me over the past few years, this thesis would have never been completed. Bob has a wonderful sense of optimism, and together we have embarked on projects I would have never attempted on my own.

I am also deeply grateful to Dean Krafft and Dan Zlatin. The algorithms I proved in Chapters 5, 6, and 7 were developed in collaboration with Dean and Dan, each of whom worked on the implementation of the V-code Interpreter. They carefully read drafts of the thesis and found countless errors. I want to especially thank Dean for performing the Herculean task of wading through an exceedingly rough draft of Chapter 6.

I would also like to mention the other people who have worked on the PL/CV project. Mike O'Donnell was the principle designer of PL/CV1, and gave helpful advice during the design of PL/CV2. Carl Hauser helped write the manual for PL/CV1. Mike O'Donnell and Tat-Hung Chan developed the arithmetic package used in PL/CV2; Carl Eichenlaub later added some extensions this package. Ryan Stansifer is currently translating the arithmetic package and V-code Interpreter into the C programming language so that it can be used by AVID, an interactive version of PL/CV being developed by Dean Krafft.

While I was working on the PL/CV project I was supported by the

National Science Foundation. Grant MCS-76-14293 got the project started, and grant MCS-78-00953 helped get it finished. While supported by grant SED-79-18966, I taught an experimental course that used PL/CV.

There is no way I can think back through all the seminars and discussions in which I participated and remember everyone who gave me ideas that eventually made it into my thesis. But I do recall fruitful discussions with Barry Bakalor, Alan Demers, Corky Cartwright, Carl Eichenlaub, Carl Hauser, Ralph Johnson, and Gary Levin.

This document was prepared on the UNIX system at Cornell University. I want to thank everyone at Cornell responsible for obtaining and maintaining that computer system, especially Alan Demers. I simply cannot conceive of writing this thesis without the aid of a word processing system.

Because I didn't quite finish my thesis on schedule, my stay in Ithaca was a little longer than I expected it to be. I want to express my appreciation to everyone who gave me support during those last two months of frenzy. I would like to thank Lynn Kivell for giving me a floor to sleep on, Paula Zebrowski for helping me with typing, and Dean Krafft for helping me produce the final version. I am especially grateful to my new employers at Ford Aerospace showing patience and understanding during this period.

It was inevitable by the Law of the Innate Perversity of Inanimate Objects, that during the final week the disk drive on which my thesis was stored would break. I am indebted to Alan Demers, Ann Kratzer, Hans Boehm, and Dean Krafft, who, armed with oscilloscopes and logic probes, valiantly rescued Chapter 4 from limbo.

And finally, I must mention Steve Jackson and Howard Thompson.

Without the materials they provided, my stay in Ithaca would have been much shorter, but not nearly so enjoyable.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1 The PL/CV Project

In 1975, a project was begun at Cornell University to produce a programming logic in which formal proofs of programs could be written and mechanically checked. The result of this project is the programming logic PL/CV; this thesis concerns the computer system used to check proofs written in PL/CV.

Actually, there have been two proof checkers written for PL/CV. The PL/CV1 Proof Checker [22], completed in 1976, provided no automatic assistance whatsoever. The purpose this first version was to get some experience with formal proofs spelled out in complete detail.

Proofs written in PL/CV1 are rather unwieldy. However, the valuable experience gained with PL/CV1 was used to guide the design of the PL/CV2 language and Proof Checker, the subject of this thesis. A third version of PL/CV is currently being designed by Constable and Zlatin [23]. In this thesis, the name "PL/CV" will be used to refer to PL/CV2.

The usual approach to program verification is typified by the Stanford Pascal Verifier [43]. From a program annotated with a specification and loop invariants a collection of formulas called verification conditions are generated. The verification conditions have the property that they are valid if and only if the program is correct.

Typically the verification conditions are large and messy. Before they are presented to the user, they are processed by a program called a simplifier. The simplifier is an automatic theorem prover that performs transformations on the verification conditions that produce equivalent but simpler formulas. Any verification conditions that cannot be simplified to "true" are presented to the user. To prove his program, the user must prove these formulas, either informally or with some automated assistance. If the formulas output by the verifier are not valid, there is an error in the program.

A problem with this approach to of verification is that the programming logic is buried within the verification condition generator. As a result, the program and its proof become uncoupled. A user can understand how his program works but not understand where a particular verification condition he must prove has come from.

The PL/CV system does not have this problem. The programming logic is explicit, and the program and its proof are integrated into a single object. The logic is simple enough that it has been used in undergraduate computer science courses. The PL/CV system does not contain an automatic theorem prover; it is not a system designed to automatically construct proofs. Instead, an attempt was made to design the PL/CV logic so that a programmer's intuition about why his program works could easily be expressed. The programmer, not the verification system, produces the proof. In PL/CV, the role of the simplifier is fulfilled by abbreviation rules that allows parts of the formal proof to omitted in well-defined ways.

Because a theorem prover is not required, it should be possible to check PL/CV proofs efficiently. The original goal was to make the Proof Checker as efficient as an optimizing compiler. Although the algorithms used in the Proof Checker, which are presented in this thesis, are all fairly efficient, the implemented system uses more time and memory than I would prefer. It is not clear whether I have simply underestimated the magnitude of the problem or the inefficiencies are due to the overhead incurred with the PL/I system with which the Proof Checker was implemented. As of this writing, the V-code Interpreter (which is the bottleneck in the system) is being translated to the C programming language so that it can be run on a VAX computer. The results ought to be interesting.

1.2 The PL/CS Programming Language

The PL/CV project was never intended to be a project in programming language design. Therefore an existing programming language had to be chosen on which to base the verification system. It seemed natural to base the system on PL/C, a dialect of PL/I used in most of the computer science courses at Cornell. Basing the system on a PL/I dialect made it easier to use PL/CV for teaching at Cornell. Of course, a programming logic based on full PL/C, a large subset of PL/I, was out of the question. Instead, PL/CV is based on PL/CS, a "disciplined" subset of PL/I that was designed by Richard Conway, with the help of various other members of the Computer Science Department at Cornell. The PL/CS language was an attempt to delete most of the notorious features of PL/I, leaving a subset with clean semantics, acceptable for teaching structured programming. Conway and Constable [24] discuss the factors that influenced the design of PL/CS.

The PL/CV logic can be viewed as providing axiomatic semantics for PL/CS; for the logic to be correct, these semantics must match the official definition of PL/CS. Unfortunately, the "official definition of PL/CS" is spread over three documents. PL/CS is defined by a Cornell technical report [11], in terms of how it differs from PL/C. PL/C, in turn, is defined [72] in terms of how it differs from PL/I-F. And PL/I-F is defined by an IBM document [45].

The situation is not as bad as it appears. The PL/CS language is simple enough that its semantics are for the most part obvious to anyone acquainted with an ALGOL-like language. When it is necessary to resolve a question about a fine point, recourse to the definition of PL/I-F is usually necessary.

The rest of this section is an overview of the PL/CS language as defined by Bishop and Conway [11]. The differences between PL/CS and PL/CV, and the reasons for those differences, will be noted.

The PL/CS language contains the following data types: integers, floating point numbers, character strings, Booleans, and arrays thereof.

To maintain compatibility with PL/I, short names were chosen for the data types that would be interpreted in a reasonable fashion by the rules for default attributes in PL/I. For example, the name FIXED is used for integers, because PL/I interprets:

```
DECLARE J FIXED;
```

as:

```
DECLARE J DECIMAL FIXED; .                                    (1.1)
```

One might suggest the name DECIMAL as being slightly more suggestive of the concept of integer. And while PL/I will interpret:

```
DECLARE J DECIMAL;
```

as the declaration (1.1), it interprets:

```
DECLARE X DECIMAL;
```

as:

```
ROOT: PROCEDURE(X) RETURNS(FLOAT);
DECLARE X FLOAT;
ASSUME X >= 0;
ATTAIN ROOT(X)*ROOT(X) = X;
DECLARE (LO, HI, MID) FLOAT;
LO := 0;
HI := X;
DO WHILE(LO ~= HI);
    /* LO <= ROOT(X) <= HI */
    MID := (LO + HI) / 2;

    /* test if MID <= ROOT(X) */
    IF (MID*MID <= X) THEN LO := MID;
                      ELSE HI := MID;
END;
RETURN(LO);
END ROOT;
```

Figure 1.1 A program that computes square roots using bisection

```
DECLARE X DECIMAL FLOAT;
```

The first letter of the identifier name is used to determine a default type. (This convention is a holdover from FORTRAN [7].)

The PL/CS names for integer, floating point, Boolean, and character strings are FIXED, FLOAT, BIT, and CHARACTER, respectively. The types FLOAT and CHARACTER were omitted from PL/CV because it is difficult to give axioms that can be used to reason about them.

When working with programs that use floating point arithmetic, one typically proves the correctness of the program assuming real arithmetic is being used, and then (when possible) proves some bound on roundoff errors. This method allows one to prove that the program yields results that are close to the actual solution, which is as strong a statement as can be expected.

This method is far less acceptable when proving, for example, that the program halts. As an example, consider the program in Figure 1.1. If real arithmetic is used, the program never halts. But given floating point arithmetic as it is implemented on most machines, the program will always halt when X >= 0, and will return a value very close to the square root of X. But to prove that the program halts, it is necessary to know some intimate details concerning the system of floating point arithmetic used by the machine on which it is to be run.

Holm [42] has investigaged the problem of formally reasoning about programs that use floating point arithmetic.

Reasoning about character strings ought to be a more tractable problem. Indeed, the character strings of SNOBOL4 [38], for axample, have a reasonable axiomatization [27]. But PL/I strings do not obey simple properties such as:

x = y implies (x || a) = (y || a)

where || denotes concatenation. There are two problems with PL/I strings:

(1) There is a maximum length associated with each string variable. If a longer string is assigned to the variable, it is truncated.

(2) When two strings are being compared, trailing blanks are ignored.

A reasonable way of dealing with the first problem would be to add the restriction that strings can only be assigned to variables large enough to hold them. This restriction would be enforced in the way that array subscripts are required to be in bounds. In a situation in which the programmer desired the truncation, he be required to explicitly perform the truncation with the substring function.

The second problem could be solved by considering a string comparison of the form X ρ Y appearing in the programming language to be so-called syntactic sugar for:

STRIP(X) ρ STRIP(Y)

where STRIP(X) is define to be X with any trailing blanks stripped off.

The following programming constructs are included in PL/CS: declarations, assignment statements, procedure and function definitions and calls, the conditional statements IF and SELECT, WHILE loops, UNTIL loops, indexed loops, loop exits, the RETURN statement, the GOTO statement, and some simple input and output statements. The SELECT, LEAVE and UNTIL statements do not appear in PL/C. These statements were included in PL/CS after they were included in Release 3 of IBM's PL/I-X [44].

Input and output statements were omitted from PL/CV. From a technical standpoint, they are no more difficult than arrays. However, to include them, it would be necessary to have a way of referring to the input and output of the program in the assertion language.

There is no block structure in PL/CS. All variables and parameters must be declared at the beginning of each routine, and procedure and functions definitions cannot be nested. The PL/CS compiler checks that variables and parameters declared in a routine with the READONLY attribute are not modified by that routine. Variables declared with the EXTERNAL attribute are global to every routine in the program. All other variables are local, and are reallocated each time the routine is entered.

PL/CS also allows variables to be declared STATIC. A static variable is allocated in the same manner as an external variable. The difference the two kinds of variables is if the same identifier is declared to be STATIC in two different routines, two different variables are allocated. Static variables are used by a routine to keep its own private history concerning calls to that routine. To prove a program that uses static variables, assertions about the values of the static variables would have to be written in the calling routine. Since the calling routine must have a name for these variables, the simplest solution is to omit static variables from PL/CV, and use external variables instead.

It is usually poor programming practice to use a construct that has been excised from a language to make program verification easier. Static variables appear to be an exception to this rule. When one is reading a program that contains a static variable, one knows that the variable will not be changed between calls to the routine. When the static variable is replaced by an external variable, this information is lost. The reader must now inspect all the other routines in the program to convince himself that the variable is left unchanged.

The coroutine construct provides the same localizing function as static variables, but in a more general fashion. For example, instead of writing a random number generator that maintains a state in static variable, it can be written as a coroutine. The coroutine is written as a loop that outputs a series of random numbers; each call to the routine causes another iteration of the loop to be performed and the next number in the sequence to be output. Clarke [15] has investigated the problem of proving programs that use coroutines.

In PL/C the distinction between between procedures and functions is somewhat nebulous. The usual practice among PL/C programmers is to supply a RETURNS option, which indicates the type of the value to be returned, on procedures if and only if the procedure is expected to return a value. However, if no RETURNS options is present, a default one is supplied; the default type depends on the first letter of the procedure name. It is the responsibility of the programmer to ensure that if a routine is called with the CALL statement it returns a value, and if the routine is called

```
EITHER: PROCEDURE(X);
DECLARE X FIXED;
IF X=1 THEN RETURN(1.0);
       ELSE RETURN;
END EITHER;

BOTH: PROCEDURE;
DECLARE Y FLOAT;
Y := EITHER(1);
CALL EITHER(2);
  .
  .
  .
```

Figure 1.2 A routine that can be called as either a procedure or function

as a function it does not. (There is no notion of voiding, as in ALGOL 68 [70].) The program in Figure 1.2, for example, is perfectly legal in PL/C.

The situation has been rectified in PL/CS. Routines are classified as procedures or functions on the basis of whether or not they are defined with a RETURNS options. Procedures are called using the CALL statement, and functions are called from within expressions. Also, when a routine is called, the types of the formal and actual parameters must match exactly so that no data conversion is necessary.

Functions are not allowed to have side effects. All function parameters implicitly have the READONLY attribute, as do external variables declared in a function. Input and output statements may not be used in functions. In PL/CS, functions are not allowed to call procedures. This restriction has been relaxed in PL/CV: a function cannot initiate a sequence of procedure calls that can ultimately lead to the modification of a function parameter or external variable, or to the execution of an input or output statement. (The latter condition is trivial to enforce, because there are no input or output statements in PL/CV.)

The syntax for the rest of the programming constructs of PL/CS are

```
V := E;
A := C;
A1 := A2;
A(E1, E2, ... En) := En+1;

CALL R(E1, E2, ... En);

RETURN;
RETURN(E);

L: DO WHILE(B);
   :
   END L;

L: DO UNTIL(B);
   :
   END L;

L: DO J := E1 TO E2 BY E3;
   :
   END L;

LEAVE L;

GOTO L;
:
L:;
```

```
IF (B)
THEN DO;
     :
     END;
ELSE DO;
     :
     END;

L: SELECT;
   WHEN (B1)
       DO;
       :
       END;
   WHEN (B2)
       DO;
       :
       END;
   :
   WHEN (Bn)
       DO;
       :
       END;
   OTHERWISE
       DO;
       :
       END;
   END L;
```

Figure 1.3 PL/CS programming constructs

shown in Figure 1.3. In the figure, V stands for a scalar variable, A stands for an array variable, C stands for a constant, E stands for an expression, B stands for a Boolean expression, J stands for a scalar FIXED variable, and L stands for a statement label. Although the symbol = is used for assignment in PL/CS and PL/CV, the symbol := will be consistently used to denote assignment in this thesis, for reasons explained in section

4.2. The labels required in PL/CS after END and before DO and SELECT and the parenthesis required in the IF and SELECT statements are optional in PL/CV.

The RETURN statement is used to exit a procedure; RETURN(E) is used to exit a function and pass the value E back to the calling program. Control is not allowed to flow "off the end" of a routine. In PL/CS, this restriction is enforced by requiring every routine to end with an explicit RETURN statement, even if that statement can never be executed. In PL/CV, the user must prove that control never flows off the end of a routine. For example, a routine in which the last statement is an IF statement whose THEN and ELSE clauses both end with a RETURN statement is allowed in PL/CV but not PL/CS.

The assignment statement evaluates the the expresaion on the right, and assigns that value to the variable on the left. The statement A := C sets every element of the array A to C. Although PL/CS allows type conversion on assignment, in PL/CV no type conversion of any kind is allowed. Because the only simple types in PL/CV are FIXED and BIT, this additional restriction is not very harsh.

The CALL statement is used to invoke a procedure. In PL/CS, arguments are not allowed to be readonly variables. This restriction has been relaxed in PL/CV; readonly variables may be passed to procedures if the corresponding formal parameter is readonly. The CALL statement has also been restricted in PL/CV so that aliasing cannot be introduced. For example, the procedure call:

```
CALL P(X, X);
```

is not allowed unless both formal parameters of P are READONLY. Aliasing is discussed more fully in section 4.11.

To execute a SELECT statement, the Boolean expressions B_1, B_2 ... are evaluated until one is found to be true or this list is exhausted. If one of the Boolean expressions is found to be true, the corresponding WHEN clause is executed. If none of the Boolean expressions are true, the OTHERWISE clause is executed. The IF statement is equivalent to a SELECT statement with one WHEN clause.

The WHILE and UNTIL loops execute their bodies until the Boolean expression evaluates to false or true, respectively. In the WHILE loop, a test is made to see if the execution of the loop is complete before the loop body is executed the first time. This initial test is omitted in an UNTIL loop; its body is guaranteed to be executed at least once.

When an indexed loop is executed, the index variable J is initialized to E_1. Alternately, the loop body is executed and the index variable is incremented by E_3, until the index variable passes E_2. (The semantics of the indexed loop are described more precisely in section 4.17.) In PL/CS, the the variables in E_1, E_2, and E_3 are given the READONLY attribute inside tbe loop body, and a run time check is used to ensure E_3 is not zero. These restrictions are enforced by PL/CV, as are two other restrictions: the expressions E_1 and E_2 cannot depend on the index variable J, and E_3 is required to be 1 or -1. So that E_1 and E_2 cannot depend implicitly on J by calling a function that depends on J, the variable J is not allowed to have the EXTERNAL attribute. The reasons for all these restrictions are discussed in section 4.17.

After an indexed loop completes execution, the value of the index variable J not defined. In PL/CS run time checking similar to that performed for newly declared variables is performed to ensure the variable J is given a value by an assignment statement or procedure call before it is used again. In PL/CV, the variable may be used before it is given a value, but the the program must be proved to work no matter what value J has when the loop completes.

The LEAVE L statement must be contained in a loop labeled L. The statement is used to exit that loop.

The GOTO statement in PL/CS is required to branch to a labeled null statement. The label must lexically follow the GOTO so that all loops must be written using explicit loop constructs. If the label L is in a loop body or one of the clauses of an IF or SELECT statement, every GOTO L statement must also be contained therein. This restriction was included in PL/CS to enforce good programming style. However, as will be seen in Chapter 4, the restriction also eases the task of checking proofs.

1.3 An Overview of the Thesis

The thesis is a detailed discussion of the design of the PL/CV language and Proof Checker. The correctness of the overall design of the Proof Checker and several of the algorithms employed are argued.

The PL/CV logic contains rules for the first order predicate calculus, Peano arithmetic, the theory of equality, and the PL/CS programming language. A complete discussion of PL/CV language is beyond the scope of this thesis; interested readers are referred to the PL/CV2 Reference Manual [17]. It is my intent that, except for a few examples in Chapter 4, unfamiliarity with PL/CV will not hinder the reader's comprehension of the material presented.

The thesis was written after the implementation of the Proof Checker was complete. In the course of writing the thesis, I discovered some errors in the Proof Checker and improvements to some of the proof rules.

Because the reason for building the Proof Checker and writing this thesis was to gain knowledge about program verification, not to write and document a production software system, I have occasionally taken the liberty of describing parts of the PL/CV system as they ought to be, instead of as they are. It is my hope that these improvements will eventually find their way into the implemented system.

The Proof Checker is divided into two phases. The first phase of the Proof Checker is the Local Context Checker (LCC), which inputs a proof and processes it in much the same fashion as a compiler. But instead of producing machine code, the LCC produces a series of instructions called V-code. The LCC performs checking that only requires looking at a small window in the proof. The global aspects of the proof are represented by the V-code. The global checking is performed during the second phase, the V-code interpreter, which reads and processes the V-code produced by the LCC.

Because the LCC is so similar to a compiler, it could be designed using well-understood principles [6, 33]. However, unlike a typical compiler, the LCC must manipulate formulas containing quantifiers and bound variables. In Chapter 2, a method of representing quantified formulas is presented that eliminates some technical problems often associated with bound variables and substitution. Bound variables are represented by storing in lieu of a variable name the number of quantifiers along the path from the variable to the binding quantifier. Operations that must commonly be performed on quantified formulas by the LCC can be efficiently applied using this representation. An extension to the representation is proposed for use with higher-order quantification.

Chapter 3 deals with the predicate calculus. PL/CV predicate calculus is proved equivalent to Gentzen's Natural Deduction [64] by giving constructions for translating proofs written in each system to the other system. Next, the syntax of PL/CV predicate calculus and its macro feature are discussed from the standpoint of human engineering. Then the relationship between ALGOL scope rules and the logic is shown. Finally, the way that V-code is used to check predicate calculus proofs is explained.

Chapter 4 is concerned with proofs of programs. It begins with a discussion of what is meant by a proof of a program. Then each of the language constructs of PL/CS is examined. For each construct, the proof rule used in PL/CV is presented and its strengths and weaknesses are discussed. Then the V-code generated for that construct is presented. A general theorem is proved that is used to derive the V-code generated for each of the control constructs. Finally, a generalized control construct called the omnibus statement is proposed as a replacement for all but one of the control constructs in PL/CS. A method is presented by which V-code can be used to check proofs about a restricted version of the omnibus statement.

The V-code interpreter is based on algorithms presented in Chapters 5 and 6. Solutions to two similar problems are given in these chapters. In each case, we are given a set S of pairs of formulas that are known to be equivalent. An equivalence relation $\equiv$ between formulas is defined as follows. $P_1 \equiv P_2$ is defined to hold whenever it is possible to transform P_1 to P_2 by applying to P_1 a sequence of substitutions resulting in P_2, where each substitution replaces an occurrence of some Q_1 by a Q_2 such that (Q_1, Q_2) or (Q_2, Q_1) is a pair in S. The problem is to compute the relation $\equiv$.

In Chapter 5, a solution is given for a particular infinite set S. The problem of computing the equivalence relation defined by this set will be called testing for equivalence by "transformation." Two formulas P_1 and P_2 are tested for this type of equivalence by applying a normalizing function N to each formula; $P_1 \equiv P_2$ if and only if $N[\![P_1]\!] = N[\![P_2]\!]$.

If the set S is finite, the problem of computing the equivalence relation defined by S will be called testing for equivalence by "substitution." A solution to this problem is given in Chapter 6. A graph is maintained in which each node represents a set of formulas. Two formulas are tested for equivalence by determining whether are represented by the same node. An algorithm is presented for making the necessary updates the graph to reflect the addition of a new pair of formulas to S.

How the algorithms presented in Chapters 5 and 6 are combined to form the V-code Interpreter is discussed in Chapter 7. Automatic deductions performed by the Proof Checker are expressed in terms of equivalences. A formula is considered to be proved if it is determined to be equivalent to "true." Transformational reasoning is included by maintaining a hash table that is used to associate normal forms obtained from the normalizing function N with nodes in the graph.

In Chapter 8, the major results of the thesis are summarized. The thesis is concluded with a short essay on the direction I feel future research in program verification should take.

1.4 A Few Words about Notation

Whenever possible, the standard notation of computer science and mathematics is used. Nonstandard notation is used when either:

(1) no standard notation exists, or

(2) the typewriter used to produce this document cannot produce the standard notation.

When one of these cases arises, the syntax of PL/CV, which was designed to cope with even more stringent typographical limitations, is used if possible. Readers who are unfamiliar with PL/CV are requested not to form an opinion of its syntax on the basis of this "sneak preview," but rather to wait until after they have read section 3.6.

The universal quantifier will be written:

```
ALL x. P(x)         (untyped)
ALL x type. P(x)    (typed)
```

and the existential quantifier will be written:

```
SOME x. P(x)        (untyped)
SOME x type. P(x)   (typed) .
```

The following symbols will be used for logical connectives and constants:

~	&	|	=>	<=>	'0'B	'1'B
not	and	or	implication	equivalence	false	true.

The logical connectives have been listed in order of precedence, with the quantifiers and ~ binding tightest, and <=> binding loosest. For example the formula:

```
P => ~ALL x. Q(x) | R(x)
```

is equivalent to:

```
P => ((~ALL x. Q(x)) | R(x))
```

so that the occurrence of x in R(x) is free.

These symbols used in PL/CV for comparisons are ~=, <= and >=. Those symbols are used when writing PL/CV examples and proof rules. But the more standard symbols $\neq$, $\leq$, and $\geq$ are used elsewhere.

The notation P[t//v] stands for the formula P with every free occurrence of the variable v replaced by the expression t, renaming bound variables as necessary to avoid capture. Capture is discussed in Chapter 2.

If x and y are integers, the notation {x ... y} is used to denote the set of all integers greater or equal to x and less than or equal to y. Note that if x > y, then {x ... y} is empty.

If A and B are sets, let A - B denote the set of all x such that x is in A but x is not in B.

The Kleene star * will be used to denote zero or more repetitions. If ρ is a relation, ρ^* denotes the reflexive transitive closure of ρ. That is, ρ^* is the smallest reflexive and transitive relation that is a superset of ρ.

The programming language that will be used to present algorithms is a mixture of ALGOL 68, PASCAL, and English. The three control constructs used:

<u>if</u> B_1 <u>then</u> S_1 <u>elif</u> B_2 <u>then</u> S_2 ... <u>elif</u> B_n <u>then</u> S_n <u>else</u> S_{n+1} <u>fi</u>
<u>while</u> B <u>do</u> S <u>od</u>
<u>repeat</u> S <u>until</u> B

correspond exactly to the SELECT statement, WHILE loop, and UNTIL loop of PL/CS. The ALGOL/PASCAL syntax is used rather than PL/CS syntax to emphasize that the algorithms are being presented for their content and not as examples of PL/CS, and because I feel the ALGOL/PASCAL syntax is aesthetically more pleasing.

Comments in algorithms are enclosed in cent signs. The statement:

v +:= e

is used to increment the variable v by e, and the statement:

v -:= e

is used to decrement the variable v by e.

I suspect that many readers will be surprised by my use of semicolons, and that some will even bo annoyed. When used in PL/I, the semicolon terminates statements, and is put at the ends of lines. When used in ALGOL or PASCAL, the semicolon separates statements and is put at the beginning of lines. If for example, the sequence (S_1 ; S_2 ; S_3) does not fit on one line, it will be written:

```
(  S1
;  S2
;  S3
)
```

with matching parentheses (or keywords) vertically aligned.

The notation r.f will be used to indicate field f of the record r. If p is a pointer variable, the statement:

p.f := x

will executed by replacing the f field of the record to which p points is with the value of x. If p.f appears anywhere else, it denotes the contents of the f field of the record to which p points.

CHAPTER 2

REPRESENTATIONS FOR BOUND VARIABLES

2.1 The Problem of Capture

Because the Local Context Checker is similar to a compiler, it could be constructed using techniques that are well-documented in the literature [6, 33]. But unlike typical compilers, the Proof Checker must create and manipulate representations of quantified formulas. The problems associated with representations for bound variables are discussed in this chapter.

Implicit renaming of bound variables is allowed in PL/CV. For example, if one proves the formula:

```
ALL X FIXED. X*X >= 0;                                          (2.1)
```

and later in the proof, one needs:

```
ALL Y FIXED. Y*Y >= 0;                                          (2.2)
```

as a hypothesis to a rule, formula (2.1) will suffice. Because (2.1) and (2.2) are considered to be essentially the same formula, they ought to have the same representation.

Purists might argue the point, preferring the user to cite rules to derive (2.2) from (2.1). But consider the following example. I think it safe to say that no computer scientist would claim that the following:

```
(X*2) + 3 > 5                                                   (2.3)
X*2 + 3 > 5                                                     (2.4)
```

are different formulas, and insist on rules to insert or delete parentheses. We make a distinction between the marks we put on paper, and the formula, which is really the parse tree that corresponds to those

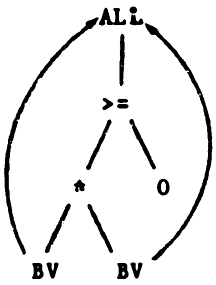


Figure 2.1 Parse graph for ALL Y FIXED. Y*Y >= 0

marks. Since formulas (2.3) and (2.4) have the same parse tree, they are considered to be the same formula.

I propose that the situation is the same for quantified formulas. The only difference is that instead of a parse tree, a quantified formula has a parse graph. Figure 2.1 contains the parse graph for formulas (2.1) and (2.2). In a parse graph, all bound variables are represented by a special symbol, BV. Every occurrence of BV in the graph is connected by an edge to the quantifier to which the represented variable is bound. The name of the bound variable is completely abstracted away. Since (2.1) and (2.2) both have the same parse graph, they are considered to be the same formula.

All of this may seem like a lot of trouble to go through just to allow a user to change the names of his bound variables at will. But there is a reason (other than making proofs hard to understand) for renaming bound variables. Suppose a user creates a global lemma file, containing a number of useful theorems, such the following:

ALL X FIXED. (PRIME(X) => SOME Y FIXED. (PRIME(Y) & Y>X)) (2.5)

which states that given any prime number, there is always a larger one. The ALL elimination rule allows one to instantiate this theorem, substituting any FIXED value for X, such as:

PRIME(2) => SOME Y FIXED. (PRIME(Y) & Y>2)

But consider what happens when we attempt to instantiate formula (2.5) with the free variable Y:

PRIME(Y) => SOME Y FIXED. (PRIME(Y) & Y>Y) (2.6)

Renaming the bound version of Y for the sake of clarity, we get:

PRIME(Y) => SOME Z FIXED. (PRIME(Z) & Z>Z)

Since there can be no Z such that Z>Z, we can conclude from the above that Y is not prime. But this argument is clearly absurd; something has gone awry. The problem is that when we substituted Y for X within the scope of the quantifier SOME Y FIXED, the meaning of Y was changed. Logicians call this phenomenon "capture." The way to avoid capture is by renaming bound variables when their names would conflict with free variables being substituted. Before we can substitute the free variable Y for X in (2.5), we must first rename the bound variable Y. The resulting formula is then:

PRIME(Y) => SOME Z FIXED. (PRIME(Z) & Z>Y) (2.7)

which is perfectly fine.

Capture can only occur when the same identifier is being used as both a bound and free variable. A sledgehammer approach to avoiding capture is to disallow the reuse of variables. Although this approach has the advantage of being conceptually simple, in practice it would be quite unpleasant for the user. Before undertaking the writing of a large proof, he would be required to inspect his lemma file, find all the lemmas he expected to use in the proof, and make a list of variables bound in those lemmas, which he therefore could not use in his proof. If one is designing a logic for the purpose of proving metamathematical results, one can

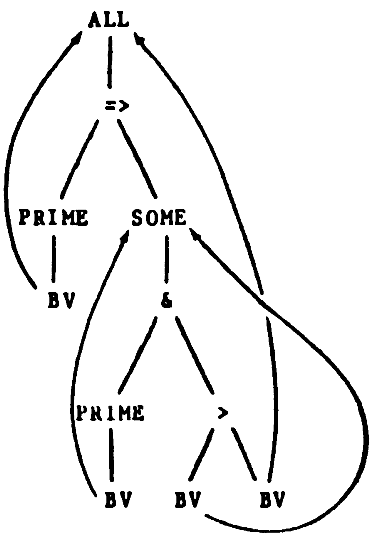


Figure 2.2 Formula graph for (2.5)

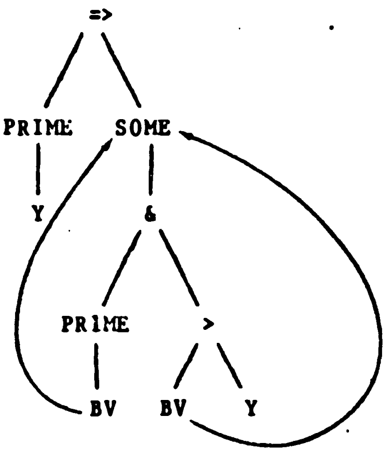


Figure 2.3 An instantiated parse graph

make such harsh restrictions, but not if one is designing a logic in which people are expected to write proofs.

There is a cleaner way of avoiding the problem of capture. If we take the point of view that substitution is an operation performed on parse trees and parse graphs, and not text, the problem of capture evaporates. When one substitutes X+2 for Y in Y*2>Z, the proper result is (X+2)*2>Z, not X+2*2>Z, even though simple textual substitution would yield the latter formula. Likewise, if one substitutes X+2 for Y in Y1<Y, the proper result is Y1<X+2, not X+21<X+2. One is actually performing the substitution on trees, not formulas, and creating a new tree T. What is written as the "result" of the substitution is any formula whose parse tree is T.

The process of instantiating a parse graph is quite similar. The first step is to delete the quantifier being instantiated, which will always be the root of the graph. The deletion will leave edges extending from the formerly bound variables dangling. These variables are replaced in the graph by whatever term is being used to instantiate the quantifier, yielding a new graph G. The "result" of the substitution is any formula whose parse graph is G. For example, Figure 2.2 contains the parse graph for formula (2.5), and Figure 2.3 contains the result of instantiating that graph with Y. Formula (2.7) has Figure 2.3 as its graph, but formula (2.6) does not. Thus, formula (2.7), and not formula (2.6) is considered to be the result of instantiating formula (2.5) with Y. Since the names of bound variables do not appear in the parse graphs, there is no possibility of a free variable being captured.

2.2 Representations of Parse Graphs

Let us now turn to the problem of representing parse graphs in the memory of a computer. Two different methods for representing formulas are used in the Proof Checker both extensions of methods for representing trees described by Knuth [54]. The first method uses a LISP-like data structure, and the second uses a string representation, so-called Polish notation. A preorder traversal is used in conjunction with the string representation, but the following discussion applies equally well to a postorder traversal.

The problem of extending parse tree representations to parse graph representations is this: How are the back edges connecting bound variables to their corresponding quantifiers to be represented? In the case of the LISP-like representation, an obvious suggestion is to put in the node representing the bound variable a pointer back to the node representing the quantifier. This is, in fact, the method used in the Proof Checker, though we shall later see there is a better method.

For now, let us concentrate on the string representation. The prefix string representation for parse trees (less some grubby details) is as follows. If the main connective of an expression or formula is a function or operator ρ with n operands P_1, P_2, ... P_n, then the expression is represented in consecutive bytes of computer memory by the following string:

ρ n R_1 R_2 ... R_n

where for each i in {1 ... n}, R_i is the prefix string representation of P_i. Variables and constants are represented as operators with zero operands. For example, the expression A*B+C would be represented:

plus 2 times 2 free-var-A 0 free-var-B 0 free-var-C 0

In the implementation, suitable byte encodings replace the names "plus," "times," "free-var-A," etc.

This representation has the following desirable properties, which the extended representation should have as well:

(1) Location independence. If a prefix string is copied from one area of memory to another area of memory, the copy is a valid representation of the expression represented by the original.

(2) Ease of instantiation. The operation "substitute the expression f for every occurrence of the variable v in the expression e" can be carried out efficiently, using a simple pattern matching algorithm.

(3) Ease of combination and decomposition. The expression f(x, y), for example, is represented by f 2 Rx Ry if and only if the expressions x and y are represented by Rx and Ry, respectively.

One difficulty in finding a representation of quantified formulas is ensuring the extended representation have property (2). Because PL/CV allows a limited form of second-order substitution, it is necessary to properly handle the case in which a quantified formula is substituted for some variable of another quantified formula. For example, when the ALL elimination rule is applied to the the formula:

ALL P BIT. ALL Q BIT. (P & Q => P)

P may be instantiated by any term of type BIT, including a quantified formula.

The basic strategy used to represent quantified formulas is to treat quantifiers as unary operators, and bound variables as nullary operators. But encoded in the name of the operators used for bound variables will be information sufficient to locate the representation of the quantifier to which the variable is bound.

By the discussion in section 2.1, the name of the bound variable need not, and should not, appear anywhere in the representation. However, the type of the bound variable must appear as part of the corresponding quantifier name. If type information were not included in the representation, one could prove the formula:

SOME (T, F) BIT. ALL B BIT. (B=T | B=F)

and later use it as:

SOME (J, K) FIXED. ALL X FIXED. (X=J | X=K)

since both these formulas would have the same representation.

In a language where each operand of each operator is required to have a specific type, and there are no empty types, all type information could be discarded in the representation of quantified formulas, since the type of any bound variable, and hence the corresponding quantifier, could be deduced if necessary. (In PL/CV, equality is the only operator that does not have this strong-typing property.) But the type of a quantifier can be deduced from the type of its bound variable only if the bound variable appears somewhere in the formula. Normally a quantifier for which there are no corresponding bound variables, such as the quantifier in:

ALL X FIXED. 2+2 = 4 (2.8)

adds nothing substantial to the meaning of the formula. There is no harm done if formula (2.8) and:

ALL B BIT. 2+2 = 4 (2.9)

have the same representation, since formulas (2.8) and (2.9) are both equivalent to the formula 2+2 = 4. However, in a the language that has a type EMPTY containing no elements, one could prove the assertion:

ALL E EMPTY. '0'B

which is vacuously true, since it is impossible to instantiate. One could then mistake the formula for:

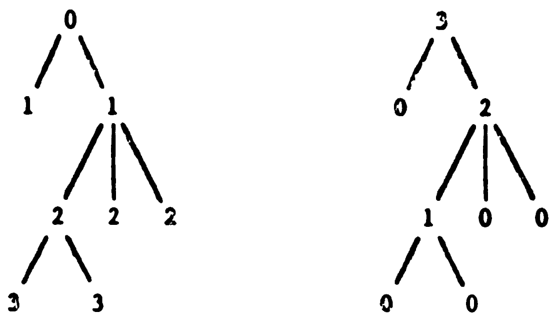
ALL J FIXED. 'O'B

which could be instantiated to get a contradiction. Therefore, the optimization of omitting type information from representations cannot be used in languages with empty types.

Let us consider various proposals for what can be put in the representation of bound variables that will determine to which quantifier the variable is bound, and still maintain the above three properties. The simplest suggestion is to use what was proposed for the LISP-like representation method: put in the representation of bound variables the address of the the corresponding quantifier. However, since this method does not have property (1), we might try to amend it by using a relative offset instead of a pointer. That is, put in the representation of the bound variable the number of bytes by which it follows the binding quantifier. But this method does not have property (2). Any time a free variable that occurs between a quantifier and one if its bound variables is instantiated, the offset stored in the bound variable would have to be changed. Thus, the instantiation process would involve locating and changing other bound variables.

There is another class of solutions that use the structure of the parse graph, which is a tree with back pointers. Since all the variables bound to a quantifier must occur in the subtree whose root is the quantifier. if the level at which the binding quantifier appears in the tree is stored with the bound variable, the quantifier to which the variable is bound is uniquely determined.

But "level" must be defined. There are two ways in which level numbers can be assigned to the nodes of trees: top-down and bottom-up. In top-down numbering, the root of the tree is assigned level 0, and the



(a) (b)

Figure 2.4 Top-down (a) and bottom-up (b) level numbering

level assigned to a nonroot node is one plus the level of its parent. In bottom-up numbering, each leaf node is assigned level 0, and the level assigned to a nonleaf node is one plus the largest level assigned to any of its offspring. Figure 2.4 contains examples of these two kinds of level numberings.

Unfortunately, putting the level number of a quantifier in a bound variable is not satisfactory, for either definition of "level." The problem with using top-down numbering is that the resulting representation does not have property (3). For example, when the formula that is represented by f 2 Rx Ry is broken down into the two formulas Rx and Ry, if either Rx or Ry contains any quantifiers, all the level numbers change.

Although a representation using bottom-up numbering has property (3), it does not have property (2). For example, consider the formula:

ALL Q BIT. (P & Q => P)

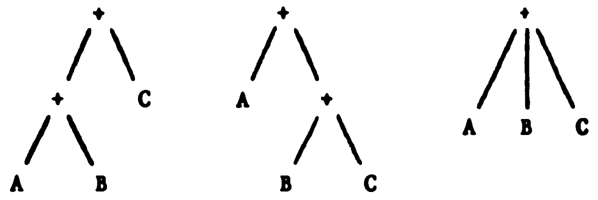
which, using bottom-up numbering, would be represented:

all-bit 1 implies 2 and 2 free-var-P 0 bound-var-3 0 free-var-P 0

The 3 in bound-var-3 refers to the level of the quantifier. The problem is that when a formula is substituted for free-var-P, the level number of the left offspring of the & is increased, which can change the level number of the the & node and the quantifier node. Therefore, the number stored with the bound variable must be changed.

There is an acceptable variation on top-down level numbering. Rather than storing in the bound variable representation the level of the corresponding quantifier, one stores the difference between the level of the variable and the level of the quantifier. Since this difference is is the distance in the tree between the bound variable and the quantifier, the quantifier to which the variable is bound can be uniquely determined by counting. This representation has all three of the desired properties. Because no machine addresses are used, property (1) obtains. All of the nodes on the path in the tree from the variable node to the corresponding quantifier node must be function nodes and quantifier nodes. When a formula or expression is substituted for a free variable, the distance from any bound variable to its corresponding quantifier does not change, since there are no variables on the path from the bound varinble to the quantifier. Therefore, property (2) obtains. Likewise, when trees are combined and broken apart, distances in the subtrees between bound variables and corresponding quantifiers do not change, so that property (3) obtains.

A modification of the above representation system is used by the Proof Checker. Rather than storing in the bound variable representation the length of the path from the variable to its binding quantifier, the number of quantifiers (not counting the binding quantifier) on that path is stored. The correspondence between bound variables and quantifiers can still determined by counting quantifier nodes. This system, which will henceforth be referred to as "quantifier count" representation, has all three of the desired properties, but it also has one more.



(a) (b) (c)

Figure 2.5 Unnormalized (a), (b) and normalized (c) parse trees

Another operation that is performed on formulas is normalization. For example, the expression (A+B)+C is parsed as in Figure 2.5a, whereas the expression A+(B+C) is parsed as in Figure 2.5b. As is discussed in Chapter 5, these two expressions are deduced by the Proof Checker to be equal by normalizing them both to the tree shown in Figure 2.5c. But now consider what happens when the formula:

SOME A FIXED. (A+B)+C = 0

is normalized. The normalization operation changes the length of the path from the bound variable A to its quantifier, but not the number of quantifiers (zero) along the path. Since none of the normalizations performed by the proof checker alter the quantifier structure of a formula, these operations can be performed on quantifier count representations without having to update the bound variable representations.

Quantifier count representation can also be profitably used with the LISP-like representation. There is a certain elegance in representing a bound variable by a pointer to the corresponding quantifier, but it complicates some programming tasks. The problem with this representation is that it creates circular list structures, which can be annoying, especially in programming environments that employ garbage collection. To copy a circular structure, one needs to use "list-marking" algorithms [66], which are more complicated than the straightforward recursive algorithm used to copy trees. For these reasons, to represent the link between bound variables and quantifiers, quantifier counts are preferable to pointers.

2.3 Creating Quantifier Count Representations

It is easy to create a parser that builds quantifier count representations. Because of the nature of proof blocks in the PL/CV language, a block-structured symbol table of the type described by Gries [36] is required. The symbol table routines have the following entry points:

(1) BlockOpen. Increase the current block level by one, creating a new block.

(2) Declare(I). Create a new variable record in the current block, and associate it with the identifier I.

(3) BlockClose. Decreases the current block level by one, discarding any variables and associations created in the block being closed.

When parsing a formula, the number of quantifiers along the path from the element currently being processed and the root of the parse tree is maintained in a global variable called QuantifierLevel. This variable is initialized to zero, increased by one when the scope of quantifier is entered, and decreased by one when the scope is left.

Here is how a quantified formula of the form:

ALL V_1, V_2, ... V_n type . formula

is parsed. A recursive descent model of parsing [35] is assumed. When the ALL is encountered, a new block is created by calling BlockOpen. Then Declare is called for each identifier V_i, creating variables local to the newly created block. The variables are marked in the symbol table as being bound variables, and QuantifierLevel+i is stored in the variable record for each V_i. Then QuantifierLevel is increased by n, the formula is recursively parsed, QuantifierLevel is decreased by n, and BlockClose is called.

When an identifier is encountered during parsing, it is looked up in the symbol table to get the variable with which it is currently associated. In the case of a bound variable, the number of quantifiers on the path from the variable to the quantifier Q to which it is bound should be put into whatever representation the parser is building. The symbol table record for the bound variable contains the number N of quantifiers on the path from the root to Q (including Q). The variable QuantifierLevel contains the number of quantifiers on the path from the root to the bound variable. Therefore, the number that goes in the representation of the bound variables is QuantifierLevel - N.

A similar algorithm is used for instantiating quantified formulas. Actually, instantiation is a two-step process. Suppose, for example, in the formula:

```
ALL P BIT. ALL Q BIT. (P & Q => P)
```

one wished to instantiate the variable P with the formula X > 0. The first step is to delete the quantifier:

```
ALL Q BIT. (P & Q => P)
```

transforming P from a bound variable to a free variable. The second step is to substitute X > 0 for every occurrence of P. Notice that these operations must be performed on internal representations. By property (2), the second step can be performed easily.

When working with external representations, the first step appears

```
( integer QuantifierLevel := 0
        . i := beginning of the subtree to be searched

; procedure Bfind():
  begin integer n := 1
  :     while n ≠ 0
        do  if   R[i].optr is a quantifier
            then i +:= 1
            ;    QuantifierLevel +:= 1
            ;    Bfind()
            ;    QuantifierLevel -:= 1
            ;    n -:= 1
            elif R[i].optr is a bound variable
            then if QuantifierLevel ≤ the level number
                    encoded in the name of the variable
                 then the variable is dangling
                 fi
            ;    i +:= 1
            ;    n -:= 1
            else n +:= R[i].order - 1
            ;    i +:= 1
            fi
        od
  end

; Bfind()
)
```

Figure 2.6 Algorithm D: The location of dangling variables

completely trivial. But performing the operation on a formula in quantifier count representation requires a bit of processing. After deleting the leading quantifier from the representation, all the bound variables which formerly referred to the quantifier must be located and changed. One could change each located bound variable to a new free variable that does not occur anywhere in the formula, and then substitute the representation of X > 0 for each occurrence of that free variable, but it is simpler to combine these two operations, and substitute X > 0 for each bound variable as it is located.

Thus, the problem of instantiation is reduced to that of locating each bound variable associated with a particular quantifier. Algorithm D, shown in Figure 2.6, will accomplish this task. Recall that the string representation of a formula consists of a sequence of pairs. We will assume that the array R contains such a sequence. Each element R[i] has two fields: "optr," the name of the represented operator, and "order," the number of operands the operator takes. Let T be a subtree of the representation in R to be searched. Some bound variables in T will be bound to quantifiers within T; others will be bound to quantifiers outside of T. These latter variables will be called "dangling" variables, because when viewing just the subtree T, these variables are not bound to anything. Algorithm D locates all the dangling variables in T. Given a formula of the form:

ALL X FIXED. T

in which the main connective is a quantifier, if algorithm D is applied to the subformula T, the dangling variables located will be precisely all the occurrences of the bound variable X in T.

Routine Bfind assumes on entry that i points to the beginning of a subtree S of T. Bfind moves i just past S, locating all the dangling variables in S.

An invariant of the while loop in Bfind is that the index i must be moved past n operands before the end of S is reached. There are three cases, depending on what R[i].optr is. If the operator is a quantifier, i is moved past the quantifier, and Bfind is called recursively to move i past its operand. The result of these two operations is to move i past one complete operand, so that n is decreased by one. Likewise, if R[i].optr is a bound variable, i is moved past it, and n is decreased by one. Finally, consider the case in which R[i].optr is some other operator. Then i points to the beginning of n operands, the first of which is an operator with m (where m ≥ 0) operands:

$\langle f,m \rangle\ P_1\ P_2\ \ldots\ P_m\ Q_2\ Q_3\ \ldots\ Q_n$

where <f,m> is an optr-order pair, and <f,m> through P_m make up Q_1. Once i is moved past <f,m>, then there are m+n-1 operands left to be moved past, so that to maintain the loop invariant, n must be increased by R[i].order-1.

The function of the variable QuantifierLevel is the same in this algorithm as in the parsing algorithm. QuantifierLevel contains the number of quantifiers that are on the path from the root of the tree to R[i] and also contained in the subtree T. Whenever we encounter a bound variable whose level number is greater to or equal QuantifierLevel, the quantifier to which it is bound must occur outside the subtree T, and hence that variable is dangling.

Algorithm D must be recursive to locate the ends of the operands of each quantifier, so that the variable QuantifierLevel can be decremented in the appropriate place. If the representation of formulas is modified slightly, a nonrecursive algorithm can be used. Recall that in the standard string representation, the formula ALL X FIXED. P(X) is represented:

all-fixed 1 R_P

where R_P is the representation of P(X). In the modified system, the formula would be represented:

```
( integer QuantifierLevel := 0
        , i := the beginning of the subtree to be searched
; while QuantifierLevel ≥ 0
  do  if  R[i].optr is a quantifier
      then QuantifierLevel +:= 1
      elif R[i].optr is "end-scope"
      then QuantifierLevel -:= 1
      elif R[i].optr is a bound variable
      then if QuantifierLevel ≥ the level number
              encoded in the name of the variable
          then the variable is dangling
          fi
      fi
  ;   i +:= 1
  od
)
```

Figure 2.7 Algorithm D': a simplified algorithm

all-fixed 1 R_p end-scope 0

appending a special marker to delimit the end of the scope of the quantifier. The 0 after the marker has no significance; it is present only to preserve the rule that a string representation consists of an alternating sequence of operators and numbers. A string in the modified representation can be converted to the standard representation simply by deleting all the "end-scope 0" markers. Likewise, a string in the standard representation can be converted to the modified representation using an algorithm similar to Algorithm D. The advantage of the modified representation is that the simple "parenthesis counting" algorithm D', shown in Figure 2.7, can be used to locate dangling variables. For algorithm D' to work, the subtree being searched must be followed by an end-scope marker. The operand of quantifier will always will always be followed by an end-scope marker, but an arbitrary subtree cannot be searched. Another disadvantage is that the modified representation takes more space than standard quantifier count representation.

2.4 Representations for Second-Order Logic

The PL/CV3 [23] system will be extended to include second-order and higher-order formulas. The new complexity introduced in higher-order logics is quantification over functions and predicates. For example, integer induction can be expressed by the following second-order formula:

```
ALL P(FIXED) BIT.
     (  P(0) & ALL X FIXED. (X>=0 & P(X) => P(X+1))                (2.10)
     => ALL X FIXED. (X>=0 => P(X))
     )  .
```

Formula (2.10) can be instantiated with any predicate that takes a FIXED value and yields a BIT. Predicates may be written using lambda notation. For example, the predicate "is greater than 5" would be written:

```
LAMBDA Y FIXED. Y > 5                                                (2.11)
```

Formula (2.10) can be instantiated with predicate (2.11) to yield:

```
(  ( (LAMBDA Y FIXED. Y>5)(0)
   & ALL X FIXED. (  X>=0 & (LAMBDA Y FIXED. Y>5)(X)
                  => (LAMBDA Y FIXED. Y>5)(X+1)
   )              )
=> ALL X FIXED. (X>=0 => (LAMBDA Y FIXED. Y>5)(X))
)
```

which, in turn can be simplified to:

```
(  0>5 & ALL X FIXED. (X>=0 & X>5 => X+1>5)
=> ALL X FIXED. (X>=0 => X>5)
)
```

Of course, lambda expressions may themselves contain quantifiers and other lambda expressions. For example, the predicate "is even" can be written:

LAMBDA Y FIXED. SOME M FIXED. 2*M = Y

The capture situation is more complicated in higher-order logics than in first-order logic. For example, consider the following valid second-order formula:

ALL P(FIXED) BIT. ((ALL X FIXED. P(X)) => ALL X FIXED. P(X+C)) (2.12)

which states that if a predicate holds for all integers, the predicate can be "shifted" by a constant factor C and still remain valid. Here is an example of capture that is analogous to the capture that occurs in first-order logic. The predicate:

LAMBDA Y FIXED. (X~=0 | Y~=C) (2.13)

has two free variables, X and C, and is true for all Y if X ~= 0. But when formula (2.12), is instantiated by predicate (2.13), the result, when simplified, is:

ALL X FIXED. (X~=0 | X~=C) => ALL X FIXED. (X~=0 | X+C~=C)

which is not valid if C ~= 0. The problem here is that the variable X, which is free in (2.13), is being captured (twice) by quantifiers in (2.12).

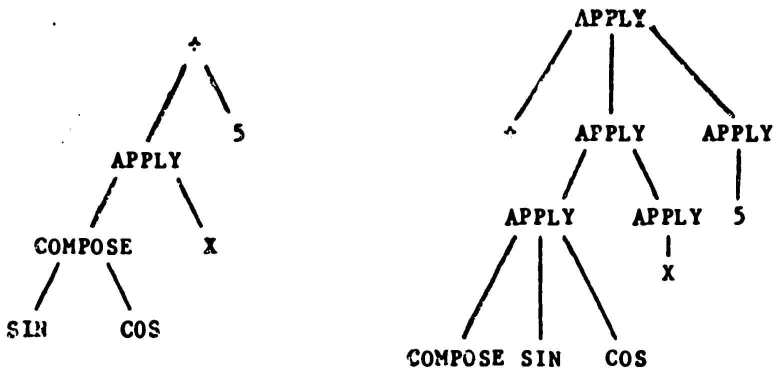
Once we begin instantiating second-order formulas with lambda expressions containing quantifiers, new types of capture emerge. For example, if (2.12) is instantiated with the formula:

LAMBDA Y FIXED. SOME C FIXED. C<Y (2.14)

the result, when simplified, is:

ALL X FIXED. SOME C FIXED. C<X => ALL X FIXED. SOME C FIXED. C<X+C

an invalid formula. Here the variable C, which is free in (2.12), is



(a) (b)

Figure 2.8 Two possible representations for expression (2.16)

being captured when the argument X+C of P in (2.12) is substituted for Y in (2.14). A third form of capture occurs when (2.12) is instantiated with the lambda expression:

LAMBDA Y FIXED. SOME X FIXED. X=Y (2.15)

The result is:

ALL X FIXED. SOME X FIXED. X=X => ALL X FIXED. SOME X FIXED. X=X+C .

an invalid formula. In the process of argument substitution, the variable X, which is bound to quantifiers in (2.12), is being stolen by quantifiers in (2.15).

As it was with first-order logic, the solution to these problems is to view substitution as a process that is applied to parse graphs, not textual formulas. However, a problem encountered when dealing with the representation of higher-order formulas is what to do with functions and predicates that are not simple variables, as in the expression:

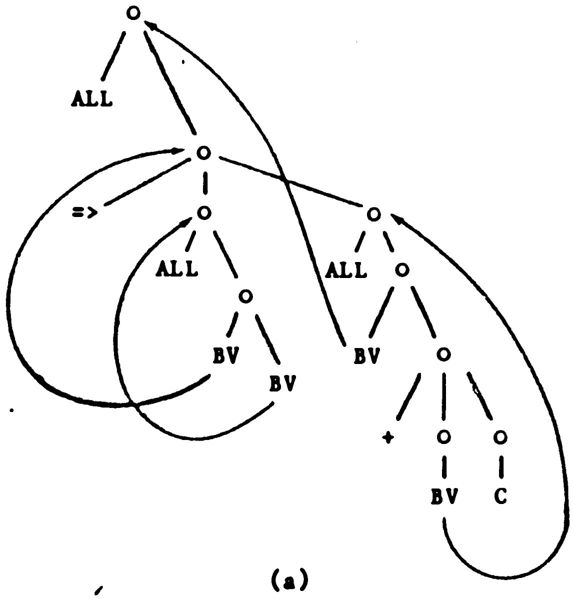
COMPOSE(SIN, COS)(X) + 5 (2.16)

which indicates that the function obtained when SIN and COS are composed is applied to X, and the result increased by 5. The tree corresponding to this expression must contain the special node "APPLY." Two ways of using such a node are shown in Figure 2.8. The tree shown in Figure 2.8a is the more traditional method; APPLY is used only to invoke functional expressions on arguments. I feel that the method shown in Figure 2.8b is more elegant. The APPLY operator is used uniformly to invoke all functions on their arguments.

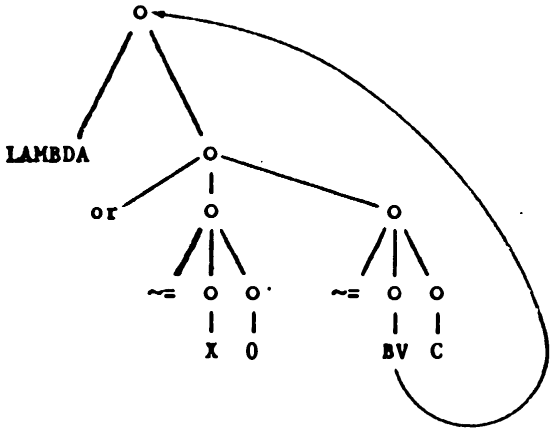
At first it may seem strange to see APPLY used on 5 and X. But recall that constants and variables are being represented as functions with zero arguments; APPLY must be used to indicate that the values 5 and X, as opposed to 5 and X themselves, are desired. There is precedent for this notion in the programming language Russell [12], in which, for example, 5 actually is a function whose domain is a universal value space, and whose value is an element of that space used to represent the fifth integer. When a Russell programmer writes "5", he is using an abbreviation for "5()," the function "5" applied to zero arguments.

Looking at Figure 2.8, the second representation appears to use more space than the first. But things are not as bad as they seem. In the second representation, every interior node must be an APPLY; therefore the name of the node can be omitted from all the interior nodes. In the discussion that follows, the use of the second representation will be assumed. This representation can easily be converted to the first representation by finding and eliminating APPLYs whose first arguments are leaves.

The techniques for representing bound variables in first-order logic extend nicely to high-order logics. Each variable bound to a quantifier is represented by the symbol BV and linked to the quantifier operator by a



(a)



(b)

Figure 2.9 Graphs for (a) formula (2.12), and (b) expression (2.13)

back edge. Variables bound to LAMBDAs are treated in exactly the same fashion. Quantifiers and LAMBDAs will be generically referred to as "binding operators." When using quantifier count representation, stored with the bound variable is the number of binding operators (to be precise, the number of APPLY nodes whose first operand is SOME, ALL, or LAMBDA) on the

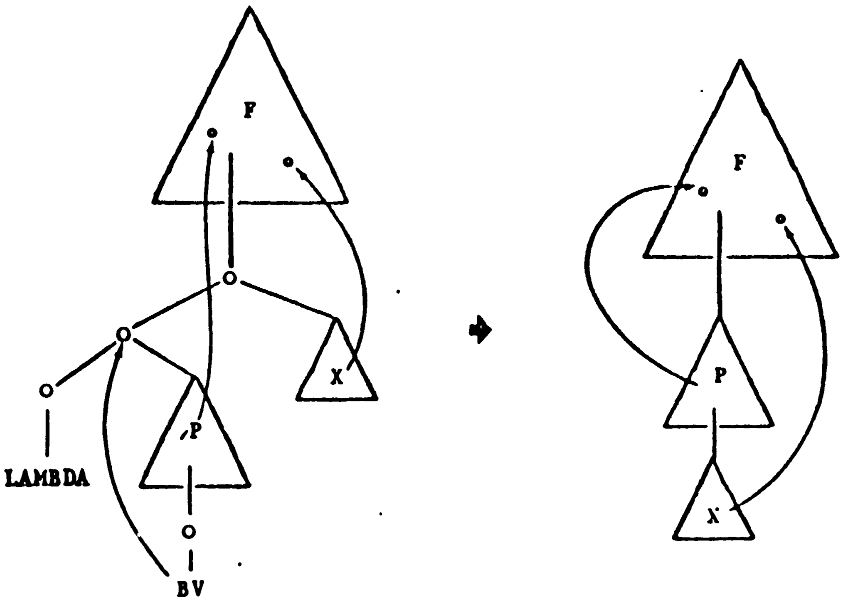


Figure 2.10 A LAMBDA reduction

path from the bound variable to (but not counting) the operator to which it is bound. Figure 2.9 shows the representation for formula (2.12) and expression (2.13). For brevity, o is used instead of APPLY.

The process of substituting a lambda expression for a bound variable is identical to the instantiation process used with first-order logic; LAMBDAs behave just like quantifiers, and there are no capture problems.

However, care must be taken when performing lambda reductions. For example, performing a lambda reduction on:

(LAMBDA X FIXED. P(X)) (T) (2.17)

is a matter of deleting the initial LAMBDA and substituting T for X in P, yielding P(T). This process is the analogous to of the instantiation of a quantifier, if the LAMBDA is the main connective of the formula. The problem arises when (2.17) appears within some other formula, and T contains bound variables.

The substitution process is diagrammed in Figure 2.10. (For the sake of clarity, the case of a LAMBDA expression with only one bound variable is discussed; the methods generalize to multi-argument functions and predicates.) F is the part of the formula above the lambda expression being reduced. The problem is this: any bound variables in P and X that are linked back to binding operators in F must remain linked to the same operators after the substitution is performed. Also, when a copy of X is substituted into P (copies may need to be made when the variable bound to the LAMBDA appears more than once in P), variables in X that are bound to operators within X itself must, in the copy, be linked to binding operators in the copy of X, not to the operators in the original.

If the back links are represented by pointers, programming this substitution operation is likely to be tricky. The problem of copying circular structures is still with us, and determining which pointers in X point to nodes within X and which point to nodes outside X looks particularly messy.

However, the operation is not too difficult to program when quantifier count representation is used. First let us consider what must be done to P. First, all the variables that are bound to the lambda must be located. If Algorithm D (modified to treat LAMBDAs as quantifiers) is applied to P, all the dangling variables (that is, variables bound outside P) will be located. But not every dangling variable in P is necessarily bound to the LAMBDA; some may be bound to operators in F. If N is the level number stored in a variable representation, Algorithm D tests whether the variable is dangling by checking if QuantifierLevel ≤ N. In fact, if QuantifierLevel = N, then the variable is bound to the LAMBDA, and if QuantifierLevel < N, then the variable is bound to an operator in F.

The variables bound to the LAMBDA have X substituted for them. Each of the other dangling variables has in its representation the number of binding operators on the path from it to the operator to which it is bound. But after the reduction is performed, there will be one fewer binding operator on this path because the LAMBDA that was eliminated was on this path. Therefore, the count stored with each of these variables must be reduced by one. Variables bound to operators within P are unaffected.

Next, consider what must be done when a copy of X is substituted for a variable V_P that is bound to the LAMBDA. Algorithm D must be used on X to locate every dangling variable V_X contained in X. The count stored with V_X is the number of binding operators on the path from V_X to the operator in P to which it is bound. When a copy of X is substituted for V_P, this path is lengthened; it now goes through P. Thus, the number N of binding operators on the path through P from V_P to the root of P (not counting the LAMBDA at the root; it is being deleted) must be added to the count stored with V_X. But N is precisely the number stored with V_P. Note that the number N that is added to the count stored with each dangling variable in X depends on the location within P of V_P. If more than one variable in P is bound to the LAMBDA, a new N must be used for each such variable. Variables in X that are bound to operators in X need not be changed.

In summary, when a lambda reduction is performed, the following modifications must be made to the bound variables in P and X. Every variable in P that is bound to an operator in F must have its count reduced by one. For every variable in V_P that is bound to the LAMBDA, a copy Y is made of X, every dangling variable in Y has its count increased by the count stored with V_P, and Y is substituted for V_P. The various copying and substitution operations can be combined with algorithm B, which executes in linear time, so that the entire operation can be performed efficiently.

CHAPTER 3
THE PREDICATE CALCULUS

3.1 Introduction

Although PL/CV was developed primarily as a tool for reasoning about programs, a facility for reasoning about the predicate calculus is a necessary component. The predicate calculus portion of PL/CV has been quite successful in its own right.

In this chapter three related systems for writing predicate calculus proofs are discussed, and the relationships among them made clear. The first system, called "Natural Deduction," [64] was originally devised by Gerhard Gentzen. In their book _A Programming Logic_, Constable and O'Donnell [19] present a logic that is derived from Gentzen's Natural Deduction. In this chapter, Constable and O'Donnell's logic will be referred to as APL/CV (APL being an acronym for _A Programming Logic_), to differentiate it from PL/CV, the language processed by the PL/CV Proof Checker.

First, the equivalence of Natural Deduction and APL/CV is shown by giving constructions for converting proofs written in each system to proofs in the other system. Next, the modifications that were made to APL/CV to get PL/CV are discussed. Many of the design decisions made concerning the PL/CV language were heavily influenced by the design of the Proof Checker. Therefore, the presentations of the two designs have been interleaved.

3.2 Natural Deduction

Natural Deduction is a formal logic proposed by Gentzen to mimic informal mathematical reasoning. The presentation given here has been adapted (the notation has been changed somewhat) from one given by Prawitz [64].

A Natural Deduction proof takes the form of a tree whose nodes are formulas. The formula at the root is being proved, and the formulas at the leaves are assumptions.

Each of the nonleaf nodes must be an instance of one of the rules of inference in Figure 3.1. The following notation is used to describe the rules. The letters P, Q, and R represent formulas. The letter x represents a bound variable, the letter v represents a free variable, and the letter t represents a term. The notation P[t//v] means "first rename the bound variables in P as necessary to avoid capture, and then substitute t for every occurrence of v in P." Capture was discussed in Chapter 2. (Prawitz avoids capture by insisting that the set of bound variables be distinct from the set of free variables.)

The form of the rules in Figure 3.1 is:

$$\frac{H_1 \; H_2 \; \ldots \; H_n}{C}$$

where H_1, H_2, ... H_n are called the hypotheses, and C is called the conclusion. Intuitively, the rule means that the conclusion follows logically from the hypotheses. In the proof tree, every node C that is not a leaf must be the conclusion of a rule of inference, and the offspring of C must be the the corresponding hypotheses.

In some of the rules, one or more of the hypotheses have above them a formula enclosed in parentheses. The application of these rules are said to "discharge" the formulas in parentheses. As an example, consider

$$\frac{P \quad Q}{P \,\&\, Q} \qquad \frac{P \,\&\, Q}{P} \qquad \frac{P \,\&\, Q}{Q} \qquad (1,2,3)$$

$$\frac{P}{P \mid Q} \qquad \frac{Q}{P \mid Q} \qquad \frac{P \mid Q \quad \begin{matrix}(P)\\ R\end{matrix} \quad \begin{matrix}(Q)\\ R\end{matrix}}{R} \qquad (4,5,6)$$

$$\frac{\begin{matrix}(P)\\ Q\end{matrix}}{P \Rightarrow Q} \qquad \frac{P \quad P \Rightarrow Q}{Q} \qquad (7,8)$$

$$\frac{\begin{matrix}(P)\\ \text{'0'B}\end{matrix}}{\sim P} \qquad \frac{P \quad \sim P}{\text{'0'B}} \qquad (9,10)$$

$$\frac{P[v//x]}{\text{ALL } x.\ P} \qquad \frac{\text{ALL } x.\ P}{P[t//x]} \qquad (11,12)$$

$$\frac{P[t//x]}{\text{SOME } x.\ P} \qquad \frac{\text{SOME } x.\ P \quad \begin{matrix}(P[v//x])\\ Q\end{matrix}}{Q} \qquad (13,14)$$

$$\frac{\text{'0'B}}{P} \qquad \frac{\begin{matrix}(\sim P)\\ \text{'0'B}\end{matrix}}{P} \qquad (15,16)$$

Figure 3.1 Gentzen's rules of inference

rule (14). An instance of this rule will be a formula Q whose offspring are the formulas SOME x. P and Q. The formula P[v//x] above the hypothesis Q indicates that everywhere P[v//x] appears as an assumption in the subtree whose root is Q, the assumption is discharged.

The set of the assumptions in a subtree T of the proof that are not discharged by instances of rules in T itself is referred to as the "undischarged assumptions of T." A complete proof is not allowed to have any undischarged assumptions.

Rules (1-14) are organized as "Introduction" and "Elimination" rules for the connectives &, |, =>, ~, ALL, and SOME. In Figure 3.1, the Introduction rules appear on the left, and the Elimination rules appear on the right. In an Introduction rule for a connective ρ, the main connective of the conclusion is ρ. Conversely, one of the hypotheses to an Elimination rule for a connective ρ has ρ as its main connective.

Rule (16), the Rule of Contradiction, is a more powerful version of rule (15). The reason for having both rules is that in intuitionistic logic, rule (16) is disallowed, but rule (15) may be used. In classical logic, all the rules may be used, and rule (15) is redundant.

There are some important restrictions on the use of two of the rules. In rule (11), ALL Introduction, v must not occur in any undischarged assumption of the subtree whose root is P. In rule (14), SOME Elimination, v must not occur in SOME x. P, in Q, or in any undischarged assumption of the subtree whose root is the hypothesis Q.

An example of a Natural Deduction proof appears in Figure 3.2. Each of the nodes is followed by some annotation in braces. Assumptions are annotated with small letters. Nonleaf nodes are annotated with the number of the rule used, and any assumptions discharged.

3.3 APL/CV

The presentation of APL/CV given here has been adapted from *A Programming Logic* [19]. The logic has been modified slightly to simplify the equivalence proof. The most significant change is that Constable and O'Donnell's logic has typed quantifiers, and the logic presented here has

```
          ((P|Q)=>R)=>((P=>R)&(Q=>R) {7bd}
                        |
               (P=>R)&(Q=>R) {1}
              /                 \
      P=>R {7a}                  Q=>R {7c}
          |                        |
        R {8}                    R {8}
        /   \                    /   \
 P|Q {4}   P|Q=>R {b}     P|Q {5}   P|Q=>R {d}
   |                        |
  P {a}                    Q {c}
```

Figure 3.2 A Natural Deduction proof tree

untyped quantifiers.

An APL/CV proof is a sequence of lines, each of which must be one of the following:

(1) A conclusion of a rule of inference, written P, where P is a formula

(2) The special symbol PROOF

(3) The special symbol QED

(4) An assumption of the form ASSUME P, where P is a formula

(5) A qualifier of the form ARBITRARY v or CHOOSE v, where v is a variable

There are some constraints on the ordering of the lines in a proof. The symbols PROOF and QED must occur in pairs, properly nested in the same fashion that <u>begin</u> and <u>end</u> are nested in ALGOL, and that parentheses are nested in formulas. A PROOF, its matching QED, and the text between them is collectively referred to as a "proof block." Qualifier lines may occur only immediately after PROOF lines, and ASSUME lines may occur only after PROOF or CHOOSE lines.

```
PROOF           PROOF           PROOF
ASSUME P        CHOOSE v        ARBITRARY v
.               ASSUME P        .
.               .               .
.               .               .
Q               .               Q
QED             Q               QED
                QED
```

Figure 3.3 Notation for proof blocks used in rules

The last line of the proof must be a conclusion line; it contains the formula being proved.

Central to APL/CV is the notion of "accessibility." A formula P is said to be accessible to a conclusion line q whenever P appears on an assumption or conclusion line p, and the following three conditions hold:

(1) Line p precedes line q in the proof.

(2) Every proof block containing line p also contains line q.

(3) No proof block that contains line q but not line p qualifies a free variable of P. A proof block is said to qualify a variable v whenever its PROOF line is immediately followed by an ARBITRARY v or CHOOSE v line.

A proof block p is accessible to a conclusion line q whenever the following two conditions hold:

(1) Every line of p precedes line q in the proof.

(2) Every proof block containing p also contains line q.

Figures 3.4 and 3.5 contain the rules of inference of APL/CV. Every conclusion line of an APL/CV proof must be the conclusion of one of the rules of inference, and the corresponding hypotheses must be accessible to the conclusion line.

The APL/CV rules have been derived from Gentzen's rules. Corresponding rules in the Gentzen system and APL/CV have been labeled

$$\frac{P \quad Q}{P \,\&\, Q} \qquad \frac{P \,\&\, Q}{P} \qquad \frac{P \,\&\, Q}{Q} \qquad (1,2,3)$$

$$\frac{P}{P \mid Q} \qquad \frac{Q}{P \mid Q} \qquad \frac{P \mid Q \quad \begin{array}{l}\text{PROOF}\\ \text{ASSUME } P\\ \vdots\\ R\\ \text{QED}\end{array} \quad \begin{array}{l}\text{PROOF}\\ \text{ASSUME } Q\\ \vdots\\ R\\ \text{QED}\end{array}}{R} \qquad (4,5,6)$$

$$\frac{\begin{array}{l}\text{PROOF}\\ \text{ASSUME } P\\ \vdots\\ Q\\ \text{QED}\end{array}}{P \Rightarrow Q} \qquad \frac{P \quad P \Rightarrow Q}{Q} \qquad (7,8)$$

$$\frac{\begin{array}{l}\text{PROOF}\\ \text{ASSUME } P\\ \vdots\\ \text{'0'B}\\ \text{QED}\end{array}}{\sim P} \qquad \frac{P \quad \sim P}{\text{'0'B}} \qquad (9,10)$$

Figure 3.4 APL/CV rules of inference (Part 1)

with the same number. Except for rule (11), the rules that do not involve discharging assumptions are identical in both systems. The other rules have been modified in APL/CV to have proof blocks as hypotheses. Figure 3.3 shows the notation used to indicate proof blocks in the inference rules. The letter Q denotes the formula appearing on the line preceding the QED, either alone on the line or as part of an assumption. In the latter case, the vertical ellipsis is empty, and the letters P and Q

```
PROOF
ARBITRARY v
   :
P[v//x]
QED                    ALL x. P
----------------       ----------            (11,12)
ALL x. P               P[t//x]


                              PROOF
                              CHOOSE v
                              ASSUME P[v//x]
                                 :
                              Q
 P[t//x]          SOME x. P   QED
----------       ------------------------    (13,14)
SOME x. P                  Q


                  PROOF
                  ASSUME ~P
                     :
                  '0'B
  '0'B            QED
  -----           -----                      (15,16)
   P               P
```

Figure 3.5 APL/CV rules of inference (Part 2)

denote the same formula.

An important restriction on the use of rule (14), SOME Elimination, is that Q may not contain any free occurrences of v.

Figure 3.6 contains an example of an APL/CV proof. The proof corresponds to the Natural Deduction proof in Figure 3.2. Each line of the proof has been labeled with a letter. The conclusion lines have been annotated by the number of the rule used and references to its hypotheses, contained in braces.

It is easier to reason about APL/CV proofs if they are presented as trees rather than sequences of lines. A tree is obtained from a sequence

```
a)  PROOF
b)  ASSUME P|Q=>R
c)      PROOF
d)      ASSUME P
e)      P|Q                     {4d}
f)      R                       {8eb}
g)      QED
h)  P=>R                        {7c}
i)      PROOF
j)      ASSUME Q
k)      P|Q                     {5j}
l)      R                       {8kb}
m)      QED
n)  Q=>R                        {7i}
o)  (P=>R)&(Q=>R)               {1hn}
p)  QED
q)  (P|Q=>R)=>(P=>R&Q=>R)       {7a}
```

Figure 3.6 An example APL/CV proof

of APL/CV lines as follows.

If the sequence of lines contains no proof blocks, the resulting tree is a vertical chain of nodes, one for each proof line. This vertical chain will be called the "main branch." The formula at the bottom of the main branch is called the "terminus." If the sequence contains one or more proof blocks, for each block, a branch that angles off to the right and then down is hung on the main branch at the point corresponding to the block. This branch connects to a subtree that is obtained recursively from the sequence of lines between the matching PROOF and QED. Back on the main chain, a square is inserted just below the branch point; this square represents the proof block when determining accessibility.

Note that the formula being proved is the terminus of the main branch of the proof. The main branch of a proof never contains any qualifiers or assumptions, and the main branch of a proof block only has qualifiers or assumptions at the top, as indicated by the appropriate proof rule.

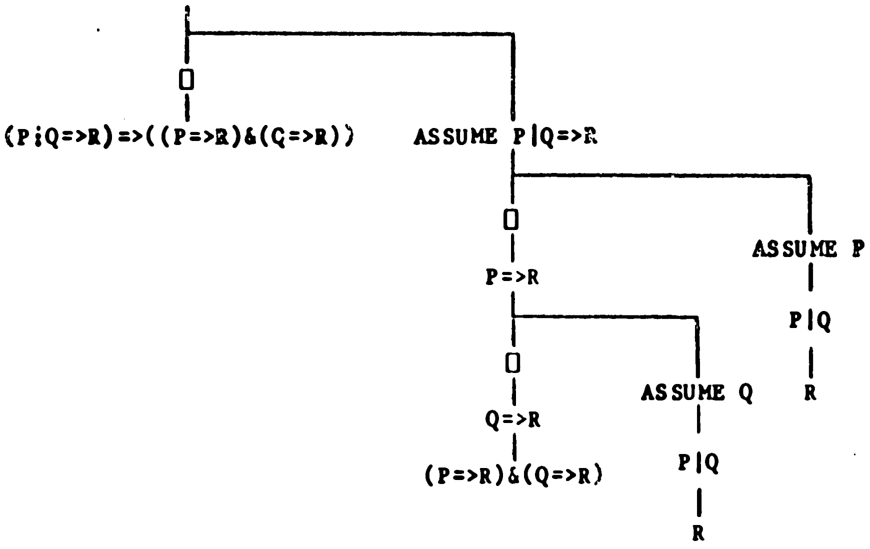


Figure 3.7 An APL/CV proof tree

Figure 3.7 contains an APL/CV proof tree for the proof in Figure 3.6.

The advantage of using APL/CV proof trees is that the accessibility rules are simpler for trees than for the sequential system. A formula P is accessible to a node q if and only if a node p containing the conclusion P or ASSUME P is accessible to the node q. A proof block is accessible to the node q if and only if the square representing the proof block is accessible from the node q. A node p (whether assumption, conclusion, or proof block square) is accessible from a node q in an APL/CV tree if and only if:

(1) There is a (unique) path in the tree from q to p that goes only upward and leftward. Hereafter, this path will be referred to as an "access path."

(2) If p contains a formula P, the access path from q to p does not pass through any CHOOSE or ARBITRARY node that qualifies a free variable in P.

The reason the square representing the proof block must appear below the branch point is to prevent a proof block from being accessible to nodes within itself.

We are now equipped to see the correspondence between Natural Deduction and APL/CV. Two constructions will be given. The first construction will be used to convert a valid Natural Deduction tree (hereafter called an ND tree) into a valid APL/CV tree. Then, the reverse construction will be given, establishing the equivalence of the two systems.

3.4 Converting Natural Deduction proofs to APL/CV

The algorithm for converting an ND tree to an APL/CV tree has these steps:

(1) Break the tree apart at the root, forming from one to three subtrees.

(2) Recursively apply the the algorithm to the subtrees to form APL/CV trees.

(3) Combine the resulting APL/CV subtrees to an APL/CV tree corresponding to the original ND tree.

However, there is a slight problem. Although we begin with a complete Natural Deduction proof, with all its assumptions discharged, after step (1) is performed, the resulting subtrees will, in general, contain some undischarged assumptions. These subtrees are not proofs, and therefore cannot be converted into APL/CV proofs in step (2).

In this case, the output of the algorithm will be an APL/CV proof tree with some extra assumptions, the undischarged assumptions of the ND tree, attached to the root. These assumptions will be accessed from within the tree. When the algorithm is applied to a complete ND tree,

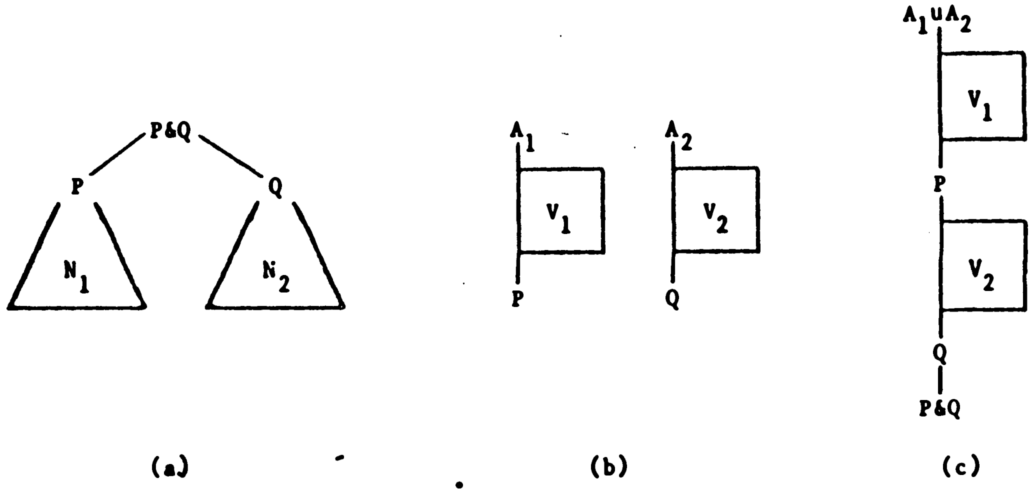


Figure 3.8 The conversion of an And Elimination ND tree

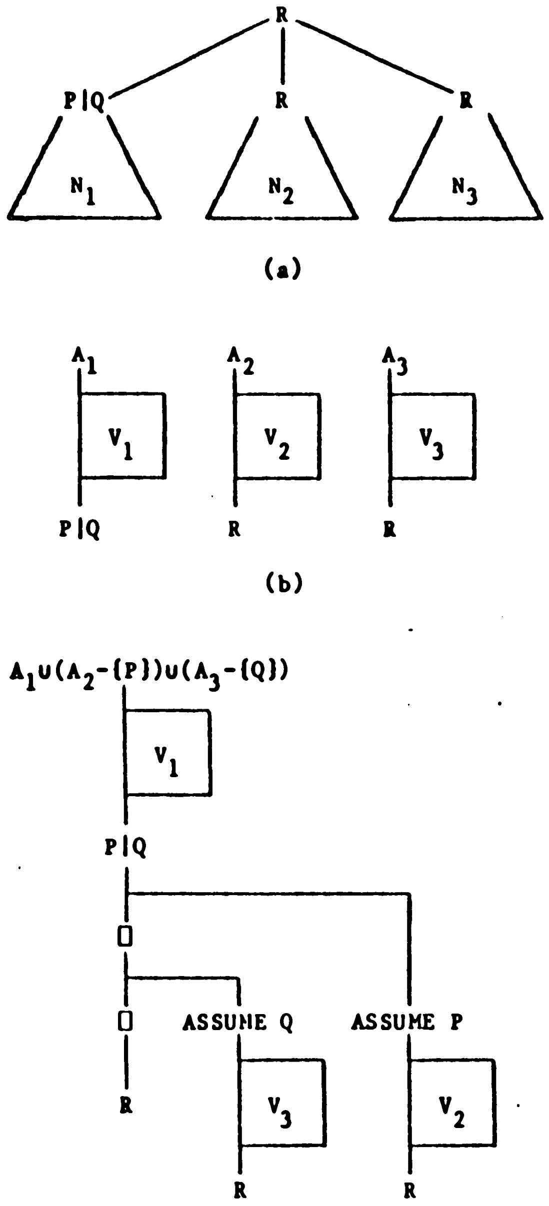
which has all its assumptions discharged, the resulting APL/CV tree will not have any extra assumptions.

For the base case of the recursion, we must convert a leaf of an ND tree, a single undischarged assumption, to an APL/CV tree. It is converted to a null tree with the assumption attached.

The actual work of the construction is performed in step (3). How the APL/CV trees are combined varies, depending on the rule applied at the root of the ND tree. Separate constructions will not be given for each of the sixteen rules. It should be sufficient to examine in detail only four exemplary rules; the construction for each of the rest of the rules is similar to one of the four.

Each of the four cases is illustrated with a three-part diagram (Figures 3.8-11), containing (a) the ND tree, (b) the APL/CV trees obtained from step (2), and (c) the APL/CV tree obtained from step (3).

Rule (1), And Elimination, is one of the simplest, and is a good place to start. The construction is illustrated in Figure 3.8. The ND trees N_1 and N_2 are recursively converted to the APL/CV trees V_1 and V_2.

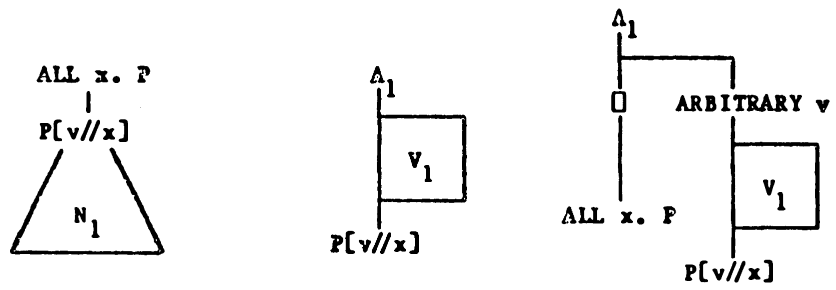


(c)

Figure 3.9 The conversion of an Or Elimination ND tree

A_1 and A_2 are sets of undischarged assumptions from the ND trees.

The APL/CV tree for P&Q is constructed by stacking V_1 on top of V_2. The set A_2 of assumptions is removed from the top of V_2, and is united



(a) (b) (c)

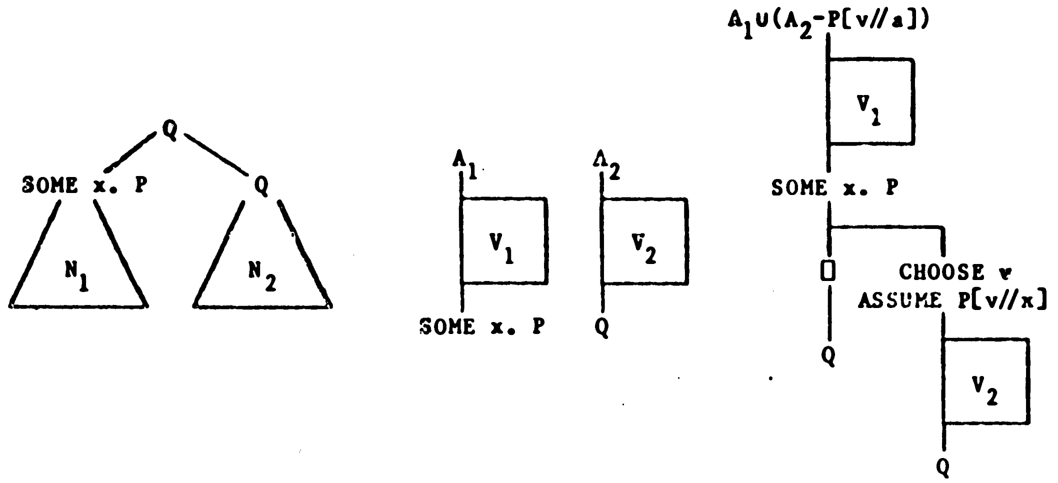
Figure 3.10 The conversion of an ALL Introduction ND tree

(eliminating duplicates) with A_1. The formula P&Q is appended to the bottom of the tree.

We must verify the validity of the resulting APL/CV tree. The main branch of V_1 and V_2 will not contain any qualifiers; those will be off to the right in the proof blocks. Therefore, formulas P and Q are accessible to the formula P&Q. All the rules used in V_1 and V_2 are still properly applied. The only change is that any access paths from nodes in V_2 to assumptions in A_2 must be extended through the main branch of V_1, which is allowed.

Constructions similar to the one for And Introduction are used for rules (2, 3, 4, 5, 8, 10, 12, 13, 15). Let us next consider a rule in which assumptions are discharged. The most complicated of them is rule (6), Or Elimination.

Figure 3.9 illustrates the construction used for Or Elimination. Recall that the set of assumptions at the top of the tree in (c) are allowed to consist only of undischarged assumptions of the tree in (a). Since the use of the Or Elimination rule discharges the assumptions P and Q in N_2 and N_3 (respectively), these assumptions may not be promoted to the top of the tree in (c). Therefore, V_1 and V_2 are turned into proof



(a) (b) (c)

Figure 3.11 The conversion of a SOME Elimination ND tree

blocks, and the assumptions P and Q are left behind to head the proof blocks.

As was the case with And Elimination, access paths from nodes in V_2 and V_3 to assumptions in A_1-{P} and A_2-{Q} must be extended up the main branch of V_1.

Similar constructions are used for rules (7.9.16), leaving only rule (11), ALL Introduction, and rule (13), SOME Elimination, for consideration.

The ALL Introduction construction is illustrated in Figure 3.10. The access paths from nodes in V_1 to assumptions in A_1 must be extended in (c) to pass through the ARBITRARY v node. The APL/CV accessibility rules allow such a path only if the assumptions in A_1 do not contain the variable v. But recall that A_1 is the set of undischarged assumptions from N_1. No assumption in A_1 can contain v, by the restriction placed on Gentzen's ALL Introduction rule.

Finally, let us consider the conversion for SOME Elimination, which is illustrated in Figure 3.11. Of all the rules, this one has the most complicated construction, because an assumption is being discharged and a qualifier is being introduced. As with the ALL Introduction rule, we must make sure that access paths from nodes in V_2 to assumptions in A_2-{P[v//x]}, which must be extended through the CHOOSE v node, do not access assumptions containing a free occurrence of v. But since Gentzen's SOME Elimination rule cannot be used if v occurs in SOME x. P, Q, or any of the undischarged assumptions in A_2, no access path is blocked. Moreover, this restriction ensures that the tree (c) will obey the APL/CV restriction on SOME Elimination, that v is not in Q.

The above construction completes the proof of the following theorem: for every Natural Deduction proof of a theorem P, there is a corresponding APL/CV proof of P.

To complete the proof of the equivalence of the two systems, the converse of the above theorem will be shown by giving the reverse construction.

1.1 Converting APL/CV proofs to Natural Deduction

The first step of this construction involves renaming some of the variables used in the APL/CV tree. If there are n qualifiers in the proof, n distinct variables that do not appear anywhere in the proof tree are chosen. Then, in each subtree whose root is a qualifier of the form CHOOSE v or ARBITRARY v, every occurrence of v is replaced with one of the new variables.

To see that this renaming preserves the correctness of an APL/CV proof, it is only necessary to inspect each rule of inference, and verify that if a new variable (one that does not already appear in the proof) is consistently substituted for another in the hypotheses and conclusion of an instance of a rule, the result is still an instance of the rule.

The only way we can get into trouble is if one or more of the hypotheses of the rule are not in the subtree being modified, and therefore do not have the substitution performed on them. Because the access path to the hypothesis must pass through the qualifier for v, by the APL/CV accessibility rules, the hypothesis cannot contain v. and effectively the substitution has been performed.

If the same variable is qualified in more than one place in the proof, care must be taken in choosing the order in which the renamings are performed. If proof block B_1 contains proof block B_2, and both qualify the same variable, B_2 must be renamed before B_1. If this ordering restriction is observed, the resulting proof tree will have have the following property: any variable that is qualified at the root of a subtree does not appear anywhere in the proof except in that subtree.

After the renaming operation has been performed, the APL/CV tree is traversed by starting at the root and heading down the main branch, visiting each node. As each proof block is encountered, it is traversed recursively. This traversal is exactly the order in which the lines appear in a sequential proof. The order has been chosen so that a node V is not visited before every node accessible to V has been visited.

When each conclusion or assumption node containing a formula P is visited, an ND tree with P as its root will be constructed and associated with that node. Not every node in an APL/CV tree contains a valid formula; therefore the ND trees constructed must in general have some undischarged assumptions. The ND tree constructed for a node V will have the following property: if the access path from V to the root of the APL/CV tree is traced, the set of assumptions through which this path passes will contain the set of assumptions that are undischarged in the constructed ND tree. Since the access path from the formula P proved by the APL/CV tree to its root is the main branch of the proof, the path cannot pass through any assumptions. Therefore, the ND tree constructed for P will have all its assumptions discharged.

The ND tree constructed for a node V depends on whether the node is an assumption or conclusion, and if it is a conclusion, which rule has been used. As before, it is not necessary to consider all the rules, the same set of representative examples is used.

If the node V is an assumption, the ND tree constructed consists of just the assumption as a leaf. Since V is on the access path from itself to the root, it is allowed to be undischarged.

Next, consider the case when V is a conclusion node; then V consists of just a formula. As in the previous construction, we consider four representative cases.

First, suppose V is of the form P&Q, and is the conclusion of rule (1), And Elimination. Then there must be formulas of the form P and Q accessible to V. The nodes containing these formulas will have already been visited, and ND trees N_P and N_Q will have been constructed for them. The ND tree for V has P&Q as its root, and N_P and N_Q as its two subtrees. Call this new tree N_{PQ}.

We must verify that the undischarged assumptions of N_{PQ} are all on the path from V to the root of the APL/CV tree. Because P and Q are accessible to V in the APL/CV tree, the access path from V to the root of the APL/CV tree passes through both P and Q. Therefore, the path from V

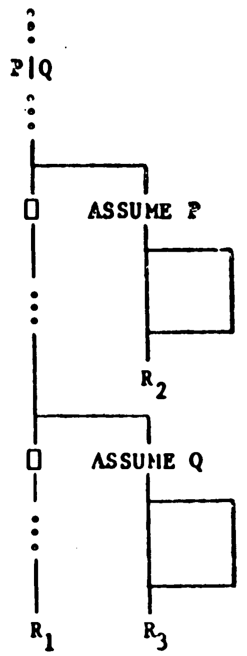


Figure 3.12 An Or Elimination node

to the root contains both the path from P to the root and the path from Q to the root. The set of undischarged assumptions of N is the union of the undischarged assumptions in N_P and N_Q. By construction, the undischarged assumptions in N_P and N_Q are the path from P and Q (respectively) to the root of the APL/CV tree.

Similar constructions are used if the APL/CV node is the conclusion of any of rules (2, 3, 4, 5, 8, 10, 12, 13, 15).

Next, let us consider the case in which the node is the conclusion of rule (6), Or Elimination. Figure 3.12 contains a diagram of the situation. (Although a node containing P|Q and the proof blocks shown must be on the path from R_1 to the root, they need not appear in the order shown in Figure 3.12.) Node R_1 is currently being visited; nodes containing P|Q, R_2, and R_3 have already been visited and ND trees constructed for them.

The ND tree N_R is formed by joining these three trees.

We must verify that all the undischarged assumptions in N_R are on the path from R_1 to the root of the APL/CV tree. By the same reasoning used for the And Introduction rule, all the undischarged assumptions in the ND tree for P|Q are on the path from R_1 to the root. The same is true for all the undischarged assumptions, except P, in the ND tree for R_2, and all the undischarged assumptions, except Q, in the ND tree for R_3. The exceptions P and Q are not a problem, however, because Gentzen's Or Elimination rule discharges all the assumptions of P in the proof tree for R_2 and all assumptions of Q in the proof tree for R_3, so that P and Q are not undischarged assumptions of N_R.

Similar constructions are used for rules (7, 9, 16), leaving rule (11), ALL Introduction, and rule (14), SOME Elimination for consideration.

The ALL Introduction case is pictured in Figure 3.13. The ND tree N_P that was constructed for node P is used as the single offspring of ALL x. P[x//v] in the ND tree constructed for the formula.

Checking that the undischarged assumptions in N_P are on the path from ALL x. P to the root is no different for this rule than for the previous rules. What makes this rule special is that we must verify that the restriction on the use of Gentzen's ALL Elimination rule, that none of the undischarged assumptions in N_P contain v, is met. By construction, all of the these assumptions are on the path from P[v//x] to the root. But all of the assumptions must be outside the subtree whose root is ARBITRARY v, since there can be no assumptions on the main branch of the subtree. And the renaming operation guarantees that none of the assumptions outside the subtree contain any occurrences of v.

Finally, consider the SOME Elimination rule, pictured in Figure 3.14

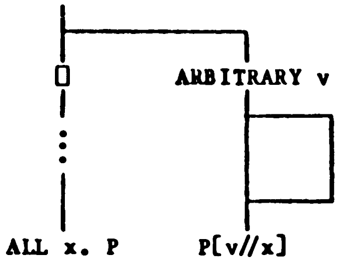


Figure 3.13 An ALL Elimination node

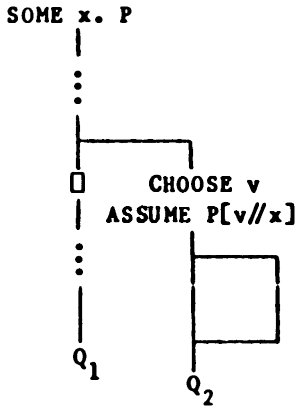


Figure 3.14 A SOME Elimination node

(Once again, the order of the hypotheses need not be as shown). The ND trees already constructed for SOME x. P and Q_2 are used to build the ND tree for Q_1. We must verify that the three restrictions on Gentzen's SOME Elimination rule, that:

(1) the variable v must not occur in SOME x. P,

(2) the variable v must not occur in Q_1, and

(3) the variable v must not occur in an undischarged assumption in the proof tree for Q_2

are met. Restriction (2) will be met by the ND tree because it is also a restriction on the APL/CV rule. The other two restrictions are guaranteed to be met by the renaming operation.

The proof of the equivalence of Gentzen's Natural Deduction and APL/CV is now complete.

There are obvious similarities between the construction that translates an APL/CV proof to a Natural Deduction proof and the one that goes the other way, but the constructions are not inverses. APL/CV is more economical than Natural Deduction in the sense that if a proof written in APL/CV is translated into a Natural Deduction proof and then translated back into APL/CV, it will typically get larger. The reason for the expansion is that in APL/CV, once a formula is proved it may be used repeatedly as a hypothesis in subsequent lines. When the proof is translated into a Natural Deduction tree, one copy of the proof tree for a formula is added to the tree for each time the formula is used as a hypotnesis. A common technique for shortening proofs in APL/CV is to prove a universally quantified lemma and then apply it several times using the ALL Elimination rule. When applied to Natural Deduction proofs, this technique only makes them longer.

3.6 PL/CV versus APL/CV

The language APL/CV defined in section 3.3 is very close to the language processed by the PL/CV1 Proof Checker. Several modifications were made to APL/CV to get PL/CV, the language processed by the Proof Checker described in this thesis. The modifications made to APL/CV were aimed at achieving the following goals:

(1) Brevity. The biggest drawback of PL/CV1 is the great length of the proofs. Much of the design of PL/CV was driven by the goal of allowing the user to get sway with writing as little as possible.

(2) Legibility. PL/CV was designed to mimic as closely as possible the

style of informal mathematical proofs.

(3) Consistency. PL/CV predicate calculus is the core of a program verification system for PL/CS, a PL/I subset. The syntax of PL/CV was designed to be consistent with that of PL/CS.

(4) Efficiency. The PL/CV language and Proof Checker were designed hand-in-hand so that the language could be efficiently checked. The goal was to make the Proof Checker as efficient as an optimizing compiler.

The changes made to APL/CV, as motivated by the above goals, will now be discussed. There is no reason to present the entire PL/CV syntax in detail here. The interested reader is referred to the PL/CV Reference Manual [17].

A syntax had to be chosen for quantifiers and qualifiers. As we shall soon see, the purpose served by these constructs is the same as the purpose served by declarations in the programming language. Therefore, a similar syntax is used for all three constructs, as shown by these examples:

```
DECLARE (A, B) FIXED;
ALL (A, B) FIXED. (A**2 + B**2) >= 0;
ARBITRARY (A, B) FIXED;
```

So that every variable in PL/CV has a specific type associated with it, information must be included on quantifiers and qualifiers to indicate the type of the variable being introduced. The FIXED (integer) type is used in all the examples in this chapter.

The most common complaint heard from the "casual observer" of the PL/CV language concerns the use of FIXED, BIT, '1'B, and '0'B instead of

INTEGER, BOOLEAN, TRUE, and FALSE (the former being the names used in PL/CS for the latter). An actual user of PL/CV is going to have to learn the PL/CS names anyway, and it would only confuse matters if "good" names were used in just the predicate calculus part of the language. Then, not only would the user have to learn two different names for each concept, but he would have to keep straight when to use which name.

One concession was made on the consistency issue. In PL/CV, a string of operands connected by relational operators, for example:

```
A <= B < C
```

is considered to be an abbreviation for what is obviously meant:

```
(A <= B) & (B < C).
```

Unfortunately, the expression A <= B < C has another meaning in PL/CS: compare A and B yielding a BIT value, convert (using the legendary PL/I conversion rules) the BIT value to FIXED, and compare the result to C. Because Boolean expressions of this nature are used so frequently in proofs, considerations of legibility and brevity outweighed those of consistency. I am bewildered by the fact that this feature has not found its way into any of the major modern programming languages.

For consistency with PL/CS, the syntax of PL/CV is sprinkled liberally with semicolons. The keywords PROOF and QED are followed by semicolons because DO and END are followed by semicolons in PL/CS.

The syntax chosen for a conclusion line, inherited from PL/CV1, is:

```
P BY <rule name>, <justification>;
```

Some critics of PL/CV1 complained about the large number of rule names that had to be remembered. For this reason, in PL/CV, all the lines using Introduction rules use the rule name INTRO. The Proof Checker can determine which introduction rule is being used from the main connective of the conclusion. In retrospect, it is not clear how wise this decision was. The overloading of the name INTRO decreases the legibility of the kind of proofs discussed in section 3.13.

The exact nature of <justification> depends on which rule is being used. It consists of some hints to the Proof Checker about how the rule is being applied, and what its hypotheses are. The goals of legibility and efficiency demanded that some justification be present, but the goal of brevity required that it be kept to a bare minimum.

The justification for each rule includes the minimum amount of information needed to deduce the hypotheses of the rule from the conclusion. For example, the SOME Introduction rule has the format:

```
SOME x FIXED. P BY INTRO, t;
```

The proof checker substitutes t for x in P to deduce the hypothesis of this rule.

In PL/CV1, the justification included a reference to the line at which each of the hypotheses could be found. This situtation was intolerable. For some PL/CV rules, the user must include one or more hypotheses as part of the justification, but he need never indicate where those lines can be found. How the Proof Checker locates the hypotheses is explained in section 3.12.

3.7 Macros

If a hypothesis of a rule is lengthy, it can be annoying to be required to write it several times: once when the formula is assumed or proved, and then each time it is used as a hypothesis to a later rule. To ease this burden, a macro facility is provided. At various points in the syntax, a user can put a label on a formula, as in:

```
L: ALL X FIXED. X**2 >= 0 BY INTRO, ... .
```

Subsequently, whenever the identifier L is used, it is expanded to the quantified formula. The major use of label-macros is the shortening of justifications, but they may appear anywhere a formula can appear, as in:

```
L => K**2+1 >= 1 BY INTRO, ... .
```

A parameterized macro facility is also provided. For example, one can write:

```
FOR (X, Y) FIXED DEFINE DIVIDES(X, Y) = SOME M FIXED. M*X = Y;
```

If one later writes, for example, DIVIDES(A, B*A), it is expanded to the formula:

```
SOME M FIXED. M*A = B*A .
```

Experience has shown this feature to be vital for the serious user of PL/CV.

Unlike PL/I [47] or PL/C [72] macros, which are add-ons implemented with a preprocessor, the macro facility of PL/CV is built into the language. Macro calls are part of the syntax of the language. They are allowed only where formulas or expressions are allowed, and they are type-checked. Macro substitutions are performed on parse trees, not input text. For example, in the expansion of DIVIDES(A+10, 10*A), parentheses are implicitly added:

```
SOME M FIXED. M*(A+10) = 10*A
```

and in the expansion of DIVIDES(M, 2*M), bound variables are implicitly renamed:

```
FOR (X, Y) FIXED DEFINE DIVIDES(X, Y) =
    SOME M FIXED. M*X = Y;

FOR P FIXED DEFINE PRIME(P) =
    P>1 & ALL D FIXED. (D>1 & DIVIDES(D, P) => D=P);

FOR (P, X) FIXED DEFINE PRIME_FACTOR(P, X) =
    PRIME(P) & DIVIDES(P, X);

FOR (P, X) FIXED DEFINE GREATEST_PRIME_FACTOR(P, X) =
    PRIME_FACTOR(P, X) &
    ALL P1 FIXED. (PRIME_FACTOR(P1, X) => X <= P);
```

Figure 3.15 Building a complex definition in PL/CV

```
SOME M1 FIXED. M1*M = 2*M
```

because macro parameters are represented internally in the same way as bound variables. Textual substitution macros would handle both these examples incorrectly.

Complex definitions can be built by nesting macro definitions. For example, Figure 3.15 shows how the relation GREATEST_PRIME_FACTOR can be defined in PL/CV. A feature of this nature is an absolute requirement if one is to express even moderately complex concepts in PL/CV.

A problem with the system as implemented is that the length of a complete macro expansion can grow exponentially with the depth of the macro definition nesting. Strictly speaking, the definitions are not part of the logic; the parser simply expands the macros before any proof checking is performed. Because the algorithm discussed in Chapter 5 runs in time $O(n^2)$, where n is the length of a formula in the proof, there is interest in keeping the individual formulas short.

One possible solution to this problem involves the partial expansion of macros. For example, to prove anything about:

```
GREATEST_PRIME_FACTOR(P, X)
```

the user would have to expand the macro somewhat, perhaps until he got to a formula involving just PRIME and DIVIDES. At that point, he might simply reference previously proved lemmas about PRIME and DIVIDES to get his result. The macros PRIME and DIVIDES would not have to be expanded. For an approach like this to work, definitions have to be made a more integral part of the logic.

3.8 Proof Blocks

Experience with PL/CV1 showed that, though a formula is often used as the hypothesis to more than one subsequent line, a proof block is almost always used to prove just one formula. In APL/CV, if a formula P is the conclusion of a rule that uses a proof block as a hypothesis, arbitrarily many lines can come between the end of the proof block and P. Allowing this separation only serves to decrease the legibility of proofs. If the following restrictions:

(1) the hypotheses that are formulas must precede the hypotheses that are proof blocks, and

(2) if more than one hypothesis is a proof block, they must be adjacent in the proof, and

(3) each proof block can be used as the hypotheses to just one rule, and the conclusion of that rule must immediately follow the last hypotheses that is a proof block,

are added to the logic, no power is lost. Given an APL/CV proof that does not obey these restrictions, if the constructions of sections 3.4 and 3.5 are used to translate it to a Natural Deduction proof, and then back into an APL/CV proof, the above restrictions will then be met.

Here is what the Implication Introduction rule looks like when the above restrictions are obeyed:

```
PROOF;
ASSUME P;
:
Q BY ...
QED;
P => Q BY INTRO;
```

But this syntax seems backwards. In mathematics books, the theorem always comes first and is followed by the proof. The following syntax suggested itself:

```
P => Q BY INTRO,
PROOF;
ASSUME P;
:
QED;
```

The above syntax is used in PL/CV. In this syntax, the proof block is considered to be part of the justification. Experience has shown this syntax to be more natural to use than the proof-then-theorem style of APL/CV. Another advantage of this syntax is that if the ASSUME line is omitted, what the missing line should be can be deduced by the Proof Checker. Therefore, all of the ASSUME lines that begin proof blocks are optional in PL/CV.

The theorem-before-proof syntax requires that the definition of accessibility be modified so that the formula P => Q cannot be accessed from within its proof.

Another consequence of this syntax is that proof blocks cannot be shared. This restriction is not a harsh one, because the first time the proof block is written, it can be used to prove P => Q. If the proof block is needed again, the second block can contain a single instance of the

```
    or
  / | \
 P  Q  R
```

Figure 3.16 The parse tree for P|Q|R

Implication Elimination rule, rather than a copy of the first block.

The Or Elimination rule has two proof blocks as hypotheses. Rather than requiring two adjacent proof blocks as in:

```
R BY OREL, P|Q,
PROOF;
ASSUME P;
:
R BY ...
QED;
PROOF;
ASSUME Q;
:
R BY ...
QED;
```

the syntax was cleaned up and the name of the rule changed to more closely correspond to informal mathematical use. The following syntax is used in PL/CV:

```
R BY CASES, P|Q
PROOF;
CASE P;
:
R BY ...
CASE Q;
:
R BY ...
QED;
```

Unlike the ASSUME line, the CASE line is not optional, for reasons explained in section 3.13.

Another extension of the CASES rule involves the way the disjunction ("or") operator is parsed. In most computer languages, every operator is assigned an "associativity." For example, in PL/I, addition is "left associative," meaning that A+B+C is parsed as (A+B)+C, whereas exponentiation is "right associative," meaning that A**B**C is parsed as A**(B**C). In PL/CV, disjunction is neither left nor right associative; it is called a "multiary" operator. The expression P|Q|R is not parsed as (P|Q)|R or as P|(Q|R); the parse tree built for this expression is shown in Figure 3.16. When using the Or Elimination rule, one proof block is given for each operand of the disjunction, as shown in the following example:

```
P BY CASES. Q | (R|S) | T.
PROOF;
CASE Q;
:
P BY ...
CASE R | S;
:
P BY ...
CASE T;
:
P BY ...
QED;
```

3.9 Qualifiers and Scope Rules

Every variable in a PL/CS program must be declared. This restriction was included in PL/CV, and fits into the language quite nicely. There are three kinds of variables in PL/CV: program variables, bound variables, and logic variables. Program variables are declared by the PL/CS DECLARE statement, bound variables are declared by the quantifier to which they are bound, and logic variables are declared by an ARBITRARY or CHOOSE line. PL/CV uses ALGOL scope rules [61], where the blocks used to determine variable scopes are proof blocks and quantified expressions.

From the point of view of the implementation, here is how variable scopes are handled. Every time a declaration is processed, a new variable is created, associated with the current block, and stored in a symbol table. Given ALGOL scope rules, it is permissible to have different variables with the same name; the implementation keeps track them by assigning them different internal names. When an identifier is parsed, and there is more than one variable with that name, the identifier is taken to denote the variable declared in the innermost block.

None of the above is particularly interesting; block-structured symbols tables [36] have been used in compilers for years. What is interesting is that the ALGOL scope rules automatically enforce one of the accessibility conditions of APL/CV. Suppose a line L in a proof block headed by an "ARBITRARY v" needs a hypothesis P_1 that contains the variable v. The line L cannot possibly use a formula from outside the proof block for a hypothesis.

The reason it cannot is that the assignation of internal names to logic variables performed by the symbol table routines is precisely the renaming step of the construction that converts APL/CV trees to Natural Deduction trees. Even if there is an assertion P_2 outside the block that has the same external appearance as P_1, the identifier v in P_2 will be associated with a variable that is different from the variable associated with the identifier v in P_1. Because the two variables will have different internal names, the internal representation of P_2 will be different from that of P_1, and P_2 will not be usable as a hypothesis.

Here is the PL/CV ALL Introduction rule:

```
ALL x FIXED. P BY INTRO.
PROOF;
ARBITRARY v FIXED;
:
P[v//x] BY ...
QED;
```

as it has been modified so far. The user is free to use any variable he likes on the ARBITRARY line. But no power is lost in the logic if x and v are required to have the same name. Any PL/CV proof that does not obey this restriction can be converted to one that does by following these steps:

(1) Rename every bound and logic variable in the proof so that no variable is ever used twice.

(2) Find every ALL Introduction proof block and rename the logic variable to agree with the corresponding quantifier.

Each logic variable introduced by the And Introduction rule is required to have the same as its corresponding bound variable so that the Proof Checker can always deduce what the ARBITRARY line should be. The ARBITRARY line is an optional part of the syntax; if one is present, the Proof Checker simply makes sure what is given matches what should be there. The restriction is a bit confusing to novices, who have trouble keeping track of the difference between the logic variable and the bound variable. In retrospect, it probably would have been better to have the name of the logic variable default to the name of the bound variable if the ARBITRARY line is omitted, but if an ARBITRARY line is given, allow it to declare any name. However, each degree of freedom allowed in the syntax increases the complexity of the Proof Checker.

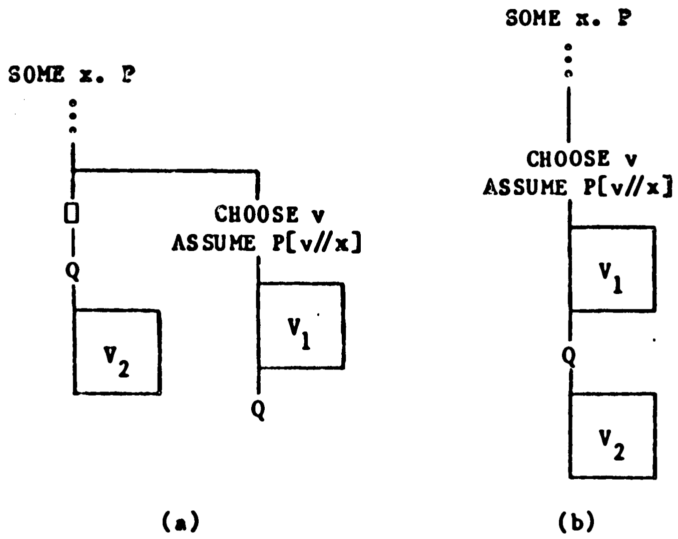


Figure 3.17 Removing a proof block from the SOME Elimination rule

3.20 SOME Elimination

The most substantial difference between PL/CV and APL/CV is in the SOME elimination rule. Here is the rule as modified so far:

```
Q BY SOMEL,
PROOF;
CHOOSE v FIXED;
ASSUME P[v//x];
:
Q BY ...
QED;
```

where the formula:

```
SOME x FIXED. P
```

is a hypothesis. Experience with PL/CV1 indicated that this rule was annoying to use, because the formula Q must be written twice. The SOME Elimination rule decided upon for PL/CV does not use a proof block, and the qualifier and assumption are combined into a single line. It is simply:

CHOOSE v FIXED WHERE P[v//x];

The formula SOME x FIXED. P must be accessible to the CHOOSE line. The scope of the variable v begins at the CHOOSE line and ends at the end of the innermost block containing the CHOOSE line.

To show that this abbreviated rule is equivalent to the rule that uses a proof block, a construction will be given to convert proofs using each rule to proofs using the other rule.

Figure 3.17a shows a proof tree using the block rule, and Figure 3.17b shows how it is changed to use the abbreviated rule. Assume without loss of generality that the free variables in the tree in (a) have been renamed so that there are no occurrences of the variable v outside of V_1. It should be clear that all the accessibility paths in (a) have corresponding paths in (b). The only possible problem is a path routed up the main branch of (a) being stopped by the "CHOOSE v" node inserted on that branch in (b). But only paths to formulas containing v are stopped, and the only such formulas are in V_1.

Figure 3.18a shows a proof tree that uses the abbreviated rule. The rule will be used somewhere on the main branch of the entire proof, or of some proof block. Let Q be the terminus of that branch.

The corresponding proof using a proof block is in Figure 3.18b. Every accessibility path in (a) has a corresponding path in (b). Since the terminus of the main branch of the subproof is still Q, whatever rule is being used in (a) is still being used properly in (b).

But recall there is a restriction on the use of the APL/CV SOME Elimination rule: the formula Q cannot contain v. We must verify that this restriction is enforced in (b). There is no problem as long as the main branch shown in (a) is part of a proof block. Then Q is either 'O'B

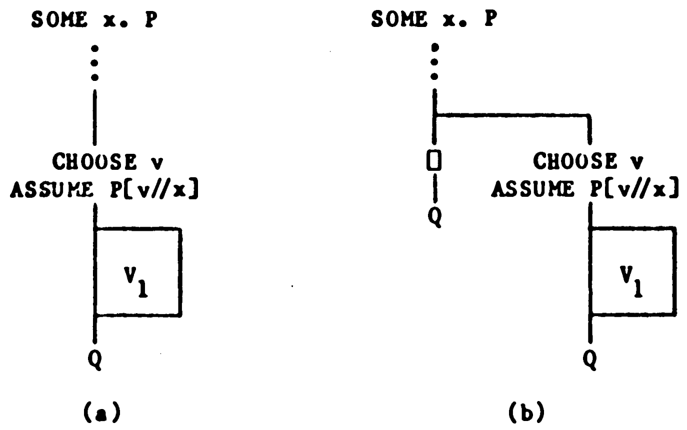


Figure 3-18 Adding a proof block to the SOME Elimination rule

or a subformula of the conclusion of the rule being used, which, by the scope rules, cannot contain any occurrences of v.

There is a slight problem in the implemented system if Q is the final conclusion of the proof. For example, one could prove the formula A>0 by:

```
1 > 0 BY ARITH;
SOME X FIXED. X > 0 BY INTRO, 1;
CHOOSE A FIXED WHERE A > 0;
A > 0; .
```

This formula could not be proved in APL/CV. Realize that PL/CV formulas with free variables are not implicitly universally quantified; the above is not a proof of ALL A FIXED. A>0. However, it is somewhat anomalous. One way of solving the problem would be to require that the conclusion of a proof be free of CHOOSE variables. A cleaner solution would be to enclose the entire proof in a proof block, by making the syntax for a proof of P be:

```
P BY
PROOF;
:
P BY ...
QED;
```

3.11 Arithmetic Inferences

The pure predicate calculus and rules for programming constructs are not sufficient to prove very many interesting facts about programs; it is also necessary to be able to reason about the data domain. The ARITH rule is used for this purpose in PL/CV. The simplest form of the rule is:

P BY ARITH, Q_1, Q_2, ... Q_n;

where P can be derived from Q_1, Q_2... Q_n using a specified set of arithmetic axioms.

Further discussion of the ARITH rule is beyond the scope of this thesis. The interested reader is referred to the PL/CV Reference Manual [17] for a complete description of this rule. Chan [14] discusses how this rule is implemented.

3.12 V-code Generation

In the previous sections, we saw how the use of ALGOL scope rules in PL/CV frees us from worrying about the annoying little restrictions in APL/CV, that are usually (erroneously) ignored anyway. To completely validate a proof, it is only necessary to check that:

(1) the hypotheses and conclusion of the rules properly match each other, and

(2) every proof block containing the hypothesis of a rule also contains the conclusion.

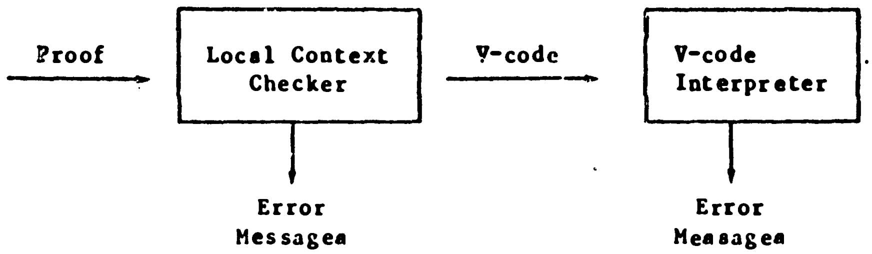


Figure 3.19 The organization of the Proof Checker

These conditions are checked with the help of an automaton called a "V-machine." In the memory of the V-machine is a set of formulas, which is called the "Assertion Table." Initially the Assertion Table is empty. The input to the V-machine is a sequence of commands, called V-code, which are instructions to the V-machine for the manipulation of the Assertion Table. Here is a list of the V-code instructions used to check predicate calculus proofs, and the actions they specify:

Instruction	Action
PROCLAIM P	Add P to the Assertion Table.
REQUIRE P	If P is not in the Assertion Table, report and error.
OPEN	Push a copy of the Assertion Table onto a stack.
CLOSE	Replace the Assertion Table by the top of the stack and pop the stack. It is assumed that OPEN and CLOSE instructions will be properly nested so that the stack will not underflow.

An interpreter for V-machine described above can be implemented using the same techniques used to implement symbol tables in languages with block structure [36].

As shown in Figure 3.19, the Proof Checker is divided into two

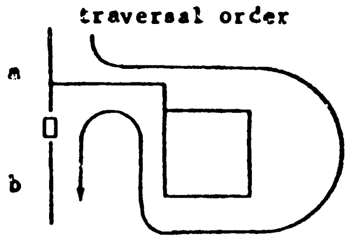


Figure 3.20 V-code generation for a proof block

modules. The Local Context Checker (LCC) inputs the proof and generates V-code, in much the same way a compiler would input a program and generate machine language. The generated V-code is sent to the V-code Interpreter, which is a software implementation of a V-machine. To simplify the discussion, we will for the time being assume that as soon as a V-code instruction is generated by the LCC it is executed by the V-code Interpreter. (In actuality, all the V-code is generated, and then it is sent to the V-code Interpreter. The reason for this two-pass operation will be explained in Chapter 4.)

Appropriate V-code is generated by the LCC so that the following invariant is maintained: when the LCC is at line p of the proof, the Assertion Table consists of exactly the set of formulas accessible to line p.

The invariant is quite easy to maintain. Recall that the sequential processing of a PL/CV proof is equivalent to the traversal of the APL/CV tree. The LCC can be described as a recursive procedure whose input is an APL/CV tree. The nodes of the main branch are processed top down, from the root to the terminus. If the node is an ASSUME P node, a PROCLAIM P instruction is generated. If the node is the conclusion Q of an inference rule, for each hypothesis P_i of the rule, the instruction REQUIRE P_i is generated to check that P_i is accessible to Q. Then a PROCLAIM Q instruction is generated so that Q will be added to the set of accessible formulas.

When a proof block is encountered (see Figure 3.20), an OPEN instruction is generated to save the state of the Assertion Table at point a. Then the V-code generator is called recursively on the subtree contained in the proof block. Upon return, the Assertion Table will contain the set of formulas accessible from the terminus of the proof block. Since the set of formulas accessible from point a is the same as the set accessible from point b, a CLOSE instruction is generated to restore the Assertion Table to the state it was in at point a.

The above translates quite simply into a recursive descent parsing algorithm. For example, the Implication Introduction rule is shown in Figure 3.21a, and the generated V-code is in Figure 3.21b. Note the PROCLAIM P => Q instruction is generated after the CLOSE instruction because the formula P => Q is not accessible from any of the lines within the proof block.

3.13 Automatic Rules and Abbreviations

The V-machine described in the previous section is quite easy to implement. However, the PL/CV Proof Checker uses a more powerful V-machine. In particular, the REQUIRE P instruction has been augmented so that it will succeed if P is in the Assertion Table, or if it is deducible from the formulas in the table. Precisely what is meant by "deducible" is the subject of Chapter 7. For the time being, note that the rules And Introduction, And Elimination, Or Elimination, Implication Elimination, Not Elimination, and False Elimination are "automatic," in the sense that if all the hypotheses to one of these rules have been operands of PROCLAIM

P => Q BY INTRO,	OPEN
PROOF;	PROCLAIM P
[ASSUME P;]	. (V-code recursively generated
.	. for the proof block)
.	REQUIRE Q
Q BY ...	CLOSE
QED;	PROCLAIM P => Q
(a)	(b)

Figure 3.21 The Implication Introduction rule and the generated V-code

instructions, then a REQUIRE P instruction, where P is the conclusion of the rule, will succeed.

If a formula is simply written without any justification:

```
P;
```

then the following V-code:

```
REQUIRE P
PROCLAIM P
```

is generated. The effect of the above V-code is to check that P follows from the application of automatic rules to formulas already in the Assertion Table, and then to add P to the Assertion Table so that subsequent rules (automatic or otherwise) can use it as a hypothesis.

Therefore, justification syntax for the automatic rules is unnecessary, and, in an effort to reduce the complexity of the Proof Checker, and none was provided. In retrospect, this omission appears to have been bad idea. There are several reasons why one might want to write, for example:

```
Q BY IMPEL, P, P => Q;
```

even though:

```
Q;
```

and sometimes nothing at all, is sufficient. The automatic rules are fairly powerful, and it is not always clear to a human reader how a particular line is to be derived, even though it is perfectly clear to the Proof Checker. For this reason, the inclusion of proof lines or justifications that are, strictly speaking, unnecessary is often desirable. And even if the unnecessary syntax were never used in practice, its existence would simplify the description and teaching of the system.

Let us reconsider the V-code generated for the Implication Introduction rule. Given the automatic rules, it is not necessary for Q to appear as the last line of the proof block; it is only necessary for Q to be deducible from the formulas that are accessible to the end of the proof block. Therefore, it is perfectly plausible to have a proof of the form:

```
P => Q BY INTRO,
PROOF;
QED;
```

as long as the V-code in Figure 3.21b is successfully executed, that is, as long as Q is deducible from P and the other formulas accessible from the QED. The above proof is silly, and can be abbreviated as:

```
P => Q BY INTRO;
```

for the same V-code will be generated. Note the use of a semicolon to mark the end of the justification.

If the proof of Q is a one-liner:

```
P => Q BY INTRO,
PROOF;
Q BY <justification>;
QED;
```

and Q is a large formula, it can be unpleasant to have to write Q twice. Therefore, the above proof can be abbreviated:

```
P => Q BY INTRO, <justification>;
```

```
P => (Q => (R => (S => P))) BY INTRO.
PROOF;
ASSUME P;
   Q => (R => (S => P)) BY INTRO.
   PROOF;
   ASSUME Q;
      R => (S => P) BY INTRO.
      PROOF;
      ASSUME R;
         S => P BY INTRO.
         PROOF;
         ASSUME S;
         QED;
      QED;
   QED;
QED;
```

Figure 3.22 An example of deeply nested proof blocks

no matter how complex <justification> is, even if it contains a proof block. For example, the proof in Figure 3.22 can be abbreviated as:

```
P => (Q => (R => (S => P))) BY INTRO, INTRO, INTRO, INTRO;
```

by starting with the innermost proof block and working outward, applying an abbreviation rule at each level. The same V-code is generated for the above line as for the proof in Figure 3.22.

There are similar abbreviation rules for every proof rule that uses proof blocks, and there are no restrictions on how they may be combined. For example, the proof in Figure 3.23a can be abbreviated to the proof in Figure 3.23b. Because empty cases can be omitted, the proof can be further abbreviated as:

```
P|Q => Q|P BY INTRO, CASES, P|Q.
PROOF;
QED;
```

or just:

```
P|Q => Q|P BY INTRO, CASES, P|Q;
```

Because the commutativity of disjunction is an automatic rule, the proof:

```
P|Q => Q|P BY INTRO.
   PROOF;
   ASSUME P|Q;
   Q|P BY CASES, P|Q.
      PROOF;
      CASE P;
      P|Q;
      CASE Q;
      P|Q;
      QED;
   QED;
```

(a)

```
P|Q => Q|P BY INTRO, CASES, P|Q.
   PROOF;
   CASE P;
   CASE Q;
   QED;
```

(b)

Figure 3.23 Abbreviating a proof

```
P|Q => Q|P BY INTRO;
```

is also sufficient.

The generation of V-code for abbreviated proofs may appear to be difficult, but it is actually a simple exercise in recursive programming. The LCC contains a recursive routine called JUSTIFY(P) that reads the justification for P and generates the appropriate V-code. That is, after the LCC has read:

```
P BY ...
```

it calls JUSTIFY with P as a parameter to read the rest of the justification.

Here is how JUSTIFY operates. First of all, it reads a token from the input. According to the PL/CV syntax, that token must be the name of a rule of inference. As an illustration, suppose the rule is INTRO. Then JUSTIFY uses the main connective of the formula P to determine which introduction rule is being used. If, for example, P has the form Q => R, Implication Introduction is being used.

Here is how JUSTIFY handles Implication Introduction. No matter what the justification is, the generated V-code must begin with the header:

```
   OPEN
   PROCLAIM Q
```

and end with the trailer:

```
   REQUIRE R
   CLOSE  .
```

What V-code is generated between the header and the trailer depends on what appears in the justification. First JUSTIFY generates the header; what happens next depends on the next token in the input.

If the token is a semicolon, there is no more justification, so the trailer is generated and JUSTIFY returns. The only other token allowed by the syntax at this point is a comma, which is used to indicate that there is more justification. If a comma is found, the token T after the comma, is peeked at but not actually read. (That is, the token will be available to subsequent read operations.)

If T is PROOF, the user has supplied a proof block. The PROCLAIM Q corresponding to the optional ASSUME Q has already been generated as part of the header; the only processing performed on a ASSUME line following the PROOF is to check that Q is the formula on that line. Next, a routine is called to parse and generate V-code for the contents of the proof block. Note that during the course of the V-code generation for the proof block, JUSTIFY can be called many times; it is a _very_ recursive routine. A proof block always ends the justification; after the proof block has been read and V-code generated, the trailer is generated and JUSTIFY returns.

If the token T is not PROOF, the remaining justification is part of an abbreviation. In that case, JUSTIFY(R) is called recursively to read and and generate V-Code for the remaining justification. If, for example.

the rest of the justification begins with the rule name INTRO. which introduction rule is applied depends on the main connective of R. After the recursive call JUSTIFY(R) returns, the trailer is generated, and JUSTIFY(P) returns.

3.14 Suggestions for Improvement

I am rather pleased with the abbreviation rules. No matter how powerful the automatic rules are made, there will always be proofs that the automatic rules can almost, but not quite, find. Often the abbreviation rules can be used to sketch an outline of the proof, and the automatic rules will fill in the details. I would like to see more work done on the design of flexible "justification languages," in which directions that automated theorem provers should proceed can be expressed. One such approach are the "tacticals" of LCF [32].

The abbreviation rules also encourage a kind of top-down refinement style of proofs reminiscent of structured programming. One way of seeing this influence is by looking at two examples of nonstructured proofs.

The PL/CV SOME Introduction rule is:

```
SOME x FIXED. P BY INTRO, t;
```

where P[t//x] is a hypothesis. If P[t//x] cannot be obtained from automatic rules, a proof of P[t//x] must precede the SOME Introduction line. This is an example of the proof-before-theorem style of APL/CV that the PL/CV syntax was designed to avoid. One improvement would be to extend the syntax to allow:

```
SOME x FIXED. P BY INTRO, t, <justification of P[t//x]>;
```

in cases where a one-line justification is possible.

The following pattern occurs quite often in PL/CV proofs:

```
X<=0 | X>0 BY ARITH;
P BY CASES, X<=0 | X>0,
PROOF;
CASE X <= 0;
.
.
.
```

The above is another example of the proof-before-theorem style. The proof of X<=0 | X>0 is really a part of the proof of P, and belongs somewhere after P is mentioned. Some syntax is needed to be able to formally say, "P BY CASES, X<=0 | X>0 (which is true by ARITH), PROOF..."

The abbreviation rules allow one to move away from the practice of writing formal proofs. Instead of demonstrating that a formal proof exists by exhibiting one, a description of how a proof can be obtained is given. We shall return to this idea in section 8.3.

CHAPTER 4
PROGRAMS

4.1 Proofs of Programs

The primary purpose of the PL/CV system is to check proofs of programs. But to understand how the system checks proofs of programs, one must have an understanding of just what is meant by a proof of a program.

A program can be defined as a collection of "points" and "atomic actions." One point is singled out as the "start" of the program; another is singled out as the "finis." The execution of the program consists of the execution of a sequence of atomic actions. Between the execution of two atomic actions, we say that control is at some "point" in the program. Whenever control is at some point other than the finis, there is always a unique atomic action that will be executed next.

The state of a program is defined to be a function that maps variables of the program to their values. There are two kinds of atomic actions:

(1) Unconditional actions, in which a function is applied to the state to obtain a new state. and control is transferred to the (lexically) next point in the program.

(2) Conditional actions, in which a new point is selected on the basis of some function of the state, and control is transferred to that point.

A special case of (2) is the atomic action that always transfers control to the same point, independent of the state.

For every point in the program, there is a corresponding place in the text of the program. There is no harm in being sloppy about distin-

```
/* a */
IF A > B
THEN DO; /* b */ MAX := A; /* c */ END;
ELSE DO; /* d */ MAX := B; /* e */ END;
/* f */
```

Figure 4.1 A program fragment with its points labeled

guishing places from points. In this chapter, points in programs will be labeled with lower case letters enclosed in comment brackets. For example, Figure 4.1 contains a program fragment with its points labeled. The atomic actions that follow points a through e in Figure 4.1 are:

(a) Evaluate the Boolean expression A > B. If the result is '1'B, transfer control to point b, and if the result is '0'B, transfer control to point d.

(b) Evaluate the expression A and store the value in MAX. (Control is implicitly transferred to point c, the point that lexically follows point b.)

(c) Transfer control to point f.

(d) Evaluate the expression B and store the value in MAX. (Control is implicitly transferred to point e, the point that lexically follows point d.)

(e) Transfer control to point f.

Given a program, an assertion is defined to be an ordered pair (P, q), where P is a formula, and q is a point in the program. The formula P will in general contain variables from the program.

Validity can be defined for assertions in a fashion similar to the way validity is defined in mathematical logic [31] for formulas. A "structure" is given that assigns meanings to all the operators, constants, and function symbols that can appear in formulas. The structure

```
IF A ~= 0
THEN DO;
     /* q */
     :
```

Figure 4.2 A program fragment

can be used to determine the validity of any formula that contains no free variables. A state s is said to "satisfy" a formula P whenever substituting for every free variable v in P the value of v specified by s yields a valid formula. The formula P is defined to be valid whenever every state s satisfies P.

When control is at a particular point q in the program, some states cannot occur. For example, at point q in the program fragment in Figure 4.2, a state in which the value of the variable A is zero is impossible. An assertion (P, q) is defined to be valid whenever P satisfies all the states s that are possible when control is at point q. For example, given the program fragment in Figure 4.2, (A ~= 0, q) is a valid assertion.

For each programming construct, there are proof rules of the form:

$$(P_1, q_1), (P_2, q_2), \ldots (P_n, q_n) \vdash (R, s)$$

where for some nonnegative n, $q_1, q_2, \ldots q_n$ and s are points in the program that are located in the program relative to an instance of the construct. The proof rules are actually theorems about the semantics of the programming language. These theorems state that if the hypotheses of the rule are all valid assertions, then the conclusion of the rule must be a valid assertion as well. For example, the proof rules for the IF statement are shown in Figure 4.3.

Here is what it means to "prove" a program. If s is the start of the program, one assertion of the form (P, s) is assumed; the idea being

```
/* a */
IF B
THEN DO;
     /* b */
     :
     /* c */
     END;
ELSE DO;
     /* d */
     :
     /* e */
     END;
/* f */
```

(P, a) ⊢ (P, b)
(P, a) ⊢ (P, d)
⊢ (B, b)
⊢ (~B, d)
(P, c), (P, e) ⊢ (P, f)

Figure 4.3 Proof Rules for the IF statement

that the program will only be begun in states that satisfy P. Using (P, a) as a starting point, the proof rules can be used to derive other assertions. The usual goal is to derive some assertion (Q, f), where f is the finis of the program, that says what the program is supposed to do.

However, there is another set of "required" assertions that must be proved. For example, given the assignment statement:

```
/* r */
A := B/C;
```

the assertion (C ~= 0, r) must be proved to demonstrate that when control reaches point r the program does not immediately abort. That (P, q) is one of these required assertions will be designated by writing:

(P, q) ⊢ .

The above can be thought of as a proof rule that does not give back a valid assertion in exchange for being provided with one. It simply must be satisfied for the overall proof to be correct.

There are also some required assertions associated with loops. For every loop, a loop invariant P must be specified. Some of the proof rules

```
/* a */                          (P, a) ⊢
DO WHILE(B);
   /* b */                       ⊢ (P, b)
   :                             ⊢ (B, b)
   :                             (P, c) ⊢
   /* c */
END;                             ⊢ (P, d)
/* d */                          ⊢ (~B, d)
```

Figure 4.4 Some of the proof rules for the WHILE loop

for the WHILE loop are shown in Figure 4.4. It is somewhat abusive to refer to the rules involving P as proof rules. It is not, for example, a theorem that (P, b) is a valid assertion. The actual proof rule is closer to:

$$(P, a), [(P, b) \vdash (P, c)] \vdash (P, d)$$

in which one deduction is used as a hypothesis to another deduction. But the proof checker verifies that the actual proof rule has been properly used by checking the rules in Figure 4.4. These rules are based on a model in which before the loop body is executed, the state is tested to see if it satisfies P, and if it does not the program aborts.

When writing PL/CV proofs of programs, one does not use the notation (P, q) to write assertions. Instead, one writes the formula P at the point q in the program. The formula R may be written at point s, (meaning the assertion (R,s) is to be proved) if the formulas P_1, P_2, ... P_n each have been written at q_1, q_2, q_n, respectively, and if

$$(P_1, q_1), (P_2, q_2), \ldots (P_n, q_n) \vdash (R, s)$$

is a proof rule. Because the proof rules all have the property that for each i in {1 ... n}, point q_i lexically precedes the point s in the program, there are no problems with circularity. Once the formulas R_1, R_2, ... R_m have been written at point s, predicate calculus proofs of other formulas $S_1, S_2, \ldots S_k$ that use $R_1, R_2, \ldots R_m$ at hypotheses can be written at point s; for each i in $\{1 \ldots k\}$, (S_i, s) is a valid assertion.

4.2 Shielding

Occasionally, after a program has been proved correct it is compiled and executed. The effort required to produce a document that has been certified to be free of human errors (not counting the humans that produced the programming logic, the Proof Checker, the machine on which the Proof Checker is run, etc.) will have been wasted if a human must then modify this document to obtain a program acceptable to a compiler. There are two solutions to this problem:

(1) design the input to the Proof Checker so that it is also acceptable to the compiler, or

(2) have the Proof Checker produce from the proof a program that is acceptable to the compiler.

In hindsight solution (2) is superior, but the Proof Checker was designed using solution (1). All the assertions and predicate calculus proofs in the input must be "shielded" from the compiler by the special comment brackets /*/ and */.

The syntax of PL/CV does not specify the positions of the shielding brackets. Instead, it is defined assuming different colors of ink are available, so that programs can be written in black ink, and assertions and proofs can be written in red ink. The structure representing a token that is returned by the scanner contains a field indicating its color. Of course in reality, only one color of ink is available. The bracket /*/ can be read "switch to red ink." After reading this bracket, the scanner returns red tokens, until it reads a */, which can be read "return to black ink." (As discussed in section 4.9, the syntax uses a third color of ink, which is introduced by the bracket /*: .) Thus, assertions and proofs can be shielded on a line-by-line basis, or as a block.

The Local Context Checker uses the same linkage to its scanner found in most language translators [34]. There is a global variable that contains the "current symbol." A call to the routine SCAN has the effect of reading the next symbol from the input and storing it in the global variable, where it can be inspected by the various parsing routines. There is another global variable, SHIELD_REQUIRED, that can hold a shielding color or none. Just before the scanner replaces the current symbol, if SHIELD_REQUIRED is not none, the current symbol is tested to see if its color matches SHIELD_REQUIRED. If not, an error is reported and a flag is set that prevents further messages until the color of the tokens is again synchronized with SHIELD_REQUIRED.

Therefore, the parser need not explicitly test each symbol it reads to see if it has the proper color. When it reads the first symbol of a construct that must be completely shielded, it sets the value of SHIELD_REQUIRED to "red." When it reads the first symbol that should not be shielded, it sets SHIELD_REQUIRED back to "black." In the few cases where delicate token-by-token testing must be performed, SHIELD_REQUIRED is set to none.

Shielding brackets are annoying both to write and read. There is only one place in the syntax where the color of the tokens involved is necessary to avoid ambiguity. Because of the questionable decision made by the designers of PL/I to use the same symbol to denote both assignment and equality, the color of the tokens is needed to determine whether:

```
/* p */
B > 0;

A := 0;

/* q */
```

Figure 4.5 A program fragment

A = B;

is an assignment statement or an assertion. In the PL/CV examples in this thesis, all shielding brackets have been omitted, and := is used to denote assignment.

4.3 Accessibility

The proof system described so far is a subset of PL/CV that has all the power of PL/CV, but is not as convenient to use. As an example, consider the proof fragment in Figure 4.5. Suppose the assertion (B > 0, q) is required as a hypothesis to a proof rule. Now given that (B > 0, p) has been established as a valid assertion, a proof rule for the assignment statement can be used to establish (B > 0, q). But recall what it means for (B > 0, p) to be a valid assertion: whenever control is at point p, the variable B contains a positive value. Now the only way that control can reach point q is by way of point p, and along the way the variable B is not changed. Therefore, whenever control is at point q, B must still be positive, so that (B > 0, q) must be a valid assertion.

We say that the assertion (B > 0, p) is "accessible" to the point q. The following definitions formalize the concept of accessibility as used in programs. Recall that in section 3.3, accessibility was defined for predicate calculus proofs. In the logic as defined by Constable and O'Donnell [18], there is actually only one concept of accessibility, but when discussing the Proof Checker it is useful to have two separate concepts.

Definition 4.1. A sequence of points p_0, p_1, ... p_n is defined to be an execution path from p_0 to p_n whenever:

(1) it is possible during the execution of a program for control to reach p_0, and

(2) the execution of the program thereafter consists of the atomic actions AA_1, AA_2, ... AA_n, such that for all i in {1 ... n}, after AA_i has been performed, control is at p_i.

The path is said to pass through a point q whenever q is p_i for some i in {1 ... n-1}. The path is said to pass through an atomic action B if B is AA_i for some i in {1 ... n}. For any point p, there is always a null execution path from p to p that passes through no points.

In a straight line program there will be exactly one execution path from p to q, but in a program that contains conditional actions there will in general be many execution paths from p to q.

Definition 4.2. A "simple execution path" from point p to point q is an execution path from p to q that does not pass through point p.

Definition 4.3. A point p is said to "dominate" a point q whenever every execution path from the start of the program to q passes through p.

Definition 4.4. An assertion (P, p) is "accessible" to point q whenever p and q are not the same point, p dominates q, and no simple execution path from p to q passes through an atomic action that modifies a variable in P. The notation A_q is used to denote the set of all assertions accessible to q.

The definition of accessibility was carefully constructed so that the following theorem holds.

Theorem 4.5. If (P, p) is a valid assertion accessible to point q, then (P, q) is a valid assertion.

Proof. Suppose (P, p) is a valid assertion accessible to point q, and control is at point q. We must show that the state of the program at point q satisfies P. Control reached q along an execution path from the beginning of the program. Because p dominates q, that path must pass through p at least once. If the path is cut the last time it passes through p, the result will be a simple execution path from p to q. Because (P, p) is a valid assertion, the state of the program when control was at point p satisfied P. Since no atomic action along the simple path from p to q can modify any variable in P, the state of the program when control reaches Q must also satisfy P. □

If a point p dominates a point q in a PL/CS program, p must lexically precede q. Given this fact and Theorem 4.5, the requirements on what constitutes a proof can be relaxed. The proof rule:

$$(P_1, q_1), (P_2, q_2), \ldots (P_n, q_n) \vdash (R, s)$$

can be used to introduce the formula R at point s in the program, if P_i has been written at point q_i, or if some assertion (P_i, t_i) is accessible to q_i. Likewise, if an assertion (P, t) is accessible to point s, P can be used as a hypothesis in a predicate calculus proof appearing at s.

Definition 4.6. The union of the sets of assertions accessible to point s, introduced at s by a proof rule for a programming construct, and proved in a predicate calculus proof at s is called the set of formulas "known" at s, and will be denoted K_s.

It is important to note that accessibility is a luxury; the proof rules are sufficiently powerful that if the assertion (P, p) is accessible to point q, then (P, q) can be proved without using accessibility.

```
/* p */
V > 0;

IF   F(X) < F(Y)
THEN V := 0;

IF X >= Y
THEN DO;
     /* q */
     :
```

Figure 4.6 Is (V > 0, p) accessible to q?

The reason it is important that accessibility does not add any power to the logic is that as currently defined, the problem of determining if the assertion (P, p) is accessible to q is undecidable. Consider the example in Figure 4.6. Is the assertion (V > 0, p) accessible to point q? In general the answer is no, because there is an execution path from point p to point q passing through the statement V := 0. However, if the function F is monotonically increasing, no such path is possible. For point q to be reached, it must be the case that X >= Y, and therefore F(X) >= F(Y). But if F(X) >= F(Y), the assignment statement is not executed.

The Proof Checker cannot be expected to determine whether the function F is monotonically increasing. When computing "all execution paths from p to q," it must in general make some approximations; it must include some bogus execution paths that cannot actually occur. The result of this approximation is that some assertions (P, p) that are by Definition 4.4 accessible to a point q in the program will not be recognized by the Proof Checker as being accessible. If the assertion (P, q) is needed as a hypothesis, it must be derived using a proof rule.

Three approximations are made by the the Proof Checker when determining execution paths. Approximation 4.7 is used in APL/CV, the logic as defined by Constable and O'Donnell [20]. The other two approximations have been added so that the Proof Checker can efficiently compute A_p for each point p in the program.

Approximation 4.7. To find all the execution paths from point p to point q, (and some bogus paths as well), begin at p, and trace the flow of execution through unconditional actions. When a conditional action is reached, ignore whatever computation is performed to determine which is the next point in the execution path, and nondeterministically choose any possible outcome of that computation as the next point in the path. Any such sequence by which point q can be reached is assumed to be an execution path.

Approximation 4.8. GOTO, LEAVE, and RETURN statements are presumed to be "unreliable." That is, when an execution path is traced to, for example, the statement GOTO L, the path may be extended through the corresponding label L. However, a second (bogus) path can be extended to the point that lexically follows the GOTO statement. Given the Approximation 4.7, the effect of this approximation is as if every statement of the form:

```
GOTO L;
```

were replaced by:

```
IF '1'B THEN GOTO L;
```

and likewise for LEAVE and RETURN.

Without this approximation, it is possible to have points in a program through which no execution paths pass. By Definition 4.4, every assertion is accessible to these points in the program.

Two examples in which this approximation prevents assertions from

```
/* p */
V > 0;
GOTO L1;

L2:;
V := 0;
GOTO L3;

L1:;
  .
  .
  .
/* q */
  .
  .
  .
L3:;
```

(a)

```
/* p */
V > 0;

IF X > 0
THEN DO;
     V := 0;
     RETURN;
     END;

/* q */
```

(b)

Figure 4.7 Bogus execution paths

being recognized as being accessible are shown in Figure 4.7. In each case, the assertion (V > 0, p) is not recognized as being accessible to point q because a bogus execution path passes through the assignment statement V := 0.

The third approximation is discussed in section 4.16.

4.4 Checking Program Proofs

As was the case with predicate calculus proofs, proofs of programs are checked by generating V-code and executing the resulting code with the V-code Interpreter. The generated V-code can be partitioned into contiguous sections, such that each section corresponds to a point in the program. The V-code section corresponding to a point p has three parts. The first part of the V-code updates the Assertion Table so that it contains A_p. Then, a series of PROCLAIM instructions are generated. There will be one PROCLAIM P for each proof rule of the form:

... ⊢ (P, p)

that is used in the proof. The second part of the V-code consists of the V-code generated for any predicate calculus proofs at point p. After this V-code is executed, the Assertion Table will contain K_p. The third part of the V-code may begin with one or more OPEN instructions that each saves K_p. It ends with a series of REQUIRE instructions. For every proof rule of the form:

... (P, p) ... $\vdash$ ...

used in the program, there is a REQUIRE P instruction.

4.5 The NEW instruction

The first part of the V-code generated for a point q in the program must set the Assertion Table to A_q. This operation often involves deleting assertions that contain a variable v that has been modified by an atomic action preceding q.

To accomplish this task, instructions of the form:

NEW v

are used. The V-machine has an inexhaustible queue of unique names, none of which appear anywhere in the proof being checked. The action specified by the NEW v instruction is: take a variable v' off the unique name queue, and substitute v' for every occurrence of v in the Assertion Table. The effect is to delete the formulas containing v, and then to PROCLAIM some new formulas containing v'.

The reader may wonder why the NEW v instruction does not simply delete all the formulas containing v from the Assertion Table. There are two reasons for performing the substitution instead:

(1) The substitution can be implemented more efficiently than the dele-

tion.

(2) The V-code generated by some rules makes use of the new formulas containing v'.

4.6 Templates

The turnstile notation for proof rules is somewhat difficult to read. A new notation will be used to present the proof rules for each language construct. For each construct a "template" similar to those in Figures 4.3 and 4.4 will be presented, but only the points referenced in the discussion of the construct will be labeled. For each proof rule of the form:

$$(P_1, q_1), (P_2, q_2), \ldots (P_n, q_n) \vdash (R, s)$$

the notation {? P_i} will be written at point q_i in the template, and {! R} will be written at point s. Formulas marked with question marks will usually correspond to REQUIRE instructions in the generated V-code, and those marked with exclamation points will usually correspond to PROCLAIM instructions. For certain constructs, some justification syntax is needed to indicate to the Proof Checker which proof rules are being used. The justification syntax is also shown in the templates.

There are two kinds of language constructs: simple and compound. A template for a simple construct, such as the assignment statement, will take the form shown in Figure 4.8a. A template for a compound construct, such as the IF statement, will take the form shown in Figure 4.8b. These templates have one or more "holes," which are indicated by vertical ellipses.

A PL/CV proof can be formed by laying a series of templates end to

```
                              /* r */
                              {? R}
                              <compound statement beginning>
                              /* s */
                              {! S}
                              .
                              .
                              .
/* p */                       /* t */
{? P}                         {? T}
<simple statement>            <compound statement middle>
/* q */                       /* u */
{! Q}                         {! U}
                              .
                              .
                              .
                              /* v */
                              {? V}
                              <compound statement end>
                              /* w */
                              {! W}

         (a)                             (b)
```

Figure 4.8 The format of templates

end, recursively filling in the holes in any compound templates by the same process. Predicate calculus proofs can be inserted in the interstices between templates. There are two kinds of points in each template: "face-down" points, like q, s, u, and w in Figure 4.8, and "face-up" points like p, r, t, and v. In a proof, each face-down point in a template (except for the last point in the proof) will be followed by a face-up point of another template, possibly separated by zero or more predicate calculus proofs.

V-code is generated for the various templates in the same way that machine code is generated by a compiler. The three parts of the V-code (as described in section 4.4) generated for any point p correspond to a face-down point of a template, some predicate calculus proofs, and a face-up point of another template.

Operator	Domain Predicate
I / J	J ~= 0
I ** J	J >= 0 & (I ~= 0 | J ~= 0)

Table 4.9 Domain predicates for partial operators

4.7 Domain Checking

The arguments to any partial function used in the program must be proved to be in the domain of the function. The partial functions (and operators) that are primitives in PL/CV are division, exponentiation, and array subscripting. In addition, all the user-defined functions are assumed to be partial.

For each function, there is a predicate involving its operands that formally expresses that the operands are in the domain of the function. For division and exponentiation operators, the predicates are shown in Table 4.9. The predicate for each user-defined function is found on the ASSUME line of the function definition (see section 4.12).

The case of arrays deserves special attention. If the array A has been declared by:

DECLARE A(L_1:U_1, L_2:U_2, ... L_n:U_n) type; (4.1)

then for all i in {1 ... n}, the PL/I functions LBOUND and HBOUND have the following values:

LBOUND(A, i) = L_i
HBOUND(A, i) = U_i .

The V-code generated for the declaration (4.1) is shown in Figure 4.10. The reason for the complexity of the V-code is a peculiar restriction in PL/I that every array must have at least one element [48]. That is, each lower bound of an array is required to be less than or equal to the

```
REQUIRE   L1 <= U1
REQUIRE   L2 <= U2
:
REQUIRE   Ln <= Un

PROCLAIM  LBOUND(A, 1) = L1
PROCLAIM  LBOUND(A, 2) = L2
:
PROCLAIM  LBOUND(A, n) = Ln

PROCLAIM  HBOUND(A, 1) = U1
PROCLAIM  HBOUND(A, 2) = U2
:
PROCLAIM  HBOUND(A, n) = Un

PROCLAIM  LBOUND(A, 1) <= HBOUND(A, 1)
PROCLAIM  LBOUND(A, 2) <= HBOUND(A, 2)
:
PROCLAIM  LBOUND(A, n) <= HBOUND(A, n)
```

Figure 4.10 V-code generated for a declaration

corresponding upper bound. When the array is a formal parameter, it is declared by:

```
DECLARE A(*, *, ... *) type;
```

where n asterisks appear. In this case, only the last n instructions in Figure 4.10 are generated.

LBOUND and HBOUND are used to formulate the domain predicates for the subscript operation. The domain predicate for the subscripting operation $A(J_1, J_2, \ldots J_n)$ is:

```
LBOUND(A, 1) <= J1 & J1 <= HBOUND(A, 1) &
LBOUND(A, 2) <= J2 & J2 <= HBOUND(A, 2) &
:
LBOUND(A, n) <= Jn & Jn <= HBOUND(A, n)  .
```

Note that the bound expressions that appear the declaration cannot be used in this predicate, since variables in these expressions may be modified between the declaration and use of the array.

Although LBOUND and HBOUND are written as two-place functions, they are in reality one-place functions. The value of LBOUND(A, I) does not depend on the value of A, since no modification a program makes to A can change the value of LBOUND(A, I) (assuming I does not depend on A). For every array A, the proof checker creates two unique function names $LBOUND_A$ and $HBOUND_A$. It translates every occurrence of LBOUND(A, J) to $LBOUND_A(J)$, and likewise for HBOUND. The V-code instruction NEW A does not alter the names $LBOUND_A$ and $HBOUND_A$ when they appear in an assertion, but it would, for example, change LBOUND(A, J) to LBOUND(A', J[A'//A]).

The functions LBOUND and HBOUND, which are partial, may appear in programs. The domain predicate for LBOUND(A, J) and HBOUND(A, J) is 1 <= J & J <= n, where n is the number of dimensions of A. The Proof Checker can always determine the numeric value of n.

If an atomic action in which a partial function is applied to arguments x_1, x_2, ... x_n is executed after control reaches point p, and P is the domain predicate for that function, the assertion ($P(x_1, x_2, \ldots x_n)$, p) must be proved. In the templates, the abbreviation {Dom t} will be used to mean that the domain assertions for every partial function in the expression t are required assertions.

When exhibiting V-code, the ficticious instruction DOMAIN t will be used. The DOMAIN instruction is a shorthand for zero or more REQUIRE instructions, one for each partial function application in t. For example, the notation:

```
DOMAIN    A(K**J / M) > 0
```

is an abbreviation for the sequence:

```
REQUIRE   J >= 0 & (K ~= 0 | J ~= 0)
REQUIRE   M ~= 0
REQUIRE   (LBOUND(A, 1) <= K**J / M) & (K**J / M <= HBOUND(A, 1))  .
```

Another problem with partial functions is defining what they mean when used in assertions. For example, the formula:

```
ALL (X, Y) FIXED. (X * Y)/X = Y                              (4.2)
```

would appear to be invalid, since for any Y, the value of (0 * Y)/0 is undefined. Yet it seems there should be formula very much like (4.2) that is valid. In an attempt to rule out the case in which X = 0, we might try:

```
ALL (X, Y) FIXED. (X ~= 0 => (X * Y)/X = Y)                  (4.3)
```

Yet how is formula (4.3) any more meaningful than formula (4.2)? We still have the problem of giving meaning to the formula 0 ~= 0 => (0 * Y)/0 = Y.

The solution used in PL/CV is never to reason about partial functions. The semantics of PL/CV are such that, for example, the division operator is actually a total function that has the same value as the familiar partial function on the domain of the partial function. Since a PL/CV program that has been proved correct only calls the total function using operands that are in the domain of the partial function, we can cheat when we implement division in the programming language, and use the partial function.

No axioms are provided to specify the value of the total function outside the domain of the partial function. Thus, formula (4.3) can be proved in PL/CV, since it is valid no matter what the value of 0/0 is. But formula (4.2) can neither be proved nor disproved in PL/CV, because all that can be proved about the value of 0/0 it that is an integer.

This same trick is used with all the partial functions. However, in

```
ONE   := 1;
TWO   := 2;
THREE := 3;
ZERO  := ONE/THREE + ONE/THREE + ONE/THREE;
ONE   := ONE/TWO + ONE/TWO;
```

Figure 4.11 The anomalous PL/I division operator

the case of division the "familiar partial function" is the division operator from FORTRAN [7]. That is, the value of I/J is undefined if J = 0, and otherwise is the integer K closest to zero such that |K| <= |I÷J|, where ÷ is rational division.

Although it is unpleasant to give axioms for FORTRAN's division operator, PL/I's division operator is an axiomatic nightmare. Rather than truncate the result of a division, PL/I maintains as much precision as it can while evaluating an expression, and truncates on assignment. Assuming all the variables in the program fragment in Figure 4.11 have been declared FIXED (which is short for FIXED DECIMAL), the variable ZERO is set to 0 and ONE is set to 1 (twice). The reason for the apparent discrepancy is that one half has a short decimal representation, whereas one third has an infinite one.

In PL/CV, the division operator is only allowed in contexts where its result is immediately truncated. An equivalent way of stating this restriction is that if the division operator is used in an expression, it must be the main connective of the expression. The total function one reasons about in PL/CV is an extension of the FORTRAN division operator. The implementation uses the PL/I operator, but it is not allowed in any context where it can be distinguished from the FORTRAN operator.

Intentional incompleteness can also be used to model uninitialized variables. Accessing an uninitialized variable is not considered a run-time error in PL/CV; nor are variables assumed to be initialized to zero.

```
UDEF: PROCEDURE (X) RETURNS (FIXED);
DECLARE (Y) FIXED;
RETURN(Y);
END UDEF;
```

Figure 4.12 A function whose value depends on a uninitialized variable

If p is the point in the program just after the variable V has been declared, no assertion (P, p), where P depends on the value of V, can be proved or disproved. An implementation is free to initialize a newly declared variable to whatever garbage happens to be on the stack, and any program that has been proved correct will run properly.

Unfortunately, if a routine is called more than once, newly declared variables must be initialized to exactly the same garbage every time. This restriction is required so that, for example, the formula:

ALL X FIXED. UDEF(X) = UDEF(X)

which can easily be proved, is valid given the function definition in Figure 4.12.

The reproducibility restriction is tantamount to requiring an implementation to initialize newly declared variables to some constant. A clever compiler could in some cases detect that operation of a routine is independent of the initial value of one or more of its variables, and omit the initialization of those variables.

Another solution to this problem would be to require that the specification given for each user-defined function imply that it is in fact a function. For example, if the user proves that F(x, y) always returns the greatest common divisor of x and y, then it will do so independent of the value of newly declared variables. But if he only proves that F(x, y) returns a positive value, then different calls to F with the same arguments can potentially return different, though positive,

```
/* p */
{Dom t}
{? P[t//v]}

v := t;

/* q */
{! P}
```

Figure 4.13 Template for the assignment rule

results.

4.8 The Assignment Statement

The template for the PL/CV assignment rule, which is essentially Hoare's assignment axiom [41], is shown in Figure 4.13. The first and second parts of the V-code for point p, which will have been generated for another template and any predicate calculus proofs at p, will set the Assertion Table to K_p. To set the Assertion Table to A_q, it is only necessary to delete all assertions that contain an instance of v. A NEW v instruction can be used to delete these instructions, and replace them with similar assertions, with v' substituted for v.

These new assertions are somewhat peculiar. If v' is interpreted as "the value the variable v had before the assignment statement," the assertions are clearly valid. However, there will never be a REQUIRE Q instruction such that Q contains an occurrence of v'. But recall that a REQUIRE Q instruction will succeed if Q is in the Assertion Table, or deducible via automatic rules from the formulas in the Assertion Table. For example, if the formula x + v*0 = y is in the Assertion Table before the V-code generated for the assignment statement is processed, the NEW instruction will change it to x + v'*0 = y. Subsequently, the instruction REQUIRE x = y will succeed. Extra deductions such as this one, which are not mentioned

```
DOMAIN    t
REQUIRE   P[t//v]
NEW       v
PROCLAIM  P
```

Figure 4.14 Adequate V-code for the assignment statement

the PL/CV Reference Manual [17] are, depending on one's point of view, a feature of or a bug in the Proof Checker. However, extra deductions that use the automatic substitution rules are the basis for the method used for checking the assignment rule.

A V-code sequence that would adequately check the assignment rule is shown in Figure 4.14. The drawback to this sequence is that it is necessary to know what P is to generate it. Some justification syntax, such as:

```
L: v := t
:
P BY ASSIGN, L;
```

would be necessary for each use of the assignment rule. Not only would this be cumbersome for the user, but to generate the proper V-code, either the BY ASSIGN lines would be required to immediately follow the assignment statement (in which case the labels would be unnecessary), or a fixup technique would have to be used to insert the REQUIRE instructions that must come before the NEW v statement.

A different approach is used in PL/CV. The assignment rule is completely automatic; no justification of any kind is needed. Instead of the V-code in Figure 4.14, the following V-code:

```
DOMAIN    t
NEW       v
PROCLAIM  v = (t[v'//v])
```

is used, where v' is the variable with which the NEW instruction replaces every occurrence of v. The name v' is chosen by the V-code interpreter in such a way that the Local Context Checker can predict what it will be (see section 7.4 for more details).

Recall that the interpretation for v' is "the value the variable v had before the assignment statement was executed." For example, for the assignment statement J := J+1, the V-code instruction:

PROCLAIM J = J'+1

will be generated, meaning, "henceforth the value of J will be its previous value plus one."

Proofs that use the assignment rule in Figure 4.13 are accepted because of an automatic substitution rule built into the V-code Interpreter. If the formulas P[t//s] and s = t are in the Assertion Table, the instruction REQUIRE P will succeed as a consequence of this automatic rule. Suppose the assertion:

P[t//v] (4.4)

is in K_p. Then formula (4.4) must be in the Assertion Table when the NEW v instruction is executed. This instruction will delete formula (4.4) and replace it with:

P[t//v][v'//v] (4.5)

and the PROCLAIM instruction will add:

v = (t[v'//v]) (4.6)

to the Assertion Table. The metanotation used in (4.5) means: first replace every occurrence of v in P by t, and then replace every occurrence of v in the result by v'. Any occurrences of v remaining after the first replacement must be in the copies of t that were inserted. These occurrences of v are changed to v' by the second replacement. If the occurrences of v in t are changed to v' before it is copied and inserted into P, the resulting formula will be the same. That is, the formula represented by (4.5) and by:

$$P[t[v'//v]//v] \tag{4.7}$$

are identical.

After the V-code generated for the assignment statement is executed, a REQUIRE P instruction will succeed as long as formulas (4.6) and (4.7) are in the Assertion Table, since the substitution rule can bo applied to those two formulas to obtain P. The deduction of P from formulas (4.6) and (4.7) can be performed even after (4.6) and (4.7) have been updated by a NEW w instruction, as long as w is not a variable in P, since the only part of the two formulas that can be changed are copies of t[v'//v], and all copies are changed in the same manner. Therefore the REQUIRE P will still succeed after a NEW v instruction has been performed, unless v is a variable in P.

Because the V-code Interpreter can deduce any formula obtained by any sequence of substitutions, applications of the assignment rule can be chained. For example, for the sequence of assignment statements:

```
/* p */
A := A + B;
B := B * C;
C := A + B;
/* q */
```

the V-code:

```
NEW        A
PROCLAIM   A = A' + B
NEW        B
PROCLAIM   B = B' * C
NEW        C
PROCLAIM   C = A + B
```

will be generated. Suppose the assertion (((A + B) + (B * C)) > 0, p) is accessible to p. After the above V-code is executed, the formulas:

$$((A' + B') + (B' * C')) > 0 \tag{4.8}$$

$$A = A' + B' \tag{4.9}$$

$$B = B' * C' \tag{4.10}$$

$$C = A + B \tag{4.11}$$

will be in the assertion table. The assertion (C > 0, q) can be derived by three applications of the assignment rule. But because C > 0 can be derived by performing substitutions on formulas (4.8-11), the V-code instruction REQUIRE C > 0 will succeed.

4.9 Array Assignment

An array A in PL/CV is a quintuple of the form (N, L, U, F, Z). The component N, the number of dimensions, is a positive integer. The components L and U, the lower and upper bounds, are vectors of N integers, such that for each i in {1 ... N}, $L_i \le U_i$. The component F is a function whose domain is the set of vectors $x_1, x_2, \ldots x_n$ such that for each i in {1 ... N}, $L_i \le x_i \le U_i$ and whose range is the base type of the array. The component Z is a value whose type is the base type of the array.

When applied to an array, the HBOUND and LBOUND functions return the appropriate element of its L or U component. The subscripting operation, when applied to an array and a vector X returns F(X) if X is in the domain of F, and Z otherwise. Because of domain checking (see section 4.7), no

```
REQUIRE   LBOUND(A, 1) = LBOUND(B, 1)
REQUIRE   LBOUND(A, 2) = LBOUND(B, 2)
   .
   .
   .
REQUIRE   LBOUND(A, n) = LBOUND(B, n)
REQUIRE   HBOUND(A, 1) = HBOUND(B, 1)
REQUIRE   HBOUND(A, 2) = HBOUND(B, 2)
   .
   .
   .
REQUIRE   HBOUND(A, n) = HBOUND(B, n)
NEW       A
PROCLAIM  A = (B[A'//A]).
```

Figure 4.15 V-code generated for the array assignment A := B

program that has been proved correct can apply the subscripting operation to a vector outside the domain of F. Therefore, an implementation of the programming language need not store Z.

There are three kinds of assignment statements involving arrays. The template for the first kind of array assignment is:

```
{? P[B//A]}
A := B;
{! P}
```

where A and B are both arrays with the same domain. This assignment statement is essentially the same as the one presented in the previous section. The V-code generated for such an assignment statement appears in Figure 4.15. The only difference between this assignment statement and scalar assignment is that REQUIRE statements must be generated to check that the domains of the two arrays are the same. The Local Context Checker must also ensure that A and B have the same number of dimensions.

The template for the second array assignment statement is:

A := c;
{! ALL (J_1, J_2, ... J_n) FIXED. A(J_1, J_2, ... J_n) = c}

where c is a constant of the appropriate type; the corresponding V-code is in Figure 4.16. The semantics of this statement is to modify the F com-

```
NEW        A
PROCLAIM  ALL (J1, J2, ... Jn) FIXED. A(J1, J2, ... Jn) = c
```

Figure 4.16 V-code generated for the array assignment A := c

ponent of A to be the constant function whose value is everywhere c, and the Z component of A to be c. Since the program can never determine the value of the Z component, it should be impossible to prove anything about the value of the array outside its domain. A more diligently incomplete rule would be:

```
A := c;
{! ALL (J1, J2, ... Jn) FIXED.
         (( LBOUND(A, 1) <= J1 & J1 <= HBOUND(A, 1)
          & LBOUND(A, 2) <= J2 & J2 <= HBOUND(A, 2)
            .
            .
            .
          & LBOUND(A, n) <= Jn & Jn <= HBOUND(A, n)) =>
                                  A(J1, J2, ... Jn) = c)}
```

but the temptation to simplify the rule was irresistible.

The third kind of assignment statement is of the form:

$A(x_1, x_2, \ldots x_N) := y;$

In the following discussion, N will assumed to be one, and x_1 will be written as x. The case in which N is greater than one is messy, but a straightforward generalization.

The semantics of this statement is to modify the F component of A so that it maps the value x to y. A proof rule that reflects these semantics is shown in Figure 4.17. ALT(A, x, y) is defined to be the array A with the F component modified so that it maps x to y. The following axioms:

```
{? Dom A(x)}
{? Dom y}
{? P[ALT(A, x, y)//A]}
A(x) := y;
{! P}
```

Figure 4.17 A proof rule for array assignment

```
ALL (K, X) FIXED, (A(*), Y) type.
    (LBOUND(A, 1) <= X & X <= HBOUND(A, 1) & K ~= X                 (4.12)
       => ALT(A, X, Y)(K) = A(K))

ALL X FIXED, (A(*), Y) type.
    (LBOUND(A, 1) <= X & X <= HBOUND(A, 1)                          (4.13)
       => ALT(A, X, Y)(X) = Y)

ALL X FIXED, (A(*), Y) type. LBOUND(A, 1) = LBOUND(ALT(A, X, Y), 1)  (4.14)

ALL X FIXED, (A(*), Y) type. HBOUND(A, 1) = HBOUND(ALT(A, X, Y), 1)  (4.15)
```

are sufficient to characterize the function ALT.

The proof rule in Figure 4.17 has been derived by a process I call reduction. Given a language construct C that can be expressed in terms of other constructs C_1, C_2, ... C_n for which proof rules are known, a proof rule for C can be derived from the rules for C_1, C_2, ... C_n. In this case, array assignment can be expressed in terms of scalar assignment and the ALT function. The resulting rule for C will be sound if the rule for each C_i is sound, and it will be as complete as the rules C_1, C_2, ... C_n are.

Certainly it is necessary that a proof rule be sound, and desirable that it be complete as possible. But soundness and completeness are not the ultimate yardsticks with which proof rules should be measured. A good proof rule for a construct C is one in which a programmer's intuition about C can be easily expressed. In a poor proof rule for C, the programmers's intuition must be encoded.

Proof rules that are derived using reduction are not necessarily good ones. We must ask ourselves the question, "If the construct C can be expressed in terms of the constructs C_1, C_2, ... C_n, why does the construct C appear in the language at all?" Sometimes the answer is that the designer of the language simply did not realize that C could be expressed in terms of other constructs. If it is truly easier to understand C as a combination of C_1, C_2, ... C_n, then C should be eliminated from the language. But more often the answer is that when C is built up from C_1, C_2, ... C_n, the resulting whole is somehow conceptually simpler than the parts. Any proof rule for C that does not reflect this simplification will be unpleasant to use.

The intuition behind the proof rule in Figure 4.17 is that an array is a single object, and that array assignment statement:

```
A(x) := y;
```

is so-called syntactic sugar for:

```
A := ALT(A, x, y);  .
```

But one could just as easily claim that there is only one true variable, an array called STATE, in any program, and that variables programmers write are just constants used to index into this array. The assignment statement:

```
v := y;
```

is just syntactic sugar for:

```
STATE := ALT(STATE, v, y);
```

and the array assignment statement:

```
{Dom A(x)}
{Dom y}
{? P[y//A(x)]}
A(x) := y;
{! P}
```

Figure 4.18 A more intuitive array rule

```
{Dom  A(x)}
{Dom y}
{? P[y//A(x)]}
{? B = A}†
A(x) := y;
{! P}
{! ALL J FIXED. (J ~= (x[B//A]) => A(J) = B(J))}
```

†B and A cannot be the same array

Figure 4.19 The PL/CV rule for A(x) := y

```
A(x) := y;
```

is just syntactic sugar for:

```
STATE := ALT(STATE, A+x, y) .
```

This view of assignment is used in many semantic descriptions of programming langauges, and it can be helpful when attempting to construct compilers for or to prove properties of the langauge. But it is not very helpful when attempting to establish the correctness of a particular program.

I propose viewing an array as a collection of variables. An array assignment statement selects and modifies a particular variable in the collection. A rule suggested by this alternate view is shown in Figure 4.18. But this rule is not sufficiently complete to prove most programs that use arrays. The rule used in PL/CV is shown in Figure 4.19, and the corresponding V-code is shown in Figure 4.20. An advantage of the rule in Figure 4.19 over the Rule in Figure 4.18 is that the ALT notation and the

```
DOMAIN    A(x)
DOMAIN    y
NEW       A
PROCLAIM  A(x[A'//A]) = (y[A'//A])
PROCLAIM  ALL J FIXED. (J ~= (x[A'//A]) => A(J) = A'(J))
```

Figure 4.20 V-code generated for A(x) := y

rules that must come with it are not necessary.

The disadvantage to the rule in Figure 4.19 is that in general there will not be an array B that is equal to tho array A. In almost all cases, an extra array B must be declared with the same bounds as A, and the statement B := A must be written prior to every statement A(x) := y. The array B is an "auxiliary variable" similar to those used by Owicki and Gries [63] to prove parallel programs.

So that this extra array and the extra assignment statements, which are needed only for the proof, do not reduce the efficiency of the program, the declaration of the array B and the assignments to it can be enclosed in the shielding brackets /*: and */. As long as the rules:

(1) auxiliary variables appear only in assertions and shielded statements, and

(2) nonauxiliary variables are not modified in shielded statements, and

(3) if p and q are points in the beginning and end of a block B of shielded statements, the only execution paths into B pass through p, and the only execution paths out of B pass through q

are obeyed, any programming constructs can be shielded from the compiler. Since none of the examples in this thesis are intended for execution, shielded statements will not be used.

The following example will be used as a benchmark to compare the two array rules. Suppose we wish to write a loop to set each element A(K) of

```
GIVEN: ALL K FIXED. (1 <= K < J => A(K) = K+5)

ALL K FIXED. (1 <= K < J+1 => ALT(A,J,J+5)(K) = K+5) BY INTRO, INTRO.
PROOF;
ASSUME 1 <= K < J+1;

    K = J | K < J BY ARITH, K < J+1;
    ALT(A,J,J+5)(K) = K+5 BY CASES, K = J | K < J,
    PROOF;
    CASE K = J;
        ALT(A,J,J+5)(J) = J+5 BY ALLEL, <Axiom (4.13)>, J, A, J+5;
    CASE K < J;
        J ~= K BY ARITH, K < J;
        ALT(A,J,J+5)(K) = A(K) BY ALLEL, <Axiom (4.12)>, K, J, A, J+5;
        A(K) = K+5 BY ALLEL, GIVEN, K;
    QED;
QED;

A(J) := J+5;

ALL K FIXED. (1 <= K < J+1 => A(K) = K+5);
```

Figure 4.21 A proof using the ALT rule

```
GIVEN_A: ALL K FIXED. (1 <= K < J => A(K) = K+5);

B := A;

GIVEN_B: ALL K FIXED. (1 <= K < J => B(K) = K+5);

A(J) := J+5;

ALL K FIXED. (1 <= K < J+1 => A(K) = K+5) BY INTRO, INTRO,
PROOF;
ASSUME 1 <= K < J+1;
    K = J | K < J BY ARITH, K < J+1;
    A(K) = K+5 BY CASES, K = J | K < J,
    PROOF;
    CASE K = J;†
        A(J) = J+5;†
    CASE K < J;
        K ~= J BY ARITH, K < J;
        B(K) = K+5 BY ALLEL, GIVEN_B, K;
        A(K) = B(K) BY ALLEL, ALL L FIXED. (L ~= J => B(L) = A(L)), K;
    QED;
QED;
```

†These lines may be omitted, but were included for clarity

Figure 4.22 A PL/CV Proof

the array A to K+5. To prove such a loop, given the assertion:

ALL K FIXED. (1 <= K < J => A(K) = K+5)

before the assignment statement:

A(J) := J+5

it will be necessary to prove that:

ALL K FIXED. (1 <= K < J+1 => A(K) = K+5)

is valid afterwards. This proof would be one step (or part of a step) of an informal proof. Proofs using the rules in Figures 4.17 and 4.19 appear in Figures 4.21 and 4.22, respectively. The proof in Figure 4.16 was written assuming axioms (4.14) and (4.15), but not axioms (4.12) and (4.13) are automatic rules. In each proof J is assumed to be within the bounds of the array.

Comparing the two proofs, the PL/CV rule appears to be no better than the rule involving ALT; the proofs require eighteen and seventeen lines, respectively. (The length of a proof is not a particularly good metric, but it is at least objective). Two lines of the proof in Figure 4.22 can be omitted, but in a system tuned to the ALT rule, the corresponding lines could be omitted from the proof in Figure 4.21.

What has gone wrong? The unpleasant part of the PL/CV proof, where the intuition must be encoded rather than expressed, is in the K < J case. At that point in the proof the assertion GIVEN is still valid, since it refers to a part of the array that is not modified by the assignment statement. In a good proof system for arrays, the assertion GIVEN would be accessible to the point just past the assignment statement. In such a system, the proof in Figure 4.23 (which takes advantage of abbreviations and automatic rules) could be used. This proof is similar the proof that

```
GIVEN: ALL K FIXED. (1 <= K < J => A(K) = K+5);

A(J) := J+5;

ALL K FIXED. (1 <= K < J+1 => A(K) = K+5) BY INTRO, INTRO,
PROOF;
ASSUME 1 <= K < J+1;
    K = J | K < J BY ARITH, K < J+1;
    A(K) = K+5 BY CASES, K = J | K < J,
    PROOF;
    CASE K<J BY ALLEL, GIVEN, K;
    QED;
QED;
```

Figure 4.23 A goal not yet attained

would be used for a loop invariant that did not involve reasoning about arrays.

This example exhibits the difference between the two views of array assignment. If array assignment is viewed as replacing the entire array with a new array, then no assertion involving that array can be accessible past the assignment statement. On the other hand, if the array assignment is viewed as selecting and modifying one of a class of variables, and there is an assertion P that depends only on those variables that were not modified, then in a sufficiently powerful system, P can be accessible to points beyond the assignment.

Although the proof in Figure 4.23 is better than the proof in Figures 4.21 and 4.22, it is not dramatically better. The benchmark problem was chosen for its simplicity; it does not adequately convey the failings of the two array rules. In moderately complex programs, such as the QUICK-SORT [21] example A Programming Logic, an array is divided into various segments, and assertions are made about each segment. Each time an array assignment modifies one of these segments, assertions about the other segments must be proved to be left invariant, even though informally it would hardly occur to anyone that the validity of these assertions could be

```
name: PROCEDURE(A, B, C ...);
DECLARE ...   (parameters and external varisbles)
ASSUME P;
{! P}
ATTAIN E;
  .
  .          (predicate calculus proof)†
  .
{? SOME s FIXED. (s >= 0 & T)}†
ARBITRARY s FIXED WHERE T;†
{! T}†
DECLARE ...  (local variables)
  .
  .          (procedure body)
  .
{? '0'B}
END;
```

†omitted in nonrecursive procedures

Figure 4.24 Template for a procedure definition

affected by the assignment. In a good proof system for arrays, the notion of accessibility would be strengthened so that these assertions would not have to be reproved. Ideally, the notion of accessibility would be strengthened to the point where the rule in Figure 4.18 would be sufficiently complete.

4.10 Procedure Definitions and Calls

The description of a program presented in section 4.1 was simplified somewhat. What was described in that section as a program is in fact a routine, either a procedure or a function. A PL/CV program is a collection of routines that call each other.

Figure 4.24 contains the template for a procedure definition. Two predicates, the input assumption P, and the exit condition E, are associated with every procedure. A third predicate, T, the termination predicate, is associated with recursive procedures. These predicates will in general depend on the procedure parameters and external variables, but they cannot depend on local variables. This restriction is enforced by having the "declarations" of P, E, and T precede the declaration of local variables in the syntax.

In a program that has been proved correct, the procedure will never be called unless its parameters and external variables satisfy P. If f is the finis of the procedure, the assertion (E, f) is proved valid by making (E, p) a required assertion for every point p that precedes a RETURN statement.

In PL/CS, the body of every procedure and function is required to end in a RETURN statement. A slightly less stringent restriction is enforced by PL/CV by requiring '0'B at the end of the procedure body. Recall that if an assertion (P, p) has been proved to be valid, then whenever control reaches p, the state must satisfy P. Since no state can satisfy '0'B, if ('0'B, p) has been proved valid, control can never reach p. The required assertion '0'B at the end of the procedure body guarantees that control cannot flow off the end of the procedure; the only ways out of the procedure body are via RETURN statements before which E has been shown to be valid.

The predicates P and E are used to give a specification of the procedure. As a simple example, consider the following procedure heading:

```
MAX: PROCEDURE(A, B, M);
DECLARE (A, B, M) FIXED;
ASSUME '1'B;
ATTAIN M>=A & M>=B & (M=A | M=B); .
```

This specification could be satisfied by a procedure body that simply assigns the same value to A, B, and M. To pin down the desired functionality, some way of ensuring that A and B are not modified by the procedure is needed. To correct the specification, the declaration is changed to:

```
DECLARE (A, B) FIXED READONLY;
DECLARE M FIXED; .
```

We say that A and B are readonly parameters, whereas M is a readwrite parameter. The Local Context Checker reports an error if a procedure directly modifies a readonly variable.

<u>Definition</u> 4.9. A procedure is defined to "directly modify" a variable V whenever V appears on the left side of an assignment, is used as a loop index, or is passed by reference to a readwrite parameter of a procedure.

Every external variable that appears in the body of procedure must be declared in the procedure header. External variables can be declared with the READONLY attribute, in which case the Local Context Checker ensures they are not directly modified by the procedure body. However, a readonly external variable can be modified by calling some other procedure in which it is declared readwrite.

<u>Definition</u> 4.10. If R_1 and R_2 are routines, define the relation $R_1 \Rightarrow R_2$ to hold whenever there is a call to R_2 in the body of R_1, define $\Rightarrow+$ to be the transitive closure of $\Rightarrow$, and define $\Rightarrow*$ be the reflexive transitive closure of $\Rightarrow$. If R_1 is a routine, and V is an external variable, R_1 is defined to "depend on" V whenever there is a routine R_2 such that $R_1 \Rightarrow* R_2$, and V is declared in R_2. R_1 is defined to "indirectly modify" V whenever there is a routine R_2 such that $R_1 \Rightarrow* R_2$, and V is declared readwrite in R_2.

The MAX example has the property that each procedure parameter is used solely for input or output. If we wish to write a specification for a procedure SWAP(A, B) that exchanges the values of A and B we have a problem. To express the values the parameters should have on output, the exit condition must use some notation for the values those variables had on input.

Auxiliary parameters are used for this purpose. The specification used for SWAP is:

```
SWAP: PROCEDURE(A, B /*:, A0, B0 */);
DECLARE (A, B) FIXED;
/*: DECLARE (A0, B0) FIXED READONLY; */
ASSUME A=A0 & B=B0;
ATTAIN A=B0 & B=A0;
```

The auxiliary parameters A0 and B0 follow the same rules as the auxiliary variables discussed in section 4.9. When SWAP is called, shielded actual parameters must be given that satisfy the input assumption P, as in:

```
CALL SWAP(X, Y /*:, (X), (Y) */);
```

In this example the parentheses are required around the shielded parameters to avoid aliasing (see section 4.11).

Roughly speaking, the rule for a nonrecursive call to a procedure is:

```
{Dom actuals}
{? P[actuals//formals]}
CALL R(actuals);
/* q */
{! E[actuals//formals]}
```

where P and E are, respectively, input assumption and exit condition of the procedure R. It is rather cumbersome to spell out the actual rule in detail; the V-code generated for the procedure call is easier to understand than the rule is.

The V-code generated for the procedure call consists of:

(1) a DOMAIN t instruction for each actual parameter t,

(2) a REQUIRE instruction,

(3) a sequence of NEW instructions, and

(4) a series of PROCLAIM instructions.

The operand of the REQUIRE instruction is obtained by substituting for each formal parameter in the input assumption P the corresponding actual parameter.

The purpose of the sequence of NEW instructions is to set the assertion table to A_q. A NEW v instruction must be generated for each variable v that can be modified as a result of the procedure call. The variables for which NEW instructions must be generated are:

(1) scalar or array variables V that are passed by reference through a readwrite parameter

(2) array variables A, where an array element A(X) is passed by reference through a readwrite parameter and

(3) external variables V that are indirectly modified by the procedure being called.

In PL/C, all parameters of the form V or A(x) are passed by reference [46]. That is, when the corresponding formal parameter is modified, the corresponding actual parameter is modified as well. All other expressions, including those of the form (V) and (A(x)) are passed by value; modifying the corresponding formal parameter has no effect on any other variable.

Designate by $\overline{V}$ the set of variables V for which NEW instructions are generated. The notation $Q[\overline{V}'//\overline{V}]$ will designate the formula (or expression) Q with V' substituted for each variable V in $\overline{V}$, where V' is the new variable selected by the NEW V instruction. As in the case of the assignment statement, the proper interpretation for V' is the value the variable

V contained before the procedure call.

The operand of the first PROCLAIM instruction is obtained by performing substitutions on E, the exit condition for the procedure. For each formal parameter F, let C be the corresponding actual parameter. In each case, some variation on C or F is substituted for every occurrence of F in E, in order to relate the final value of the formal parameter to some value in the calling procedure. There are four cases:

(1) If F is READONLY, the value of F at the end of the procedure is the same value it had at the beginning. Variables in C can be changed by the procedure; $C[\overline{V}'//\overline{V}]$ is the value F contained throughout the execution of the procedure. Therefore, $C[\overline{V}'//\overline{V}]$ is substituted for for every occurrence of F in E.

(2) If C is passed by value and F is readwrite, the value of F at the end of the called procedure is not related to any value in the calling procedure. Therefore, a variable F' that appears nowhere else in the proof is substituted for every occurrence of F in E.

(3) If C is a scalar or array variable passed by reference and F is readwrite, C and F have the same value when the procedure finishes execution. Therefore, substitute C for every occurrence of F in E.

(4) If C is an array element of the form A(x) and F is readwrite. the variable that is the actual parameter and F have the same value when the procedure finishes execution. However, after the procedure call, A(x) is not necessarily a name for the variable that is the actual parameter if the procedure call modifies variables in x. The expression $A(x[\overline{V}'//\overline{V}])$ is substituted for every occurrence of F in E.

Once the appropriate substitutions for each formal parameter in E have been performed to yield E', the instruction PROCLAIM E' is generated.

If one or more array elements of the form A(x) are passed by reference to a readwrite parameter, a NEW A instruction will be generated. Assertions involving elements of A that were not modified by the procedure are still valid, but are not accessible to points past the procedure call. So that these assertions can be established after the procedure call more PROCLAIM instructions must be generated. For each array A for which there are elements $A(x_1)$, $A(x_2)$ ... $A(x_n)$ passed by reference to readwrite parameters, a PROCLAIM Q instruction must be generated, where Q is of the form:

```
ALL J FIXED,
  ( J ~= x1[V̄'//V̄]
  & J ~= x2[V̄'//V̄]
  :
  & J ~= xn[V̄'//V̄]
  => A(J) = A'(J)
  ) .
```

Given the strengthened notion of accessibility discussed in section 4.9, these PROCLAIM instructions could be omitted.

A routine R is said to be recursive whenever R $\Rightarrow^+$ R. (Note that a procedure need not call itself explicitly to be recursive.) The rules for recursive procedures are exactly the same as those for nonrecursive procedures, except that termination must be proved as well. A termination predicate T with a free variable s is specified for every recursive procedure. The predicate T will, in general, also depend on the procedure parameters and external variables accessed by the procedure. Intuitively, T says that the difficulty of problem to be solved by the routine can be characterized by some metric m, and that m ≤ s. At the top of every recursive procedure, the formulas

SOME s FIXED. s >= 0 & T

must be proved, using only the input assumption as a hypothesis.

Definition 4.11. Let R_1 be a routine that calls R_2. The call is said to be a "recursive call" whenever $R_2 \Rightarrow* R_1$.

Since $R_1 \Rightarrow R_2$ and $R_2 \Rightarrow* R_1$, we have $R_1 \Rightarrow* R_1$ and $R_2 \Rightarrow* R_2$. Therefore, recursive calls can only occur between recursive routines. Note, however, that it is possible for one recursive routine to call another without the call being a recursive call.

Suppose R_1 and R_2 are recursive routines, and R_1 calls R_2. Let T_1, T_2, s_1, s_2 be the termination parameters and the arbitrary variables declared in the bodies of R_1 and R_2. If the call to R_2 is a recursive call, to ensure termination two additional REQUIRE instructions are generated for the call: REQUIRE s_1 ~= 0 and REQUIRE T', where T' is obtained by performing the following substitutions on T_2. The corresponding actual parameter in the call to R_2 is substituted in T_2 for each occurrence of each formal parameter of R_2, and $s_1 - 1$ is substituted for s_2. Intuitively, these two assertions express the idea that if R_1 is performing a recursive call, the problem it is solving is nontrivial (that is, every difficulty metric for the problem is positive) and it is using R_2 to solve a problem easier than the one it is solving (there is a difficulty metric for the problem R_2 is solving that is smaller than every difficulty metric for the problem R_1 is solving).

To see that these rules guarantee termination, suppose we have an infinite chain of routine calls. Because there are only finitely many routines in a program, the chain of calls must have an infinite tail that is a sequence of calls to routines R_{a_0}, R_{a_1}, ..., R_{a_i} ... where if routine

R_j appears in the tail, it appears infinitely often. Because every routine that is called in the chain is called again at a later point in the chain, every call in the chain must be a recursive call.

Let T_j be the termination predicate for each R_j. Define an integer m to satisfy a termination predicate T_j in a state whenever substituting m for the free variable s in T_j yields a formula that is satisfied by the state. The termination rules guarantee that:

(1) whenever a recursive routine R_j is called there will be some non-negative integer m satisfying T_j in the state in which the routine begins execution, and

(2) for any integer m, if R_j is begun in a state in which m satisfies T_j, and from that state R_j makes a recursive call, then m cannot be zero, and

(3) for any integer m, if R_j is begun in a state in which m satisfies T_j, and from that state R_j makes a recursive call to R_k, then m - 1 satisfies T_k in the state in which R_k is begun.

Let m be a nonnegative integer satisfying T_{a_0} in the state in which R_{a_0} is begun; by (1) such an m must exist. For each nonnegative i, in the state in which R_{a_i} is begun, m - i must satisfy T_{a_i}, by (3) and induction on i. Therefore, 0 must satisfy T_{a_m} in the state which R_{a_m} is begun. Since when begun in this state, R_{a_k} makes a recursive call, (2) is contradicted. Therefore there can be no infinite chains of procedure calls.

4.11 Aliasing

The V-code generated for every programming construct includes a NEW v instruction for every variable v that can be modified by the construct. To generate this V-code, the Local Context Checker be able to determine which variables are modified by a construct by examining the construct. For example, the assignment statement:

X := X + 1; (4.16)

modifies the variable X. However, if X and Y are the two parameters of the procedure R, and there is a procedure call of the form:

CALL R(Z, Z); (4.17)

then the assignment statement (4.16) can modify the variable Y as well.

Procedure calls such as (4.17) are said to "alias" the parameters X and Y. Cartwright and Oppen [13] discuss the additions necessary to a programming logic to reason about programs in which aliasing occurs. The PL/CV system uses the simple expedient of declaring any program that contains aliasing to be in error. The Local Context Checker tests each procedure call, and issues an error message if it can potentially introduce aliasing.

Every variable in a program can be considered a name for some storage location. If a variable v has been declared to be readonly in a procedure R, then v can only be used to examine the contents of the storage location. In that case, the procedure R will be said to possess v as a "read name" for the location. If the variable v has been declared to be readwrite in R, then v can be used to both examine and modify the storage location. In that case, R is said to possess v as both a read name and a "write name" for the location. To prevent aliasing, it is sufficient to prevent any procedure from possessing two distinct variables r and w such that r is a read name and w is a write name for the same location.

There are three kinds of program variables in PL/CV: local variables, external variables, and procedure parameters. Distinct variables that are not parameters are always names for distinct storage locations. Therefore, if the variables r and w are aliases, at least one of them must be a parameter. In a procedure R, it is impossible for a parameter and a local variable to be names for the same location, because the location named by the local variable and the parameter must reside in different environments. Therefore, aliasing can only occur between two parameters, or between a parameter and a external variable.

The following test is used to prevent aliasing between two parameters. For every procedure call, all the parameters that are are being passed by reference, including those being passed through readonly parameters, are examined. If the same variable is being passed through more than one parameter, and at least one of those parameters is readwrite, an error is reported.

When array variables or array elements are being passed by reference, some extra processing is required. All pairs of actual parameters involving the same array, except for pairs being passed through two readonly parameters, must be considered. If one of the pair is an array variable, an error is reported. In the other case, when the pair is two array elements A(x) and A(y), the V-code instruction:

```
REQUIRE x ~= y
```

must be generated.

This test can be extended to prevent aliasing between parameters and external variables by treating external variables as implicit parameters. If a routine R indirectly modifies an external variable V, then V is included as an extra implicit readwrite parameter to every call to the routine R. If the routine R depends on V, but does not indirectly modify

```
R: PROCEDURE;                              R: PROCEDURE;
DECLARE G FIXED EXTERNAL;                  DECLARE G FIXED EXTERNAL;
:                                          :
CALL S(G);                                 CALL S(G);
:                                          :
END R;                                     END R;

S: PROCEDURE(X);                           S: PROCEDURE(X);
DECLARE X FIXED READONLY;                  DECLARE X FIXED READONLY;
:                                          :
CALL T(X);                                 CALL T(X);
:                                          :
END S;                                     END S;

T: PROCEDURE(Y);                           T: PROCEDURE(Y);
DECLARE Y FIXED READONLY;                  DECLARE Y FIXED READONLY;
DECLARE G FIXED EXTERNAL READONLY;         DECLARE G FIXED EXTERNAL;
:                                          :
                (a)                                      (b)
```

Figure 4.25 Legal (a) and illegal (b) uses of parameter passing

V, then V is included as an implicit readonly parameter to each call. The implicit parameters are of course passed by reference.

An example is shown in Figure 4.25. In routine R in part (a), the variable G is being passed to S explicitly. But since S calls T, and G is declared in T, the routine R also passes the variable G to S implictly. However, both the implicit and explicit parameters are both readonly, so the call is legal. It can only result in multiple read names for the location occupied by G.

The difference between Figure 4.25a and 4.25b is that in (b), G is declared readwrite in T. The call to S from R is now illegal, because G is being passed explicitly readonly but implicitly readwrite. The procedure T can potentially possess Y as a read name and G as a write name for the same location.

```
name: PROCEDURE(A, B, C ...) RETURNS(type);
DECLARE ...   (parameters and external variables)
ASSUME P;
{! P}
ATTAIN E;
:            (predicate calculus proof)†
{? SOME s FIXED. (s >= 0 & T)}†
ARBITRARY s FIXED WHERE T;†
{! T}†
DECLARE ...  (local variables)
:            (procedure body)
{? '0'B}
END;
```

†omitted in nonrecursive functions

Figure 4.26 Template for a function definition

4.12 Functions and Function Calls

To reason about a programming language in which expressions can have side effects, it is necessary to make assertions about points in the program within expressions. Expressions, and in particular user-defined functions, are for this reason not allowed to have side effects in PL/CV. All function parameters are implicitly readonly, and a function is not allowed to indirectly modify any external variable. Therefore functions are much simpler than procedures. For instance, there is no need to check function calls for aliasing.

The template for a function definition is shown in Figure 4.26; it is essentially identical to the template for procedure definitions. Let N stand for name(A, B, C ...). Then the exit condition E will be some predicate involving the function parameters, external variables (which can only be accessed readonly), and N. For each statement of the form:

```
RETURN(X);
```

in the function, the assertion (E[X//N], p) must be known, where p is the

```
REQUIRE   P'
OPEN
PROCLAIM  E[actuals//formals]
REQUIRE   Q
CLOSE
```

Figure 4.27 V-code generated for the FUNCTION rule

point in the program just prior to the RETURN statement.

There is no "function call statement"; functions are simply written in any expression. The domain predicate (see section 4.7) for a recursive call to the function R_2 appearing in an expression in a routine R_1 is:

$$P[actuals//formals] \ \& \ T[actuals//formals, \ s_1-1//s_2] \ \& \ s_1 \sim= 0 \qquad (4.18)$$

where P and T are the input assumption and termination parameter of R_2, and s_1 and s_2 are the arbitrary variables declared in the definitions of R_1 and R_2. If the call is not recursive, the domain condition is just:

$$P[actuals//formals] \ . \qquad (4.19)$$

User-defined functions can also appear in predicate calculus proofs. The following rule is used to reason about user-defined functions:

Q BY FUNCTION, R(actuals); . (4.20)

The V-code generated for (4.20) is shown in Figure 4.27, where E is the exit condition of R, and P' is either (4.18) or (4.19). Thus, Q is any conclusion that can be derived via automatic rules from the exit condition of R.

When computing the relation $\Rightarrow*$ to determine which routines and routine calls are recursive, an application of the FUNCTION rule (4.20) counts as a "call" to the function R. In the V-code in Figure 4.27, if the rule application is a recursive "call" then P' is (4.18); otherwise P' is (4.19).

```
WRONG: PROCEDURE(X) RETURNS(FIXED);
DECLARE X FIXED;
ASSUME '1'B;
ATTAIN WRONG(X) = X & WRONG(X) = X+1;
WRONG(X) = X & WRONG(X) = X+1 BY FUNCTION, WRONG(X);
X = WRONG(X) = X+1;
X = X & X = X+1;
RETURN(X);
END;
```

Figure 4.28 A fallacious argument

It may seem peculiar that a function that is called from a within comment must be proved to terminate. But Figure 4.28 shows the kind of nonsensical arguments that can be made without this requirement. When one is using the function rule, one is proving that a function has some property by induction on the size of its arguments. The termination rule is used to show the induction to be well-founded.

Suppose we wish to prove a function FACT(N) that returns N factorial. What do we put on the ATTAIN line of the function body? We cannot write ATTAIN FACT(N) = N!, because the factorial function is not a primitive in the assertion language. A solution to this problem is to write:

```
ATTAIN (N = 0 => FACT(N) = 1) & (N > 0 => FACT(N) = N * FACT(N-1)); .
```

If we write the function recursively, it is a simple matter to prove it correct. If we write the function iteratively, the FUNCTION rule must be used recursively to prove the function, as shown in Figure 4.29.

Note that the technique of recursive specifications used to define the function FACT cannot be used with procedures. A specification for a procedure FACTP(N, K) that stores N! in K must be written in terms of the FACT function in Figure 4.29.

Although functions are not allowed to modify external variables, they are allowed to depend on external variables. Suppose the function

```
FACT: PROCEDURE(N) RETURNS(FIXED);
DECLARE N FIXED;
ASSUME N >= 0;
ATTAIN (N = 0 => FACT(N) = 1) & (N > 0 => FACT(N) = N * FACT(N-1));

SOME S FIXED. (S >= 0 & N <= S) BY INTRO. N;
ARBITRARY S FIXED WHERE N <= S;

DECLARE (J, F) FIXED;

F := 1;

1>N => (N=0 => F=1) & (N>0 => F=N*FACT(N-1)) BY INTRO.
PROOF;
   N = 0 BY ARITH. 1>N. N>=0;
   ~(N>0) BY ARITH. N = 0;
QED;

DO J := 1 TO N BY 1;
ASSUME (J-1 = 0 => F = 1) & (J-1 > 0 => F = (J-1) * FACT(J-2));
    P1: ~(J = 0) BY ARITH. J >= 1;

    /* show FACT(J-1) well-defined */
    S ~= 0 BY ARITH. 1 <= J. J <= N. N <= S;
    J-1 <= S-1 BY ARITH. J <= N. N <= S;
    J-1 >= 0 BY ARITH. J >= 1;

    C: J-1 = 0 | J-1 > 0 BY ARITH. J >= 1;
    F = FACT(J-1) BY CASES. C.
    PROOF;
    CASE J-1 = 0;
        F = 1; /* from the invariant */
        FACT(J-1) = 1 BY FUNCTION. FACT(J-1);
    CASE J-1 > 0;
        F = (J-1) * FACT(J-2); /* from the invariant */
        FACT(J-1) = (J-1) * FACT(J-2) BY FUNCTION. FACT(J-1);
    QED;
    P2: F * J = J * FACT(J-1) BY ARITH. F = FACT(J-1). *. J > 0;

    /* from P1 and P2 we get: */
    (J+1-1 = 0 => F*J = 1) & (J+1-1 > 0 => F*J = (J+1-1) * FACT(J+1-2));

    F := F * J;

    /* from the assignment rule we get: */
    (J+1-1 = 0 => F = 1) & (J+1-1 > 0 => F = (J+1-1) * FACT(J+1-2));
END;

RETURN(F);
END;
```

Figure 4.29 A nonrecursive function proved using recursion

$F(X)$ is defined to be the sum of X and the external variable Z, and the assertion $(F(Y) > 0, p)$ has been proved to be valid. If point p dominates point q and the variable Y is not modified along any simple path from p to q, then by Definition 4.4, the assertion is accessible to q, and by Theorem 4.5, $(F(Y) > 0, q)$ is a valid assertion.

However, if the varible Z is modified along a simple path from p to q the assertion $(F(Y) > 0, q)$ need not be valid. For Theorem 4.5 to be correct, the definition of accessibility needs to be modified slightly to accomodate functions that depend on external variables. If a formula contains a function that depends on an external variable v, then for purposes of computing accessibility, the formula is considered to contain an instance of v.

Whenever the Local Context Checker generates a NEW v instruction for an external variable v, it also generates a NEW f instruction for every function f that depends on v. In the example above, $F(Y) > 0$ is in the Assertion Table when the V-code interpreter gets to point p. But between points p and q the variable Z is modified, so that NEW Z and NEW F instructions are executed. When the interpreter gets to point q, the formula in the Assertion Table will have been replaced by $F'(Y) > 0$, and a REQUIRE $F(Y) > 0$ will not succeed.

4.13 A Theorem about Accessibility

For each point q in the program, the Local Context Checker must generate V-code that sets the Assertion Table to A_q, given that it contains K_p, where p is the point in the program that immediately precedes q lexically. For the simple constructs (the assignment statement and procedure call), the claim was made that this V-code consists of simply a NEW v for each variable modified by the construct. In this section, a theorem is proved that justifies this claim. The theorem will be used to show how the computation of A_q can be performed for more complex control structures.

Two assumptions will be made about the nature of execution paths. First, every point in the routine is assumed to be reachable. That is, for each point p in the routine, an execution path from the start of the routine to p is assumed to exist. This assumption is valid given Approximation 4.8. Second, if there is an execution path from p to q, and a path from q to r, it is assumed that these paths can be concatenated to form a path from p to r. This assumption is valid given Approximation 4.7.

The fact that an execution path from p to r that passes through q can be cut at q to yield a path from p to q and a path from q to r is also used, but this property of execution paths holds no matter what (if any) approximations are being used.

Lemma 4.12. If point p dominates point r, there is a simple execution path from p to r.

Proof. Because r is assumed to be reachable, there is an execution path from the start of the routine to r. Because p dominates r, p must be on that path. If the path is cut the last time it passes through p, the result will be a simple path from p to r. □

Definition 4.13. If p and q are points in a program, define $V_{p,q}$ to be the set of all variables that are modified by atomic actions through which any simple execution path from p to q passes. Define $M_{p,q}$ to be the set of all assertions (R, r) such that r is any point and R contains a variable in $V_{p,q}$.

In Lemma 4.14 and Theorem 4.15, p and q will refer to points in a routine with the following two properties:

p dominates q (4.21)

for any point r, there is a simple execution path from p to q that does not pass through r. (4.22)

Lemma 4.14. If r is a point that is distinct from p, and r dominates q, every path from r to q passes through p.

Proof. Let r be a point that is distinct from p and dominates q. Consider any path T from r to q. We must show that T passes through p.

Since every point is assumed to be reachable, there is a path from the beginning of the routine to r. Cutting this path the first time it passes through r, we obtain a path S from the beginning of the routine to r that does not pass through r. If S is concatenated with T, the result is a path from the beginning of the routine to q. Since, by property (4.21), p dominates q, this path must pass through p. Since p is distinct from r, either S or T must pass through p.

If T passes through p we are done. Suppose S passes through p. Cut S at p to obtain a path from the beginning of the routine to p. Concatenate this path with any path from p to q that does not pass through r; one must exist by property (4.22). The result is a path from the beginning of the routine to q that does not pass through r, contradicting the assumption that r dominates q. □

Theorem 4.15. $A_q = K_p - M_{p,q}$.

Proof. It is easy to show from properties of set subtraction that $A_q = K_p - M_{p,q}$ if and only if the following three conditions hold:

$$A_q \subseteq K_p \tag{4.23}$$

$$A_q \cap M_{p.q} = \emptyset \tag{4.24}$$

$$K_p - M_{p.q} \subseteq A_q \,. \tag{4.25}$$

The theorem will be proved by proving each of the above conditions.

Condition (4.23) will be proved by contradiction; suppose it does not hold. Then there is an assertion (R, r) that is accessible to point q. but is not known at point p. Therefore (R, r) cannot be accessible to point p, so that either:

(1) there is a path from the start of the routine to p that does not pass through r, or

(2) a variable in R may be modified along a simple path from r to p.

By property (4.22) there must be a simple path from p to q that does not pass through r. Let T be one such path.

In case (1), concatenate the path from the start of the routine to p that does not pass through r with T. The result is a path from the start of the routine to q that does not pass through r, contradicting the hypothesis that (R, r) is accessible to q.

In case (2), the simple path from r to p along which a variable in R is modified is concatenated with T. The result is a simple path from r to q along which a variable in R is modified. again contradicting the hypothesis that (R, r) is accessible to q.

Condition (4.24) is also proved by contradiction; suppose it does not hold. Then there is an assertion (R, r) that is accessible to q. and there is a simple path T from p to q along which a variable in R is modified. The points r and p must be distinct, since if they were the same point then (R, r) could not be accessible to point q, because of path T. Because r dominates q, by Lemma 4.12, there is a simple path from r to q, and by Lemma 4.14, this path passes through p. Cutting the path at p, we obtain a simple path S from r to p. Now either:

(1) the path T does not pass through the point r or

(2) T does pass through point r.

In case (1), concatenate S with T to obtain a simple path from r to q along which a variable of R is modified, contradicting the assumption that (R, r) is accessible to q.

In case (2), cut T at the point where it passes through r. The result is a path from r to q. Since, by lemma 4.14, every path from r to q passes through p, the path T must also pass through p, contradicting the assumption that T is a simple path from p to q.

To establish condition (4.25), suppose:

(1) an assertion (R, r) is known at p, and

(2) no variables of R are modified along any simple path from p to q.

We must show that (R, r) is accessible to point q. We first show that point r dominates point q. Let T be any path from the start of the routine to q. By property (4.21), p dominates q, so that T passes through p. By hypothesis (1), either r and p are the same point, or r dominates p. In either case, T must pass through r.

Next we must show that no atomic action along any simple path U from r to q modifies a variable in R. If r and p are the same point, by hypothesis (2) we are done. Otherwise, by hypothesis (1), r must dominate p, and then by Lemma 4.14, the path U must pass through p. By cutting U the last time it passes through p, it can be split into to a simple path S from r to p and a simple path T from p to q. By hypothesis (1), no variable in R can be modified along S, and by hypothesis (2), no variable in R can be modified along T. □

Theorem 4.15 is used to derive the V-code that is to be generated for the various control structures. Given a point q in a template, a point p is located that satisfies conditions (4.21) and (4.22). After the V-code generated for point p that will set the Assertion Table to K_p, an OPEN instruction is generated that will save K_p. At point q, a CLOSE instruction is generated to restore the Assertion Table to K_p, and a NEW instruction is generated for every variable in $V_{p,q}$. For this approach to work, it is necessary that the V-code generated for each template is balanced with respect to OPEN and CLOSE instructions, in the same way that open and close parentheses are balanced in expressions. The notions of "balanced" V-code, and "nests" are defined recursively in terms of each other as follows.

Definition 4.16. A sequence of V-code instructions is defined to be balanced whenever it consists of a sequence of zero or more elements $a_1, a_2, \ldots a_n$, where for each i in {1 ...n}, a_i is either a nest or a V-code instruction other than OPEN, CLOSE, or CLOSE-NEW. A "nest" is defined to be an OPEN instruction, followed by a balanced sequence of V-code instructions, followed by a (matching) CLOSE or CLOSE-NEW instruction. The CLOSE-NEW instruction is discussed in the next section.

4.14 Conditional Statements

The template for the PL/CV IF rules is shown in Figure 4.30. Point p dominates both q and r, since the only way to get to the THEN or ELSE part is by way of the IF. Since the only simple execution path from point p to point q or r does not pass through any points in the program, by

```
{Dom B}
ATTAIN E;

/* p */
IF B
THEN DO;
     /* q */
     {! B}
     :    (body of the THEN part)
     {? E}
     END;
ELSE DO;
     /* r */
     {! ~B}
     :    (body of the ELSE part)
     {? E}
     END;
/* s */
{! E}
```

Figure 4.30 Template for the PL/CV IF rules

```
OPEN
OPEN
DOMAIN   B
PROCLAIM B
:        (V-code generated for the THEN part)
REQUIRE  P
CLOSE
PROCLAIM ~B
:        (V-code generated for the ELSE part)
REQUIRE  P
CLOSE
NEW      <Each variable modified in THEN and ELSE parts>
PROCLAIM P
```

Figure 4.31 V-code generated for an IF statement

Theorem 4.15. $A_q = A_r = K_p$.

Because the GOTO statement in PL/CV has been restricted so that it cannot branch from outside the IF statement into the THEN part or ELSE part, point p dominates point s. Note that GOTO statements that branch out of the THEN or ELSE part, which are allowed, do not prevent point p from dominating point e. Suppose a simple path from p to s passes through a point r. That path must be entirely within the THEN part or the ELSE part of the IF statement. If r is in the THEN part, there is a path through the ELSE part that avoids r (given Approximation 4.8), and vice versa. Therefore, by Theorem 4.15, $A_s = K_p - M_{p,s}$. Given Approximation 4.8, $V_{p,s}$ can be determined by examining all the simple statements in the THEN and ELSE parts of the IF that modify variables.

The V-code generated for an IF statement is shown in Figure 4.31. The two OPEN instructions each save copies of K_p. The first CLOSE instruction sets the Assertion Table to A_q by restoring it to K_p. The second CLOSE instruction again restores the Assertion Table to K_p, and then a NEW instruction is generated for each variable in $V_{p,s}$. The result is to set the Assertion Table to A_s (plus some innocuous assertions about values that variables modified by the IF statement contained before the statement was executed).

After every construct, a NEW instruction will need to be generated for every variable that can be modified by the construct. If a simple statement that modifies a variable is nested within n compound statements, then n NEW instructions will be generated. Thus, the length of the V-code generated for a proof can be quadratic in the size of the proof.

The effect of this quadratic expansion can be diminished by increasing the power of the V-machine. A new instruction, CLOSE-NEW, is introduced that is a combination of a CLOSE instruction and a NEW instruction. The V-machine is augmented so that it keeps track of NEW instructions as it performs them. To execute a CLOSE-NEW instruction, first a CLOSE operation is performed. Let NV be the set of variables v for which there is a NEW v instruction in the nest ending with the CLOSE-NEW instruction.

For each v in NV, the CLOSE-NEW instruction performs a NEW v operation. The V-code generated for the IF statement can be shortened by replacing the CLOSE and NEW instructions generated for point s in the IF template (Figure 4.30) with a CLOSE-NEW instruction.

When the line ATTAIN P precedes a construct in PL/CV, the formula P will be known at the point in the program following the construct. The formula P is considered to be the conclusion of the proof rule for the construct. The normal convention in logic is to have the conclusion of a rule be written last. But having the conclusion come first is more consistent with the theorem-before-proof syntax chosen for the predicate calculus rules. In fact, the syntax:

```
P BY
IF B THEN
:
.
```

in which the IF statement is considered to be the justification for P, might have been more appropriate. (This view is taken by Bates [10] in his Program Refinement Logic.) Another reason for having the conclusion of the rule precede the statement is to simplify the generation of V-code.

The SELECT statement is a straightforward generalization of the IF statement. The template for the SELECT statement is shown in Figure 4.32, and the V-code generated is shown in Figure 4.33.

Rather than generating PROCLAIM instructions for the conjunctions shown in the template, the PROCLAIM $\sim B_i$ instructions are placed so that $\sim B_i$ is known in all the WHEN and OTHERWISE clauses following the point for which the PROCLAIM instruction is generated. Given the automatic rules, the instruction:

```
ATTAIN E;

SELECT;
{Dom B1}
WHEN (B1) DO;
     {! B1}
     :         (first WHEN part)
     {? E}
     END;
{Dom B2}
WHEN (B2) DO;
     {! ~B1 & B2}
     :
     {? E}      (second WHEN part)
     END;
:
{Dom Bn}
WHEN (Bn) DO;
     {! ~B1 & ~B2 & ... ~Bn-1 & Bn}
     :
     {? E}      (nth WHEN part)
     END;
OTHERWISE DO;
     {! ~B1 & ~B2 & ... ~Bn}
     :
     {? E}      (OTHERWISE part)
     END;
END;
{! E}
```

Figure 4.32 Template for the PL/CV SELECT rules

PROCLAIM B_1 & B_2 & ... B_n

is equivalent to the sequence of instructions:

PROCLAIM B_1
PROCLAIM B_2
⋮
PROCLAIM B_n .

```
OPEN
OPEN
DOMAIN     B1
PROCLAIM   B2
  .
  .        (V-code generated for first WHEN part)
  .
REQUIRE    E
CLOSE
PROCLAIM   ~B1
OPEN
DOMAIN     B2
PROCLAIM   B2
  .
  .        (V-code generated for second WHEN part)
  .
REQUIRE    E
CLOSE
PROCLAIM   ~B2
  .
  .
  .
OPEN
DOMAIN     Bn
PROCLAIM   Bn
  .
  .
  .
REQUIRE    E
CLOSE
PROCLAIM   ~Bn
  .
  .        (V-code generated for OTHERWISE part)
  .
REQUIRE    E
CLOSE-NEW
PROCLAIM   E
```

Figure 4.33 V-code generated for a SELECT statement

4.15 WHILE Loops

A slightly simplified version of the WHILE rule is shown in Figure 4.34. The rule is Hoare's WHILE rule [41] combined with a proof of termination.

The rule used to prove termination is essentially the same rule presented in section 4.10 for proving the termination of recursive routines. A rule requiring ~T[0//s] at the top of the loop body is equivalent to a rule requiring s ~= 0 there. If s ~= 0 were required, it would in most cases be proved by assuming s = 0, substituting 0 for s in T, and

```
/* p */
{? P}
{? SOME s FIXED. (s >= 0 & T)}
{Dom B}
DO WHILE(B);
/* q */
ASSUME P;
{! P}
ARBITRARY s FIXED WHERE T;
{! T}
{! B}
:       (Predicate Calculus proof)
{? ~T[0//s]}
:       (Body of loop)
{? P}
{? B => T[s-1//s]}
{Dom B}
END;
/* r */
{! P}
{! ~B}
```

Figure 4.34 The basic WHILE rule

```
REQUIRE    P
REQUIRE    SOME s FIXED. (s >= 0 & T)
DOMAIN     B
NEW        <variables modified in the body of the loop>
OPEN
PROCLAIM   P
PROCLAIM   T
PROCLAIM   B
:          (V-code for predicate calculus proof)
REQUIRE    ~T[0//s]
:          (V-code for loop body)
REQUIRE    P
REQUIRE    B => T[s-1//s]
DOMAIN     B
CLOSE
PROCLAIM  P
PROCLAIM ~B
```

Figure 4.35 V-code generated for a WHILE loop

deriving a contradiction. Therefore the rule requiring ~T[0//s] is usually easier to use.

A typical way of proving termination is to find some expression e involving variables that are modified in the loop body that has the two properties:

(1) just before the loop begins execution, and at the top and bottom of the loop body, e is nonnegative, and

(2) Each execution of the loop body reduces e by at least one.

This termination proof can be expressed in PL/CV by letting the predicate T be e <= s. Note that the formula:

```
SOME S FIXED. (S >= 0 & E <= S)
```

is always valid no matter what E is; choose S to be the maximum of 0 and E.

Because GOTO statements have been restricted so that they cannot branch from outside to inside the body of the loop, point p dominates both q and r. There is a simple path from p to q that passes through no other points in the program. There also is a simple path from p to r, the one in which the loop body is executed zero times, that passes through no other points in the program. Therefore, by Theorem 4.15, $A_q = K_p - M_{p,q}$ and $A_r = K_p - M_{p,r}$. But $V_{p,q}$ and $V_{p,r}$ are each the set of variables that can be modified by the loop body. Given Approximation 4.8, this set is just the set of variables that can be modified by any statement in the loop body.

The V-code generated for the WHILE statement is in Figure 4.35. A sequence of NEW instructions generated for point q are used to set the Assertion Table to $A_p - M_{p,q}$. An OPEN instruction is used to save a copy of $A_p - M_{p,q}$, and the CLOSE instruction generated for point r restores the Assertion Table to that value.

The NEW instructions that must be generated for point q are not known until the body of the loop has been processed. This forward referencing problem is the major reason that the V-code instructions cannot be interpreted as they are produced. Instead, the output V-code is saved and interpreted after the entire proof has been processed.

In the implementation there is a V-code instruction of the form MULTINEW b, where b is a bit string. There is one position in the string for every program variable. The action specified by a MULTINEW instruction is to perform a NEW for every variable whose corresponding bit is on. When the V-code for a loop is being generated, room is saved for a MULTINEW instruction, and the location of that room is remembered. After the body of the loop has been processed, and it is known which variables are changed, the appropriate MULTINEW instruction is generated and installed in the appropriate place.

The use of the GOTO statement is generally discouraged by advocates of structured programming. One exception to this dictum is the use of the GOTO statement to exit a loop. [25] In an attempt to clearly distinguish between acceptable and unacceptable uses of the GOTO, the LEAVE statement has been introduced to PL/I. A LEAVE statement is just a euphemism for GOTO L, where L is the statement just after the loop being left.

If a WHILE rule being proved using the rule in Figure 4.34 contains a LEAVE statement, the assertions P and ~B must be known at the points preceding each LEAVE statement for that loop. But in general these assertions do not have to be valid when a LEAVE statement is executed. For example, if the statement:

```
IF B THEN LEAVE;
```

appears in the body of a loop (in a place where the LEAVE can ever be executed), ~B will not be known just before the LEAVE.

The solution to this problem is to associate a third predicate E, called the "exit condition," with every loop. The exit condition is what is proved to hold when the execution of the loop is complete. WHILE loops can optionally be preceded by a line of the form ATTAIN E to indicate what the exit condition is. If the ATTAIN line is not present, the exit condition defaults to P & ~B.

There are two parts to the proof that E holds when a loop execution completes. The formula E must be known just before every LEAVE that exits the loop, and E must be shown to hold when the loop terminates normally.

In the case of WHILE loops, this second condition is equivalent to showing that E follows from the set of formulas:

$$A_q \cup \{P\} \cup \{\sim B\} \quad . \tag{4.26}$$

To accomodate the LEAVE statement, the V-code in Figure 4.35 must be modified as follows. The final PROCLAIM is changed to PROCLAIM E, and the following V-code:

```
OPEN
PROCLAIM ~B
:               (V-code generated for proof of E)
REQUIRE E
CLOSE
```

is inserted just after the PROCLAIM P and before the PROCLAIM T. In many cases, E will follow by automatic rules. If a proof is required to establish E, it may be placed between the ASSUME and ARBITRARY lines in the loop body.

This mechanism is fairly clumsy. A cleaner solution would be to require the assertion ~B => E before the loop begins and at the end of the loop body. The problem with this solution is that the user must write essentially the same proof, that ~B => E follows from (4.26), twice. A better solution is proposed in section 4.18.

4.16 UNTIL Loops

Wirth [71] gives the following rule rule for UNTIL loops:

$$\frac{\{P\}\ S\ \{Q\} \qquad \{Q\ \&\ \sim B\}\ S\ \{Q\}}{\{P\}\ \text{DO UNTIL(B) S END}\ \{Q\}}\ .$$

This rule cannot be easily adapted to PL/CV, because two proofs involving the body of the loop must be written, and in a PL/CV proof, the body of the loop is only written once.

Any UNTIL loop of the form:

```
DO UNTIL(B)
:
END
```

can be reduced to a WHILE loop and two assignment statements as follows:

```
f := '1'B;
DO WHILE(~B | f);
:
f := '0'B;
END;
```

where f is a Boolean variable that does not appear anywhere else in the program. Reduction can therefore be used to obtain a proof rule for UNTIL loops suitable for use with PL/CV. Combining the proof rules for assignment and WHILE loops yields the proof rule in Figure 4.36. (The rule is valid only for loops for which there are no LEAVE statements.)

The UNTIL rule in Figure 4.36 is another example of a poor use of reduction to obtain proof rules. The rule is as complete as the assignment and WHILE rules are, but in practice it is clumsy to use. The loop

```
{? P['1'B//f]}
{? SOME s FIXED. (s>=0 & T)}

DO UNTIL(B);
ARBITRARY f BIT WHERE P & (~B | f);
ARBITRARY s FIXED WHERE T;
:           (Predicate calculus proof)
~T[0//s]
:           (Loop body)
{? P['0'B//f]}
{? ~B => T[s-1//s]}
END;

{! P['0'B//f]}
```

Figure 4.36 A proof rule obtained by reduction

```
/* p */
{? P}
{? SOME s FIXED. (s>0 & T)}

ATTAIN E;
DO UNTIL(B);
/* q */
ASSUME P;
ARBITRARY s FIXED WHERE T;
:           (Loop body)
/* r */
{? B => E}
{? ~B => P}
{? ~B => ~T[0//s]}
{? ~B => T[s-1//s]}
END
/* s */
{! E}
```

Figure 4.37 The PL/CV UNTIL loop rule

invariant P is usually of the form (f => P_1) & (~f => P_2), where P_1 holds at the beginning of the first execution of the loop body, and P_2 holds at subsequent executions. Proofs in the body of the loop will make heavy use of the CASES rule, split on f being either '0'B or '1'B. Essentially, the use of the variable f is a trick that allows the two proofs required by Wirth's rule to be packaged together in one loop body.

```
REQUIRE    P
REQUIRE    SOME s FIXED. (s>  & T)
NEW        <Variables modified by loop body>
OPEN†
PROCLAIM   P
PROCLAIM   T
:          (V-code for loop body)
REQUIRE    B => E
REQUIRE    ~B => P
REQUIRE    ~B => T[s-1//s]
REQUIRE    ~B => ~T[0//s]
CLOSE†
PROCLAIM   E
```

†necessary only when loop has LEAVE statements

Figure 4.38 V-code for UNTIL loops

The rule used for the UNTIL loop is shown in Figure 4.37. In that rule, the loop invariant P is shown to hold at the top of the loop body by showing it holds before the loop begins, and by showing that it holds at the end of the loop body if the loop body is to be executed again. The exit condition E is shown to hold after the loop completes by showing that it holds just before every LEAVE statement that exits the loop, and that it holds at the end of the last execution of the loop body.

The termination rule is similar to the one used for WHILE loops. The assertion required before the loop is SOME s FIXED. (s > 0 & T), rather than SOME s FIXED. (s >= 0 & T), reflecting the fact that the body of the UNTIL loop will be executed at least once. For the same reason, the required assertion involving ~T[0//s] appears at the bottom of the loop.

As was the case in the WHILE loop, point p dominates point q and there is a simple path from p to q that passes through no points in the program, so that $A_p = A_q - M_{p,q}$. Since the GOTO statement has been restricted so that it cannot enter a loop, point p dominates point s as well. However, Theorem 4.15 cannot be applied to points p and s. Since the loop body is executed at least once, there can be points in the program that are on every simple execution path from point p to point s.

However, there is a simple path from s to r that does not pass through any points in the program. And if there are no LEAVE statements for the loop, s dominates r. Then, by Theorem 4.15, $A_s = K_r$. The corresponding V-code is that shown in Figure 4.38, with the OPEN and CLOSE statements deleted.

But this V-code is only correct for loops for which there are no LEAVE statements. In the general case, the following approximation is used.

Approximation 4.17. An execution path may be extended from the top of the body of an UNTIL loop, to the point just after the loop. Given Approximation 4.7, the effect is as if the statement:

```
IF '0'B THEN LEAVE;
```

were inserted at the top of the body of each UNTIL loop.

Given the bogus execution path from q to r provided by Approximation 4.17, Theorem 4.15 can be applied to to points p and s, and we have $A_s = K_p - M_{p,s}$. The OPEN instruction in Figure 4.38 saves $A_p - M_{p,s}$, and the CLOSE instruction restores the Assertion Table to this value. The Proof Checker does not keep track of whether an UNTIL loop contains LEAVE statements; it unconditionally generates the OPEN and CLOSE instructions.

Because of Approximation 4.17, assertions that are known at the end of the loop body and before every LEAVE statement for the loop should be accessible to point s but will not be recognized as such by the Proof Checker. Any such assertions that are subsequently needed as hypotheses must be explicitly included as part of the exit condition of the loop.

```
DO I = L TO U BY K;                 I := L;
:                                   WHILE(I <= U);
END                                 :
                                    I := I + K
                                    END;

        (a)                                (b)
```

Figure 4.39 Reducing an indexed loop to a WHILE loop and assignments

4.17 Indexed Loops

The indexed loop in Figure 4.39a can be reduced to the WHILE loop in Figure 4.39b, assuming K is always positive.

For compatibility with PL/I [49], the semantics of PL/C indexed loops are slightly more complicated. Copies are made of the expressions L, U, and K before the loop begins execution, and the copies are used to control the loop execution. Thus, if U and K have side effects, those effects are performed exactly once, and if the body of the loop modifies variables in U or K, these modifications do not affect the loop control. However, some restrictions have been added to PL/CV: expressions cannot have side effects and the control variable I and all varisbles in L, U, and K are readonly in the body of the loop. Given these restrictions, the reduction shown in Figure 4.39 is valid.

Note that for the reduction to be valid, it is vital that the variables in K are not modified by the loop body, and that the variable I is modified only by the indexing mechanism. In particular, if the variable I or any variables in K are external, procedures that indirectly modify any of these variables are not allowed to be called in the loop body.

A second consequence of these restrictions is that every indexed loop must terminate. Therefore the termination parameter is omitted from the proof rule for indexed loops.

```
{? L > U+1 => P[U+1//I]}
{? P[L//I]}

DO I = L TO U BY 1;
ASSUME P;
{! P}
{! L <= I <= U}
:  (loop body)
P[I+1//I] BY ...
END;

{! P[U+1//I]}
```

(a)

```
{? L > U+1 => P[U//I]}
{? P[L-1//I]}

DO I = L TO U BY 1;
ASSUME P[I-1//I];
{! P[I-1//I]}
{! L <= I <= U}
:  (loop body)
{? P}
END;

{! P[U//I]}
```

(b)

Figure 4.40 Two proof rules for LEAVE-free indexed loops

```
P[L//I] BY ..
L <= L BY ARITH;
L > U+1 => P[U+1//I] BY ...

I := L;

P;
I <= L;
I > U+1 => P[U+1//I];

DO WHILE(I <= U);
ASSUME P & (L <= I) & (I > U+1 => P[U+1//I]);
ARBITRARY S WHERE I-L < S;
I <= U;
~(I-L < 0) BY ARITH, L<=I;
:  (loop body)
P[I+1//I] BY ...
L <= I+1 BY ARITH, L<=I;
~(I+1 > U+1) BY ARITH, I <= U;
I+1 > U+1 => P[U+1//I];

I := I+1;

P;
L <= I;
I > U+1 => P[U+1//I];
END;

P & (L <= I) & (I > U+1 => P[U+1//I]) & ~(I <= U);
(I = U+1) | (I > U+1) BY ARITH, ~(I <= U);
P[U+1//I] BY CASES, (I = U+1) | (I > U+1);
```

Figure 4.41 Using the WHILE rule to prove an indexed loop

In an indexed loop, the loop body is executed once for each element of the set {L ... U} that is congruent to L mod K. When this construct is used, K will almost always be 1. and the loop body will be executed once for each element of {L ... U}. So that the proof rule for indexed loops does not have to involve congruences, K is restricted to be 1 or -1 in PL/CV. Wirth [50] made the same decision when he designed PASCAL. Only the proof rule for the case of K = 1 will be discussed here; there is a similar rule for the case of K = -1.

For any indexed loop that uses an increment greater than 1. there is an an equivalent loop of the form:

```
DO J = 0 TO U BY 1;
I := L + K*J;
:
END;
```

that is allowed in PL/CV, though it might be argued that such a translation is another misguided use of reduction.

Two proof rules for indexed loops for which there are no LEAVE statements are shown in Figure 4.40. It turns out that rule (a) is more convenient for loops of the form DO I = 0 TO N-1 and that (b) is more convenient for loops of the form DO I = 1 TO N. Since the latter for of loops seems to be more common among PL/I programmers, rule (b) probably should have been used in PL/CV, but rule (a) was chosen so that the Proof Checker would not need to test that the loop invariant is of the form P[I-1//I].

In PL/CS, the value of the loop index is set to an undefined value after the loop completes execution. Newly declared variables are set to the same undefined value. If a variable containing this value is accessed, a run-time error occurs.

```
{? P[L//I]}
{? L > U => E}

ATTAIN E;
DO I = L TO U BY 1;
ASSUME P;
{! P}
{! L <= I <= U}
:
{? I ~= U => (P[I+1//I])}
{? I = U => E}
END;

{E}
```

Figure 4.42 A rule for indexed loops with exits

As was the case with newly declared variables, the undefined value is modelled in PL/CV with intentional incompleteness. The proof rules in Figure 4.40 do not allow any conclusions to be drawn concerning the value of the loop index after the completion of the loop.

Note the similarity between the rules for indexed loops and mathematical induction. Any proof involving an indexed loop can be rewritten using the WHILE rule. The resulting proof will be similar to the proof in Figure 4.41. (In the figure, assertions that are justified by "..." are the assertions one would have to prove if one used the indexed loop rule.) The proof using the WHILE rule is much more complicated than the proof that uses the indexed loop rule. The indexed loop construct is an example of a successful use of reduction to derive a proof rule.

The proof rule that is used for indexed loops with LEAVE statements is shown in Figure 4.42. Since the value of the loop index is not known after the loop completes execution, the exit condition E is not allowed to contain an occurrence of the loop index I. If no ATTAIN line is present, the default exit condition is P[U+1//I].

The rule in Figure 4.42 is not as elegant as the rule in Figure 4.40a. Let us refer to U - L + 1 as the number of times the loop body is executed. A properly written loop will work in the case in which the loop body is executed zero times; that case will fit naturally into the logic of the program. But typically, the loop body will never be executed a negative number of times. That is, the assertion U - L + 1 >= 0 can be proved at the point just before the loop begins execution.

When using the rule in Figure 4.40a, the precondition L > U+1 => P[U+1//I] can be proved vacuously for a loop that never executes its body a negative number of times. But when using the rule in Figure 4.42, proving U - L + 1 >= 0 does not vacuously prove the precondition L > U => E; one must also prove L = U+1 => E. A better rule would require that:

(1) P[I+1//I] to hold at the bottom of the loop,

(2) L > U+1 => E to hold before the loop begins, and

(3) I = U+1 => E to be a consequence of the formulas {P} u A_P.

This rule would not single out the loop executing zero times as a special case. Unfortunately, there is no good syntactic place to put the proof of (3).

The accessibility calculations performed for indexed loops are the same as those done for WHILE loops. The V-code generated for the indexed loop is shown in Figure 4.43.

4.18 Jump Statements

There are three kinds of jump statements in PL/CV: the RETURN statement, the LEAVE statement, and the GOTO statement. The proof rules for each of these statements are shown in Figure 4.44. Recall that the assertion

```
REQUIRE   P[L//I]
REQUIRE   L > U => E
NEW       <variables modified by loop body>
NEW       I
OPEN
PROCLAIM  P
PROCLAIM  L <= I <= U
   :      (V-code generated for loop body)
REQUIRE   I ~= U => (P[I+1//I])
REQUIRE   I = U => E
CLOSE
PROCLAIM  E
```

Figure 4.43 V-code generated an indexed loop

```
{? E}          {? E}          {? E}          {? E}
RETURN(x);     RETURN;        LEAVE L;       GOTO L;
{! '0'B}       {! '0'B}       {! '0'B}       {! '0'B}
```

Figure 4.44 Templates for jump statements

(P. p) means, "whenever control reaches point p, the state will satisfy P." Since no state can satisfy '0'B, the assertion ('0'B, p) means "control will never reach p."

The nature of the predicate E varies, depending on the type of the jump. In the case of the RETURN statement, E is the exit condition of the routine containing the statement. In the case of the LEAVE, E is the exit condition of the loop being left.

In the case of the GOTO L, E is a condition that will be proved to hold whenever the label L is reached. The following syntax is used to indicate to the Proof Checker what E is:

```
{? E}
ATTAIN E;
L:;
{E}  .
```

In PL/CS. the target of every GOTO is required to be a null statement. Since E is proved to be accessible to every GOTO L statement and the label

```
OPEN†
REQUIRE   E
CLOSE†
PROCLAIM  '0'B
```

†generated only for GOTO statements

Figure 4.45 The V-code generated for a Jump statement

L, E must hold whenever the label is reached. Because the GOTO statement has been restricted so that the label L must lexically follow the GOTO L, there is no question of termination. As has already been noted, the GOTO is also restricted from entering a loop, the THEN or ELSE part of an IF statement, or one of the WHEN parts of a SELECT statement. More succinctly, if there is a GOTO statement at point p, and the corresponding label is at point q, then every nest containing the V-code generated for q must also contain the V-code generated for p.

The V-code generated for a jump statement is shown in Figure 4.45. The V-code generated for a label is a little trickier. Definition 4.18 and Theorem 4.19 are used to derive the V-code generated for a label.

Definition 4.18. Say that there is a "forward branch" from p to q whenever q immediately follows p in some execution path, q follows p lexically, but q does not immediately follow p lexically.

Forward branches are the result of conditional actions and Approximation 4.17. Note that not all conditional actions result in forward branches. For example, the atomic action at the end of the ELSE part of an IF statement that transfers control to the statement after the IF does not result in a forward branch, because control is transferred from one point to its immediate lexical successor.

Theorem 4.19. Given a label L, let t be the point just after the label. Find the GOTO L that appears lexically first in the program. Next

```
                 /* p */
                    :
                 /* q */
                 OPEN
Nest N              :              Region containing r
                 /* r */
                 GOTO L
                    :
                 /* s */
                 CLOSE(-NEW)
                    :
                 L:
                 /* t */
```

Figure 4.46 Points used in the proof of Theorem 4.19

find the largest nest N that contains the REQUIRE E instruction generated for the GOTO L. but none of the V-code generated for point t. Let q be the point for which the OPEN instruction that begins N was generated. Then $A_t = K_q - M_{q,t}$.

Proof. Let r be the point just before the GOTO L, and let s be the point for which the CLOSE instruction ending N was generated. The theorem is established by showing that Theorem 4.15 can be applied to points q and t. It is necessary to show:

(1) q dominates t, and

(2) for every point in the program, there is a simple path from q to t that does not pass through that point.

To show (1), suppose there is an execution path from the start of the routine to t that does not pass through q. Since q lexically precedes t, such a path is possible only if there is a forward branch from some point p to another point x, where p lexically precedes q, and x precedes t but follows q. The situation is depicted in Figure 4.46.

By examining the V-code generated for each control construct (including GOTO statements and labels, for which the V-code has not yet been fully presented) one can establish that if there is a forward branch from p to x, then a CLOSE(-NEW) instruction is generated for point x. Moreover, if M is the nest ended by that instruction, some of the V-code generated for point p is in the nest M.

Suppose that x and s are the same point x or that point x precedes point s. Then, since point s follows point q, the CLOSE(-NEW) instruction ending nest M is contained in nest N. Therefore, either M and N are the same nest, or nest M is contained in nest N. Since some of the V-code generated for point p is contained in nest N, either p and q are the same point, or point p must lexically follow point q. In either case, the hypothesis that p precedes q has been contradicted.

The other possibility is that the point x follows the point s. Since the nest N, which ends in a CLOSE(-NEW) generated for point s, must extend back as far as the V-code generated for point p, and p precedes q, nest N must properly contain nest M. But nest N cannot contain any V-code for point t, since point s precedes point t. Thus, the hypothesis that M is the largest nest containing the REQUIRE Ė instruction generated for the GOTO L, but not any of the V-code for point t, has been contradicted.

To establish (2), it will be necessary to construct some simple paths from s to t. In general, given two points b and c such that b lexically precedes c, the following construction can be used to find a path from b to c that has the property that if point p_2 follows point p_1 on the path, the p_2 follows p_1 lexically. Any path with this property must be simple.

Start at b and trace the flow of execution toward c. When a jump statement is encountered, use Approximation 4.8 to ignore the jump, and extend the path to the point following the jump statement. When an IF or SELECT statement is encountered, if a clause (THEN, ELSE, WHEN, or OTHERWISE) of that statement contains c, extend the path into that clause. If none of the clauses contains c then trace any path through the statement. Likewise, when a loop is encountered, if the loop body contains c then extend the path into the loop body, and if the loop body does not contain c then trace a path around the loop body. The only case in which this construction can fail is if b and c are in two different clauses of an IF or SELECT statement. Then, when the path is traced to the end of the clause containing b there is no choice but to branch around the clause containing c.

The points q and r cannot be in different clauses of an IF or SELECT statement. If they were, then point q would be in a THEN or WHEN clause. But the V-code generated for a THEN or WHEN clause is a nest. Since the V-code generated for r is contained in the nest begun by an OPEN instruction generated for q, r would have to be in the same THEN or WHEN clause as q. Therefore, a simple path can be traced from q to r. Since r is the point before a GOTO L statement, the path can be extended to t, the point just after the label L. Let S be this path from point q to point t.

All the points through which the path S passes are lexically between points q and s. To establish (2), it is only necessary to construct for any point y that is lexically between points q and s a path T from q to t that does not pass through y. Now matching OPEN and CLOSE(-NEW) instructions were generated for points q and s respectively. The OPEN and CLOSE(-NEW) instructions generated for points q and s must have been the result of a predicate calculus proof, a GOTO statement, or some other control construct. In the first two cases, q and s must be the same point. In the third case, the OPEN and CLOSE(-NEW) instructions were generated because Theorem 4.15 could be applied to points q and s. In all cases, there must be a simple execution path T from point q to point s that does not pass through point y.

Suppose points s and t are in different clauses of an IF or SELECT statement. Then point s must be in a THEN or WHEN clause. Since the V-code generated for a THEN or WHEN clause is a nest, and the V-code generated for points r and s is contained in a common nest, point r must be in the same THEN or WHEN clause as s. But then the GOTO L statement must branch from one clause to another in an IF or SELECT statement, which is not allowed. Therefore points s and t cannot be in different clauses of an IF or SELECT statement, and there must be a simple path U from s to t that passes only through points lexically between s and t. If the paths T and U are concatenated, the result is a simple path from q to t that does not pass through point y. □

When the label L is encountered, an OPEN instruction is inserted in the V-code generated for point q, and and the following V-code:

```
REQUIRE    E
CLOSE-NEW
PROCLAIM   E
```

is generated for the label. The effect of the CLOSE-NEW instruction will be to set the Assertion Table to $K_q - M_{q,t}$ assuming it matches the inserted OPEN instruction.

How could the inserted instructions not match? There are two cases, illustrated in Figure 4.47. The first case, (a), is that the inserted OPEN instruction matches a CLOSE between the end of N and the label. This case cannot occur, because then a nest larger than N would contain the V-code

```
(  ( ... GOTO L ... )  ( ... )  )  ( ... )  L: )
^                    ^                          ^
Inserted OPEN      Nest N                  CLOSE-NEW
```

(a)

```
(  ( ... GOTO L ... )  ( ... )  (  ( ... )  L: )
^                    ^                          ^
Inserted OPEN      Nest N                  CLOSE-NEW
```

(b)

Figure 4.47 Cases in which inserted instructions do not match

generated for the GOTO L but not the label L. The second case, shown in (b), is that the inserted CLOSE instruction matches an OPEN instruction between the end of N and the label. In this case, the GOTO L statement is branching into a nest, which is not allowed.

Here is how the Proof Checker generates V-code for the labels of GOTO statements and detects GOTO statements that branch into nests. Each OPEN instruction contains two pointer fields, named "container" and "predecessor," and a bit field named "closed" that are used to keep track of the structure of the V-code. When a field of a nest or a pointer to a nest is referred to, the field of or a pointer to the OPEN instruction that begins the nest is meant. Recall that a nest N contains zero or more nests N_0, N_1, ... N_{n-1}. The container field of each subnest N_i contains a pointer to the nest N. The predecessor field of N_0 is nil, and for each positive i, the predecessor of field of N_i contains a pointer to N_{i-1}. The closed field of a nest will contain '1'B whenever the CLOSE(-NEW) instruction that completes the nest has been generated.

The partially generated V-code will be the prefix of a nest: a

```
( ( ) ( ) ( ( ) ( ( ) ( )
↑         ↑     ↑     ↑
 Unmatched stack (top) MRC
```

Figure 4.48 The unmatched stack and the variable MRC

sequence of one or more OPEN instructions, each separated by zero or more nests and V-code instructions other than OPEN, CLOSE, and CLOSE-NEW. Pointers to these OPEN instructions are kept on a stack called "Unmatched," with a pointer to the the OPEN instruction that occurs lexically latest on the top of the stack. Let N be the nest pointed to by the top of the Unmatched stack. Then the variable MRC ("most recently closed") is <u>nil</u> if N has no subnests, and is otherwise a pointer to the subnest of N that occurs lexically latest. An example is shown in Figure 4.48.

Finally, there is an integer field called "count" in each OPEN instruction that will contain a nonnegative integer. The count field indicates the number of OPEN operations that are to be performed when the instruction is executed. When the count field is greater than one, the single instruction is considered to be an abbreviation for a series of instructions. The container and predecessor fields of an actual instruction are associated with first instruction in the represented series. The container fields of the other members of the series implicitly point to the immediately preceding member, and the predecessor fields are implicitly <u>nil</u>. The closed fields of every member of the series implicitly have the value '1'B. When container or predecessor fields point to an OPEN instruction with a count field greater than 1, which of the represented instructions is meant is ambiguous.

Here is how the various pointers are maintained. When an OPEN instruction is generated, its container field is set to a copy of the top of the Unmatched stack, its predecessor field is set to MRC, and its closed field is set to '0'B. A pointer to the instruction is pushed onto the Unmatched stack. Since the nest begun by the OPEN instruction is completely empty, MRC is set to nil.

When a CLOSE instruction is generated (for a construct other than a label), it must match the OPEN instruction pointed to by the top of the Unmatched stack. The closed bit of that OPEN instruction is set to '1'B. The pointer on the top of the stack is popped and stored in MRC.

The V-code generated for a GOTO L instruction shown in Figure 4.45 is abbreviated in the implementation by a special instruction VGOTO that has the same effect. When the VGOTO instruction is generated, the predicate E is not known. The VGOTO instructions generated for all the GOTO L instructions are kept on a linked list whose head is stored in the symbol table entry for L. When the label L is encountered, the list of GOTO instructions is traversed, and the pointer field that was used to link the instructions together is updated to point to a REQUIRE E instruction generated for the label. The last VGOTO instruction on the list will have been generated for the lexically first GOTO L statement. An OPEN instruction must be inserted somewhere before this VGOTO instruction.

Since no V-code has yet to be generated for the label, the largest nest containing the VGOTO instruction but not the V-code for the label will be the largest closed nest containing the VGOTO. Tho following loop is used to find the OPEN instruction that begins the nest:

```
                     inserted
                        ↓
( (  ) (  ) ( (  ) (  ) (  )
↑           ↑         ↑
q           p        MRC
```

Figure 4.49 Modifying the structure of the V-code

```
( q := a pointer to the VGOTO instruction
; repeat
     p := q
  ;  q := p.container
  until ~q.closed
) .
```

Since the V-code for a routine begins with an OPEN instruction whose matching CLOSE instruction is the end of the routine, and therefore will not have been generated when the label is encountered, the execution of the above loop is always well-defined. If q.container points to an OPEN instruction with a count field greater than one, the statement "q := p.container" can traverse several implicit container links at once. When the loop completes, p will be a pointer to the largest closed nest containing the VGOTO instruction (which may be the VGOTO instruction itself) and q will be its unclosed container. If q does not point to the same instruction as the top of the Uncloaed stack, the situation is as shown in Figure 4.47b; the GOTO L statement is branching into a nest.

An OPEN instruction is effectively generated before p by incrementing p.count, and a matching CLOSE-NEW instruction is generated for the label. These insertions modify the structure of the V-code; various fields must be updated to reflect this new structure (see Figure 4.49). The inserted OPEN inherits the container and predecessor fields of the instruction before which it was inserted; these fields correctly reflect the new structure. The closed field, which is '1'B, is correct because the inserted OPEN instruction has a matching CLOSE-NEW instruction.

Finally, the container fields of the successors of the nest to which p points and the value of MRC must be corrected. The following statements are used:

```
( r := MRC
; while r ≠ p
  do r.container := p
  ; r := r.predecessor
  od
; MRC := p
).
```

During the execution of the above loop, the variable r always points to an OPEN instruction. But if the OPEN instruction has a count field greater than 1, there is an ambiguity: to which instruction in the represented series should r refer? The predecessor and container fields that are manipulated in the loop belong to the lexically first OPEN instruction in the represented series; we must convince ourselves that this is the correct instruction. We know that if the chain of predecessor fields in the instructions whose container fields must be updated is followed, the chain will eventually reach p. But the predecessor field of any but the first in a series of instructions is implicitly nil. Therefore, it is always the first instruction in the series that should be manipulated.

The V-code generated for the label is:

```
REQUIRE    E
CLOSE-NEW
PROCLAIM  E.
```

The use of the CLOSE-NEW instruction in the V-code for SELECT and IF statements has been an optimization; in each case NEW instructions could have been generated instead. The CLOSE-NEW instruction is more of a necessity

```
I := 1;

DO WHILE('1'B);
    IF   I > N
    THEN DO;
         FOUND = '0'B;
         LEAVE;
         END;

    IF   A(I) = KEY
    THEN DO;
         FOUND = '1'B;
         LEAVE;
         END;

    I := I + 1;
END;
```

Figure 4.50 An erroneous linear search program

here. Since the location of the beginning of the nest in which all the varible modifications must be located is not known until the label is encountered, a fairly elaborate system of bookkeeping would have to be devised to generate the proper NEW instructions.

The PL/CS langauge is advertised [24] as an "almost subset" of PL/C. Included in the "almost" part is the LEAVE statement, which IBM has recently added to PL/I [44]. In PL/I, the DO and END that are used to specify grouping in IF and SELECT statements actually comprise a degenerate loop whose body is executed exactly once. In fact, there is just one loop construct in PL/I. The phrases WHILE(B), UNTIL(B), and I = L TO U BY K are considered "options" and can be used in any or all combinations, in conjunction with LEAVE. In an attempt to introduce some discipline into the language, the designers of PL/CS saw fit to make the restriction that at most one option can appear on a loop construct, and only loops with such an option can be exited via a LEAVE.

A reason for forbidding LEAVE statements in degenerate loops is that they are error-prone. As an example, consider the intended linear search

(a)	(b)
	ATTAIN E;
	L: DO;
	⋮
GOTO L;	LEAVE L;
⋮	⋮
GOTO L;	LEAVE L;
⋮	⋮
GOTO L;	LEAVE L;
⋮	⋮
ATTAIN E;	END;
L: ;	

Figure 4.51 Replacing the GOTO statement with the LEAVE statement

(a)

```
<proof of P>

ATTAIN E;
L: DO WHILE(B);
ASSUME P;
:
<proof of E, given P and ~B>
:
<proof of E>
LEAVE L;
:
END;
```

(b)

```
ATTAIN E;
L: DO;

DO WHILE(B);
ASSUME P;
:

<proof of E>
LEAVE L;
:
END;

<proof of E, given P and ~B>
END;
```

Figure 4.52 Two ways of exiting a loop

in Figure 4.50. The intent is that the LEAVE statements exit the loop, but in actuality the LEAVE statements exit the THEN and ELSE parts of the IF. In the positions in which they are written, the LEAVE statements are equivalent to null statements. The program in Figure 4.50 is not allowed in PL/CS.

However, there is a use for LEAVE statements that exit degenerate loops. When the Local Context Checker is generating V-code for GOTO statements, it is in essence replacing the sequence of statements in Figure 4.51a with those in Figure 4.51b. Not only is it easier to generate V-code for the LEAVE statements than the GOTO statements, but the overall proof syntax is cleaner. The ATTAIN line is used in a manner that is more consistent with the other other constructs.

PL/CV would be simplified, without any loss of power, if the GOTO were eliminated, and the restrictions about LEAVE were reversed so that LEAVE statements could be used only to exit degenerate loops. The restriction that all LEAVE statements be labeled should prevent errors of the type shown in Figure 4.50. Elimination of LEAVE statements that exit loops directly would reduce the complexity of all the loop rules. The effect of a loop exit can still be obtained by exiting a surrounding DO group, as shown in Figure 4.52. The two programs and their proofs are essentially the same; the only difference is that in (b) there is a convenient syntactic place to put the proof of E given P and ~B.

A more radical proposal for simplifying the PL/CV control constructs is presented in section 4.20.

4.19 The Plethora Problem

One of the problems with PL/CV is that there is a fairly large collection of control constructs: IF, SELECT, WHILE loops, UNTIL loops, indexed loops, LEAVE statements, RETURN statements, and GOTO statements. Each of these constructs has a different proof rule, and it can be hard to keep the rules straight.

It is a well-known result in the theory of computation that there

$$a + b = b + a$$
$$a \times b = b \times a$$
$$a + (b + c) = (a + b) + c$$
$$a \times (b \times c) = (a \times b) \times c$$
$$a \times (b + c) = (a \times b) + (a \times c)$$

Figure 4.53 Some simple rules about addition and multiplication

are languages with an extremely small set of constructs in which programs can be written to compute any partial recursive function. One solution to the "plethora problem" is to choose one of these languages L, say Turing machine quintuples or Dijkstra's guarded commands [29], and give some simple, elegant proof rules for the constructs in L. Next, we announce to the world that all the other constructs are "harmful." We cite results in computability theory [39] to prove that any program that is written using the harmful constructs can be purified to use only the constructs in L. We can even cite relative completeness results [26] to support the claim that if there is a proof at all, there is a proof that uses the elegant rules.

This solution involves using reduction to reduce the number of constructs in the language. The problem is that simple, elegant proof rules in no way guarantee simple, elegant proofs. For example, Peano's Axioms are a simple and elegant system for reasoning about the integers. The rules in Figure 4.53 can be derived from Peano's Axioms. Using these rules and a substitution rule, one can prove:

$$(a+b+c) \times (d+e+f) = a\times d + a\times e + a\times f + b\times d + b\times e + b\times f + c\times d + c\times e + c\times f \ .$$

The proof is shown in Figure 4.54. Recall that + and × are binary operators, so that for example, $a + b + c + d$ is an abbreviation for $((a + b) + c) + d)$. The proof in Figure 4.54 uses only the simple rules in Figure 4.53. The reader can judge the elegance of the proof for him-

$$
\begin{aligned}
& (a + b + c) \times (d + e + f) \\
= \; & (a + b + c)\times(d + e) + (a + b + c)\times f \\
= \; & (a + b + c)\times d + (a + b + c)\times e + (a + b + c)\times f \\
= \; & d\times(a + b + c) + (a + b + c)\times e + (a + b + c)\times f \\
= \; & d\times(a + b + c) + e\times(a + b + c) + (a + b + c)\times f \\
= \; & d\times(a + b + c) + e\times(a + b + c) + f\times(a + b + c) \\
= \; & d\times(a + b) + d\times c) + e\times(a + b + c) + f\times(a + b + c) \\
= \; & d\times a + d\times b + d\times c + e\times(a + b + c) + f\times(a + b + c) \\
= \; & d\times a + d\times b + d\times c + (e\times(a + b) + e\times c) + f\times(a + b + c) \\
= \; & d\times a + d\times b + d\times c + (e\times a + e\times b + e\times c) + f\times(a + b + c) \\
= \; & d\times a + d\times b + d\times c + (e\times a + e\times b + e\times c) + (f\times(a + b) + f\times c) \\
= \; & d\times a + d\times b + d\times c + (e\times a + e\times b + e\times c) + (f\times a + f\times b + f\times c) \\
= \; & a\times d + d\times b + d\times c + (e\times a + e\times b + e\times c) + (f\times a + f\times b + f\times c) \\
= \; & a\times d + b\times d + d\times c + (e\times a + e\times b + e\times c) + (f\times a + f\times b + f\times c) \\
= \; & a\times d + b\times d + c\times d + (e\times a + e\times b + e\times c) + (f\times a + f\times b + f\times c) \\
= \; & a\times d + b\times d + c\times d + (a\times e + e\times b + e\times c) + (f\times a + f\times b + f\times c) \\
= \; & a\times d + b\times d + c\times d + (a\times e + b\times e + e\times c) + (f\times a + f\times b + f\times c) \\
= \; & a\times d + b\times d + c\times d + (a\times e + b\times e + c\times e) + (f\times a + f\times b + f\times c) \\
= \; & a\times d + b\times d + c\times d + (a\times e + b\times e + c\times e) + (a\times f + f\times b + f\times c) \\
= \; & a\times d + b\times d + c\times d + (a\times e + b\times e + c\times e) + (a\times f + b\times f + f\times c) \\
= \; & a\times d + b\times d + c\times d + (a\times e + b\times e + c\times e) + (a\times f + b\times f + c\times f) \\
= \; & a\times d + b\times d + c\times d + (a\times e + b\times e + c\times e) + (a\times f + b\times f) + c\times f \\
= \; & a\times f + b\times f + (a\times d + b\times d + c\times d + (a\times e + b\times e + c\times e)) + c\times f \\
= \; & a\times f + b\times f + (a\times d + b\times d + c\times d + (a\times e + b\times e) + c\times e) + c\times f \\
= \; & a\times f + b\times f + (a\times e + b\times e + a\times d + b\times d + c\times d + c\times e) + c\times f \\
= \; & a\times f + b\times f + (a\times e + b\times e + a\times d + b\times d + c\times d) + c\times e + c\times f \\
= \; & a\times f + b\times f + (a\times e + b\times e + a\times d + b\times d) + c\times d + c\times e + c\times f \\
= \; & a\times e + b\times e + a\times d + b\times d + (a\times f + b\times f) + c\times d + c\times e + c\times f \\
= \; & a\times e + b\times e + a\times d + b\times d + a\times f + b\times f + c\times d + c\times e + c\times f \\
= \; & a\times d + (a\times e + b\times e) + b\times d + a\times f + b\times f + c\times d + c\times e + c\times f \\
= \; & a\times d + (a\times e + b\times e + b\times d) + a\times f + b\times f + c\times d + c\times e + c\times f \\
= \; & a\times d + (a\times e + (b\times e + b\times d)) + a\times f + b\times f + c\times d + c\times e + c\times f \\
= \; & a\times d + (a\times e + (b\times d + b\times e)) + a\times f + b\times f + c\times d + c\times e + c\times f \\
= \; & a\times f + (a\times d + (a\times e + (b\times d + b\times e))) + b\times f + c\times d + c\times e + c\times f \\
= \; & a\times f + (a\times d + (a\times e + b\times d + b\times e)) + b\times f + c\times d + c\times e + c\times f \\
= \; & a\times f + (a\times d + (a\times e + b\times d) + b\times e) + b\times f + c\times d + c\times e + c\times f \\
= \; & a\times f + (a\times d + (a\times e + b\times d)) + b\times e + b\times f + c\times d + c\times e + c\times f \\
= \; & a\times f + (a\times d + a\times e + b\times d) + b\times e + b\times f + c\times d + c\times e + c\times f \\
= \; & a\times f + (a\times d + a\times e) + b\times d + b\times e + b\times f + c\times d + c\times e + c\times f \\
= \; & a\times d + a\times e + a\times f + b\times d + b\times e + b\times f + c\times d + c\times e + c\times f
\end{aligned}
$$

Figure 4.54 A complex proof that uses simple rules

self, but the proof can hardly be called simple.

The PL/CV ARITH rule is not a simple rule; nor is it exceedingly elegant. But using this rule, the proof in Figure 4.54 can be reduced to one step. The point is that simplicity is not necessarily a virtue in proof rules. Unnecessary complexity is certainly to be avoided, but complex proof rules are acceptable if their use simplifies proofs.

In the next section a control construct called an omnibus statement is proposed as a solution to the plethora problem. IF statements, SELECT statements, WHILE loops, UNTIL loops, LEAVE statements, RETURN statements, and GOTO statements are all special cases of the omnibus statement, and the proof rules for these constructs are all special cases of the proof rule for the omnibus statement. That is, if one writes and proves, say, a SELECT statement in PL/CV, one can write an equivalent omnibus statement, and use essentially the same proof used for the SELECT statement to prove the omnibus statement.

4.20 <u>Omnibus Statements</u>

Figure 4.55 contains a context-free grammar for omnibus statements. (Standard extensions are used; brackets indicate optional items and Kleene star indicates zero or more repetitions.) A <statement> can either be a primitive statement, such as an assignment statement or a procedure call, or it can be an omnibus statement.

The syntax in Figure 4.55 should not be taken too seriously. It uses as little punctuation as possible while still avoiding ambiguity. Constructs of any complexity written in this syntax are difficult to understand because it does not have enough redundancy. I have experimented with a few variations on the syntax, some better than the one in Figure 4.55, but I do not have a single solution with which I am completely satisfied.

Every <u>loop</u> and <u>exit</u> in a program is associated with an omnibus statement in the following straightforward manner. If the <u>loop</u> or <u>exit</u> is followed by a label, associate it with the smallest omnibus statement that

```
<omnibus>    →  [<label>:] do <tail>
<tail>       →  <statement>* <terminus>
<terminus>   →  when <Boolean expression> <tail> <tail>
<terminus>   →  loop [<label>]
<terminus>   →  exit [<label>]
<label>      →  <identifier>
```

Figure 4.55 The syntax of omnibus statements

contains the loop or exit and begins with that label. If there is no such omnibus statement, the program is syntactically incorrect. If the loop or exit is not followed by a label, associate it with the smallest omnibus statement containing the loop or exit.

The semantics of the omnibus statement then are given operationally as follows. To perform an <omnibus> perform its <tail>. To perform a <tail>, perform each <statement> in the <tail>, and then perform the <terminus>. (This sequence of executions is aborted if there is an explicit transfer of control within any <statement> or the <terminus>.) To perform a <terminus> beginning with when, first evaluate the Boolean expression. If the value is '1'B, perform the first <tail>, and if the value is '0'B, perform the second <tail>. To perform a loop, transfer control to the beginning of the <omnibus> associated with the loop. To perform an exit, transfer control to the statement following the <omnibus> associated with the exit.

In a proof, three predicates are associated with every omnibus statement: an exit condition E, a loop invariant P, and a termination predicate T that has a free variable s. The following syntax:

```
L: do
attain E
assume P
arbitrary s where T
   .
   .
   .
```

can be used in a proof to indicate what the three predicates are. If there is no loop associated with the statement, the loop invariant and termination predicates are not really necessary. Therefore, the assume and arbitrary lines can be omitted, in which case P and T default to '1'B and s = 0 respectively.

The proof rule for the omnibus statement is just a combination of rules that have already been presented. The invariant P must be known at the point preceding the do and before every loop associated with the statement. The exit condition E must be known at every exit associated with the statement. The assertion SOME s FIXED. s >= 0 & T must be known before the do, and s ~= 0 and T[s-1//s] must be known before every loop associated with the statement. The formula B is known at the beginning of the first <tail> following a when B, ~B is known at the beginning of the second <tail> following the when, and E is known after the omnibus statement.

All of the control constructs in PL/CV (except indexed loops), and some that are not, are special cases of the omnibus statement. Some examples are shown in Figure 4.56. The proof rules for these control constructs can be derived by writing the equivalent omnibus statement and the proof rule for that statement. The rule used in PL/CV for the UNTIL loop was first discovered by this process.

If the omnibus statement is used in an expression language, every

IF statement:

```
do
when B
    ⋮   <then part>
    exit
    ⋮   <else part>
    exit
```

SELECT statement:

```
do
    when B1
    ⋮  (perform if B1)
    exit
    when B2
    ⋮  (perform if ~B1 & B2
    exit
    when B3
    ⋮  (perform if ~B1 & ~B2 & B3)
    exit
    ⋮  (perform if ~B1 & ~B2 & ~B3)
    exit
```

WHILE loop:

```
do when ~B exit
    ⋮  <loop body>
    loop
```

UNTIL loop:

```
do
    ⋮  <loop body>
    when B loop
    exit
```

Dahl's [53] n + 1/2 loop:

```
do
    ⋮
    when B exit
    ⋮
    loop
```

PL/CS Restricted GOTO:

```
L: do
   ⋮
   do exit L
   ⋮
   exit
```

Figure 4.56 Special cases of the omnibus statement

exit is preceded by an expression, which is the value of the associated omnibus statement if it is terminated via that exit. Thus, loops have meaningful values just like any other construct. In ALGOL 68 [70], for example, every loop has the value **void**.

The omnibus statement was not derived by examining all the control constructs in PL/CV and trying to generalize them. When using the C programming language [52], I found myself developing loops by writing loops equivalent to:

```
DO WHILE('1'B);
  .
  .
  .
END;
```

and within the loop body using break and continue statements. The break statement is analogous to the PL/I LEAVE statement, and the continue statement is essentially a GOTO statement to the top of the loop body.

Another way of viewing the omnibus statement is as the definition of a parameterless recursive procedure in which each exit is a RETURN and each loop is a CALL. Methods for deriving programs proposed by Hehner [40] and Manna and Waldinger [59] are especially suited to the omnibus statement.

Computing accessibility efficiently for the general omnibus statement is a fairly difficult problem. However, in a language in which omnibus statements are the only control construct, and in which the statement has been restricted so that:

(1) no loop or exit is followed by a label

(2) every omnibus statement has at least one exit associated with it

it is not too difficult to generate V-code that computes accessibility. Restriction (1) is analogous to restricting GOTO statements so that they can exit a single level. Any program that meets restriction (1) but not (2) cannot terminate. If a program meets these two restrictions, there will be no unreachable points in the program. Thus, Theorem 4.15 can be applied without using the Approximation 4.8. And since there is no UNTIL statement, there is no need for Approximation 4.17.

An omnibus statement that meets restrictions (1) and (2) can be visualized as a tree in which each edge contains a series of statements. Each when is a branch point; the Boolean expression indicates which subtree should be chosen when control reaches the branch point. Every leaf of the tree is either <u>loop</u>, indicating the statement should be repeated, or <u>exit</u>, indicating the execution of the statement is complete.

Let p be the point in a program just before an omnibus statement S, and q be the point just afterwards. A consequence of restrictions (1) and (2) is that p dominates both q and any point within S. Moreover, because there is an execution path from p to any point within S, the set of all variables that can be modified on any simple execution path from p to q is simply the set of all variables that can be modified by any simple statement in S.

Let p be a point in a program just before the <u>do</u> of an omnibus statement, and let r be the point just after the <u>do</u>. We have seen that p must dominate r, and since there is a simple execution path from p to r that does not pass through any other points in the program, by Theorem 4.15, $A_r = K_p - M_{p,r}$. Using the tree model of an omnibus statement, $V_{p,r}$ can be found by examining all the paths in the tree that lead from the root to a <u>loop</u>, and computing the set of all variables that can be modified by a statement on one of those paths.

Now let us consider what A_q should be. Again using the tree model of an omnibus statement, if W is a <u>when</u>, define it to be an "upper bound" if every <u>exit</u> in the tree has W as an ancestor. If the statement has only one <u>exit</u> associated with it, define that <u>exit</u> to be an upper bound as well. It is easy to see that if W_1 and W_2 are distinct upper bounds either W_1 is an ancestor of W_2, or vice versa. Therefore, we can define the "least upper bound" as the unique upper bound W such every other upper bound is an ancestor of W.

Any execution path that extends to q, the point following the omnibus statement, must pass through an *exit*. Moreover, any path from the beginning of the program to an *exit* must pass through every *when* that is the ancestor of the *exit*, so that the path must pass through every upper bound. If S has only one *exit* associated with it, any path to q must pass through the *exit*. Therefore, every upper bound dominates q.

Let w be the point in the program just before the least upper bound W. Clearly there must be some path from w to q. Let s be any point on this path. To apply Theorem 4.15 to w and q, we must show there is a path from w to q that does not pass through s.

The upper bound W cannot be an *exit*, since no simple path from an *exit* to q can pass through any other points, and by hypothesis, there is a path from w to q that passes through s. Therefore, there is more than one *exit*, and W is a *when*.

Because W is an upper bound, every *exit* in S must be in one of the two tails following W. If one tail following W does not contain an *exit*, then the other tail must contain every *exit* in S. But then the least upper bound would not be W, but would instead be contained in the tail following W that contained every *exit*. Therefore there must be at least one *exit* in each tail.

A path that extends from w to a *loop* and back to the *do* must pass again through w before it can get to q, because W dominates every *exit*. Therefore, every simple path from w to q must extend to an *exit* and then to q. The point s, through which a simple path from w to q passes, must then be contained in one of the two tails following W, and there must be a simple path from w to q through an *exit* in the other tail that does not pass through point s.

Therefore by Theorem 4.15, $A_q = K_w - M_{w,q}$. If W is an *exit* then

V-code used	Position of least upper bound
a	(1) the least upper bound is in the second <tail>
b	(2) the least upper bound is in the first <tail>
c	(3) the *when* is the least upper bound
a or b	(4) none of the above

Table 4.57 Four cases involving *when* <tail> <tail>

```
OPEN
PROCLAIM B
  :  (V-code for first tail)
CLOSE
PROCLAIM ~B
  :  (V-code for second tail)
```

(a)

```
OPEN
PROCLAIM ~B
  :  (V-code for second tail)
CLOSE
PROCLAIM ~B
  :  (V-code for first tail)
```

(b)

```
OPEN
OPEN
PROCLAIM B
  :  (V-code for first tail)
CLOSE
PROCLAIM ~B
  :  (V-code for second tail)
CLOSE-NEW
```

(c)

Figure 4.58 V-code generated for *when* <tail> <tail>

$M_{w,q}$ is empty. If w is a *when*, $V_{w,q}$ is the set of all variables that can be modified by statements along the simple paths from w to each *exit*.

Here then, is V-code that can be generated for a restricted omnibus statement, given in terms of the grammar in Figure 4.55. For an omnibus statement, first generate:

```
REQUIRE P
REQUIRE SOME s FIXED. s >= 0 & T;
NEW     <every variable in V_p,r>
```

then generate V-code for the tail, and finally generate a PROCLAIM E. For a tail, generate V-code for each statement, and then generate V-code for the terminus.

The V-code generated for a terminus depends on which of the three kinds of terminus it is. For a <u>loop</u>, the V-code:

```
REQUIRE  P
REQUIRE  s ~= 0
REQUIRE  T[s-1//s]
```

is generated. For an <u>exit</u>, the instruction REQUIRE E is generated.

For a terminus of the form <u>when</u> B <tail> <tail>, one of the V-code sequences shown in Figure 4.58 is generated. Which of the three sequences is used depends on the position of the terminus vis-a-vis the least upper bound. The four cases, and the sequence that is used in each case, are listed in Table 4.57.

If s and t are the points in the program just before the start of each tail, by the same argument used for the IF statement, $A_s = A_t$. Therefore, the V-code generated for the first tail is enclosed in a pair of OPEN and CLOSE instructions.

The tail immediately following the <u>do</u> will be referred to as the main tail. The V-code generated for the main tail must set the Assertion Table to A_q, which we have seen is equal to $K_w - M_{w,q}$.

If the omnibus statement contains exactly one <u>exit</u>, either case (1) or (2) in Table 4.57 will apply to every terminus beginning with <u>when</u>. In each case, V-code is generated for the tail containing the *exit* second. Therefore the last V-code instruction generated for the main tail must be the REQUIRE E generated for the *exit* that is the least upper bound. Therefore, after the V-code generated for the main tail is executed, the assertion table will contain K_w. Given that there is only one *exit*, $V_{w,q}$ is empty, and $K_w = K_w - M_{w,q}$.

Next consider the case in which there is more than one *exit*. In case (1) or (2) of Table 4.57, the tail that contains the least upper bound contains every *exit*; the V-code for that tail is generated second. The last V-code instruction generated for the main tail will be the final CLOSE-NEW instruction in Figure 4.58c that is generated for the least upper bound. After the V-code generated for the main tail is executed, the Assertion Table will therefore contain $K_v - M_{v,q}$.

CHAPTER 5
A DECISION PROCEDURE FOR TRANSFORMATIONS

5.1 Transformations and Replacement Systems

The V-code Interpreter uses two decision procedures to implement the PL/CV automatic rules. The first of these two decision procedures is discussed in this chapter. An algorithm is given for finding all consequences of the axioms in Figures 5.1 and 5.2. The symbol $\equiv$ is used in the axioms to stand for either = or <=>, depending on the type of the operands.

Given any set TA of transformation axioms of the form $R \equiv S$, we say that a "transformation" can be applied to the formula P to yield Q whenever there is an axiom $R \equiv S$ such that Q can be obtained by replacing in P a single occurrence of R by S or a single occurrence of S by R.

Define the notation $P \leftrightarrow_T Q$ to mean that a transformation can be applied to P to obtain Q. Then define the relation $\equiv_T$ to be $\leftrightarrow_T{}^*$, the reflexive transitive closure of $\leftrightarrow_T$. If $\leftrightarrow_T$ is defined with respect to a set of axioms TA, then a sequence of formulas $P_0, P_1, \ldots P_n$ such that for each i in $\{1 \ldots n\}$, $P_{i-1} \leftrightarrow_T P_i$ is called a "derivation of P_n from P_0 using TA," and $P \equiv_T Q$ if and only if there is such a derivation.

The problem discussed in this chapter is that of computing the relation $\equiv_T$ defined by the axioms in Figures 5.1 and 5.2. The solution is to show how to compute a normalization function N with the property that $N[\![P]\!] = N[\![Q]\!]$ if and only if $P \equiv_T Q$.

The function N is based on a replacement system. A replacement system consists of a domain D and a binary relation $\Rightarrow$ that is a subset of $D \times D$. A sequence $P_0, P_1, \ldots P_n$, where for each i in $\{1 \ldots n\}$,

$P \equiv Q$ (where P and Q contain no variables and have the same value)

(P <=> Q)	≡	(Q <=> P)
P <=> '0'B	≡	~P
P <=> '1'B	≡	P
P <=> P	≡	'1'B
P <=> ~P	≡	'0'B
('0'B => P)	≡	'1'B
('1'B => P)	≡	P
(P => '0'B)	≡	~P
(P => '1'B)	≡	'1'B
(P => P)	≡	'1'B
(P => ~P)	≡	~P
(~P => P)	≡	~~P
(P | '0'B)	≡	P
(P | '1'B)	≡	'1'B
(P | P)	≡	P
(P | Q)	≡	(Q | P)
(P | Q) | R	≡	P | (Q | R)
(P & '0'B)	≡	'0'B
(P & '1'B)	≡	P
(P & P)	≡	P
(P & Q) & R	≡	P & (Q & R)
(P & Q)	≡	(Q & P)
(P & ~P)	≡	'0'B

Figure 5.1 The Transformation Axioms, Part 1

$P_{i-1} \Rightarrow P_i$,' is called a derivation of P_n from P_0. The length of the derivation is said to be n. Note that the notion of a derivation has been defined with respect to transformation axioms and with respect to replacement rules. The two kinds of derivations are quite similar. The difference between them is if P can be derived from Q using transformation axioms, then Q can be derived from P, but derivations using replacement rules in general work in only one direction.

Rosen [67] has defined a special class of replacement systems called subtree replacement systems. In a subtree replacement system, the domain is a set of trees, and the relation $\Rightarrow$ is defined by a set of replacement rules of the form $P \rightarrow Q$, where P and Q are trees. We say a replacement

(P = Q)	≡	(Q = P)
(P = Q)	≡	($^{-}$P = $^{-}$Q)
(P = Q)	≡	(P + R) = (Q + R)
(P < Q)	≡	(P + R) < (Q + R)
(P < Q)	≡	(P <= Q-1)
(P < Q)	≡	(Q > P)
(P <= Q)	≡	(P + R) <= (Q + R)
(P <= Q)	≡	(P < Q+1)
(P <= Q)	≡	(Q >= P)
(P > Q)	≡	(P + R) > (Q + R)
(P > Q)	≡	(P >= Q+1)
(P > Q)	≡	(Q < P)
(P >= Q)	≡	(P + R) >= (Q + R)
(P >= Q)	≡	(P > Q-1)
(P >= Q)	≡	(Q <= P)
(P + 0)	≡	P
(P + Q) + R	≡	P + (Q + R)
(P + Q)	≡	(Q + P)
(P + $^{-}$P)	≡	0
(P - Q)	≡	P + $^{-}$Q
$^{-}$(P + Q)	≡	($^{-}$P + $^{-}$Q)
$^{--}$Q	≡	Q
(P × 1)	≡	P
(P × Q)	≡	(Q × P)
(P × Q) × R	≡	P × (Q × R)
(P ** 1)	≡	P
(P / 1)	≡	P

Figure 5.2 The Transformation Axioms, Part 2

rule P → Q can applied to a tree R to obtain S whenever Q can be substituted for a single occurrence of P in R to yield S. The relation R ⇒ S is then defined to hold whenever there is a replacement rule that can be applied to R to obtain S.

Suppose it is possible to derive a formula Q from P. At any point in the derivation, there will be in general a number of replacement rules that can be applied. If it is impossible to select a rule to be applied that results in a dead end R from which Q cannot be derived, the system is said to have the Church-Rosser property. Formally, the system is defined to be "Church-Rossor" whonever for all P, Q, and R, P $\Rightarrow^*$ Q and P $\Rightarrow^*$ R imply the existence of an S such that Q $\Rightarrow^*$ S and R $\Rightarrow^*$ S. That is, if two derivations that each begin with P branch off in different directions, they can always be brought back together with a suitable sequence of replacement rules.

A replacement system is defined to be "finite" whenever there are no endless derivations. In a finite replacement system, if one begins with an element, and repeatedly applies rules, one will eventually obtain an element to which no rule can be applied. Such an element is said to be "irreducible." Whenever a finite replacement system is Church-Rosser, it is said to be "confluent." These definitions have been adapted from ones in a paper by Sethi [69].

We are now prepared to define the normalizing function N. It is defined in terms of a confluent subtree replacement system called TRANS, which will be presented in section 5.2. If P is a formula, define N⟦P⟧ to be the unique formula Q such that P $\Rightarrow^*$ Q and Q is irreducible. N⟦P⟧ is called "the normal form of P." The following theorem shows that if TRANS is confluent, then N truly is a function.

<u>Theorem</u> 5.1. Given a confluent replacement system and an element P, there exists an element Q_1 such that P $\Rightarrow^*$ Q_1 and Q_1 is irreducible. Moreover, if there exists an element Q_2 such that P $\Rightarrow^*$ Q_2 and Q_2 is irreducible then $Q_1 = Q_2$.

<u>Proof</u>. Begin with P and keep applying replacement rules. Because the system is finite, eventually an irreducible element Q_1 will be derived. If there were an irreducible element Q_2, distinct from Q_1, such

that $P \Rightarrow^* Q_2$. by the Church-Rosser property, there would be an element R such that $Q_1 \Rightarrow^* R$ and $Q_2 \Rightarrow^* R$. Since Q_1 and Q_2 are irreducible, both derivations must be of length zero. Therefore, $Q_1 = R = Q_2$, contradicting the assumption that Q_1 and Q_2 are distinct. □

A set of transformation axioms is extracted from the replacement system as follows. For formulas P and Q, $P \equiv Q$ is an axiom if and only if $P \rightarrow Q$ is a replacement rule. The following theorem ensures that the normalization function N has the desired property.

<u>Theorem</u> 5.2. If TRANS is confluent, then for any formulas P and Q, $P \equiv_T Q$ if and only if $N[\![P]\!] = N[\![Q]\!]$.

<u>Proof</u>. From the definitions of $\leftrightarrow_T$ and $\Rightarrow$, we have $P \leftrightarrow_T Q$ if and only if $P \Rightarrow Q$ or $Q \Rightarrow P$.

Suppose $N[\![P]\!] = N[\![Q]\!] = R$. Then $P \Rightarrow^* R$ so that $P \leftrightarrow_T^* R$, and therefore $P \equiv_T R$. By the same reasoning, $Q \equiv_T R$. Since $P \equiv_T R$ and $R \equiv_T Q$, $P \equiv_T Q$.

To prove that $P \equiv_T Q$ implies $N[\![P]\!] = N[\![Q]\!]$ it is sufficient. by Lemma 6.10 (which will be proved in section 6.7) to prove that $P \leftrightarrow_T Q$ implies $N[\![P]\!] = N[\![Q]\!]$. If $P \leftrightarrow_T Q$ then either $P \Rightarrow Q$ or $Q \Rightarrow P$; assume without loss of generality that $P \Rightarrow Q$. $N[\![P]\!]$ can be computed by starting with P and repeatedly applying replacement rules until the result is irreducible. Because TRANS is assumed to have the Church-Rosser property, no matter what the first step of the derivation is, it will lead to $N[\![P]\!]$. Therefore, we can require the first step of the derivation to be $P \Rightarrow Q$, and there will be a derivation $Q \Rightarrow^* R$, where R is irreducible. Because $P \Rightarrow^* R$, $N[\![P]\!] = R$. Because $Q \Rightarrow^* R$, $N[\![Q]\!] = R$. Therefore $N[\![P]\!] = R = N[\![Q]\!]$. □

In the next section, the replacement system TRANS is presented. In sections 5.3 and 5.4, TRANS is shown to be confluent. Then, in section 5.5, an algorithm for computing the normalization function is presented.* Let TR1 be the set of axioms obtained directly from the replacement system. In section 6.6, the axiom set TR1 is shown to be equivalent to the axioms in Figures 5.1 and 5.2.

5.2 The replacement system TRANS

The domain of TRANS is the set of all PL/CV formula trees, with two extensions. The pairs of terms "formula" and "tree," "operator" and "node," "operand" and "offspring," and "main connective" and "root" will be used interchangably.

As in most computer languages, there are no negative constants in PL/CV. When one writes -3, one is applying a unary operator to the constant 3. The domain of TRANS includes negative constants. It is important to keep straight the differences among the sign of a negative constant, the unary negation operator, and the binary subtraction operator. In the rest of this chapter, the unary negation operator will be raised slightly, as in ^{-}A, to help distinguish it from the binary subtraction operator, an occurrence of which appears in (0 - A). Negative constants will be indicated by underlining. For example, the additive inverse of 3 is written $\underline{3}$.

Trees containing nodes of arbitrary out degree are also needed. In most computer languages, there are unary operators and binary operators, and possibly some functions with a fixed number of arguments. The corresponding nodes in parse trees have one, two, or some fixed number k of offspring. For example, the expression (A + B + C + D) is usually parsed as (((A + B) + C) + D). The domain of TRANS will include trees with associative operators that have any nonnegative number of offspring. If a node ρ has two or more offspring, it will be written $(P_1\ \rho\ P_2\ \rho\ \ldots\ P_n)$. If it has one offspring, it will be written $(\rho\ P)$, and if it has zero offspring, it will be written (ρ).

As discussed in section 3.8, the | and & operators in PL/CV are not restricted to be binary. Though I believe that these "multiary" operators correspond more closely with people's intuition about & and | than the binary operators, they appear to be more trouble than they are worth. They make data structures, algorithms, and proof rules more complex, and in terms of what is and is not a valid PL/CV proof, the only place it makes a difference that these operators are not binary is in the CASES rule (see section 3.8). In this and the remaining chapters, the PL/CV operators & and | will be assumed to be binary.

The domain of TRANS includes quantified formulas. Quantifier count representation (see section 2.2) will be used to represent these formulas as trees. Since the only replacement rules that alter the quantifier structure of a formula are the Folding Rules:

```
SOME x. '1'B → '1'B
SOME x. '0'B → '0'B
ALL  x. '1'B → '1'B
ALL  x. '0'B → '0'B
```

there is no need to worry about changing the quantifier counts within bound variables.

A set of schemata will be presented from which the rules can be obtained. A schema consists of an ordered pair of trees. Each leaf of each tree is either a constant or a variable called a parameter. Some of the schemata are shown in Table 5.3. To obtain a rule from a schema, one consistently substitutes a tree for each parameter in the schema. Some of

<u>Implication Rules</u>

'1'B => P → P
'0'B => P → '1'B
P => '1'B → '1'B
P => '0'B → ~P
P => P → '1'B
P => ~P → ~P
~P => P → ~~P

<u>Equivalence Rules</u>

'1'B <=> P → P
P <=> '1'B → P
'0'B <=> P → ~P
P <=> '0'B → ~P
P <=> P → '1'B
P <=> ~P → '0'B
~P <=> P → '0'B

<u>Comparison Rules</u>

P = Q → P - Q = 0 (where Q is not 0)
P < Q → P - Q < 0 (where Q is not 0)
P > Q → Q - P < 0
P <= Q → P - Q - 1 < 0
P >= Q → Q - P - 1 < 0

<u>Subtraction Elimination Rule</u>

P - Q → P + ¯Q

<u>Double Negation Rule</u>

¯¯Q → Q

<u>Negation Distribution Rule</u>

$^-(P_1 + P_2 + \ldots P_n) \rightarrow (^-P_1 + {}^-P_2 + \ldots {}^-P_n)$

<u>Exponentiation Rule</u>

P ** 1 → P

<u>Division Rule</u>

P / 1 → P

Table 5.3 Schemata for replacement rules

the schemata have restrictions on the trees that can be substituted for their parameters.

The associative operators complicate the presentation of the replacement rules because an infinite set of schemata is required. It is easier to describe replacement rules that operate on associative operators algorithmically than to give complicated diagrams littered with ellipses and subscripts. However, bear in mind that each description corresponds to a set of schemata. For example, the set of schemata for the Associativity Rules is all schemata of the form:

$$\begin{aligned}&(P_1\ \rho\ P_2\ \ldots\ P_i\ \rho\ (Q_1\ \rho\ Q_2\ \ldots\ Q_m)\ \rho\ P_{i+1}\ \ldots\ P_n)\\ &\quad \rightarrow\ (P_1\ \rho\ P_2\ \ldots\ P_i\ \rho\ Q_1\ \rho\ Q_2\ \ldots\ Q_m\ \rho\ P_{i+1}\ \ldots\ P_n)\end{aligned}$$

where ρ is an associative operator, $1 \le i$, $i+1 \le n$, $n \ge 1$, and $m \ge 0$.

In the following rule descriptions, ρ will represent the main connective of the left side of the replacement rule. The offspring of ρ will be referred to as "the operand list."

Folding Rules. The name "folding" comes from terminology used when discussing compiler optimization [37]. If ρ is a unary or binary operator, the folding rules are of the form:

$$\begin{aligned}&(\rho\ c_1) && \rightarrow\ v\\ &(c_1\ \rho\ c_2) && \rightarrow\ v\end{aligned}$$

where c_1 and c_2 are constants, and v is the constant that is obtained by evaluating the left side. If ρ is an associative operator, the operand list is searched for constants. For the rule to be applied, the list must contain at least two constants. The constants c_1, c_2, ..., c_n, where $n \ge 2$, are deleted from the list, the constant V is obtained by evaluating $(c_1\ \rho\ c_2\ \rho\ \ldots\ c_n)$ and V is inserted where c_1 was. This rule is valid only because all of the associative operators in PL/CV are also commutative. If there were an associative but not commutative operator in PL/CV, the restriction would have to be added that c_1, c_2, ... c_n be adjacent in the operand list. The schemata for the Folding Rules do not contain parame-

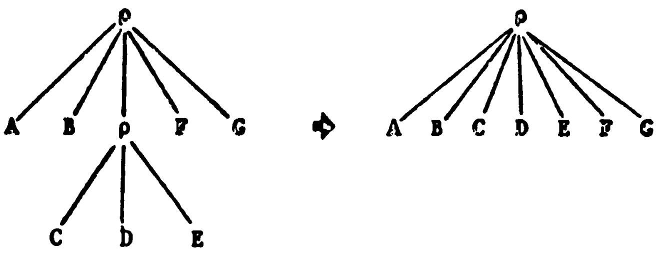


Figure 5.4 Flattening an associative operator

ters that may only have constants substituted for them; instead there is a different schema for each collection of constants.

<u>Associativity Rules</u>. If ρ is one of the operators &, |, +, or ×, the operand list is searched for operands whose main connective is ρ. For the rule to be applied there must be at least one such operand. The result of the rule is obtained by inserting the operands of the subtree into the operand list of the root, which flattens the tree as shown in Figure 5.4.

<u>Commutativity Rules</u>. If ρ is one of the commutative operators &, |, <=>, =, +, or ×, the preorder traversal of each of its operands are tested to see if they are in lexicographic order. For the rule to be applied, the operands must not be in order. The ordering among symbols is chosen so that constants are greater than all other symbols. In particular, the Commutativity Rule does not apply to a formula $P = 0$ or $P < 0$ unless P is a constant. The result of the rule is obtained by sorting the operand list. Note that the set of schemata for the Commutativity Rules must have restrictions placed on which trees can be substituted for parameters.

<u>Identity Elimination Rules</u>. If ρ is one of the operators &, |, ×, or +, let E be the corresponding identity element '1'B, '0'B, 1, or 0. For the rule to be applied, there must be at least one occurrence of E in the operand list. An occurrence of E is deleted to obtain the result of the rule.

Identity Introduction Rules. If ρ is one of the operators &, |, ×, or +, let E be the corresponding identity element '1'B, '0'B, 1, or 0. The Identity Introduction Rules are then of the form:

$(\rho) \rightarrow E$

Reduction Rules. If ρ is one of the operators &, |, ×, or +, the Reduction Rules are then of the form:

$(\rho\ P) \rightarrow P$

Annihilation Rules. If ρ is one of the operators &, |, or ×, let A be the corresponding annihilating element '0'B, '1'B, or 0. For the rule to be applied, there must be an occurrence of A in the operand list. The result of the rule is A.

Idempotent Rules. If ρ is one of the operators & or |, the operand list is searched for duplicates. For the rule to be applied, there must be at least two identical operands. One of the duplicates is deleted to obtain the result of the rule.

Cancellation Rules. If ρ is +, the operand list is searched for pairs of the form P and $^-$P. For the rule to be applied, there must be one such pair. One occurrence of each of P and $^-$P are deleted to obtain the result.

False Introduction Rules. If ρ is &, the operand list is searched for a pair of the form P and ~P. For the rule to be applied, there must be at least one such pair. The result of the rule is '0'B.

Equality Negation Rules. For an Equality Negation Rule to be applied, a number of conditions must be met. The tree must be of the form P = 0, where P is $(P_1 + P_2 + \ldots P_n)$, where n = 1 if + is not the root of P. Each P_i is of the form Q_i or ^-Q_i, where the root of Q_i is neither $^-$

nor +. If there are distinct i and j in {1 ... n} such that Q_i and Q_j are identical, the corresponding P_i and P_j must be identical. (That is, the Cancellation Rule must not be applicable.) Finally, not every element of $\{Q_1, Q_2, \ldots Q_n\}$ can be a constant, and if Q_m is a smallest (using lexicographic ordering) nonconstant in the set, Q_m must be P_m (as opposed to ^-P_m). To get the result of the Equality Negation Rule, each P_i of the form Q_i is replaced by its additive inverse if it is a constant (for example, $\underline{3}$ is replaced by 3), and by ^-Q_i otherwise, and each P_i of the form ^-Q_i is replaced by Q_i. Note that the set of schemata for the Equality Negation Rules must have restrictions placed on what trees can be substituted for parameters.

The enumeration of the replacement rules is now complete.

5.3 The Confluence of Replacement System TRANS

Aho, Sethi, and Ullman [5] define a replacement system to have "Property P2" whenever for all P, Q_1, and Q_2, where $P \Rightarrow Q_1$ and $P \Rightarrow Q_2$, there is an R such that $Q_1 \Rightarrow^* R$ and $Q_2 \Rightarrow^* R$. They prove the following theorem, which will be used to show that TRANS is a confluent replacement system.

Theorem 5.3. A replacement system is confluent if and only if it is finite and has Property P2.

Theorem 5.4. Replacement system TRANS is finite.

Define the size of a tree as the number of nodes it contains. (Note that the size of a tree is independent of the number of edges it contains.) The replacement rules can be divided into three classes. The Comparison Rules, Subtraction Elimination Rule, and the Negation Distribution Rule will be called "addition rules" because they increase the size of a tree when they are applied. The Commutativity Rules and the Identity Introduction Rules leave the size of a tree unchanged. All the rest of the rules, except for the Equality Negation Rule, will be called "deletion rules" because they decrease the size of a tree when they are applied. The Equality Negation Rules do not fit into any of these classes. There are Equality Negation Rules that, when applied to a tree, leave it smaller, the same size, and larger.

Suppose an endless derivation is possible. If the derivation uses a deletion rule infinitely often, it must use an addition rule infinitely often as a source of nodes to delete. But let us consider the addition rules. The Comparison Rules each replace a subtree whose root is a comparison operator with a subtree of the form $P < 0$ or $P = 0$. There are no rules that will replace 0 by something else or replace $P < 0$ or $P = 0$ by a subtree of the form $P' < Q$ or $P' = Q$, where Q is not 0. Since the Comparison Rules cannot be applied to subtrees of the form $P < 0$ or $P = 0$, only as many Comparison Rules can be used in a derivation as there were comparison operators in the original tree. Since the Subtraction Elimination Rule deletes a - operator, and only Comparison Rules introduce - operators, Subtraction Elimination Rules can only be used finitely many times in a derivation. The Negation Distribution Rule deletes a $^-$ node and then creates n more. But the nodes that are created are closer to the leaves of the tree than the node that was deleted. Therefore, this rule cannot be used infinitely often unless another addition rule that provides $^-$ nodes is used infinitely often as well, as a source of $^-$ operators. The only possible source that has not yet been eliminated from consideration are the Equality Negation Rules. Therefore, if a deletion rule is to be used infinitely often, then Equality Negation Rules must be used infinitely often as well.

The only additions an Equality Negation Rule can make to a tree is to replace a node P by $^-$P. There are five classes of rules that can be applied at the subtree $^-$P: Folding, Cancellation, Double Negation, Negation Distribution, and the Equality Negation Rule itself. The Equality Negation Rule cannot add a $^-$ onto P if P is a constant or its root is + or $^-$. Therefore, the first four rules cannot be applied immediately after the Equality Negation Rule. There are Folding, Identity Elimination, Annihilation, Division, and Exponentiation Rules that will take a P that is not a constant and whose root is not + or $^-$, and replace it with a Q that is a constant or has a root of + or $^-$, but not without deleting a node other than $^-$.

Therefore, it is impossible to use a deletion rule infinitely often. Nor is it possible to use the Identity Introduction Rule infinitely often, since it deletes an associative operator, and the only source of associative operators is the Subtraction Elimination Rule. Therefore, in every derivation there will exist a point after which it must either stop or consist entirely of Commutativity Rules and and Equality Negation Rules.

Define a cost function that maps trees to nonnegative integers as follows. Assign each node in the tree the "basic cost" 2 if it is the root of a subtree of the form 0 = P, 1 if it is any other node at which the Commutativity Rule or Equality Negation Rule can be applied, and 0 otherwise. Then, for each node, count the number of commutative operators and = operators (since = is also commutative, it gets counted twice) on the path from the node to the root. If the count is n, multiply the basic cost of the node by 2^n to get the total cost. The cost of the tree is then the sum of the total costs of all its nodes.

To prove that every derivation must terminate, it is sufficient to show that applying a Commutativity or an Equality Negation Rule will decrease the cost of the tree. After a Commutativity or Equality Negation Rule is applied at an operator ρ, one of these rules cannot immediately be applied again at that operator, unless the operator was the root of a subtree of the form $0 = P$, and the Commutativity Rule was applied. In either case, the basic cost of the operator is reduced by at least one.

Let n be the number of commutative operators on the path from ρ to the root of the tree. For each of these operators it is possible that a rule that formerly could not be applied there now can be applied. That is, the basic cost of each of these operators could be increased by 1; since a subtree not of the form $0 = P$ cannot have been changed to that form, the cost could not be increased by 2. Therefore the total cost of the tree could be increased by as much as $2^0 + 2^1 + \ldots + 2^{n-1} = 2^n - 1$. But since the total cost of ρ has been decreased by 2^n, the total cost of the tree has been reduced by at least 1. □

<u>Theorem</u> 5.5. Replacement system T has Property P2.

<u>Proof</u>. Suppose there are trees P, Q_1, and Q_2 such that $P \Rightarrow Q_1$ and $P \Rightarrow Q_2$. We must find a tree R such that $Q_1 \Rightarrow^* R$ and $Q_2 \Rightarrow^* R$. Let $S_1 \rightarrow T_1$ and $S_2 \rightarrow T_2$ indicate the subtree replacements that are performed by $P \Rightarrow Q_1$ and $P \Rightarrow Q_2$, respectively. There are three possibilities: S_1 and S_2 are disjoint, S_1 contains (not necessarily properly) S_2, or vice versa. If S_1 and S_2 are disjoint, each replacement can be performed independently; let R be the tree after both replacements have been performed.

If S_1 and S_2 are not disjoint, assume without loss of generality that S_1 contains S_2. There are two cases: either the subtree matched by

S_2 is contained in the subtree matched by a parameter of the schema used to obtain S_1, or it is not. It is easiest to explain the first case with an example. Suppose $S_1 \to T_1$ is an Idempotent Rule, produced by the schema $(U_1 \& U_2 \& U_1) \to (U_1 \& U_2)$, and S_2 matches a subtree contained in U_1.

If the replacement $S_1 \to T_1$ is performed first, the result is of the form $(U_1 \& U_2)$. Now perform the replacement $S_2 \to T_2$ on U_1, yielding $(U' \& U_2)$ for some U'.

Next consider what happens if the replacement $S_2 \to T_2$ is performed first. The result is $(U' \& U_2 \& U_1)$. The replacement can be performed a second time on the other copy of U_1 to yield $(U' \& U_2 \& U')$. Now $(U' \& U_2 \& U') \to (U' \& U_2)$ is also an Idempotent Rule, and can be applied to get $(U' \& U_2)$.

The same kind of argument works for all rules on which there are no restrictions on the trees that can be substituted for parameters. It is, however, necessary to examine each of the schemata for which there are restrictions: the first two Comparison Rules, the Commutativity Rules, and the Equality Negation Rules.

First consider the case of the Comparison Rules. Suppose $S_1 \to T_1$ is of the form $(U_1 \; \rho \; U_2) \to (U_1 - U_2 \; \rho \; 0)$, where ρ is either $<$ or $=$, and $S_2 \to T_2$ is the rule $U_2 \to 0$. The problem is that if the rule $S_2 \to T_2$ is applied first, the result is $(U_1 \; \rho \; 0)$, to which $S_1 \to T_1$ cannot be applied. However, there is a derivation of $(U_1 \; \rho \; 0)$ from $(U_1 \; \rho \; U_2)$ in which the first step is $S_1 \to T_1$:

$$(U_1 \; \rho \; U_2) \underset{S_1 \to T_1}{\Rightarrow} (U_1 - U_2 \; \rho \; 0) \underset{S_2 \to T_2}{\Rightarrow} (U_1 - 0 \; \rho \; 0) \underset{\text{Folding}}{\Rightarrow} (U_1 \; \rho \; 0) \;.$$

The next case is that of the Commutativity Rules. Suppose $S_1 \to T_1$ is, for example, $(U_1 \& U_2 \& U_3) \to (U_3 \& U_1 \& U_2)$, where the subtrees on the right side, but not the left, are in lexicographic order, and the rule $S_2 \rightarrow T_2$ transforms U_2 to U'. The rule $S_2 \rightarrow T_2$ can be applied to U_2 before or after the Commutative Rule is applied. The sequences of rules $S_1 \rightarrow T_1$, $S_2 \rightarrow T_2$, Commutativity Rule (if it can be applied), and $S_2 \rightarrow T_2$, Commutativity Rule (if it can be applied) will each yield a tree whose root is & and offspring are U_1, U', and U_3 arranged in lexicographic order.

The final (and messiest) case is that of the Equality Negation Rules. Suppose, for example, $S_1 \rightarrow T_1$ is of the form:

$$(U_1 + U_2 + {}^-U_3 + {}^-U_4 = 0) \rightarrow ({}^-U_1 + {}^-U_2 + U_3 + U_4 = 0)$$

This rule can be applied only if no Cancellation Rule applies, if none of the U_i have + or $^-$ as their main connective, if at least one of the U_i is not a constant and if a minimal U_i that is not a constant is not preceded by $^-$. If the rule $S_2 \rightarrow T_2$ is applied to change some U_i to U', one or more of the above restrictions can be violated.

However, after the rule $S_1 \rightarrow T_1$ is applied, the rule $S_2 \rightarrow T_2$ can still be applied to the occurrence of U_i in T_1. Let V_1 be the result of applying $S_1 \rightarrow T_1$ and then $S_2 \rightarrow T_2$, and let V_2 be the result of just applying $S_2 \rightarrow T_2$. Now V_1 and V_2 are quite similar; they differ only by the placement of some $^-$ operators and constants of opposite sign. There will be a sequence of Cancellation, Associativity, Negation Distribution, and Double Negation Rules that will transform V_1 into a formula V'_1 that meets all but possibly the last restriction on the Equality Negation Rule. There is a similar sequence of rules (for example, a cancellation of P_1 and ${}^-P_2$ in the first sequence corresponds to a cancellation of ${}^-P_1$ and P_2 in this sequence) that will transform V_2 into V'_2 such that the Equality Negation Rule can be used to transform V'_1 into V'_2, or vice versa.

Let us pause and take stock of where we are in the proof that TRANS has property P2. We have a tree P to which the two subtree replacement rules $S_1 \rightarrow T_1$ and $S_2 \rightarrow T_2$ both can be applied, where S_1 contains S_2. We want to show there is a tree R that can be derived from the result of applying either rule to P.

We have just completed the proof for the case in which S_2 is contained entirely within a subtree matched by a parameter of the schema from which S_1 was obtained. There are two other cases.

The first case is that S_1 and S_2 are identical trees. An example of this case is where S_1 and S_2 are each '1'B & ~'1'B. Either the Identity Elimination Rule '1'B & P → P can be applied to yield ~'1'B. or the False Introduction rule P & ~P → '0'B can be applied.

The second case is that S_2 matches more than just a parameter of the schema used to obtain S_1, but not the entire tree. An example of this case is where S_1 is $^-(x + 0)$ and S_2 is $(x + 0)$ (a small letter is used to distinguish the variable x from a schema parameter). The Negation Distribution rule $^-(P + Q) \rightarrow (^-P + {}^-Q)$ can be applied to obtain $(^-x + {}^-0)$, or the Identity Elimination Rule $(P + 0) \rightarrow P$ can be applied to obtain ^-x.

To find all the instances of the first case, it is necessary to consider all pairs of schema (R_1, R_2) whose left sides have the same main connective. To find all instances of the second case, it is necessary to consider all pairs of schema (R_1, R_2) such that the main connective of the left side of R_1 appears in the left side of R_2, but not as the main connective. In each case, it is necessary to argue that, given a tree P to which both R_1 and R_2 can be applied, there is a tree P' that can be derived from the result of the application of either rule.

The various cases are summarized in Tables 5.5, 5.6, 5.7, and 5.8.

	Folding	Associativity	Commutativity	Identity Elim.	Annihilation	Idempotent	Cancellation	False Intro.	Equality Neg.	Equivalence	Comparison	All Others
Folding	a											
Associativity	b	b										
Commutativity	c	c	d									
Identity Elim.	a	b	c	b								
Annihilation	e	e	e	e	f							
Idempotent	a	g	c	a	e	a						
Cancellation	a	h	c	a	-	-	a					
False Intro.	a	i	i	j	f	i	-	f				
Equality Neg.	d	-	d	-	-	-	-	-	d			
Equivalence	a	-	k	-	-	-	-	-	-	a		
Comparison	a	-	l	-	-	-	-	-	d	-	d	
All Others	a	-	-	-	-	-	-	-	-	-	-	m

Table 5.5 Cases in which S_1 and S_2 are the same

$S_2 \rightarrow T_2$ \ $S_1 \rightarrow T_1$	Associativity	Equality Neg.	Neg. Distrib.
Folding	b	b	n
Associativity	b	d	n
Commutativity	c	c	c
Identity Elim.	b	b	n
Annihilation	o	-	-
Idempotent	b	-	-
Cancellation	b	d	n
False Intro.	o	-	-

Table 5.6 Cases in which S_2 matches S_1 at an associative operator

$S_2 \rightarrow T_2$	Equality Negation	Cancellation	Double Negation
		$S_1 \rightarrow T_1$	
Folding	p	a	q
Negation Distr.	d	h	r
Double Negation	d	p	f

Table 5.7 Cases in which S_2 matches S_1 at a $^{-}$ operator

$S_2 \rightarrow T_2$	False Intro.	Implication	Equivalence
		$S_1 \rightarrow T_1$	
Folding	a	a	a

Table 5.8 Cases in which S_2 matches S_1 at a ~ operator

Given any two distinct schemata R_1 and R_2, locate the class in which each schema was presented in section 5.2, and use these classes to obtain the appropriate entry from Table 5.5. If the entry is "-", the left sides of the two schemata cannot have the same main connective. If the entry is a letter, it refers to an argument in section 5.4, which explains how the results of the two rules can be unified. Tables 5.6, 5.7, and 5.8 are used in a similar fashion, when the main connective of the left side of R_2 is an associative operator, the operator $^{-}$, or the operator ~ respectively, and the left side of R_2 matches part, but not all, of the left side of a rule R_1. If there is no entry for the pair, or the entry is "-", then the main connective of the left side of R_2 does not appear (except possibly as the main connective) in the left side of R_1. If the entry is a letter, it refers to an argument in section 5.4.

The arguments in section 5.4 are sketchy, and none are particularly deep. Many readers will no doubt wish to skip the section completely.

However, the proof was definitely worth performing. In the process of constructing the proof, I discovered a small bug in the normalization system of the Proof Checker.

The arguments in section 5.4 complete the proof of Theorem 5.5. □

<u>Theorem</u> 5.6. Replacement system T is confluent.

<u>Proof</u>. The theorem follows immediately from Theorems 5.3, 5.4, and 5.5. □

5.4 A <u>Number of Cases</u>

This section contains the arguments referenced by Tables 5.5, 5.6, 5.7, and 5.8.

(a) Two rules R_1 and R_2 each can be applied. Either each rule can be applied independently (see b), or if T_1 and T_2 are the results of applying R_1 and R_2, T_1 and T_2 can be made the same by applying to each some sequence of zero or more Folding, Commutativity, Annihilation, and Identity Elimination Rules.

(b) The two rules R_1 and R_2 are independent. That is, after R_1 has been applied, R_2 still can be applied, and vice versa. The sequence R_1, R_2 and the sequence R_2, R_1 have the same result.

(c) A Commutativity Rule and some other rule R both can be applied. If the Commutativity Rule is applied first, R may be applied afterwards to yield T_1. Call T_2 the result of applying just R. Apply the Commutativity Rule, where possible, to T_1 and T_2; the results will be the same.

(d) It is impossible to apply two distinct rules of each class.

(e) The Annihilation Rule and another rule R both can be applied. If the Annihilation Rule is applied first, the result is A, the annihilating element. If R is applied first, the Annihilation Rule can be applied afterwards, and the result is also A.

(f) No matter which of the two rules is applied, the result is the same.

(g) The Idempotent and Associativity rules both can be applied. Either the rules can be applied independently (see b), or there are two identical subtrees, each of which can be flattened or deleted. No matter which rule is applied first, there is a sequence of Idempotent and Associativity Rules that leads to one subtree deleted and the other flattened.

(h) Expressions of the form $(P_1 + P_2 + \ldots P_n)$ and $^-(P_1 + P_2 + \ldots P_n)$ each appear as operands to a +, so that a Cancellation, Associativity, or Negation Distribution Rule can be applied. If either the Associativity Rule or Negation Distribution Rule is applied first, apply the other one, apply the Associativity Rule to the result of the Negation Distribution Rule, and then apply the Cancellation Rule n times. The result will be the same as when the Cancellation Rule is applied first.

(i) The False Introduction Rule and another rule R both can be applied. If the False Introduction rule is applied first, the result is '0'B. If R is applied first, the False Introduction Rule can be applied afterwards, and the result is also '0'B.

(j) False Introduction and Identity Elimination both can be applied. If False Introduction is applied first, the result is '0'B. If Identity Elimination is applied first, then False Introduction can still be applied, unless P and ~P were '1'B and ~'1'B, and the '1'B was deleted by the Identity Elimination Rule. In that case, Folding is applied to ~'1'B to obtain '0'B, and then Annihilation is applied.

(k) An Equivalence Rule R_1 and a Commutativity Rule both can be applied. For every Equivalence Rule R_1 there is a corresponding Equivalence Rule (in some cases the same one) R_2 that differs only by the reversal of the order of the operands on the left side. Therefore Commutativity followed by R_2 has the same effect as just R_1.

(l) The Commutativity Rule and the first Comparison Rule (equality) can both be applied. If the Commutativity Rule is applied first, then the Comparison Rule can be applied to the result, or the Comparison Rule can be applied first. The result will be either $P - Q = 0$ or $Q - P = 0$. Repeated applications of the Subtraction Elimination, Associativity, Double Negation, Negation Distribution, and Cancellation Rules to each result will put them into the forms: $P_1 + P_2 \ldots P_n = 0$ and $Q_1 + Q_2 \ldots Q_n = 0$, where there is a permutation $j_1, j_2, \ldots j_n$ of the integers $\{1 \ldots n\}$ such that for each i, ^-P_i is Q_{j_i} or P_i is $^-Q_{j_i}$, and the Equality Negation Rule can be applied to one of the forms. Then, by using the Commutativity Rule on each tree to which it can be applied, the trees can be made the same.

(m) The two rules must be two different Implication Rules. Folding Rules can be applied to the results of each rule to make them the same.

(n) A Negation Distribution Rule applies to a tree of the form $^-(P_1 + P_2 + \ldots P_n)$, and a rule R applies to $(P_1 + P_2 + \ldots P_n)$. Let T_1 be the result of applying R and then Negation Distribution, and T_2 be the result of applying just Negation Distribution. After applying Double Negation, Negation Distribution, or Folding to T_2, rule R can be applied to yield T_3. Applying Folding to T_1, if necessary, makes it the same as T_3.

(o) A Rule R, either False Introduction or Annihilation, can be performed within a subtree that can be flattened by the Associativity Rule. If R is applied first to the subtree, the result will be to put '0'B into the main tree, and then Annihilation can be performed, yielding '0'B. If Associativity is applied first, R can then be performed on the main tree, also yielding '0'B.

(p) Formulas P and $^-$P appear as operands to a +, so that a Cancellation Rule applies, and a Double Negation Rule applies to $^-$P. The Cancellation Rule can still be applied after the Double Negation Rule is applied.

(q) The tree is of the form $^{--}$c, where c is a constant, so that either Double Negation can be applied or Folding can be applied twice to obtain c.

(r) Either a Double Negation or a Negation Distribution Rule can be applied to a tree of the form $^{--}(P_1 + P_2 + \ldots P_n)$. Applying Double Negation yields $(P_1 + P_2 + \ldots P_n)$, and applying Negation Distribution twice, followed by Double Negation n times yields the same result.

5.5 The Computation of Normal Forms

Given that replacement system TRANS is confluent, any algorithm that begins with P and repeatedly applies replacement rules until no more rules can be applied will compute the function N⟦P⟧. However, choosing the proper order in which to apply replacement rules can make the difference between an algorithm that uses exponential time and one that uses quadratic time.

The following is a recursive algorithm for computing N⟦P⟧. In general, P will be an operator ρ with operands $P_1, P_2, \ldots P_n$. The algorithm is applied recursively to obtain $Q_1, Q_2, \ldots Q_n$, where for each i in

{1 ... n}, $Q_i = N[\![P_i]\!]$. For each i in {1 ... n} we have $P_i \Rightarrow^* Q_i$, and letting Q be $(Q_1\ \rho\ Q_2\ \rho\ \ldots\ Q_n)$, we have $P \Rightarrow^* Q$, so that $N[\![P]\!] = N[\![Q]\!]$. $N[\![P]\!]$ can be computed by applying replacement rules to Q until no more rules can be applied. And since, by induction, each Q_i is a normal form, no rules can be applied that replace subtrees completely within any Q_i. If a replacement rule can be applied to Q, the main connective of its left side must be ρ, and the rule must replace all of Q.

How $N[\![P]\!]$ is obtained from Q depends on what ρ, the main connective of Q, is. There are five cases.

The first case is that ρ is =>, <=>, **, /, or some operator for which there are only Folding Rules. First check whether every Q_i is a constant; if so, apply an appropriate Folding Rule. The result is irreducible because there are no rules that can be applied to constants. If no Q_i is a constant, use any of the rules in Table 5.3 that can be applied. If a rule is applied, the result will either be a constant, or of the form Q_i, $\sim Q_i$, or $\sim\sim Q_i$. The result cannot be of the form $\sim Q_i$ or $\sim\sim Q_i$, where Q_i is a constant, for if it were, both Q_1 and Q_2 would have to be constants, and a Folding Rule could have been applied. Since each Q_i is, by induction, a normal form and the only replacement rules in which the main connective of the left side is ~ are Folding Rules, the result is irreducible. Finally, if ρ is <=> and no rules have yet been applied, arrange the two operands in lexicographic order.

The second case is that ρ is an associative operator. First, flatten each Q_i whose root is ρ. By induction, the subtrees introduced are irreducible, and therefore cannot have ρ as their root. Next, use a lexicographic sort [1] to put the subtrees into order in linear time. After the subtrees have been sorted, all the constants will be adjacent, all identical subtrees will be adjacent, and all subtrees of the form ~R will be adjacent, as will subtrees of the form $^-$R.

All the constants can be easily located and folded together without disturbing the sort order. If the resulting constant is the identity element, delete it; if it is the annihilating element, replace the entire subtree with it. If ρ is & or |, make a pass through the list of subtrees, deleting all but one duplicate of any subtree in the list.

All cancellations can be performed in linear time as well. All of the subtrees $^-R_1$, $^-R_2$, ... ^-R_n will appear together. Moreover, if any of the subtrees R_1, R_2, ... R_n are present, they will be in that order. A simple linear algorithm that is similar to a merge can be used to locate and delete all the $^-$R and R pairs. The same approach can be used to locate ~R and R pairs for the False Introduction Rule.

Finally, if the above deletions result in a ρ having a single operand R, the normal form is R. If the deletions result in ρ having zero operands, the normal form is the identity element of ρ.

The third case is that ρ is $^-$. Then the only rules that can be applied are Folding, Double Negation, and Negation Distribution. The first two rules will result in irreducible forms. Let us consider the subcase in which the tree is of the form $^-P_1$, where P_1 is of the form $(Q_1 + Q_2 + \ldots Q_n)$, and each Q_i is irreducible. Applying Negation Distribution yields $(^-Q_1 + {^-Q_2} + \ldots {^-Q_n})$; now each ^-Q_i must be normalized. Because P_1 is irreducible, no Q_i can have + as a root; if it did, the Associativity Rule could be applied to P_1. Therefore, each Q_i need only be checked for Double Negation and Folding, and these rules result in irreducible forms.

However, the resulting tree $(R_1 + R_2 + \ldots R_n)$ will in general not be irreducible. Most of the rules for + cannot be applied, because if they could be, there would be corresponding rules that could be applied to P_1, and P_1 is irreducible. Folding cannot be applied, since two constants R_i and R_j must correspond to constants Q_i and Q_j. Associativity cannot be applied, since if R_i is of the form (S + ...), Q_i must be of the form $^-$(S + ...) and then the Negation Distribution Rule could be applied to P_1. Identity Elimination cannot be applied, because if R_i is 0, Q_i must be 0. This leaves Commutativity as the only rule that can be applied. But there is a trick that can be used to make it easy to apply the Commutativity Rule. If the ordering among symbols used to represent trees is chosen so that $^-$ is the smallest symbol, and all constants are greater than nonconstants, P_1 will take the form:

$$({}^-S_1 + {}^-S_2 + \ldots {}^-S_n + T_1 + T_2 + \ldots T_m + c)$$

where the root of each S_i and T_i is not $^-$, and c is a constant. (In general, some parts of the above may not be present). Then the normal form of $^-P_1$ is:

$$({}^-T_1 + {}^-T_2 + \ldots {}^-T_n + S_1 + S_2 + \ldots S_m + \underline{c})$$

where $\underline{c}$ is the constant whose value is the same as $^-$c. The normal form of $^-P_1$ can be computed from P_1 in linear time without sorting.

The fourth case is that ρ is −. Subtraction Elimination Rules are used to replace P − Q by P + $^-$Q. Replace $^-$Q by the normal form of Q (as was done in the third case) and proceed as if ρ were +.

The final case is that ρ is a comparison operator. The Comparison Rules are used to turn comparisons to subtractions. Then the Subtraction Elimination Rules are used to turn the subtractions into additions, and a Folding Rule is applied to $^-$1. The result will be of one of the forms

$(P + {}^{-}Q)\ \rho\ 0$ or $(P + {}^{-}Q + 1)\ \rho\ 0$, where P, Q and 1 are all irreducible and ρ is either < or =. Replace ^{-}Q by the normal form of Q as is done in the third case, and normalize the left operand of ρ as is done in the second case.

The result of the thoso operations will bo of tha form $P\ \rho\ 0$, where ρ is either < or =. If P is a constant, a Folding Rule must be applied. If P is not a constant, and ρ is =, the result of the above operations will be:

$$({}^{-}S_1 + {}^{-}S_2 + \ldots {}^{-}S_n + T_1 + T_2 + \ldots T_m + c) = 0$$

and the Equality Negation Rule must be applied if possible. Because P is irreducible, the Cancellation Rule cannot apply in P, and no S_i or T_i can have a main connective of $^{-}$ or +. Because P is not a constant, at least one S_i or T_i must not be a constant. Therefore, the Equality Negation Rule can be applied if and only if T_1 is lexically less than S_1 or n = 0. The Equality Negation Rule is applied in the same way the Negation Distribution Rule is applied in the third case.

The presentation of the algorithm for computing N⟦P⟧ is now complete.

Recall that only the Comparison Rules, Subtraction Elimination Rules, Negation Distribution, and Equality Negation Rules can increase the size of the tree, and only by adding $^{-}$ operators, and the constant 1. Only as many occurrences of 1 can be added as there are <= and >= operators in P. If n is the size of P, there can be only O(n) $^{-}$ operators in N⟦P⟧; no more can "fit" without having subtrees of the form ^{--}Q. Therefore, N⟦P⟧ can be larger than P by at most a linear factor.

A simple charging argument can be used to show that N⟦P⟧ is computed in $O(n^2)$ time, where n is the size of P. The algorithm processes operator node ρ by first normalizing its operands, and then combining the normalized forms. The size of the normalized operands is linear in the size of the original operands, and the time needed to combine them is linear in the size of the normalized operands. This time is charged to the operator ρ. Each operator in the formula gets O(n) time charged to it, so the entire tree is normalized in O(n^2) time.

A rule for the distribution of multiplication over addition could have been included, but then N⟦P⟧ could grow exponentially with P. DeMorgan's Laws, the logical analogue of the Negation Distribution Rule, could also have been included, but to maintain the linear space bound between N⟦P⟧ and P, the rule ~~P → P would have been required. This rule and one of DeMorgan's four rules are not constructively valid. So that constructive proofs can be checked, all the automatic reasoning performed by the Proof Checker is constructive. For the same reason, the rule (P | ~P) → '0'B was omitted.

It is not implausible that the O(n^2) run time can be improved upon. A variation on the linear tree isomorphism algorithm found in Aho, Hopcroft and Ullmans's book [2] can be used to a compute normal form based on a subset of the rules of TRANS in linear time. The Folding, Commutativity, Annihilation, Idempotent, and the Comparison Rules can all be handled. Because the algorithm processes a tree from the bottom up by levels, replacement rules that alter the level of a subtree, such as the Associativity and Double Negation Rules, are a problem. If this problem cannot be solved, a hybrid algorithm that does not have a better worst-case asymptotic running time, but in practice is faster, may be possible.

5.6 The Transformation Axioms

We have seen that if a set of axioms TA_1 is constructed by including an axiom of the form $P \equiv Q$ for every replacement rule $P \rightarrow Q$ in TRANS, then $P \equiv_T Q$ if and only if $N[\![P]\!] = N[\![Q]\!]$, where $\equiv_T$ is defined in terms of TA_1.

Because the replacement rules of TRANS were carefully constructed so that the replacement system would be confluent, the rules are rather messy to state. But there are no corresponding constraints on the axiom system. In this section, the set of axioms appearing in Figures 5.1 and 5.2, which will be referred to as TA_4, are shown to be equivalent to TA_1 in the sense that the relation $\equiv_T$ defined by TA_4 is the same as the relation defined by TA_1.

__Theorem__ 5.7. The transformation axiom sets A and A ∪ B are equivalent if for every axiom $P \equiv Q$ in B, there is a derivation of Q from P using axiom set A.

__Proof__. To prove two axioms sets equivalent, we must show for any two formulas X and Y, if Y can be derived from X using one axiom set, then it can be derived from X using the other set, and vice versa.

Suppose it is possible to derive the formula Y from X using the axioms in A. Certainly the same derivation is possible using the axioms in A ∪ B.

Now suppose there is a derivation D of Y from X using the axioms in A ∪ B; we wish to construct a derivation that uses just axioms from A. Each step of D is of the form $R[P_0/\!/x]$, $R[P_n/\!/x]$, where R is a formula in which x appears exactly once, and either either $P_0 \equiv P_n$ or $P_n \equiv P_0$ is an axiom in B. This step will be replaced by a series of steps of the form $R[P_0/\!/x]$, $R[P_1/\!/x]$, ... $R[P_n/\!/x]$, each of which uses an axiom from A. Such a replacement is possible because, by hypothesis, there is a derivation

P_0, P_1, ... P_n that only uses axioms in A. The derivation of Y from X using only axioms in A is obtained by performing the above replacement on every step in D. □

To obtain TA_4 from TA_1, we will make a series of additions and deletions to the set of axioms, all of which will, by Theorem 5.7, result in equivalent sets of axioms. Of the equivalent intermediate axiom sets in the sequence between TA_1 and TA_4, only two (TA_2 and TA_3) will merit names.

The Commutativity Axioms are of the form:

$$(P_1 \;\rho\; P_2 \;\ldots\; P_n) \equiv (Q_1 \;\rho\; Q_2 \;\ldots\; Q_n)$$

where ρ is a commutative operator, one side is a permutation of the other, and the right side is sorted lexicographically. We can always derive a formula of the form $(S \;\rho\; T)$ from $(T \;\rho\; S)$, because one of those two formulas will have sorted operands. Therefore, by Theorem 5.7, the set of axioms of the form $(S \;\rho\; T) \equiv (T \;\rho\; S)$, called the Binary Commutativity Axioms, can be added.

The Binary Commutativity Axioms and the Associativity Axioms can be used to derive the Swapping Axioms, which are of the of the form:

$$(P_1 \;\rho\; P_2 \;\ldots\; P_i \;\rho\; P_{i+1} \;\ldots\; P_n) \equiv (P_1 \;\rho\; P_2 \;\ldots\; P_{i+1} \;\rho\; P_i \;\ldots\; P_n)$$

The derivation is as follows:

$$\begin{array}{ll}
(P_1 \;\rho\; P_2 \;\ldots\; P_i \;\rho\; P_{i+1} \;\ldots\; P_n) & \\
 & \text{[Associativity]} \\
(P_1 \;\rho\; P_2 \;\ldots\; (P_i \;\rho\; P_{i+1}) \;\ldots\; P_n) & \\
 & \text{[Binary Commutativity]} \\
(P_1 \;\rho\; P_2 \;\ldots\; (P_{i+1} \;\rho\; P_i) \;\ldots\; P_n) & \\
 & \text{[Associativity]} \\
(P_1 \;\rho\; P_2 \;\ldots\; P_{i+1} \;\rho\; P_i \;\ldots\; P_n) &
\end{array}$$

Therefore, by Theorem 5.7, we may add the Swapping Axioms. But the Swapping Axioms are sufficient to derive the Commutativity Axioms, since they can be used to implement a bubble sort. Therefore all the Commutativity

Axioms could be deleted. We will delete all of them except the Binary Commutativity Axioms, which along with the Associativity Axioms are sufficient to derive the Swapping Axioms. Those axioms, having served their purpose, will now be deleted. The net effect has been to replace the Commutativity Axioms with the Binary Commutativity Axioms.

In the same fashion we can reduce the other axioms to simple schemata. For example, the Identity Elimination Axioms, which are of the form:

$$(P_1 \ \rho \ \ldots \ P_i \ \rho \ E \ \rho \ P_{i+1} \ \ldots \ P_n) \equiv (P_1 \ \rho \ P_2 \ \rho \ \ldots \ P_n)$$

where E is the identity element, can be used to derive the Binary Identity Elimination Axiom $(P \ \rho \ E) \equiv P$. The derivation is:

$(P \ \rho \ E)$
[Identity Elimination]
$(\rho \ P)$
[Reduction]
P

But all of the Identity Elimination Axioms can be derived using the Binary Identity Elimination, Binary Commutativity, and Associativity Axioms. Therefore, the Identity Elimination Rules can be deleted. The same kind of reductions can be applied to the Annihilation, Idempotent, Cancellation, and False Introduction Axioms.

The Folding Axioms will be generalized. Clearly any valid statement of the form $P \equiv Q$, where P and Q are trees whose leaves are all constants can be derived using the Folding Axioms. All such statements will be added as axioms.

From P we can certainly derive $P - 0$. It follows that from $P = 0$ we can derive $P - 0 = 0$, and from $P < 0$, we can derive $P - 0 < 0$, so that the axioms $(P = 0) \equiv (P - 0) = 0$ and $(P < 0) \equiv (P - 0) < 0$ can be added. The effect of adding these axioms is to remove the restriction that Q cannot be 0 in the axioms $(P = Q) \equiv (P - Q) = 0$ and $(P < Q) \equiv (P - Q) < 0$.

The axiom $(P = Q) \equiv (^{-}P = {}^{-}Q)$ can be derived as follows. In general, P will be of the form $(P_1 + P_2 \ldots + P_n)$, letting n be 1 if the root of P is not +. Similarly, let Q be $(Q_1 + Q_2 \ldots + Q_m)$. The derivation will begin:

$$(P_1 + P_2 \ldots + P_n) = (Q_1 + Q_2 \ldots + Q_m)$$

[Comparison]

$$(P_1 + P_2 \ldots + P_n) - (Q_1 + Q_2 \ldots + Q_m) = 0$$

[Subtraction Elim.]

$$((P_1 + P_2 \ldots + P_n) + {}^{-}(Q_1 + Q_2 \ldots + Q_m)) = 0$$

[Negation Distrib.]

$$((P_1 + P_2 \ldots + P_n) + ({}^{-}Q_1 + {}^{-}Q_2 \ldots + {}^{-}Q_m)) = 0$$

[Associativity]

$$(P_1 + P_2 \ldots + P_n + ({}^{-}Q_1 + {}^{-}Q_2 \ldots + {}^{-}Q_m)) = 0$$

[Associativity]

$$(P_1 + P_2 \ldots + P_n + {}^{-}Q_1 + {}^{-}Q_2 \ldots + {}^{-}Q_m) = 0 \tag{5.1}$$

By applying a sequence of Cancellation, Double Negation Elimination, Negation Distribution, and Associativity Axioms, we can get a formula:

$$(R_1 + R_2 \ldots + R_k) = 0 \tag{5.2}$$

where each R_i is of the form S_i or ${}^{-}S_i$, no S_i has ${}^{-}$ or + as its main connective, not every S_i is a constant, and if S_i and S_j are the same, then the corresponding R_i and R_j are the same.

At least one of the following:

$$((T_1 + T_2 + \ldots T_k) = 0) \equiv ((R_1 + R_2 + \ldots R_k) = 0) \tag{5.3}$$

$$((R_1 + R_2 + \ldots R_k) = 0) \equiv ((T_1 + T_2 + \ldots T_k) = 0) \tag{5.4}$$

where T_i is $\underline{c}$ if R_i is a constant c, ${}^{-}S_i$ if R_i is not a constant S_i, and S_i if R_i is ${}^{-}S_i$, will be an axiom. If every S_i is a constant, then both (5.3) and (5.4) will be Folding Axioms. Otherwise, let S_m be a smallest nonconstant in $\{S_1, S_2, \ldots S_k\}$. If S_m is in $\{T_1, T_2, \ldots T_k\}$ then (5.3) will be an Equality Negation Axiom; if S_m is in $\{R_1, R_2, \ldots R_k\}$ then

(5.4) will be an Equality Negation Axiom. Using the appropriate axiom, we obtain:

$(T_1 + T_2 + \dots T_n) = 0.$

Now the derivation of (5.2) from (5.1) can be reversed to obtain:

$(^-P_1 + {}^-P_2 \dots + {}^-P_n + Q_1 + Q_2 \dots + Q_n) = 0$

and then be continued as follows:

$$((^-P_1 + {}^-P_2 \dots + {}^-P_n) + Q_1 + Q_2 \dots + Q_n) = 0$$
[Associativity]
$$((^-P_1 + {}^-P_2 \dots + {}^-P_n) + (Q_1 + Q_2 \dots + Q_n)) = 0$$
[Associativity]
$$(^-(P_1 + P_2 \dots + P_n) + Q_1 + Q_2 \dots + Q_n) = 0$$
[Negation Distrib.]
$$(^-(P_1 + P_2 \dots + P_n) + (Q_1 + Q_2 \dots + Q_n)) = 0$$
[Double Negation]
$$(^-(P_1 + P_2 \dots + P_n) + {}^{--}(Q_1 + Q_2 \dots + Q_n)) = 0$$
[Subtraction Elim.]
$$(^-(P_1 + P_2 \dots + P_n) - {}^-(Q_1 + Q_2 \dots + Q_n)) = 0$$
[Comparison]
$$^-(P_1 + P_2 \dots + P_n) = {}^-(Q_1 + Q_2 \dots + Q_n)$$

Therefore, by Theorem 5.7, we can add the axiom $(P = Q) \equiv (^-P = {}^-Q)$. But using this new axiom and the Folding, Negation Distribution and Double Negation, we can derive the Equality Negation Axiom; we now joyously delete that axiom.

Let TA_2 be the set of axioms produced by all the additions and deletions to TA_1 made so far. The set TA_2 can be generated by a set of schemata that has no restrictions on how trees can be substituted for parameters (the Folding Axioms are generated by an infinite set of schemata that use no parameters). Since none of the schemata contain variables, if x is a variable (not a parameter), TA_2 has the property:

if $P \equiv Q$ is in TA_2, then $P[R//x] \equiv Q[R//x]$ is in TA_2 (5.5)

since a derivation of $Q[R//x]$ from $P[R//x]$ can be obtained by substituting

```
(P = Q)    ≡  (P + R) = (Q + R)

(P < Q)    ≡  (P + R) < (Q + R)
(P < Q)    ≡  (P <= Q-1)
(P < Q)    ≡  (Q > P)

(P <= Q)   ≡  (P + R) <= (Q + R)
(P <= Q)   ≡  (P < Q+1)
(P <= Q)   ≡  (Q >= P)

(P > Q)    ≡  (P + R) > (Q + R)
(P > Q)    ≡  (P >= Q+1)
(P > Q)    ≡  (Q < P)

(P >= Q)   ≡  (P + R) >= (Q + R)
(P >= Q)   ≡  (P > Q-1)
(P >= Q)   ≡  (Q >= P)
```

Figure 5.9 Extended axioms for comparison relations

R for every occurrence of x in a derivation of Q from P. We can now easily show that all the axioms in Figure 5.9 can be derived. Let $P \equiv Q$ be one of the axiom schemata in the figure, and let P' be P with every parameter replaced by a variable with the same name, and Q' be obtained from Q in a similar fashion. By applying the normalization function to P' and Q' (not in an abstract sense, but by actually performing the steps of the computation) we can see that $N[\![P']\!] = N[\![Q']\!]$. Therefore there is a derivation of Q' from P' using TA_1. Since TA_2 and TA_1 are equivalent, there is a derivation of Q' from P' using the axioms in TA_2. Let $P'' \equiv Q''$ be an axiom obtained from the schema $P \equiv Q$ by substituting trees for parameters. If those same substitutions are made for variables in the derivation of Q' from P' using axioms in TA_2, the result is, by property (5.5), a derivation of Q" from P" in TA_2. Therefore, all the axioms produced by the schemata in Figure 5.9 can be added to TA_2.

The right side of each of the axioms:

$$
\begin{aligned}
P = Q &\equiv P - Q = 0 \\
P < Q &\equiv P - Q < 0 \\
Q > P &\equiv P - Q < 0 \\
P <= Q &\equiv P - Q - 1 < 0 \\
Q >= P &\equiv P - Q - 1 < 0
\end{aligned}
$$

can each be derived from its left side using the extended axioms in Figure 5.9. Transform each inequality to one involving <, add ^{-}Q or $^{-}(Q+1)$ to both sides, apply cancellation on the right, and simplify on the left. Therefore, once the extended axioms are added, the above axioms can be deleted.

If ρ is an associative operator, the derivation:

$$
\begin{array}{ll}
((P\ \rho\ Q)\ \rho\ R) & \\
 & \text{[Associativity]} \\
(P\ \rho\ Q\ \rho\ R) & \\
 & \text{[Associativity]} \\
(P\ \rho\ (Q\ \rho\ R)) &
\end{array}
$$

allows us to add the Binary Associativity Axiom:

$$((P\ \rho\ Q)\ \rho\ R) \equiv (P\ \rho\ (Q\ \rho\ R)) \quad .$$

Recall that the domain of the replacement system is a slight extension of PL/CV formulas. Let us call these two domains formulas in the "combined language" and formulas in the "strict language." The difference between the two languages is that the combined language contains negative constants and formulas of the form $(P_1\ \rho\ P_2\ \rho\ \ldots\ P_n)$ for associative operators ρ and nonnegative n. If $n \neq 2$, these operators will be called "extended associative operators." Any formula that is in the combined language but not in the strict language will be said to be in the "extended language."

We are only interested in derivations that begin and end with formulas in the strict language. However, there could be some derivations that have intermediate formulas that are in the extended language. What we will now see is that if there is such a derivation, there is another derivation that only uses formulas in the strict language. We will then be justified in deleting any axioms of which at least one side is a formula in the extended language.

Any formula in the extended language can be translated into the strict language as follows. Any negative constant $\underline{c}$ is replaced by the expression ^{-}c. Any formula $(P_1\ \rho\ P_1\ \rho\ \ldots\ P_n)$ is replaced by E, the identity element of ρ if $n = 0$, P_1 if $n = 1$, and by:

$$((\ldots((P_1\ \rho\ P_2)\ \rho\ P_3)\ \rho\ \ldots)\ \rho\ P_n)$$

if $n > 2$.

If P is a formula in the extended language, denote its translation into the strict language as $M[\![P]\!]$. $M[\![P]\!]$ can be thought of as the "meaning" of P. If P is in the strict language, define $M[\![P]\!]$ to be P.

Now suppose we have a derivation P_0, P_1, ... P_n in which P_0 and P_n are in the strict language, and the other formulas are in the combined language. What we must show is that for each i in {1 ... n} that $M[\![P_i]\!]$ can be derived from $M[\![P_{i-1}]\!]$ using only intermediate steps that are in the strict language.

If an application of an axiom to a formula in the extended language transforms P_{i-1} to P_i leaving negative constants and extended associative operators intact, then the same axiom can be used to transform $M[\![P_{i-1}]\!]$ to $M[\![P_i]\!]$. Let TA_3 be the current set of transformation axioms. TA_3 consists of the axioms in Figures 5.1 and 5.2, plus the Identity Introduction Axiom:

$$(\rho) \equiv E$$

where ρ is an associative operator and E is its identity element. the

Reduction Axiom:

$$(\rho\ P) \equiv P$$

the Negation Distribution Axiom:

$$^{-}(P_1 + P_2 + \ldots P_n) \equiv ({}^{-}P_1 + {}^{-}P_2 + \ldots {}^{-}P_n)$$

and the Associative Axiom:

$$(P_1\ \rho\ P_2 \ldots P_i\ \rho\ (Q_1\ \rho\ Q_2 \ldots Q_m)\ \rho\ P_{i+1} \ldots P_n)$$
$$\equiv\ (P_1\ \rho\ P_2 \ldots P_i\ \rho\ Q_1\ \rho\ Q_2 \ldots Q_m\ \rho\ P_{i+1} \ldots P_n)$$

where ρ is an associative operator. By inspecting all the axioms in TA_3, we find that the only ones that do not leave extended associative operators and negative constants intact are the Folding Axioms and the ones listed above.

There are Folding Axioms, such as $\underline{3} \equiv 2 - 5$, that do not leave negative constants intact. But if $P \equiv Q$ is a Folding Axiom, $M[\![P]\!] \equiv M[\![Q]\!]$ is as well. In the above example, the corresponding step in the strict language could use the Folding Axiom $^{-}3 \equiv 2 - 5$.

If a step uses an Identity Introduction or Reduction Axiom of the form $P \equiv Q$, an extended associative operator will be modified. In this case, $M[\![P]\!]$ is the same as $M[\![Q]\!]$, and no corresponding step is necessary in the strict language.

The application of the Negation Distribution Axiom can modify an extended associative operator. Translating this step into the strict language, there are three cases. If $n > 2$, we must be able to derive:

$$((\ldots(({}^{-}P_1 + {}^{-}P_2) + {}^{-}P_3)\ldots) + {}^{-}P_n)$$

from:

$^{-}((\ldots((P_1 + P_2) + P_3)\ldots) + P_n)$.

This derivation can be performed by n-1 applications of the axiom $^{-}(P + Q) = (^{-}P + {}^{-}Q)$. If n is 1 or 0, we must be able to derive $^{-}P_1$ from $^{-}P_1$, or 0 from 0, each of which is trivial.

Finally, consider what happens when the Associativity Axiom is applied. Suppose we have a formula P of the form $(P_1 \rho P_2 \rho \ldots P_n)$, where, for some i in $\{1 \ldots n\}$, P_i is of the form $(Q_1 \rho Q_2 \rho \ldots Q_m)$. Applying an Associativity Axiom flattens the tree. Let R be the flattened tree. We must show that $M[\![R]\!]$ can be derived from $M[\![P]\!]$ using intermediate steps that are all in the strict language. The cases in which $n = 1$ or $m < 2$ are all trivial; we will assume that $n \geq 2$ and $m \geq 2$.

Consider binary trees whose nonleaf nodes are all ρ, and whose leaves are formulas. Define inductively what it means for these binary trees to be in "left normal form" as follows. All leaves are in left normal form. A tree of the form $(L \rho x)$ is in left normal form if and only if L is in left normal form and x is a leaf.

The notion of left normal form can be used to state more precisely what the translations of P and R are. One way of viewing $M[\![P]\!]$ is as a binary tree in left normal form whose leaves are, in traversal order, $M[\![P_1]\!]$, $M[\![P_2]\!]$, ... $M[\![P_n]\!]$. $M[\![P_i]\!]$ can, in turn be viewed as a binary tree in left normal form whose leaves are, in traversal order, $M[\![Q_1]\!]$, $M[\![Q_2]\!]$, ... $M[\![Q_m]\!]$. But if $M[\![P_i]\!]$ is not viewed as a leaf of $M[\![P]\!]$, then $M[\![P]\!]$ can be viewed as a binary tree whose leaves are, in traversal order:

$$M[\![P_1]\!], M[\![P_2]\!], \ldots M[\![P_{i-1}]\!], M[\![Q_1]\!], M[\![Q_2]\!], \ldots M[\![Q_m]\!], M[\![P_{i+1}]\!], \ldots M[\![P_n]\!]$$

but which is not, in general in left normal form. $M[\![R]\!]$ can be viewed as a binary tree that has the same leaves, but is in left normal form.

When the Binary Associativity Axiom $((P \rho Q) \rho R) \equiv (P \rho (Q \rho R))$ is applied to a tree, its leaves and their order are unchanged. Thus, to prove that M⟦R⟧ can be derived from M⟦P⟧ using the Binary Associativity Axiom, it is sufficient to prove the following theorem.

<u>Theorem</u> 5.8. Any tree can be transformed to one in left normal form by repeated applications of the Binary Associativity Axiom.

<u>Proof</u>. The proof is by induction on the size of the tree. The base case is when the tree is a leaf. Since all leaves are in left normal form. no tranalstion ia necessary.

If the tree is not a leaf it will be of the form $(S \rho T)$. If T is a leaf, S can, by induction, be transformed to a tree U that is in left normal form. The tree $(U \rho T)$ will then be in left normal form. If T is not a leaf it can, by induction, be transformed, into left normal form, yielding a tree of the form $(S \rho (V \rho x))$, where x is a leaf. Apply the Binary Associativity axiom to get $((S \rho V) \rho x)$. $(S \rho V)$ can, by induction, be transformed to a tree W that is in left normal form. Then the tree $(W \rho x)$ will be in left normal form. □

We have shown that any derivation that begins and ends in the strict language can be performed using only intermediate steps that are in the strict language. The extended language is only an implementation detail; it is not essential to the logic. Therefore, any axiom in TA_3 that has as one of its sides a formula in the extended language can be deleted. The result, TA_4, is the list of transformation axioms in Figures 5.1 and 5.2.

CHAPTER 6

A DECISION PROCEDURE FOR THE THEORY OF EQUALITY

6.1 Introduction

As we saw in section 4.8, it is necessary for the V-code Interpreter to perform the substitution of equals for equals to properly implement the automatic assignment rule. For example, the V-code generated for the following PL/CV proof fragment:

```
((D + C) + 1)*2 > 2 BY ...
A := D + C;
B := A + 1;
C := B * 2;
C > 2;
```

is:

```
PROCLAIM   ((D + C) + 1)*2 > 2
NEW        A
PROCLAIM   A = D + C
NEW        B
PROCLAIM   B = A + 1
NEW        C
PROCLAIM   C = B * 2;
REQUIRE    C > 2
PROCLAIM   C > 2
```

The V-code interpreter deduces C > 2 from the PROCLAIMed formulas using a decision procedure for the theory of equality with uninterpreted function symbols.

The substitutions that must be performed to check assignment statements are just a special of case the more general equality rules. The traditional axiom and rules of inference for equality are shown in Figure 6.1. However, the Symmetry and Transitivity Rules are redundant; they can be derived from the Reflexivity Axiom and Substitution Rule. To derive the Symmetry Rule, suppose we have s = t. To get t = s, substitute t for

$$\frac{}{s = s} \quad \text{Reflexivity}$$

$$\frac{s = t}{t = s} \quad \text{Symmetry}$$

$$\frac{s = t \quad t = u}{s = u} \quad \text{Transitivity}$$

$$\frac{s = t \quad P[s//x]}{P[t//x]} \quad \text{Substitution}$$

Figure 6.1 The axioms and rules of inference for equality

the first s in the axiom s = s. To derive the Transitivity rule, suppose we have s = t and t = u. To get s = u, substitute u for t in s = t.

Therefore to decide the theory of equality it is only necessary to recognize all consequences of the Reflexivity Axiom and the Substitution Rule.

Algorithm Eq, which recognizes all consequences of the Substitution Rule and the Reflexivity Axiom, is presented and proved correct in this chapter. And as we will see in Chapter 7, Algorithm Eq is not only used in the V-code Interpreter to implement the automatic equality rules; it forms the foundation on which the algorithm used by the V-code Interpreter to perform propositional reasoning is built. The next section contains a formal specification for Algorithm Eq.

6.2 A Formal Definition of the Problem

Suppose we are given an alphabet A of function symbols. The syntactic object "expression" can be inductively defined as follows. If f is an element of A, and x_1, x_2, ... x_k are zero or more expressions, then

$f(x_1, x_2, \ldots x_k)$ is also an expression. The only expressions are objects that can be obtained by repeated applications of the preceding rule. The "degree" of an expression is defined to be the number of function symbols it contains.

The exact nature of the object "expression" has been deliberately left a bit vague. An expression could be a string of function symbols, commas, and parentheses, but representations that can more easily be manipulated by computer programs are possible.

Suppose there is also a set, EQS, of unordered pairs of expressions that have been declared to be "equal." Adding the pair $\{s_1, s_2\}$ to EQS is an indication that in any context where s_1 appears, s_2 would do equally well. Two expressions are defined to be "equivalent" whenever one can be transformed to the other by a sequence of substitutions of expressions for their equals.

Definition 6.1. Let x_1 and x_2 be any two expressions. Define $x_1 \leftrightarrow_S x_2$ to mean there exist expressions s_1 and s_2 such that $\{s_1, s_2\}$ is in EQS, and substituting s_2 for a single occurrence of s_1 in x_1 yields x_2. Then define $\equiv_S$ to be $\leftrightarrow_S{}^*$, the reflexive transitive closure of $\leftrightarrow_S$. Since $\leftrightarrow_S$ is a symmetric relation, $\equiv_S$ is an equivalence relation. The input to Algorithm Eq is a series of instructions of the form "Assert(x,y)" and "Query(x,y)." The operation of algorithm Eq on these instructions is described in terms of the set EQS. (EQS is not actually used in the computations; it is merely used for specification.) Initially, EQS is empty. When an Assert(x,y) instruction is encountered, the pair $\{x,y\}$ is added to EQS. When a Query(x,y) instruction is encountered, a boolean value is output indicating whether $x \equiv_S y$, where $\equiv_S$ is the relation defined by the current value of EQS.

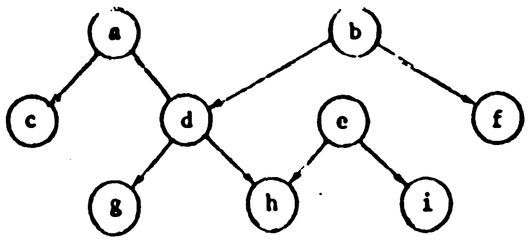


Figure 6.2 A graph created by EqInsert

The problem of computing $\equiv_S$ from EQS is equivalent to the uniform word problem for finitely presented algebras. Several algorithms for solving this problem have appeared in the literature. Nelson and Oppen [62] and Downey, Sethi and Tarjan [30] give algorithms that run in O(N log N) time. However, these algorithms are offline, which means that all the Assert instructions must come before all the Query instructions. Samet [68] gives an online solution to the problem, but he does not give a complete run time analysis. He simply states that the worst case run time (his algorithm uses hashing) for a single Assert instruction is quadratic. Algorithm Eq processes a list of instructions in an expected time (it also uses hashing) of O(N log N), where N is the total size of all the expressions in the input.

6.3 An Overview of Algorithm Eq

In Algorithm Eq, a linked data structure is used to represent expressions. The routine EqInsert(e) is used to obtain the linked representation of the expression e. If e is of the form $f(x_1, x_2, \ldots x_k)$, EqInsert first calls itself recursively to obtain a node n_i representing each x_i. The node representing e will contain the label f and pointers called operand links to the nodes $n_1, n_2, \ldots n_j$. A hashed search is used to determine if the data structure already contains such a

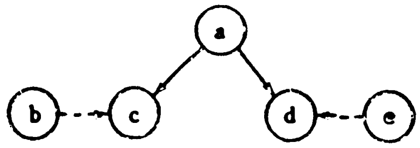


Figure 6.3 A graph with reprlinks

node; if not, a new one is created.

After a few calls to EqInsert have been made, the data structure will take the form of a directed acyclic graph, in which each node represents an expression. For example, if EqInsert is called on the expressions a(c(),d(g(),h())), b(d(g(),h()),f()) and e(h(),i()), the result will be the graph in Figure 6.2.

When two expressions e_1 and e_2 are asserted to be equal, this fact is recorded in the graph by connecting the the node representing e_1 to the node representing e_2 with a pointer called a reprlink. After a number of expressions have been asserted to be equal, the nodes will be partitioned into equivalence classes. McIlroy and Morris's UNION-FIND algorithm [3] is used to keep track of the equivalence classes. All the nodes of a given equivalence class are connected by reprlinks to form a tree. One node in each equivalence class, the root of the tree, is singled out as the "representative" of the class.

Operand links are allowed to point only to representative nodes. Two equivalence classes are merged by connecting their representatives n_1 and n_2 with a reprlink. One of the nodes, say n_1, is chosen as the representative of the new equivalence class. All of the operand links pointing to n_2 must be located and changed to point to n_1.

Once reprlinks have been put into the data structure, a node can represent a class of equivalent expressions. For example, the node with

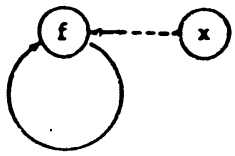


Figure 6.4 A graph with a cycle

label a in the graph in Figure 6.3 (reprlinks are shown as dashed lines in figures) represents the following set of formulas: .

{a(b(),d()), a(b(),e()), a(c(),d()), a(c(),e())}.

Once reprlinks are introduced, the graph is no longer constrained to be acyclic. Nodes in graphs with cycles can represent an infinite set of expressions. For example, the graph in Figure 6.4 is the result of asserting x and f(x) to be equal. The node with label x represents the infinite set {x(), f(x()), f(f(x())), f(f(f(x()))), ...}.

A Query(x,y) instruction is processed by testing whether the two expressions x and y are represented by the same node. For this strategy to work, it is crucial that enough reprlinks are added so that all nodes representing the same expression are in the same equivalence class.

Suppose we are given the graph in Figure 6.5a. As the result of the instruction Assert(d(), e()), the nodes representing d() and e() are "merged" by adding the reprlink and moving the operand links as shown in Figure 6.5b. But now the two nodes that have b as a label, both representative nodes, have identical operand links. Only one of these nodes can be used as the representative of, for example, b(d(),e()). Arbitrarily, the node on the right is chosen as the representative, and the two nodes are merged, as shown in Figure 6.5c. But now nodes that have the label a are equivalent and must be merged, as shown in Figure 6.5d.

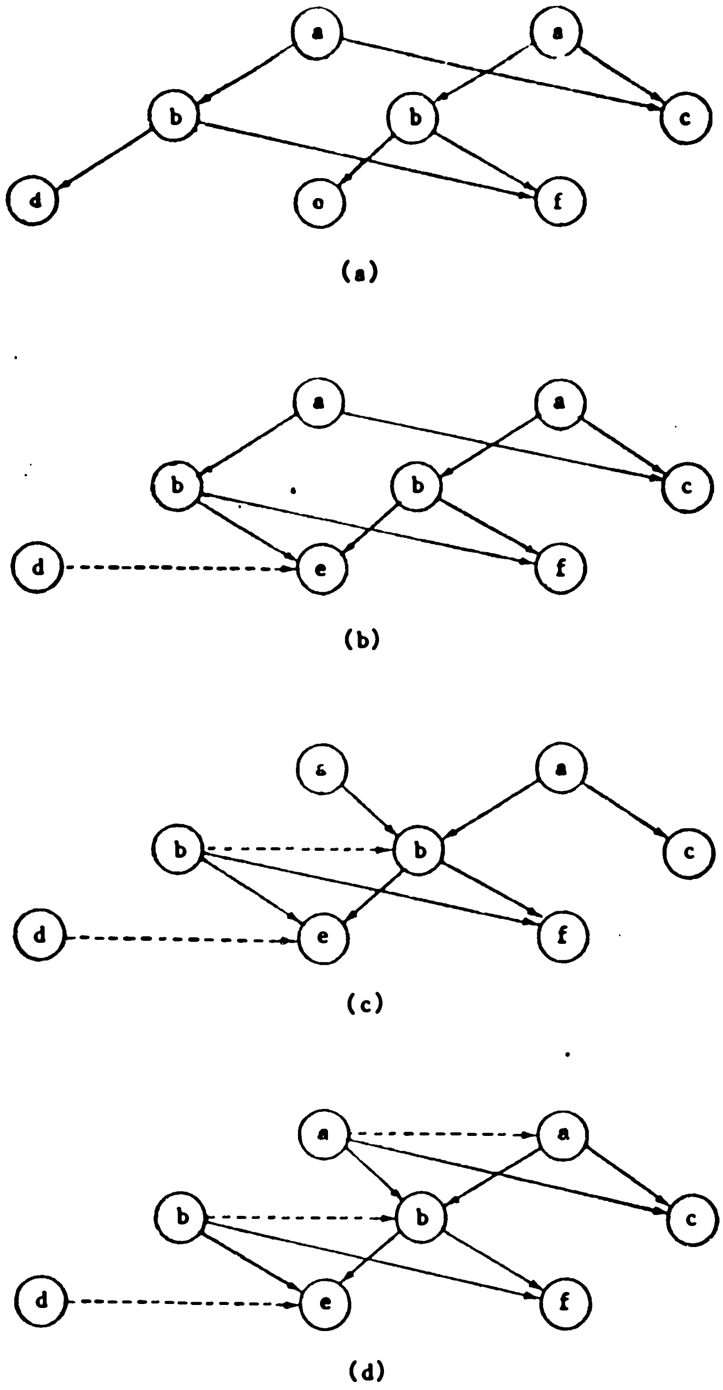


Figure 6.5 Merging equivalent nodes

field	contents
label	a function symbol
operands	a list of pointers to nodes
containers	a list of pointers to operand fields
reprlink	a pointer to a node, or nil
reprcount	a nonnegative integer
next	a pointer to a node, or nil
previous	a pointer to a node, or nil
eligible	a bit

Table 6.6 The fields of a node

The problem is to find all the nodes that need to be merged. Nelson and Oppen [62] use an iterative technique in their offline algorithm. First they add all the reprlinks for all the Assert instructions, and then they step through the all the nodes of the graph serially, merging all the equivalent nodes they find. The algorithm does not halt until it has made an entire pass through all the nodes without finding any nodes that can be merged. Each pass through the list of the nodes takes O(N) time, and they prove that they can make at most log N passes, giving a run time of O(N log N).

An efficient online algorithm cannot afford the luxury of making a pass through the entire list of nodes each time a reprlink is added. Algorithm Eq maintains another set of links, called container links, that are used to locate the nodes that need to be updated when a reprlink is added.

6.4 The Details of Algorithm Eq

Algorithm Eq consists of the routines shown in Figures 6.7-10 and a global data structure, or "graph," as it will be called. The graph has three parts, the first of which is a collection of "nodes." The fields of the node are listed in Table 6.6. The operands and containers fields are

```
Eq:
¢ Process a list of instructions ¢
begin
     while there are instructions left
     do    Read an instruction i
     ;     if    i is of the form Assert(x, y)
           then  EqAssert(x, y)
           elif  i is of the form Query(x, y)
           then  Output EqQuery(x, y)
           fi
     od
end
```

Figure 6.7 Algorithm Eq

```
EqQuery(xx, yy):
¢ Determine if expressions xx and yy are equal ¢
begin
       nx := EqInsert(xx)
;      ny := EqInsert(yy)
;      if   nx = ny
       then EqQuery := true
       else EqQuery := false
       fi
end
```

Figure 6.8 Routine EqQuery

heads of singly-linked lists of nodes. However, there is no reason to present the details of list manipulation here. For the purposes of this discussion, we will assume that nodes are flexible, and the operand and container fields contain variable-sized lists.

Note that each pointer on the container list points to a specific operand field of a node. Some mechanism must also be provided for obtaining from a container link to an operand field a pointer to the node containing the operand field. One simple method is to implement the con-

```
EqInsert(e):
¢ Add expression e to the graph ¢
begin
     e will be of the form f(x1, x2, ... xk)
        for a function symbol f and some k ≥ 0
;    for each xi do a[i] := EqInsert(xi) od
;    h := hash(f, (a[1], a[2], ... a[k]))
;    Search bucket[h] for a node with operator f
        and operands a[1], a[2], ... a[k]
;    if   the search succeeds in finding a node n
     then while n.reprlink ≠ nil do n := n.reprlink od
     else n := a pointer to a newly allocated node
     ;    Put n in bucket[h]
     ;    n.label := f
     ;    n.operands := (a[1], a[2], ... a[k])
     ;    Add a pointer to the i'th operand field of n
             to the container list of a[i], for each i in {1 ... k}
     ;    n.containers := the empty list
     ;    n.reprlink := nil
     ;    n.reprcount := 1
     fi
;    EqInsert := n
end
```

Figure 6.9 Routine EqInsert

tainer link as a pointer to a node and an offset.

The term "node" will be used to refer to an area of storage, and not its contents. For example, if n_1 and n_2 are nodes, $n_1 \neq n_2$ means that n_1 and n_2 occupy different locations in memory, but does not exclude the possibility that corresponding fields of n_1 and n_2 have identical contents. Initially, there are no nodes. As the algorithm executes, new nodes can be allocated, but nodes are never deallocated.

The second part of the graph is "bucket," an array of pointers to nodes. Each element of bucket is the head of a list of nodes that is dou-

```
       EqAssert(xx, yy):
       ¢ assert the equality of expressions xx and yy ¢
       begin
             nx := EqInsert(xx) ; ny := EqInsert(yy)
II     ;     Add (nx, ny) to ToDo
             ¢ add {xx, yy} to EQS ¢
       ;     EqLoop:
             while ToDo is not empty
             do   Remove an element (x, y) from ToDo
             ;    while x.reprlink ≠ nil do x := x.reprlink od
             ;    while y.reprlink ≠ nil do y := y.reprlink od
             ;    if   x ≠ y
                  then if y.reprcount > x.reprcount
                       then swap x and y fi
                  ;    y.reprlink := x
                  ;    x.reprcount := x.reprcount + y.reprcount
                  ;    ConLoop:
                       for every w on y.containers
                       do  w.operand := x
III                    ;   z := a pointer to the node containing w.operand
                           if z is eligible
                           then Remove z from the bucket it is in
                           ;    h := hash(z.label, z.operands)
                           ;    Search bucket[h] for a node with
                                   the same label and operands as z
                           ;    if   the search finds a duplicate d
                                then Add (d, z) to ToDo
                                else Put z in bucket[h]
                                fi
                           fi
                       od
                  ;    Append y.containers to x.containers
                  fi
             od
       end
```

Figure 6.10 Routine EqAssert

bly linked through the next and previous fields of each node. If a node is in one of the buckets, the node will be said to be "eligible"; otherwise it is said to be "ineligible." A bit in each node is used to keep track of whether the node is eligible; the bit is set when a node is put in a bucket, and cleared when it is removed.

The function "hash" takes as input a label and a list of the addresses of zero or more nodes, and produces the index of one of the buckets. Initially, there are no nodes, so that all the buckets are empty.

The third part of the graph is "ToDo," which is a set of pairs of pointers to nodes. The notation "$\{n_1, n_2\}$ is in ToDo" will be defined to mean that either (n_1, n_2) or (n_2, n_1) is in ToDo. Initially, ToDo is empty.

The set EQS, which contains the pairs of expressions that have been asserted to be equal, will also be considered to be part of the graph. The set is not used anywhere in the computation, but it is used in invariant assertions about the graph. The comment on line 6 of EqAssert indicates exactly when, for the purposes of invariant assertions about EQS, a new pair is added to EQS.

6.5 An Overview of the Correctness Proof

The method used to prove the correctness of Algorithm Eq was inspired by the methods of Owicki and Gries [63] for proving parallel programs. Three areas of the algorithm, lines 12-20 of EqInsert, lines 5-6 of EqAssert, and lines 9-32 of EqAssert are designated as Critical Regions I, II, and III respectively. All modifications to the graph are performed within critical regions.

A number of invariant assertions ("invariants" for short) about the graph are proved to hold whenever the program is not in a critical region. There are two ways of establishing an invariant. The first way is to show that it is a consequence of other invariants.

The second way is a two-step procedure. First, the invariant must be shown to hold when the algorithm begins, that is, when EQS and ToDo are empty, and there are no nodes in the graph. For all the invariants in this proof this step is trivial, and will be omitted.

The second step is to show the invariant holds at the end of each critical region. In this step, we are allowed to assume that every invariant, including the one being established, holds at the beginning of the critical region. (Therefore if none of the structures on which an invariant depends are changed by a critical region, the proof that the invariant holds at the end of that region is trivial.) Essentially, the theorem "every invariant holds at the end of every critical region" is being proved by induction on the number of critical regions executed. Since the graph can only be modified within critical regions, once an invariant has been proved to hold as control leaves a critical region, it must remain valid until another critical region is entered.

To formulate the invariants, some equivalence relations and a function will be defined.

The relation $\equiv_R$, defined on nodes, is the equivalence relation determined by the reprlinks. A second relation, $\equiv_N$ is defined on nodes. Its definition is quite technical, depending on the reprlinks, operand links, labels of nodes, and on the ToDo list. The relation $\equiv_R$ is always refinement of $\equiv_N$, and the two relations are the same whenever EqQuery is called. EqAssert(x, y) operates by first adding the pair of nodes that represent x and y to ToDo. This addition updates $\equiv_N$, but not $\equiv_R$. The body of ConLoop adds reprlinks to the graph; it is repeated until $\equiv_R$ and $\equiv_N$ are once again the same relation.

In section 6.3, the notion of a node representing an expression was introduced. To be more precise, an expression is represented by an equivalence class of nodes. The function recl(e) is defined to map the expression e to an equivalence class of $\equiv_N$ that represents e. (The name "recl" is an abbreviation for "representative class.") For any e, recl(e) is always a finite set of nodes, even for cyclic graphs. If C is an equivalence class of $\equiv_N$, the set of expressions $\{e \mid \text{recl}(e) = C\}$ can be infinite, but this does not cause any problems.

The definition of recl mirrors the computation performed by EqInsert. When the routine EqQuery(x, y) calls EqInsert on x and y and compares the results, it is testing whether recl(x) = recl(y). The relation $\equiv_E$, which is defined over expressions, is intended to mirror the computation performed by EqQuery. For expressions x and y, $x \equiv_E y$ is defined to mean recl(x) = recl(y).

The key to proving the correctness of Algorithm Eq lies in establishing the invariant that $\equiv_E$ and $\equiv_S$ are (almost) the same. We will see that Critical Regions I and III leave $\equiv_E$ (essentially) unchanged. Critical Region II updates both EQS and ToDo, changing both $\equiv_E$ and $\equiv_S$. We will see, however, that it makes the same change to both relations.

6.6 Some Simple Invariants and an Easy Theorem

There are two levels of understanding of a complex algorithm. First, one understands _what_ the algorithm is doing, and then he can understand _why_ it works. The invariants and theorem presented in this section are of the "what" variety; they present more formally some of the ideas introduced in section 6.3.

Since only ToDo and EQS are changed in Critical Region II, and none of the invariants in this section depend on ToDo or EQS, Critical Region II can be ignored.

Invariant 6.2. The directed graph whose vertices are nodes and whose edges are reprlinks is a forest in which the reprlinks of offspring point to their parents; and the roots have *nil* reprlinks.

Proof. When a node is created in Critical Region I, its reprlink is set to *nil*, creating a new tree of size 1 in the forest. In Critical Region III, lines 10 and 11 guarantee that x and y are roots of trees in the forest. In line 15, the tree whose root is y is made a subtree of the tree whose root is x. □

Invariant 6.2 guarantees the termination of the loops on line 11 of EqInsert and lines 10 and 11 of EqAssert.

Invariant 6.3. If n is the root of a subtree t, n.reprcount is the number of nodes in t. If m is an offspring of n, n.reprcount is at least double m.reprcount.

Proof. When a new tree of size 1 is created in Critical Region I, its reprcount field is set to 1 at line 20. When the tree y is added to the tree x in Critical Region III, x.reprcount is suitably updated at line 16. Because x.reprcount was at least double the reprcount fields of each of its sons upon entry to the region, and x.reprcount is being increased, x.reprcount is still at least double the reprcount fields of each of its old sons. Because lines 13-14 guarantee that x.reprcount ≥ y.reprcount, after line 16 is executed x.reprcount will be at least double y.reprcount. □

Invariant 6.3 is not needed to prove the correctness of Algorithm Eq. It is used in section 6.15 to derive the O(N log N) running time.

Invariant 6.4. Each eligible node with label f and operands $n_1, n_2, \ldots n_k$ is in bucket[h], where h = hash(f, $(n_1, n_2, \ldots n_k)$).

Proof. When a node is created in Critical Region I, it is placed in the proper bucket at line 13. The label field of a node is never changed, and the operand field of a node, or the bucket in which it resides, is changed only in Critical Region III. When an operand field of an eligible node is changed at line 19, the node is removed from the bucket it is in, and if it is put back into a bucket, the bucket is selected at line 23 on the basis of its updated operand fields. □

Invariant 6.5. There are not two eligible nodes with identical operand and and label fields.

Proof. Before a node with a label f and operands $a_1, a_2, \ldots a_k$ is created in Critical Region I, bucket[hash(f, $(a_1, a_2, \ldots a_k)$)] is searched for a node with those fields. By Invariant 6.4, if such a node exists, it will be in that bucket. When the operand field of a node z is changed in line 19 of Critical Region III, it is made ineligible. If another node d exists with label and operand fields identical to those of z, node z is left ineligible. □

Definition 6.6. A node n is said to be a representative whenever n.reprlink is nil.

Invariant 6.7. There is a one-to-one correspondence between operand links of representative nodes and container links. If an operand link of a representative node n points to node m, there is a corresponding container link in node m that points back to the operand field.

Proof. When a node is created in Critical Region I, for every operand link created, a corresponding container link is created as well at lines 16-17. Since at the time of the creation of the node no operand link can possibly point to it, its container list is made empty at line 18.

Whenever an operand link of a representative node is changed in Critical Region III, the corresponding container link is changed as well. ConLoop searches all the nodes on the container list of y for the corresponding operand links pointing to y, and changes them to point to x. Then, at line 22, the container list of y is appended to the container list of x to preserve Invariant 6.7. The container list of y is no longer relevant, becsuae y is no longer a representative. □

<u>Invariant</u> 6.8. Every operand link points to a representative node.

<u>Proof</u>. First observe that EqInsert always returns a representative node, either by finding one with the loop at line 11, or by creating one at line 19. When a node is created in Critical Region I, its operands are set to nodes that are returned by a recursive call to EqInsert. Hence, those nodes must be representstive nodes.

When the node y is changed from a representative to a nonrepresentative at line 15 of Critical Region III, nodes with operand links pointing to y no longer point to representatives. ConLoop searches the container list of y, which by Invariant 6.7 allows it to find all the operand links pointing to y, and to move them to point to x, reestablishing Invariant 6.8. □

<u>Theorem</u> 6.9. Outside of routine EqAssert, ToDo is empty.

<u>Proof</u>. Initially ToDo is empty. The only place ToDo is modified is in EqAssert, which adds a pair to ToDo and executes the body of EqLoop until it is empty again. □

6.7 Some Lemmas About Relations

In the following sections, it will be necessary several times to prove that one relation is the same as another relation. These tasks will be made easier by the following lemma.

Lemma 6.10. Let $\leftrightarrow$ be a relation defined over a domain D, and $\equiv$ be $\leftrightarrow^*$. Let ρ be any relation defined over D that is both reflexive and transitive. To establish that:

$$\text{for all } x \text{ and } y \text{ in D, } x \equiv y \text{ implies } x \,\rho\, y \tag{6.1}$$

it is sufficient to show:

$$\text{for all } x \text{ and } y \text{ in D, } x \leftrightarrow y \text{ implies } x \,\rho\, y. \tag{6.2}$$

Proof. Suppose (6.2) holds. Then let x and y be any elements of D such that $x \equiv y$. If $x = y$ then, since ρ is by hypothesis reflexive, $x \,\rho\, y$. If $x \neq y$ then, since $\equiv$ is defined to be $\leftrightarrow^*$, there is a sequence of one or more elements $v_1, v_2, \ldots v_k$ from D such that $x = v_1 \leftrightarrow v_2 \ldots v_k = y$. But by (6.2), $x = v_1 \,\rho\, v_2 \ldots v_k = y$. Since ρ is by hypothesis transitive, $x \,\rho\, y$. □

As an example, Lemma 6.10 is used to prove a simple lemma on substitution.

Lemma 6.11. For any expressions e_1, e_2, d, and x, if $e_1 \equiv_S e_2$, then $d[e_1//x] \equiv_S d[e_2//x]$.

Proof. By Lemma 6.10, it suffices to show that $e_1 \leftrightarrow_S e_2$ implies $d[e_1//x] \equiv_S d[e_2//x]$. By definition $e_1 \leftrightarrow_S e_2$ means there is a pair of expressions (s_1, s_2) in EQS such that substituting s_1 for a single occurrence of s_2 in e_1 yields e_2. If this same substitution is applied to each copy of e_1 in $d[e_1//x]$, the result will be $d[e_2//x]$. Therefore we have $d[e_1//x] \equiv_S d[e_2//x]$. □

The following lemma is used in conjuction with induction on the relation $\equiv_S$.

Lemma 6.12. For expressions e_1 and e_2, if $e_1 \leftrightarrow_S e_2$, then either $\{e_1, e_2\}$ is in EQS, or for some $k > 0$, e_1 is of the form $f(x_1, x_2, \ldots x_k)$, e_2 is of the form $f(y_1, y_2, \ldots y_k)$, and for each i in $\{1 \ldots k\}$, $x_i \equiv_S y_i$.

Proof. The substitution that transforms e_1 to e_2 either operates on all of e_1 or somewhere in one of its operands. If it operates on all of e_1, then $\{e_1, e_2\}$ must be in EQS. If it operates on one of its operands, the rest of e_1 is unchanged. Therefore e_1 and e_2 must begin with the same function symbol and have the same number k of operands. One operand x_j of e_1 can be transformed to the corresponding operand y_j of e_2 by a single substitution. That is, $x_j \leftrightarrow_S y_j$. The other operands must be identical. Therefore, for all i in $\{1 \ldots k\}$, $x_i \equiv_S y_i$. □

6.8 Definitions

In this section, definitions and proofs are given that formalize the concepts introduced in section 6.5.

The relation $n_1 \equiv_R n_2$ is defined to mean that n_1 and n_2 are in the same UNION-FIND tree.

Definition 6.13. For any nodes n_1 and n_2 define $n_1 \leftrightarrow_R n_2$ to mean n_1.reprlink = n_2 or n_2.reprlink = n_1. Then define $\equiv_R$ to be $\leftrightarrow_R{}^*$.

Informally, $n_1 \equiv_N n_2$ means that either $n_1 \equiv_R n_2$, or that nodes n_1 and n_2 must eventually be merged. When EqAssert detects that n_1 and n_2 must be merged, it puts the pair on the ToDo list. Any other nodes that have identical labels and equivalent operands will need to be merged as well.

Therefore a definition should be chosen for $\equiv_N$ such that any of the following conditions:

$m \equiv_R n$ (6.3)

$\{m, n\}$ is in ToDo (6.4)

m.label = n.label,
m has operands $p_1, p_2, \ldots p_k$, n has operands $q_1, q_2, \ldots q_k$, (6.5)
and for each i in $\{1 \ldots k\}$, $p_i \equiv_N q_i$

imply $m \equiv_N n$. Conditions (6.3), (6.4), and (6.5) can be used as the basis for inductive definition for $\equiv_N$. However, because the graph can be cyclic, the induction cannot be on the structure of the graph.

<u>Definition</u> 6.14. Define over the set of nodes a sequence of equivalence relations ${}_h{\equiv}_N$, for nonnegative integers h. The relation ${}_0{\equiv}_N$ is defined to be the identity relation. For $h > 0$, define $m \;{}_h{\leftrightarrow}_N\; n$ to mean that (6.3) or (6.4) holds, or that m.label = n.label, m has operands $p_1, p_2, \ldots p_k$, n has operands $q_1, q_2, \ldots q_k$, and for each i in $\{1 \ldots k\}$, $p_i \;{}_{j_i}{\equiv}_N\; q_i$, for some $j_i < h$. Then define ${}_h{\equiv}_N$ as ${}_h{\leftrightarrow}_N{}^*$. Finally, define $m \equiv_N n$ to mean there is some $h \geq 0$ such that $m \;{}_h{\equiv}_N\; n$.

<u>Theorem</u> 6.15. For any nodes m and n, if conditions (6.3), (6.4), or (6.5) hold, then $m \equiv_N n$.

<u>Proof</u>. If $m \equiv_R n$ or $\{m, n\}$ in ToDo, then $m \;{}_1{\leftrightarrow}_N\; n$, and therefore $m \equiv_N m$. If m and n are eligible nodes with identical labels, and operands $p_1, p_2, \ldots p_k$ and $q_1, q_2, \ldots q_k$ such that $p_i \equiv_N q_i$, then there are $j_1, j_2, \ldots j_k$ such that $p_i \;{}_{j_i}{\equiv}_N\; q_i$. Letting h be the maximum of $j_1, j_2, \ldots j_k$ (zero, if $k = 0$), we have that $m \;{}_{h+1}{\leftrightarrow}_N\; n$, so that $m \;{}_{h+1}{\equiv}_N\; n$, and $m \equiv_N n$. □

The converse of Theorem 6.15 is also true, but will not be needed.

<u>Definition</u> 6.16. Let e be an expression of the form $f(x_1, x_2, \ldots x_k)$, for $k \geq 0$. The function recl(e) is defined inductively

on the degree of e to map e onto a set of nodes as follows. A node n is defined to be in recl(e) whenever there is an eligible node m such that $m \equiv_N n$, and m has the label f and operands $p_1, p_2, \ldots p_k$ such that for each $i \in \{1 \ldots k\}$, $p_i \in recl(x_i)$. An expression e is said to be "in the graph" whenever recl(e) is nonempty.

Theorem 6.17. For any expression e, the set recl(e) is either empty, or an equivalence class of the relation $\equiv_N$.

Proof. If recl(e) is empty, we are done. Suppose recl(e) is nonempty. Then let n_1 be a node in recl(e), and n_2 be a node such that $n_1 \equiv_N n_2$. The eligible node m (see Definition 6.16) that guarantees $n_1 \in recl(e)$ works equally well for n_2. Therefore $n_2 \in recl(e)$. The preceding argument generalizes to show that if $n_1 \in recl(e)$, the equivalence class of $\equiv_N$ that contains n_1 is a subset of recl(e). Therefore recl(e) must consist of the union of equivalence classes of $\equiv_N$. The following lemma demonstrates that recl(e) contains at most one equivalence class. □

Lemma 6.18. For any nodes n_1 and n_2, if n_1 and n_2 are in recl(e), then $n_1 \equiv_N n_2$.

The proof is by induction on the degree of e. Let e be of the form $f(x_1, x_2, \ldots x_k)$, for some $k \geq 0$. Let n_1 and n_2 be nodes in recl(e). Then there must be eligible nodes m_1 and m_2, each with label f, such that $m_1 \equiv_N n_1$ and $m_2 \equiv_N n_2$, and if $p_1, p_2, \ldots p_k$ are the operands of m_1 and $q_1, q_2, \ldots q_k$ are the operands of m_2, both p_i and q_i are in $recl(x_i)$ for all i in $\{1 \ldots k\}$. By induction, $p_i \equiv_N q_i$. Then, by Theorem 6.15, $m_1 \equiv_N m_2$, giving us $n_1 \equiv_N m_1 \equiv_N m_2 \equiv_N n_2$. □

Definition 6.19. The domain of the equivalence relation $\equiv_E$ is the set of expressions in the graph, that is, $\{e \mid recl(e) \neq \emptyset\}$. For two expressions e_1 and e_2 that are in the graph, define $e_1 \equiv_E e_2$ to mean $recl(e_1) = recl(e_2)$.

6.9 Invariants

Invariant 6.20 is the key to the proof of correctness of Algorithm Eq. It says that $\equiv_E$, the relation computed by EqQuery, is essentially the same as $\equiv_S$, the relation determined by EQS.

Invariant 6.20. If expressions e_1 and e_2 are in the graph, then $e_1 \equiv_E e_2$ if and only if $e_1 \equiv_S e_2$.

To establish Invariant 6.20, two less spectacular invariants will be needed as well.

Invariant 6.21. For every node n there is an expression e such that $n \in recl(e)$.

Invariant 6.22. If $\{e_1, e_2\}$ is in EQS, then both e_1 and e_2 are in the graph.

In the next few sections, Invariants 6.20, 6.21, and 6.22 are proved to hold at the end of each critical region. When discussing the effect a critical region has on a graph, "graph a" will refer to the state of the graph after the critical region has been left, and "graph b" will refer to the state of the graph before the region has been entered. (A mnemonic is "b" means "before" and "a" means "after.") The function and relations that have been defined in terms of the graph will have the superscripts "b" and "a" placed on them to distinguish which graph is meant.

6.10 Critical Region I

Critical Region I does not change the set EQS, and therefore does not change the relation $\equiv_S$. However, Critical Region I does add a node to the graph, which adds more expressions to the graph and extends the domain of $\equiv_E$. To prove Invariant 6.20, we must show that $\equiv_E$ is extended to be consistent with $\equiv_S$.

<u>Invariant</u> 6.23. If ToDo is empty, then $\equiv_N$ and $\equiv_R$ are the same relation.

<u>Proof</u>. This invariant is proved by showing that it is a consequence of Invariants 6.2, 6.5, and 6.8.

Let m and n be any two nodes. If m $\equiv_R$ n, then, by Theorem 6.15, m $\equiv_N$ n. Thus, it is only necessary to show that m $\equiv_N$ n implies m $\equiv_R$ n. We shall see, by induction on h, that m $_h{\equiv}_N$ n implies m $\equiv_R$ n.

If h = 0, then m = n, and clearly m $\equiv_R$ n. For the induction step, assume h > 0. By Lemma 6.10, it is sufficient to show that m $_h{\leftrightarrow}_N$ n implies m $\equiv_R$ n. Definition 6.14 admits three possibilites. The first is tbat m $\equiv_R$ n, which is what we need to prove. The second possibility, that {m, n} is in ToDo, is ruled out, since ToDo is assumed to be empty. The final possibility is that m and n are eligible nodes with identical labels and equivalent operands. That is, if p_1, p_2, ... p_k are the operands of m, and q_1, q_2, ... q_k are the operands of q, then p_i $_{j_i}{\equiv}_N$ q_i for some $j_i < h$. Then by induction, each $p_i \equiv_R q_i$. Since they are pointed to by operand links of m and n, nodes p_i and q_i must both be representative nodes, by Invariant 6.8. Since p_i and q_i are both roots in the same tree (see Invariant 6.2), they must be the same node. Since m and n are both eligible, and have identical labels and operand links, by Invariant 6.5, m = n, and hence m $\equiv_R$ n. □

<u>Invariant</u> 6.24. If ToDo is empty, for any expression e, recl(e) is either empty or contains exactly one representative node (and possibly some nonrepresentative nodes).

By Theorem 6.17, recl(e) is either empty, or an equivalence class of $\equiv_N$. Since ToDo is assumed empty, by Invariant 6.23, recl(e) is either empty or an equivalence class of $\equiv_R$. But by Invariant 6.2, an equivalence class of $\equiv_R$ contains exactly one representative node. $\square$

Since $\equiv_E$ is defined in terms of recl, the effect Critical Region I has on $\equiv_E$ is determined by the effect it has on recl. For the rest of this section, "b" will refer to the graph before Critical Region I, and "a" will refer to the graph afterwards. Also, n_I will refer the node created at line 12 of Critical Region I.

Lemma 6.25. For every expression e, any node m that is in $recl^b(e)$ is also in $recl^a(e)$.

Proof. If $m \in recl^b(e)$, it is because there are a collection of nodes with fields having values specified by Definition 6.16. Since the only difference between graphs b and a is the addition of the node n_I, the same nodes that guarantee $m \in recl^b(e)$ also guarantee $m \in recl^a(e)$. $\square$

Lemma 6.26. For any expression e and node m such that $m \neq n_I$ and $m \in recl^a(e)$, $m \in recl^b(e)$.

Proof. The proof is by induction on the degree of e. The expression e will be be of the form $f(x_1, x_2, \ldots x_k)$. Let m be a node other than n_I in $recl^a(e)$. By Definition 6.16, there will be in graph a an eligible node p with label f, and operands $q_1, q_2, \ldots q_k$ such that each $q_i \in recl^a(x_i)$, and $p \equiv_N^a m$. Since, by Theorem 6.9, ToDo is empty whenever EqAssert is called, by Invariant 6.23, $p \equiv_R^a m$. Since n_I is not in graph b, no nodes in graph b can have any operand links pointing to it. Since no operand links pointing to n_I are added in Critical Region I, there can be no links to n_I in graph a, either. Since operand links of node p point in graph a to each q_i, none of them can be n_I, so that by induction, each

$q_i \in recl^b(x_i)$. Therefore to show $m \in recl^b(e)$ it is only necessary to show $m \equiv_R^b p$. The only effect Critical Region I has on the relation $\equiv_R$ is to add the equivalence class $\{n_I\}$. Since $m \neq n_I$ by hypothesis, and $m \equiv_R^a p$, $m \equiv_R^b p$. □

Theorem 6.27. For any expression e, either $recl^a(e) = recl^b(e)$, or $recl^b(e) = \emptyset$ and $recl^a(e) = \{n_I\}$.

Proof. Let e be some expression. If $recl^a(e) = recl^b(e)$, we are done, so assume $recl^a(e) \neq recl^b(e)$. Suppose n_I is not in $recl^a(e)$. Then, by Lemmas 6.25 and 6.26, for any node m, $m \in recl^b(e)$ if and only if $m \in recl^a(e)$, contradicting the assumption $recl^a(e) \neq recl^b(e)$. Therefore $n_I \in recl^a(e)$. But since ToDo is empty and $recl^a(e)$ is nonempty, by Theorem 6.17 and Invariant 6.23, $recl^a(e)$ is an equivalence class of $\equiv_R^a$. Since $n_I \in recl^a(e)$ and the equivalence class containing n_I is $\{n_I\}$, $recl^a(e) = \{n_I\}$. But by Lemma 6.25, $recl^b(e)$ is a subset of $recl^a(e)$. But n_I cannot be in $recl^b(e)$, since n is not in graph b. Therefore $recl^b(e)$ is empty. □

Corollary. If e_1 is in graph b, it is in graph a as well. Therefore Invariant 6.22 must hold at the end of Critical Region I.

Theorem 6.28. Invariant 6.21 holds at the end of Critical Region I.

Proof. Let m be a node in graph a. We must exhibit an expression e such that $m \in recl^a(e)$. If m is in graph b, then by Invariant 6.21, there is an expression e such that $m \in recl^b(e)$, and by Theorem 6.27, $recl^b(e) = recl^a(e)$.

The other possibility is that $m = n_I$. Let $p_1, p_2, \ldots p_k$ be the nodes pointed to by the operand links of n. Since these nodes are located by line 6 of EqInsert, before n_I was created, they cannot be n_I, and must be in graph b. Using the same reasoning as before, there must be expressions $x_1, x_2, \ldots x_k$ such that $p_i \in recl^a(x_k)$. Then, if f is the label of m, by Definition 6.16, $m \in recl^a(f(x_1, x_2, \ldots x_k))$. □

Lemma 6.29. If $e_1 \equiv_S^b e_2$ and e_1 is in graph b, e_2 is in graph b as well.

Proof. If e_1 and e_2 are expressions, let $e_1 \rho e_2$ mean that e_1 is in graph b implies e_2 is in graph b. We must then prove that $e_1 \equiv_S^b e_2$ implies $e_1 \rho e_2$. Because ρ is both reflexive and transitive, by Lemma 6.10, it suffices to prove, by induction of the degree of e_1, that $e_1 \leftrightarrow_S^b e_2$ implies $e_1 \rho e_2$.

Suppose $e_1 \leftrightarrow_S^b e_2$ and e_1 is in graph b. By Lemma 6.12, either $\{e_1, e_2\}$ are in EQS^b, or e_1 is of the form $f(x_1, x_2, \ldots x_k)$ and e_2 is of the form $f(y_1, y_2, \ldots y_k)$, where for all $i \in \{1 \ldots k\}$, $x_i \equiv_S^b y_i$. In the first case, e_2 is in graph b by Invariant 6.22.

Let us consider the second case. Because e_1 is in graph b, there is in graph b an eligible node p with operands $m_1, m_2, \ldots m_k$ such that each $m_i \in recl^b(x_i)$. Therefore each x_i is in graph b. Then, by induction, each y_i is in graph b, so that by Invariant 6.20, $x_i \equiv_E^b y_i$. But then $recl^b(y_i) = recl^b(x_i)$, so that $m_i \in recl^b(y_i)$, and $p \in recl^b(e_2)$. Therefore e_2 is in graph b. □

Lemma 6.30. For any expressions e_1 and e_2, if $recl^a(e_1)$ is equal to $recl^a(e_2)$ and contains a node in graph b, then $e_1 \equiv_S e_2$.

Proof. Let $C = recl^a(e_1) = recl^a(e_2)$. Because C contains a node that is in graph b, C cannot be $\{n_I\}$. Therefore by Theorem 6.27, $recl^b(e_1) = recl^a(e_1)$ and $recl^b(e_2) = recl^a(e_2)$. Then, since $recl^b(e_1) = C = recl^a(e_2)$, $e_1 \equiv_E^b e_2$, and by Invariant 6.20, $e_1 \equiv_S e_2$. □

Theorem 6.31. Invariant 6.20 holds at the end of Critical Region I.

Proof. Let e_1 and e_2 be two expressions in graph a. We must show that $e_1 \equiv_E^a e_2$ if and only if $e_1 \equiv_S e_2$.

First, suppose that $e_1 \equiv_E^a e_2$, which means that $recl^a(e_1) = recl^a(e_2)$. Let $C = recl^a(e_1) = recl^a(e_2)$. If C contains a node in graph b, then by Lemma 6.30, $e_1 \equiv_S e_2$. Suppose C contains no nodes from graph b. C cannot be empty, because e_1 and e_2 are by hypothesis in the graph. Therefore C must consist of just $\{n_I\}$.

Let f be the label, and $p_1, p_2, \ldots p_k$ be the operands of node n_I. Since $recl^a(e_1) = recl^a(e_2) = \{n_I\}$, by Definition 6.16, e_1 must be of the form $f(x_1, x_2, \ldots x_k)$, where $p_i \in recl^a(x_i)$, and e_2 must be of the form $f(y_1, y_2, \ldots y_k)$, where $p_i \in recl^a(y_i)$. Since $recl^a(x_i)$ and $recl^a(y_i)$ have an element, p_i, in common, by Theorem 6.17, $recl^a(x_i) = recl^a(y_i)$. Now each of $p_1, p_2, \ldots p_k$ must be in graph b, since they are located by line 6 of EqInsert, before n was created. Therefore, by Lemma 6.30, $x_i \equiv_S y_i$. Each y_i can be substituted for x_i in e_1 to yield e_2. Therefore, by Lemma 6.11, $e_1 \equiv_S e_2$.

Next we must show that $e_1 \equiv_S^a e_2$ implies $e_1 \equiv_E^a e_2$. Suppose that $e_1 \equiv_S e_2$. Now either one of e_1 and e_2 is in graph b, or neither of them is. Suppose one of them is in graph b. Then, by Lemma 6.29, both are in graph b, and by Invariant 6.20, we have that $e_1 \equiv_E^b e_2$, and hence $recl^b(e_1) = recl^b(e_2)$. Since e_1 is in graph b, $recl^b(e_1)$ is nonempty, and by Theorem 6.27, $recl^b(e_1) = recl^a(e_1)$. Likewise, $recl^b(e_2) = recl^a(e_2)$, so that:

$$recl^a(e_1) = recl^b(e_1) = recl^b(e_2) = recl^a(e_2),$$

which means that $e_1 \equiv_E^a e_2$.

The other possibility is that neither e_1 nor e_2 is in graph b. But both of them are in graph a. Since $recl^b(e_1)$ is empty and $recl^a(e_1)$ is not, by Theorem 6.27, $recl^a(e_1) = \{n_I\}$. By the same reasoning,

$recl^a(e_1) = \{n_1\}$. Therefore $e_1 \equiv_E^a e_2$. □

6.11 The Correctness of EqInsert

Before moving on to Critical Region II, we will pause to prove the correctness of EqInsert. In this section, b and a will refer to the state of the graph before and after a call to EqInsert(e), and n_r is the node returned by the call.

Theorem 6.32. For any expression d, if d is in graph b, then $recl^b(d) = recl^a(d)$.

Proof. EqInsert modifies the graph only by repeated executions of Critical Region I. If recl(d) is nonempty, by Theorem 6.27, an execution of Critical Region I does not change recl(d). Therefore $recl^b(d) = recl^a(d)$. □

Theorem 6.33. Expression e is in graph a, and n_r is a representative node in $recl^a(e)$.

Proof. The proof is by induction on the degree of e. After the i'th recursive call to EqInsert at line 6, by induction and Theorem 6.32, $recl(x_i)$ will be nonempty, and s[i] will be the representative node in $recl(x_i)$. Then a node n with label f and operands s[1], s[2], ... s[k] will be found on line 8 or created on lines 12-20. By Definition 6.16, $n \in recl^a(e)$. The representative of n, n_r is returned as the value of EqInsert. Since $n_r \equiv_R n$, and therefore $n_r \equiv_N n$, $n_r \in recl(e)$. □

6.12 Critical Region II

Critical Region II modifies both $\equiv_E$ and $\equiv_S$. To prove Invariant 6.20 holds at the end of Critical Region II, we must prove that both relations are updated in the same fashion. In this section, "b" will refer to the graph before Critical Region II is executed and "a" will refer to the graph afterwards.

Lemma 6.34. For any nodes m and n, and any expressions d and e:

if d $\equiv_S^b$ e then d $\equiv_S^a$ e (6.6)

if m $\equiv_N^b$ n then m $\equiv_N^a$ n (6.7)

if n $\in$ $recl^b(e)$, then n $\in$ $recl^a(e)$. (6.8)

Proof. Suppose d $\equiv_S^b$ e. Then there is a sequence of substitutions using expression pairs in EQS^b that will transform d to e. But every expression pair in EQS^b is also in EQS^a, so that d $\equiv_S^a$ e. Thus, (6.6) is established. A similar argument will establish (6.7) and (6.8). If m $\equiv_N^b$ n (or n $\in$ $recl^b(e)$), it is because of a collection of nodes and fields in graph b. Since the only difference between graphs b and a is the addition of a pair to ToDo and a pair to EQS, the same collection of nodes and fields are in graph a, so that m $\equiv_N^a$ n (or n $\in$ $recl^a(e)$). □

Theorem 6.35. Invariant 6.22 holds at the end of Critical Region II.

Proof. We must show that if $(e_1, e_2) \in EQS^a$, then e_1 and e_2 are in graph a. Let (e_1, e_2) be a pair in EQS^a. Then, either $(e_1, e_2) = (xx, yy)$ or (e_1, e_2) is in EQS^b. In the first case, line 4 of EqInsert and Theorems 6.33 and 6.32 guarantee that e_1 and e_2 are in graph b. In the second case, e_1 and e_2 are in graph b by Invariant 6.22. But if e_1 is in graph b, there is a node n such that n $\in$ $recl^b(e_1)$. By Lemma 6.34, the same node n is in $recl^a(e_1)$. Therefore e_1 is in graph a. By the same reasoning, e_2 is in graph b. □

Theorem 6.36. Invariant 6.21 holds at the end of Critical Region II.

Proof. We must show for every node m, there is an expression e such that $m \in recl^a(e)$. Let m be a node in graph a. By Invariant 6.21, there is an expression e such that $n \in recl^b(e)$. By Lemma 6.34, $n \in recl^a(e)$ as well. $\square$

Theorem 6.37. If expressions e_1 and e_2 are in graph a and $e_1 \equiv_S^a e_2$, then $e_1 \equiv_E^a e_2$.

Proof. For expressions d_1 and d_2, define the relation $d_1 \rho d_2$ to mean d_1 is in graph a implies that d_2 is in graph a and $d_1 \equiv_S^a d_2$. To establish the theorem, it is sufficient to prove that $e_1 \equiv_S^a e_2$ implies $e_1 \bar{\rho} e_2$. But notice that ρ is both reflexive and transitive. Therefore, by Lemma 6.10, it suffices to prove that $e_1 \leftrightarrow_S^a e_2$ implies $e_1 \rho e_2$.

The proof will be by induction on the sum of the degrees of e_1 and e_2. Assume e_1 is in the graph a, and $e_1 \leftrightarrow_S^a e_2$. By Lemma 6.12, either $\{e_1, e_2\}$ is in EQS^a, or all of the following:

e_1 is of the form $f(x_1, x_2, \ldots x_j)$ for some $j > 0$. (6.9)

e_2 is of the form $f(y_1, y_2, \ldots y_j)$ (6.10)

for all i in $\{1 \ldots j\}$, $x_i \equiv_S^a y_i$ (6.11)

hold.

If $\{e_1, e_2\}$ is in EQS^a, then either $\{e_1, e_2\}$ is in EQS^b or $\{e_1, e_2\} = \{xx, yy\}$. If $\{e_1, e_2\}$ is in EQS^b, then by Invariant 6.22, both e_1 and e_2 are in graph b, and by Invariant 6.20, $e_1 \equiv_E^b e_2$. By Lemma 6.34, e_2 is in graph a and $e_1 \equiv_E^a e_2$.

Now suppose $\{e_1, e_2\} = \{xx, yy\}$. We know by line 4 of EqAssert and Theorems 6.33 and 6.32 that $nx \in recl^b(xx)$ and $ny \in recl^b(yy)$, so by Lemma 6.34, $nx \in recl^a(xx)$ and $ny \in recl^a(yy)$. Therefore e_2 is in graph a. Since $\{nx, ny\}$ is in $ToDo^a$, by Theorem 6.15, $nx \equiv_N^a ny$. Then, by Theorem 6.17, $recl^a(xx) = recl^a(yy)$, and therefore $xx \equiv_E^a yy$.

Finally, consider the case in which (6.9), (6.10), and (6.11) hold. Because e_1 is in graph a, $recl^a(e_1)$ must be nonempty. There must be an eligible node p with label f and operands $m_1, m_2, \ldots m_j$ such that $m_i \in recl^a(x_i)$. Since each $x_i \equiv^a_S y_i$, by induction, $x_i \equiv^a_E y_i$. Thus, $recl^a(x_i) = recl^a(y_i)$, so that $m_i \in recl^a(y_i)$, and $p \in recl^a(e_2)$. We have established that p is in both $recl^a(e_1)$ and $recl^a(e_2)$. By Theorem 6.17, $recl^a(e_1) = recl^a(e_2)$, and therefore $e_1 \equiv^a_E e_2$. □

Lemma 6.38. For two expressions e_1 and e_2, if $recl^b(e_1) \cap recl^b(e_2)$ is nonempty, then $e_1 \equiv^a_S e_2$.

Proof. By Theorem 6.17, $recl^b(e_1)$ and $recl^b(e_2)$ are equivalence classes and are therefore equal. Then, by Definition 6.19, $e_1 \equiv^b_E e_2$. Since $recl^b(e_1)$ and $recl^b(e_2)$ are both nonempty, by Invariant 6.20, $e_1 \equiv^b_S e_2$, and by Lemma 6.34, $e_1 \equiv^a_S e_2$. □

Lemma 6.39. Given two nodes n_1 and n_2 such that $n_1 \equiv^a_N n_2$, there are expressions e_1 and e_2 such that $n_1 \in recl^b(e_1)$, $n_2 \in recl^b(e_2)$, and $e_1 \equiv^a_S e_2$.

Proof. By Invariant 6.21, there must be an expression d_1 such that $n_1 \in recl^b(d_1)$. Since $n_1 \equiv^a_N n_2$, $n_1 \ {}_h\!\equiv^a_N n_2$ for some $h \geq 0$. We will now see, by induction on h, that $n_1 \ {}_h\!\equiv^a_N n_2$ implies the existence of an appropriate e_1 and e_2. If $h = 0$, ${}_h\!\equiv^a_N$ is the identity relation, so that $n_1 = n_2$. It suffices to choose $e_1 = e_2 = d_1$.

For the induction step, assume $h > 0$. If m_1 and m_2 are nodes, define the relation $m_1 \ \rho \ m_2$ to mean the existence of an expression x_1 such that $m_1 \in recl^b(x_1)$ implies the existence of an expression x_2 such that $m_2 \in recl^b(x_2)$ and $x_1 \equiv^a_S x_2$. If we could just show that $m_1 \ {}_h\!\equiv^a_N m_2$ implies $m_1 \ \rho \ m_2$, we would be done, because we have $n_1 \ {}_h\!\equiv^a_N n_2$ and $n_1 \in recl^b(d_1)$.

But notice that ρ is both reflexive and transitive. Therefore, by

Lemma 6.10, it suffices to show that $m_1 \ {}_h\!\overset{a}{\leftrightarrow}_N\ m_2$ implies $m_1\ \rho\ m_2$.

Suppose that $m_1 \ {}_h\!\overset{a}{\leftrightarrow}_N\ m_2$, and there is an expression x_1 such that $m_1 \in recl^b(x_1)$. Then, Definition 6.14 admits three possibilities:

(1) $m_1 \equiv^a_R m_2$

(2) $\{m_1, m_2\}$ is in ToDo

(3) m_1 and m_2 are eligible nodes with identical labels and equivalent operands.

In each case, we will see how to find expressions x_2 and x_3 such that $m_1 \in recl^b(x_3)$, $m_2 \in recl^b(x_2)$, and $x_3 \equiv^a_S x_2$. Then, since m_1 is in both $recl^b(x_1)$ and $recl^b(x_3)$, by Lemma 6.38, $x_1 \equiv^a_S x_3$. Then $x_1 \equiv^a_S x_3 \equiv^a_S x_2$.

In case (1), choose $x_2 = x_3 = x_1$. Then trivially $x_3 \equiv^a_S x_2$, and $m_1 \in recl^b(x_3)$. Since the reprlinks in graph a are the same as the reprlinks in graph b, $m_1 \equiv^b_R m_2$, and by Theorem 6.15, $m_1 \equiv^b_N m_2$. Since by Theorem 6.17, $recl^b(x_3)$ is an equivalence class of $\equiv^b_N$, so that we have $m_2 \in recl^b(x_3)$. Since $x_3 = x_2$, $m_2 \in recl^b(x_2)$.

Next consider case (2). Either $\{m_1, m_2\}$ is $\{nx, ny\}$, the pair placed in ToDo at line 5 of EqAssert, or $\{m_1, m_2\}$ is some pair in $ToDo^b$. In the first subcase, line 4 of EqAssert and Theorems 6.33 and 6.32 guarantee that $nx \in recl^b(xx)$ and $ny \in recl^b(yy)$. Line 6 of EqAssert guarantees that $\{xx, yy\} \in EQS^a$, so that $xx \equiv^a_S yy$. If $m_1 = nx$ and $m_2 = ny$, choose x_2 to be yy and x_3 to be xx; if $m_1 = ny$ and $m_2 = nx$, choose x_2 to be xx and x_3 to be yy. In either case, $m_1 \in recl^b(x_3)$, $m_2 \in recl^b(x_2)$, and $x_3 \equiv^a_S x_1$.

In the second subcase, $\{m_1, m_2\}$ is a pair on $ToDo^b$. Then, by Invariant 6.21, there must be expressions x_3 and x_2 such that $m_1 \in recl^b(x_3)$ and $m_2 \in recl^b(x_2)$. But since $\{m_1, m_2\}$ is on $ToDo^b$, $m_1 \equiv^b_N m_2$, and therefore $x_2 \equiv^b_E x_2$. Then, by Invariant 6.20, we have

$x_2 \equiv_S^b x_2$, and by Theorem 6.34, we have $x_2 \equiv_S^a x_2$. (Actually, this subcase holds vacuously, because ToDo must be empty before Critical Region II is entered. But because of the modifications made to Algorithm Eq in Chapter 7, it is important to prove Critical Region II correct without the assumption that ToDo is empty.)

Finally consider case (3). Then m_1 and m_2 are eligible nodes with the same label f, such that for some $k \geq 0$, m_1 has operands $p_1, p_2, \ldots p_k$ and m_2 has operands $q_1, q_2, \ldots q_k$, where for each i, $p_i \; {}_{j_i}\!\equiv_N^b q_i$ for some $j_i < h$. Then by induction, there are expressions $y_1, y_2 \ldots y_k$ and $z_1, z_2, \ldots z_k$ such that for each i in $\{1 \ldots k\}$, $p_i \in \mathrm{recl}^b(y_i)$, $q_i \in \mathrm{recl}^b(z_i)$, and $y_i \equiv_S^a z_i$. Let $x_2 = f(z_1, z_2, \ldots z_k)$ and $x_3 = f(y_1, y_2, \ldots y_k)$. By Lemma 6.11, $x_2 \equiv_S^a x_3$. By Definition 6.16, $m_1 \in \mathrm{recl}^b(x_3)$ and $m_2 \in \mathrm{recl}^b(x_2)$. □

Lemma 6.40. For expressions e_1 and e_2, if $\mathrm{recl}^b(e_1) \cap \mathrm{recl}^a(e_2)$ is nonempty, then $e_1 \equiv_S^a e_2$.

Proof. The proof is by induction on the degree of e_2. Let n be a node that $\mathrm{recl}^b(e_1)$ and $\mathrm{recl}^a(e_2)$ have in common. The expression e_2 will be of the form $f(x_1, x_2, \ldots x_k)$ for some $k \geq 0$. Because $n \in \mathrm{recl}^a(e_2)$, by Definition 6.16, there must be an eligible node p with label f and operands $m_1, m_2, \ldots m_k$ such that each $m_i \in \mathrm{recl}^a(x_i)$, and $p \equiv_N^a n$. Let $y_1, y_2, \ldots y_k$ be any expressions such that $m_i \in \mathrm{recl}^b(y_i)$; by Invariant 6.21 such expressions must exist. Then, letting $e_3 = f(y_1, y_2, \ldots y_k)$, we have, by Definition 6.16, $p \in \mathrm{recl}^b(e_3)$. Since $\mathrm{recl}^a(x_i)$ and $\mathrm{recl}^b(y_i)$ both contain m_i, by induction, $x_i \equiv_S^a y_i$, and by Lemma 6.11, $e_2 \equiv_S^a e_3$. Since $p \equiv_N^a n$, by Lemma 6.39, there are expressions e_4 and e_5 such that $p \in \mathrm{recl}^b(e_4)$, $n \in \mathrm{recl}^b(e_5)$, and $e_4 \equiv_S^a e_5$. Because p is in both $\mathrm{recl}^b(e_3)$ and $\mathrm{recl}^b(e_4)$, by Lemma 6.38, $e_3 \equiv_S^a e_4$. Likewise, since n is in both

$recl^b(e_5)$ and $recl^b(e_1)$, $e_5 \equiv_S^a e_1$. Therefore we have:

$e_2 \equiv_S^a e_3 \equiv_S^a e_4 \equiv_S^a e_5 \equiv_S^a e_1$. □

Theorem 6.41. If e_1 and e_2 are in graph a and $e_1 \equiv_E^a e_2$, then $e_1 \equiv_S^a e_2$.

Proof. Assume e_1 and e_2 are in graph a, and $e_1 \equiv_E^a e_2$. Then $recl^a(e_1) = recl^a(e_2) \neq \emptyset$. Let n be any node in $recl^a(e_1)$. By Invariant 6.21, there is an expression e_3 such that $n \in recl^b(e_3)$. Since n is in both $recl^a(e_1)$ and $recl^b(e_3)$, by Lemma 6.40, $e_1 \equiv_S^a e_3$. By the same reasoning, $e_3 \equiv_S^a e_2$. Therefore $e_1 \equiv_S^a e_3 \equiv_S^a e_2$. □

Theorem 6.42. Invariant 6.20 holds at the end of Critical Region II.

Proof. We must show that if e_1 and e_2 are in graph a, then $e_1 \equiv_E^a e_2$ if and only if $e_1 \equiv_S^a e_2$. Assume e_1 and e_2 are in graph a. If $e_1 \equiv_E^a e_2$, by Theorem 6.41, $e_1 \equiv_S^a e_2$. If $e_1 \equiv_S^a e_2$, by Theorem 6.37, $e_1 \equiv_E^a e_2$. □

6.13 Critical Region III

The change Critical Region III makes to the graph can be broken down into a series of actions. When describing an action, the superscript "b" will refer to the state of the graph before the action, and "a" to the state of the graph after the action. Here are the four actions performed in Critical Region III:

(1) A pair (x, y) is removed from ToDo. If necessary, a reprlink is added so that $x \equiv_R^a y$ (lines 9-15).

(2) An operand link pointing to y_1 is changed to point to x_1, where $x_1 \equiv_R^a y_1$, and there are nodes x_2 and y_2 such that $x_1 \equiv_R^b x_2$, $y_1 \equiv_R^y x_2$, and (x_2, y_2) is on $ToDo^b$ (line 19).

(3) A node z is moved from one bucket to another (lines 22 and 28).

(4) A node z is made ineligible, and a pair (d, z) is placed on ToDo, where d is eligible, and d and z have identical labels and operand links (lines 22 and 27).

Theorem 6.43. Actions (1-4), can each be implemented by a sequence one or more of the following actions:

(5) Add (x, y) to ToDo or add a reprlink from x to y, where $x \equiv_N^b y$.

(6) Remove (x, y) from ToDo, or delete a reprlink from x to y, where $x \equiv_N^a y$.

(7) Change an operand link pointing to x to point to y, where $x \equiv_N^b y$ and $x \equiv_N^a y$.

(8) Set or clear the eligibility bit of a node x, if there is another eligible node with identical label and operand fields.

(9) Change the bucket that a node x is in.

Proof. Action (1) is implemented by using action (5) to add a reprlink from x to y, if necessary to make $x \equiv_N^a y$, and then using action (6) to delete the pair (x, y) from ToDo. Action (2) is implemented with action (7). Action (3) is implemented with action (9). Action (4) is implemented by first using action (8) to make z ineligible, and then using action (5) to add the pair to ToDo. □

For the remainder of this section, "b" will refer to the state of the graph before one of actions (5-9) is performed, and "a" will refer to the graph afterwards.

Lemma 6.44. For any two nodes n_1 and n_2, $n_1 \equiv_N^b n_2$ implies $n_1 \equiv_N^a n_2$.

Proof. We will prove, by induction on h, that $n_1 \ {}_h{\equiv}_N^b \ n_2$ implies $n_1 \equiv_N^a n_2$. The base case, in which h = 0, is trivial since ${}_0{\equiv}_N$ is the identity relation. For h > 0, by Lemma 6.10, it is sufficient to prove that $n_1 \; {}_{h}{\leftrightarrow}^{b}_{N} \; n_2$ implies $n_1 \equiv^{a}_{N} n_2$.

Suppose $n_1 \; {}_{h}{\leftrightarrow}^{b}_{N} \; n_2$. From Definition 6.14, there are three possibilities, the first of which is $n_1 \equiv^{b}_{R} n_2$. The only way that $n_1 \not\equiv^{a}_{R} n_2$ is by a reprlink from x to y being deleted by action (5). But since the reprlink is deleted only if $x \equiv^{a}_{N} y$, $n_1 \equiv^{a}_{N} n_2$. A similar argument is used for the second possibility, that $\{n_1, n_2\}$ is in ToDo^{b}. The final possibility is that n_1 and n_2 have identical labels and operand links $p_1, p_2, \ldots p_k$ and $q_1, q_2, \ldots q_k$ such that each $p_i \; {}_{j_i}{\equiv}^{b}_{N} \; q_i$, for some $j_i < h$. Then, by induction, $p_i \equiv^{a}_{N} q_i$. The operands of n_1 and n_2 may not be p_i and q_i in graph a; they could be changed by operation (7). But an operand link is changed from m_1 to m_2 only if $m_1 \equiv^{a}_{N} m_2$, so that the corresponding operands of n_1 and n_2 will be equivalent in graph a, and $n_1 \equiv^{a}_{N} n_2$. □

<u>Lemma</u> 6.45. $n_1 \equiv^{b}_{N} n_2$ if and only if $n_1 \equiv^{a}_{N} n_2$.

<u>Proof</u>. Since, by Lemma 6.44, $n_1 \equiv^{b}_{N} n_2$ implies $n_1 \equiv^{a}_{N} n_2$, it is only necessary to prove $n_1 \equiv^{a}_{N} n_2$ implies $n_1 \equiv^{h}_{N} n_2$. That is, we must show that if $n_1 \equiv_{N} n_2$ after one of operations (5-9) is performed, then $n_1 \equiv_{N} n_2$ beforehand. But since the inverse of every operation in (5-9) is among (5-9), this second implication is proved by Lemma 6.44 as well. □

<u>Lemma</u> 6.46. For any node n, if $n \in \text{recl}^{b}(e)$, then $n \in \text{recl}^{a}(e)$.

<u>Proof</u>. The proof is by induction on the degree of e. The expression e will be of the form $f(x_1, x_2, \ldots x_k)$ for some $k \geq 0$. Assuming $n \in \text{recl}^{b}(e)$, there is a node p that is eligible in graph b, with label f and operands $m_1, m_2, \ldots m_k$ such that each $m_i \in \text{recl}^{b}(x_i)$, and $p \equiv^{b}_{N} n$. By Lemma 6.45, $p \equiv^{a}_{N} b$. By induction, $m_i \in \text{recl}^{a}(x_i)$. As the result of action (7), an operand of p in graph a may no longer be m_i. But if q_i is the new operand, $q_i \equiv^{a}_{N} m_i$, so that $q_i \in \text{recl}^{a}(x_i)$. As the result of

action (7), node p may no longer be eligible. But there will be an eligible node d with label and operand fields identical with those of p. By Theorem 6.15, d $\equiv_N^a$ p. In either case, n $\in$ $recl^a(e)$. □

Lemma 6.47. For any expression e, $recl^b(e) = recl^a(e)$.

Proof. By Lemma 6.46, if n $\in$ $recl^b(e)$, then n $\in$ $recl^a(e)$. Therefore we need only prove that if n $\in$ $recl^a(e)$, then n $\in$ $recl^b(e)$. Since the inverse of every operation in (5-9) is among (5-9), Lemma 6.46 proves this implication as well. □

Theorem 6.48. Critical Region III leaves $\equiv_N$, recl, $\equiv_E$, and $\equiv_S$ invariant.

Proof. The change made to the graph by Critical Region III can be performed by a series of actions (5-9), each of which, by Lemmas 6.45 and 6.47, leaves $\equiv_N$ and recl invariant. Therefore Critical Region III leaves $\equiv_N$ and recl invariant. Since $\equiv_E$ is defined in terms of recl, Critical Region III leaves $\equiv_E$ invariant. Since Critical Region III does not change EQS, it leaves $\equiv_S$ invariant.

Corollary. Invariants 6.20, 6.22, and 6.21 hold at the end of Critical Region III.

6.14 The Correctness of EqQuery

All of the invariants have now been established. It is now easy to establish the correctness routine EqQuery, and hence, of Algorithm Eq.

Theorem 6.49. EqQuery computes $\equiv_S$.

Proof. By Theorems 6.33 and 6.32, after lines 4-5 of EqQuery have been executed expressions xx and yy will both be in the graph, nx will be a representative node in recl(xx), and ny will be a representative node in recl(yy). By Theorem 6.9 and Invariant 6.24, nx and ny will be the only representative nodes in recl(xx) and recl(yy), so that recl(xx) = recl(yy) if and only if nx = ny. By Invariant 6.20, xx $\equiv_E$ yy if and only if xx $\equiv_S$ y. Therefore:

xx $\equiv_S$ yy iff xx $\equiv_E$ yy iff recl(xx) = recl(yy) iff nx = ny. □

6.15 The Running Time of Algorithm Eq

Theorem 6.50. If N is the total size (the number of function symbols) of all the expressions in a sequence of Assert and Query instructions, that sequence can be processed by algorithm Eq in expected time O(N log N).

Proof. I make the usual assumptions about hashing: that the number of buckets is large with respect to the number of nodes, and that the hash function distributes the nodes evenly among the buckets, so that no bucket gets too full [55]. In this analysis, I will assume the searches on line 8 of EqInsert and line 24 of EqAssert are performed in O(1) time.

Another simplifying assumption made is that there is an upper bound K on the number of operands that any function can have. However, this assumption is not very restrictive. If functions with an unbounded number of operands are desired, one needs only to reserve two function symbols "cons" and "nil." Then the expression $f(x_1, x_2, \ldots x_k)$ can be encoded as $f(\text{cons}(x_1, \text{cons}(x_2, \ldots \text{cons}(x_k, \text{nil}()) \ldots)))$. Since this encoding increases the size of the input only by a linear factor, it does not effect the asymptotic running time of the algorithm.

Only O(N) time is spent outside EqInsert and EqAssert. To prove the O(N log N) running time, a scheme is presented for charging computation to function symbols in the input and nodes in the graph. We will see that the time required to perform EqInsert(e) can be accounted for by charging

O(log N) time to each of the symbols in e. The time required to perform EqAssert(xx, yy) is accounted for by charging O(log N) to each of the symbols in xx and yy, and charging "overtime" to some nodes in the graph. Throughout all of the calls to EqAssert, each node of the graph will absorb O(log N) overtime. Since every node in the graph can be associated with a unique function symbol from the input, there can be at most N nodes in the graph. Therefore the amount of overtime is O(N log N).

The loop on line 11 of EqInsert can require at most log N iterations. By Invariant 6.3, each iteration of the loop at least doubles n.reprcount, and since $1 \le$ n.reprcount $\le N$, n.reprcount can be doubled at most log N times. Similarly, the loops on lines 10 and 11 of EqAssert can require at most log N iterations.

EqInsert(e) charges O(log N) time to each of the function symbols in e. Let e be of the form $f(x_1, x_2, \ldots, x_k)$ for some $k \ge 0$. EqInsert first recursively charges O(log N) time to each of the functions symbols in x_i. Then it performs lines 6-21, which require O(log N) time, charging it to f.

Next, consider routine EqAssert(xx, yy). As we have just seen, the calls to EqInsert(xx) and EqInsert(yy) charge O(log N) to the symbols in xx and yy. The O(log N) time spent in lines 10-11 on the first iteration of EqLoop can be charged any symbol in xx or yy.

All of the time spent in ConLoop is overtime, and is charged to the nodes. For each node, separate subaccounts are kept for each operand field. ConLoop requires O(1) time for each element on the container list it is searching, and the time is charged to each operand field updated at line 19.

When an operand field is updated on line 19, it is changed from y to x, its parent in the reprlink forest. Since 1 ≤ w.reprcount ≤ N, w.reprcount can be doubled at most log N times. By Invariant 6.3, the modification at least doubles w.reprcount. Therefore, throughout the entire computation, the subaccount for each is charged O(log N). Since each node has at most K operands, each node is only charged O(log N) overtime.

The O(log N) time spent on lines 10-11 of EqAssert in the second and subsequent iterations of EqLoop is charged to y. The node y must have been the node z on line 27 of a previous iteration. The node z is changed from eligible to ineligible on line 21. Since there is no place in the algorithm where an ineligible node is made eligible again, node z remains ineligible. Therefore the node cannot be put back onto ToDo by subsequent executions of line 27, and cannot be charged O(log N) time again. □

The running time of Algorithm Eq can be improved using a technique called "path compression." Each loop of the form:

<u>while</u> x.reprlink ≠ <u>nil</u> <u>do</u> x := x.reprlink <u>od</u>

which traces a branch of a tree to its root, is replaced by the following:

```
( z := x
; while x.reprlink ≠ nil do x := x.reprlink od
; while z ≠ x do t := z.reprlink; z.reprlink := x; z := t od
) .
```

After the root of the tree is found, the reprlinks of the nodes passed through to get to the root are updated to point directly to the root.

Path compression preserves the correctness of Algorithm Eq, because it does not invalidate Invariants 6.2 or 6.3, and it does not change the relation $\equiv_R$.

Although path compression improves the asymptotic running time of the UNION-FIND algorithm [3], it does not improve the asymptotic running time of Algorithm Eq, because it does not reduce the length of the container lists. As explained in section 7.6, the Proof Checker does not use path compression.

CHAPTER 7

THE V-CODE INTERPRETER

7.1 V-code Instructions

The V-code instructions introduced in Chapters 3 and 4, and the actions specified by them, are:

PROCLAIM P	Add P to the Assertion Table.
REQUIRE P	Report an error if P cannot be deduced via automatic rules from formulas in the Assertion Table.
NEW v	Replace every occurrence of the variable v in the Assertion Table by a new variable v' that does not appear in the Assertion Table or in subsequent V-code instructions.
OPEN	Push a copy of the Assertion Table onto a stack.
CLOSE	Replace the assertion table by the top of the stack, and pop the stack.
CLOSE-NEW	Perform a CLOSE instruction as above, and re-execute every NEW instruction that occurred between the CLOSE-NEW instruction and its matching OPEN instruction.

There are other V-code instructions (VGOTO, for example), but from the point of view of the V-code Interpreter, each of these other instructions is just an abbreviation for a sequence of the above six instructions. The descriptions given above are intended as specifications, and are not descriptions of the implementation.

The implementation of the V-code interpreter will be described incrementally. In sections 7.2 and 7.3, an interpreter will be described that only executes PROCLAIM and REQUIRE instructions. Then in section 7.4, the NEW instruction is added. Finally, in sections 7.5 and 7.6 the imple-

$$\frac{}{\text{'1'B}} \qquad \frac{\text{'0'B}}{P} \qquad \frac{P \quad \sim P}{\text{'0'B}} \qquad (1,\ 2,\ 3)$$

$$\frac{P \quad Q}{P\&Q} \qquad \frac{P\&Q}{P} \qquad \frac{P\&Q}{Q} \qquad (4,\ 5,\ 6)$$

$$\frac{P}{P|Q} \qquad \frac{Q}{P|Q} \qquad (7,\ 8)$$

$$\frac{P}{\sim P{=}{>}Q} \qquad \frac{\sim P}{P{=}{>}Q} \qquad \frac{P \quad P{=}{>}Q}{Q} \qquad (9,\ 10,\ 11)$$

$$\frac{P{=}{>}Q \quad Q{=}{>}P}{P{<}{=}{>}Q} \qquad \frac{P{<}{=}{>}Q}{P{=}{>}Q} \qquad \frac{P{<}{=}{>}Q}{Q{=}{>}P} \qquad (12,\ 13,\ 14)$$

$$\frac{}{P=P} \qquad \frac{P=Q \quad R[P//x]}{R[Q//x]} \qquad \frac{P=Q \quad R[Q//x]}{R[P//x]} \qquad (15,\ 16,\ 17)$$

$$\frac{}{P{<}{=}{>}P} \qquad \frac{P{<}{=}{>}Q \quad R[P//x]}{R[Q//x]} \qquad \frac{P{<}{=}{>}Q \quad R[Q//x]}{R[P//x]} \qquad (18,\ 19,\ 20)$$

Figure 7.1 The PL/CV Automatic Rules

mentation of the OPEN, CLOSE, and CLOSE-NEW instructions is discussed.

7.2 The PROCLAIM and REQUIRE instructions

Were it not for the automatic rules, PROCLAIM and REQUIRE could be implemented using a simple table lookup scheme. The automatic rules consist of the rules shown in Figure 7.1 and the Transformation Axioms shown in Figures 5.1 and 5.2.

The algorithm used to implement PROCLAIM and REQUIRE will be obtained by modifying algorithm Eq, which was presented in Chapter 6.

Let us review Algorithm Eq. Given a set EQS of pairs of formulas and two formulas P and Q, the relation P $\equiv_S$ Q is defined to hold whenever it is possible to transform P to Q using a sequence of substitutions of one element of a pair in EQS for the other element of that pair. Algorithm Eq maintains a graph structure in which each node represents a collection of formulas. A formula is said to be "in the graph" whenever it is represented by some node in the graph. The relation $\equiv_E$ is defined over the set of formulas that are in the graph; P $\equiv_E$ Q is defined to hold whenever P and Q are represented by the same node. When a new pair is added to EQS, the graph is updated so that $\equiv_E$ and $\equiv_S$ are the same relation (when the domain of $\equiv_S$ is restricted to the set of formulas in the graph.) The set EQS is not used in the computation; it is an auxiliary variable used to argue the correctness of the algorithm.

A node with label ρ and operands $n_1, n_2, \ldots n_k$ represents all formulas or expressions of the form $\rho(x_1, x_2, \ldots x_k)$, where for all i in $\{1 \ldots k\}$, n_i represents x_i. An equivalence relation $\equiv_N$ is defined over the set of nodes. A class C of $\equiv_N$ is said to represent a formula P whenever there is a node n in C such that n represents P.

Another equivalence relation $\equiv_R$, which is also defined over the set of nodes is determined by the reprlinks of the nodes. The relation $\equiv_R$ is always a refinement of the relation $\equiv_N$. In every equivalence class of $\equiv_R$, there is exactly one node whose reprlink is <u>nil</u>; this node is called the "representative" of the class. A representative node r represents not only the formulas or expressions defined above, but also all formulas or expressions P for which there is a node n such that n represents P, and n $\equiv_N$ r. Each operand field of every node points to a representative node.

The V-code interpreter will consider a formula P to be "proved" whenever P $\equiv_E$ '1'B. When a PROCLAIM P instruction is processed, the for-

```
EqnAssert(nx, ny):
begin
          Add (nx, ny) to ToDo
          ¢ Add {xx, yy} to EQS ¢
    ;     EqLoop:
          while ToDo is not empty
          do    Remove an element (x, y) from ToDo
          ;     while x.reprlink ≠ nil do x := x.reprlink od
          ;     while y.reprlink ≠ nil do y := y.reprlink od
          ;     if   x ≠ y
                then if y.reprcount > x.reprcount
                     then swap x and y fi
                ;    y.reprlink := x
                ;    x.reprcount := x.reprcount + y.reprcount
                ;    ConLoop:
                     for every w on y.containers
                     do   w.operand := x
                     ;    z := a pointer to the node containing w.operand
                          if z is eligible
                          then Remove z from the bucket it is in
                          ;    h := hash(z.label, z.operands)
                          ;    Search bucket[h] for a node with
                                  the same label and operands as z
                          ;    if   the search finds a duplicate d
                               then Add (d, z) to ToDo
                               else Put z in bucket[h]
                               fi
                          ;    <Insert Figure 7.4 here>
                          fi
                     od
                ;    Append y.containers to x.containers
                ;    <Insert Figure 7.3 here>
                fi
          od
end
```

Figure 7.2 Routine EqnAssert

mula P is inserted into the graph, the pair {P, '1'B} is added to EQS, and the graph is updated to keep the relations $\equiv_E$ and $\equiv_S$ consistent.

In this section, we will see how to modify Algorithm Eq so that as it discovers new formulas to be equivalent to '1'B, it applies proof rules to these formulas, and adds new pairs of formulas to EQS. A pair {P, Q} is added to EQS only if the formula P = Q or P <=> Q can be deduced from formulas that have already been proved.

Before any V-code instructions are processed, the formulas '1'B and '0'B are inserted into the graph using the statements:

```
( TrueNode  := EqInsert⟦'1'B⟧
; FalseNode := EqInsert⟦'0'B⟧
) .
```

(The double brackets are used to emphasize that formulas, not bits, are being passed to EqInsert.) Algorithm Eq will be modified so that the two variables TrueNode and FalseNode are updated as necessary so that they will always contain the pointers to nodes representing the formulas '1'B and '0'B. The instruction PROCLAIM P is then implemented by:

```
EqnAssert(EqInsert(P), TrueNode) .
```

The routine EqnAssert is obtained by modifying the routine EqAssert, which appears in Figure 6.10. EqAssert is first modified so that it takes pointers to nodes as arguments instead of formulas; the result of this modification is shown in Figure 7.2. Then the statements in Figures 7.3 and 7.4 are added so that deductions are performed.

Some new fields are added to each node so that all the members of each equivalence class of the relation $\equiv_R$ can be put on a linked list called the "reprlist." The head and tail of the reprlist containing all the elements of a class are stored in the representative node for that class. It is necessary that both the head and tail of the reprlist be stored in the node, so that reprlists can be concatenated in constant time.

A bit field "established" is also added. If n is a node, n.established will be true whenever TrueNode is the representative of n.

Finally, a bit field "equiv" is added. This field is only significant when the label field of a node n is "=>". If m_1 and m_2 are the operands of the node n, and n.equiv is true then $m_1 \equiv_N m_2$. The converse of this statement is not true; n.equiv can be false when $m_1 \equiv_N m_2$. This field is not necessary to obtain a correct algorithm; its purpose is to reduce the running time of the algorithm.

When a node is created by EqInsert, it is put into a new equivalence class of $\equiv_R$. Therefore, its reprlist consists of just itself, and its established bit is set to false. When a node is created whose label is "=>", its equiv field is set to true if its operands are identical, and false otherwise.

Figures 7.3 and 7.4 contain the statements that must be added to EqAssert. If n is a node, the notation E(n) is used in Figures 7.2-4 to denote some expression such that n represents E(n). By Invariant 5.21, an appropriate E(n) must exist for each node n.

Here is a summary of what the statements in Figure 7.3 do. In lines (01-04) it is determined whether one of the two classes being merged is the class representing '1'B, and if so, a pointer to the representative of the other class is stored in the variable "merged." Lines (05-06) are used to maintain the invariant that TrueNode and FalseNode are pointers to the nodes representing the formulas '1'B and '0'B. If one of the two

```
(  merged := nil
;  if   x = TrueNode then merged := y
   elif y = TrueNode then merged := x
   fi
;  if y = TrueNode  then TrueNode  := x fi
;  if y = FalseNode then FalseNode := x fi
;  if merged ≠ nil
   then for every node z on merged.reprlist
        do  z.established := true
            let z.operands be (n1, n2, ... nk)
        ;   if   z.label = "&"
            then Add (n1, TrueNode) to ToDo
                 ¢ Add {E(n1), '1'B} to EQS ¢
            ;    Add (n2, TrueNode) to ToDo
                 ¢ Add {E(n2), '1'B} to EQS ¢
            elif z.label = "~"
            then Add (n1, FalseNode) to ToDo
                 ¢ Add {E(n1), '0'B} to EQS ¢
            elif z.label = "=" or z.label = "<=>"
            then Add (n1, n2) to ToDo
                 ¢ Add {E(n1), E(n2)} to ToDo ¢
            elif z.label = "=>"
            then h := hash("=>", (n2, n1))
            ;    Search bucket [h] for a node
                     with label "=>" and operands (n2, n1)
            ;    if such a node m is found
                 then if m.established
                      then Add (n1, n2) to ToDo
                           ¢ Add {E(n1), E(n2)} to EQS ¢
                      ;    m.equiv := true ; z.equiv := true
                      fi
                 fi
            fi
        od
   fi
; Append y.reprlist to x.reprlist
)
```

Figure 7.3 An Addition to Routine EqnAssert

```
{ let z.operands be (n1, n2)
; if    z.label = "=>" and z.established and ~z.equiv
  then  h := hash("=>", (n2, n1))
  ;     Search bucket [h] for a node
           with label "=>" and operands (n2, n1)
  ;     if the search finds a node r
        then if   r.established
             then Add (n1, n2) to ToDo
                  ¢ Add {E(n1), E(n2)} to EQS ¢
                  z.equiv := true
                  r.equiv := true
             fi
        fi
  fi
}
```

Figure 7.4 An addition to EqnAssert

classes being merged represented 'λ'B before the merge, the reprlist of the other class is traversed by lines (08-34). The "established" bit of each node in the class being merged is set to true. Deductions then are made on the basis of formulas that are newly discovered to be valid. After the reprlist of x or y has been traversed if necessary, the reprlist of y is appended to the reprlist of x by line (36).

When an operand field of a node z is changed, the new node to which the field points represents a larger class of formulas or expressions, and hence the node z also represents a larger class of formulas or expressions. If z is in the class of nodes that represents the formula 'i'B, some new formulas have been established. The statements in Figure 7.4 are used to perform deductions based on these newly established formulas.

All of the changes made to the graph by the statements in Figures 7.3 and 7.4 are of the form:

(1) add a pair of formulas $\{P_1, P_2\}$ to EQS, where P_1 and P_2 are both in the graph, and

(2) if n_1 and n_2 are nodes representing P_1 and P_2, add (n_1, n_2) to ToDo.

The modification made to the graph is the same kind of modification performed in Critical Region II of EqAssert. Because the proof in section 6.12 does not depend on the fact that ToDo is empty, the same proof can be used to show that the statements in Figures 7.3 and 7.4 maintain all the necessary invariants discussed in Chapter 6.

It is, however, necessary to convince ourselves that the pairs of formulas being added to EQS are in fact equivalent. When lines (13-15) of Figure 7.3 are executed, the formula P & Q, where P is $E(n_1)$ and Q is $E(n_2)$, is represented by node z. Since z is being merged into the class represented by TrueNode, we know that P & Q $\equiv_E$ '1'B. That is, the formula P & Q has been proved. Therefore the formulas P and Q are both valid, and the pairs {P, '1'B} and {Q, '1'B} may be added to EQS.

In a similar fashion, P $\equiv_E$ '0'B is being deduced from ~P in lines (17-18), P $\equiv_E$ Q is being deduced from P = Q or P <=> Q in lines (20-21). In lines (23-33), since a valid formula of the form P => Q has been discovered, the graph is searched for a node representing the formula Q => P. If it is discovered that TrueNode represents Q => P, then P $\equiv_E$ Q is deduced as a consequence.

The statements in Figure 7.4 are executed whenever an the operand of a node z is changed, and the class of formulas represented by z is thereby expanded. If the node z represents a formula of the form P => Q, and z $\equiv_R$ TrueNode, a search is performed for a node r representing P => Q. If r $\equiv_R$ TrueNode as well, then P $\equiv_E$ Q is deduced as a consequence.

The modifications made to Algorithm Eq do not increase its expected asymptotic running time. The extra time required by the statements in Figure 7.3 to traverse the reprlist and add a pair to ToDo can be charged to the nodes on the reprlist. Each extra pair put onto ToDo as part of the processing of an element z of a reprlist will require O(log N) time to process when the pair is taken off ToDo. The O(log N) time will be charged to the node z. Because a node can be merged into the class representing '1'B only once, only O(log N) extra time is charged to each node for the statements in Figure 7.3.

The pair added to ToDo in by the statements in Figure 7.4 will also require O(log N) time to process; this time is charged to the node z. The pair is put onto ToDo only if z.equiv is _false_, and then z.equiv is set to _true_, and left unchanged thereafter. Therefore, this extra O(log N) time can be charged to each node at most once.

Definition 7.1. We will say that a formula P is "established" whenever P is in the graph and P $\equiv_E$ '1'B.

Theorem 7.2. Whenever ToDo is empty, the graph will have the following closure properties:

If P & Q is established then P $\equiv_E$ '1'B and Q $\equiv_E$ '1'B. (7.1)

If ~P is established then P $\equiv_E$ '0'B. (7.2)

If P = Q is established, then P $\equiv_E$ Q. (7.3)

If P <=> Q is established, then P $\equiv_E$ Q. (7.4)

If P => Q and Q => P are established, then P $\equiv_E$ Q. (7.5)

Proof. Let us first establish that the graph has property (7.1). Suppose P & Q is in the graph and P & Q $\equiv_E$ '1'B. Then there must be an eligible node z whose label is "&" and whose operands are two nodes x_1 and x_2 representing P and Q. Likewise, there must be an eligible node t with no operands whose label is "'1'B", and it must be the case that z $\equiv_N$ t.

By Invariant 6.21, if ToDo is empty, then $\equiv_N$ and $\equiv_R$ are the same relation. Therefore, $z \equiv_R t \equiv_R$ TrueNode.

Therefore, at some point during the computation, the equivalence class of $\equiv_R$ containing z must have been merged with the equivalence class representing '1'B. The superscript "m" will be used to denote the state the graph was in just after this merge was performed. Let y_1 and y_2 be the operands z had in graph m. Lines (12) and (15) in Figure 7.4 guarantee that $y_1 \equiv_N^m$ TrueNode^m and $y_2 \equiv_N^m$ TrueNode^m.

Because the operands of z could have been changed by line (17) of Figure 7.2 at some time after the merge was performed, y_1 and y_2 may not be the same nodes as x_1 and x_2. And since the variable TrueNode could have been changed by line (05) of Figure 7.3, TrueNode^m may not be the same node as TrueNode. But since:

(1) an operand is changed from p to q only if $p \equiv_N q$,

(2) TrueNode is changed from p to q only if $p \equiv_N q$, and

(3) the only changes made to $\equiv_N$ make the relation coarser

we can conclude that $x_1 \equiv_N y_1$, $x_2 \equiv_N y_2$, and TrueNode$^m \equiv_N$ TrueNode. Therefore, by transitivity, $x_1 \equiv_N$ TrueNode, and $x_2 \equiv_N$ TrueNode. If follows from the definition of $\equiv_E$, that $P \equiv_E$ '1'B and $Q \equiv_E$ '1'B.

Similar arguments can be used to establish closure properties (7.2-4). But property (7.5) is more subtle. Suppose P => Q and Q => P are both established. Then there is an eligible node y with label "=>" and operands (x_1, x_2), where x_1 represents P and x_2 represents Q. Likewise there is an eligible node z with label "=>" and operands (x_2, x_1). Furthermore, $y \equiv_R z \equiv_R$ TrueNode.

At some point in the computation, the following two events must have occurred:

(1) The the class of $\equiv_R$ containing the node y was merged into the class representing 'i'B.

(2) The the class of $\equiv_R$ containing the node z was merged into the class representing 'l'B.

Further, if at the time the node y or z (whichever was created second) was created, the operands of y were not the operands of z in reverse order, the following event must have taken place:

(3) An operand field of y or z is updated as to make the operands of y the same as the operands of z in reverse order.

Let m be the state of the graph just after the last of the above three events occurred. Let (w_1, w_2) be the operands y had in graph m, and (w_2, w_1) be the operands z had in graph m. Either lines (28-29) of Figure 7.3 or lines (08-09) of Figure 7.4 (depending on whether the last of the three events was (3)) must have been executed to establish $w_1 \equiv_N^m w_2$. By the same reasoning used above, $x_1 \equiv_N w_1 \equiv_N w_2 \equiv_N x_2$, and therefore $P \equiv_E Q$.

One final subtlety should be mentioned. When the loop in lines (08-34) of Figure 7.3 is being executed, it is possible for there to be on the reprlist being traversed a node z with label "=>" and operands (n_1, n_2), and for there to be a node m with label "=>" and operands (n_2, n_1) such that m $\equiv_R$ TrueNode, but to have the node m be missed by lines (27-31). This situation can arise when m also appears on the reprlist, and m follows z. When z is being processed, m.established will not yet have been set to <u>true</u>. However, when m is processed by the loop, z.established will have been set to <u>true</u>, and (n_1, n_2) will be added to ToDo. □

Suppose as part of the initialization of the V-code Interpreter we insert the formula 'l'B & 'l'B into the graph, and assert that it is equivalent to 'l'B as follows:

EqnAssert(EqInsert['l'B & 'l'B], TrueNode) .

Then, if the formulas P and Q are established, we have P $\equiv_E$ 'l'B and Q $\equiv_E$ 'l'B, so that P & Q $\equiv_E$ ('l'B & 'l'B) $\equiv_E$ 'l'B, and P & Q is established. Rule (4) can be implemented in this fashion.

Similarly, rule (7) could be implemented by inserting all formulas of the form 'l'B | P in the graph, and asserting them equivalent to 'l'B. But to use this approach, it would be necessary to insert infinitely many nodes into the graph.

Even if we could initialize the graph to contain infinitely many nodes, it would be foolish to do so, since the algorithm could only inspect finitely many of them. Suppose the hashed search operation that appears at lines (07-09) of EqInsert, lines (21-23) of Figure 7.2, lines (23-25) of Figure 7.3, and lines (03-05) of Figure 7.4 were modified so that when it searched for a node whose label is "|" and first operand is TrueNode, and did not find one, it would create such a node, set its reprlink to TrueNode, and set its "established" bit to <u>true</u>. The effect would be the same as if the infinite initialization were performed.

But it is not necessary for the hashed search to actually create a new node. When the search is performed in EqInsert and in Figure 7.2, only the representative of the located node is used. And when the search is performed in Figures 7.3 and 7.4, only the established bit of the located node is used. Therefore, when searching for a node n, there is no harm in returning some node m, where n $\equiv_R$ m. The hashed search can be modified so that when searching for a node whose label is "|" and whose first operand is TrueNode, TrueNode can simply be returned.

The final modification to be made to Algorithm Eq is to build the following equivalences:

~'1'B	$\equiv_E$	'0'B	(7.6)
'1'B & '1'B	$\equiv_E$	'1'B	(7.7)
'1'B | P	$\equiv_E$	'1'B	(7.8)
P | '1'B	$\equiv_E$	'1'B	(7.9)
'1'B => Q	$\equiv_E$	Q	(7.10)
'0'B => Q	$\equiv_E$	'1'B	(7.11)
P => P	$\equiv_E$	'1'B	(7.12)
P = P	$\equiv_E$	'1'B	(7.13)
P <=> P	$\equiv_E$	'1'B	(7.14)

into the hashed search. FalseNode is returned when searching for a node whose label is "~" and whose operand is TrueNode. TrueNode is returned when searching for a node whose label is "&" and both of whose operands are TrueNode. and so on. If. for example, a node z whose label is "|" has an operand changed to be TrueNode by line (17) of Figure 7.2, the modified search operation at lines (22-23) must "find" a simulated duplicate. The node z is made ineligble, and (z, TrueNode) is put on ToDo. Similar sorts of subterfuge must be performed by search operations in Figures 7.3 and 7.4.

Once the equivalences (7.6-14) have been built into the hashed search. the distinction between formulas that are in the graph and not in the graph becomes somewhat fuzzy. So that we do not have to worry about the domain of $\equiv_E$, let us redefine the relation slightly. If P and Q are formulas, define P $\equiv_E$ Q to mean that after executing:

n_1 := EqInsert(P) ; n_2 := EqInsert(Q)

n_1 and n_2 would have the same value. Then $\equiv_E$ and $\equiv_S$ are identical relations. Define a formula P to be "established" whenever P $\equiv_E$ TrueNode.

Because the equivalences (7.6-14) were not established by EqnAssert, to preserve the correctness of Theorem 7.2, it is necessary that the set of formulas that are established by the simulation performed by the hashed search obeys closure properties (7.1-5). But these properties are easy to verify by inspection.

<u>Theorem</u> 7.3. If ToDo is empty, then for each of the automatic rules in Figure 7.1 except rule (2), if all the hypotheses of the rule are established, then the conclusion of the rule must be established as well.

<u>Proof</u>. For each rule, the conclusion follows from simple applications of closure properties (7.1-5), equivalences (7.6-14), and the following substitution property: if P[R//x] $\equiv_E$ Q[R//x]. and R $\equiv_E$ S or S $\equiv_E$ R, then P[S//x] $\equiv_E$ Q[S//x]. The substitution property follows from the fact that $\equiv_E$ and $\equiv_S$ are identical relations. The numbering of the following cases is consistent with the numbering of Figure 7.1.

(1) Trivially '1'B $\equiv_E$ '1'B.

(3) Since ~P $\equiv_E$ '1'B, by closure property (7.2), P $\equiv_E$ '0'B. By hypothesis, P $\equiv_E$ '1'B, so that '0'B $\equiv_E$ '1'B.

(4) If P $\equiv_E$ '1'B, and Q $\equiv_E$ '1'B, then by substitution,
P & Q $\equiv_E$ '1'B & '1'B and by equivalence (7.7), '1'B & '1'B $\equiv_E$ '1'B.
Therefore P & Q $\equiv_E$ '1'B.

(5) If P & Q $\equiv_E$ '1'B, then by closure property (7.1), P $\equiv_E$ '1'B.

(6) If P & Q $\equiv_E$ '1'B, then by closure property (7.1), Q $\equiv_E$ '1'B.

(7) If P $\equiv_E$ '1'B, then by substitution, (P | Q) $\equiv_E$ ('1'B | Q). By equivalence (7.8), ('1'B | Q) $\equiv_E$ '1'B.

(8) If Q $\equiv_E$ '1'B, then by substitution, (P | Q) $\equiv_E$ (P | '1'B). By

equivalence (7.9), (P | '1'B) $\equiv_E$ '1'B.

(9) If P $\equiv_E$ '1'B, then ~P $\equiv_E$ ~'1'B, and by equivalence (7.6), ~'1'B $\equiv_E$ '0'B. Therefore ~P => Q $\equiv_E$ '0'B => Q, and by equivalence (7.11) '0'B => Q $\equiv_E$ '1'B.

(10) If ~P $\equiv_E$ '1'B, then by closure property (7.2), P $\equiv_E$ '0'B. Then by substitution, P => Q $\equiv_E$ '0'B => Q, and by by equivalence (7.11), '0'B => Q $\equiv_E$ '1'B.

(11) If P $\equiv_E$ '1'B and (P => Q) $\equiv_E$ '1'B, then by substitution, (P => Q) $\equiv_E$ ('1'B => Q). By equivalence (7.10), '1'B => Q $\equiv_E$ Q. Therefore Q $\equiv_E$ ('1'B => Q) $\equiv_E$ (P => Q) $\equiv_E$ '1'B.

(12) If P => Q $\equiv_E$ '1'B and Q => P $\equiv_E$ '1'B, then by closure property (7.5), P $\equiv_E$ Q. By equivalence (7.14), P <=> P $\equiv_E$ '1'B. Substituting Q for P, we obtain P <=> Q $\equiv_E$ '1'B.

(13) If P <=> Q $\equiv_E$ '1'B, then P $\equiv_E$ Q by closure property (7.4). We have P => P $\equiv_E$ '1'B by equivalence (7.12). Substituting Q for P, we obtain P => Q $\equiv_E$ '1'B.

(14) If P <=> Q $\equiv_E$ '1'B, then P $\equiv_E$ Q by closure property (7.4). We have P => P $\equiv_E$ '1'B by equivalence (7.12). Substituting Q for P, we obtain Q => P $\equiv_E$ '1'B.

(15) P = P $\equiv_E$ '1'B by equivalence (7.13).

(16) If P = Q $\equiv_E$ '1'B, then by closure property (7.3), P $\equiv_E$ Q. If R[P//x] $\equiv_E$ '1'B, then R[Q//x] $\equiv_E$ '1'B by substitution.

(17) If P = Q $\equiv_E$ '1'B, then by closure property (7.3), P $\equiv_E$ Q. If R[Q//x] $\equiv_E$ '1'B, then R[P//x] $\equiv_E$ '1'B by substitution.

(18) P <=> P $\equiv_E$ '1'B by equivalence (7.14).

(19) If P <=> Q $\equiv_E$ '1'B, then by closure property (7.4), P $\equiv_E$ Q. If R[P//x] $\equiv_E$ '1'B, then R[Q//x] $\equiv_E$ '1'B by substitution.

(20) If $P \Leftrightarrow Q \equiv_E$ '1'B, then by closure property (7.4), $P \equiv_E Q$. If $R[Q//x] \equiv_E$ '1'B, then $R[P//x] \equiv_E$ '1'B by substitution. □

Suppose that for each i in {1 ... n} the instruction:

PROCLAIM P_i

has been executed, so that $P_i \equiv_E$ '1'B. Also suppose that is possible to derive the formula Q from the formulas P_1, P_2, ... P_n using the rules in Figure 7.1, except for rule (2). Then by repeated applications of Theorem 7.3, we obtain $Q \equiv_E$ '1'B.

Now suppose that rule Q can be derived from P_1, P_2, ... P_n using all the rules in Figure 7.1. It is easy to see that if a proof uses rule (2), it must contain a subproof of '0'B that does not use rule (2), by induction on the size of the proof. Therefore, by Theorem 7.3, '0'B $\equiv_E$ '1'B, so that TrueNode = FalseNode.

The instruction REQUIRE P is therefore implemented by:

```
if   TrueNode ≠ FalseNode
then if   EqInsert(P) ≠ TrueNode
     then Report an error
     fi
fi .
```

7.3 Transformations

The equivalences (7.6-7.14) that are built into the hashed search are each instances of one of the Transformation Axioms in Figures 5.1 and 5.2. However, not all the Transformation Axioms can be built into the hashed search. For example, consider the Associativity Axiom:

$$(a + b) + c = a + (b + c) \quad .$$

It would be nice if the hashed search could be modified to return the same node for (a + b) + c as for a + (b + c). But suppose the hash function is given the label "+" and pointers to two operands. The nodes to which the operand links point each represent a class of expressions, and there is no simple way to tell from that representative node if one of those classes contains an expression whose main connective is +.

Krafft [57] cites a number of theoretical results that indicate that an efficient algorithm that can deduce all consequences of the Transformation Axioms and the rules in Figure 7.1 is infeasible. The problem is with the substitution rules (16, 17, 19, 20). If the first hypotheses of these rules are restricted to come from some finite set, Algorithm Eq can be used to find all consequences of the substitution rules. If hypotheses are restricted to be Transformation Axioms, the normalization function described in Chapter 5 can be used to find all consequences of the substitution rules. The problem of efficiently finding all consequences of the Transformation Axioms plus some finite set has not been solved.

A less ambitious goal is to obtain all consequences of the Transformation Axioms and the rules in Figure 7.1, where the substitution rules are restricted to use Transformation Axioms as their first hypothesis. Deductions in this class will be referred to as being "substitution-free." The current implementation of the Proof Checker almost achieves this goal.

Here is how transformations are incorporated into the V-code interpreter. A hash table called the Normalization Table is maintained that maps formulas onto nodes in the graph. As the routine EqInsert adds a formula P to the graph, it normalizes all the subformulas and subexpressions of P and adds these normal forms to the Normalization Table.

The routine EqInsert and the routine that computes normal forms have a similar structure. Given a formula of the form $\rho(x_1, x_2, \ldots x_k)$, EqInsert produces a representative node by recursively finding representatives for each x_i, and then combining these representatives with ρ to find or create a node that represents the entire formula. Likewise, the normalization function are computed by first finding normal forms for each x_i, and then combining ρ with the normal forms to obtain a normal form for the entire formula.

The two computations are performed in tandem, so that a representative node r and normal form n are computed for each subformula and subexpression of P. As each pair (r, n) is obtained, the normal form n is looked up in the Normalization Table. If n is not present, it is added to the table, so that n is mapped to r. The other possibility is that n already appears in the table. If the table maps n to r', then EqnAssert(r, r') is called.

As a result of these extra calls to EqnAssert, the set of formulas in the graph has the following closure property:

> if P and Q are subformulas or subexpressions of formulas appearing in PROCLAIM or REQUIRE instructions, and $P \equiv_T Q$, then $P \equiv_E Q$. (7.15)

The relation $\equiv_T$ was defined in Chapter 6; $P \equiv_T Q$ is defined to hold whenever Q can be obtained by applying a series of transformations to P. Note that for such a deduction to be performed automatically by the V-Code Interpreter, it is not necessary that the intermediate steps between P and Q be subformulas or subexpressions of formulas in the proof, but both ends of the chain must be.

Let us consider the form that substitution-free deductions must take. The proof rules in Figure 7.1, except for rule (2), are divided into three classes: "introduction," "elimination," and "substitution" rules. Rules (16, 17, 19, 20) are the substitution rules. The introduction rules are the ones with the property that every hypothesis is a subformula of the conclusion (note that rules that have no hypotheses are vacuously introduction rules). The elimination rules are the ones with the property that conclusion is a subformula of one of the hypotheses. Rules (10) and (12) are classified as introduction rules, and rules (13) and (14) are classified as elimination rules even though they do not, strictly speaking, have the necessary subformula properties.

If a formula P appears in a substitution-free deduction, define P to be a "maximum" formula whenever:

(1) P is the conclusion of an introduction rule and the major hypothesis of an elimination rule, or

(2) P is the conclusion of an introduction rule, and the major hypothesis of a substitution rule, or

(3) P is the conclusion of a substitution rule and the major hypothesis of an elimination rule.

If a rule has only one hypothesis, that hypothesis is the major hypothesis. If a rule has two hypotheses, the major hypothesis is the second one shown in Figure 7.1. A minor hypothesis is one that is not a major hypothesis.

A substitution-free deduction of P is defined to be in "normal form" whenever:

(1) The proof contains no maximum formulas, and

(2) Rule (2) is used at most once, at the root of the proof tree to conclude P.

This definition of normal form is similar to the definition given by Prawitz [65] for Natural Deduction proofs. The difference between the two definitions is that there are no substitution rules in Natural Deduction.

If substitution rules are used in a proof that is in normal form, they must be used as part of a chain of reasoning that looks like this:

$$P \xrightarrow{\text{elimination rules}} Q \xrightarrow{\text{substitution rules}} R \xrightarrow{\text{introduction rules}} S$$

where P is a leaf in the proof tree, and S is either the root of the proof tree, or the minor hypothesis of rule (3) or (11). Because Q is obtained from P by a chain of elimination rules, Q is a subformula of P. Likewise, since S is obtained from R by a sequence of introduction rules, R is a subformula of S. (The technical problem of rules (10, 12, 13, 14), which are not strictly speaking introduction and elimination rules. is being ignored here.) If S is a minor hypothesis of rule (3) or (11), then the major hypothesis, either S => T or ~S, is on a chain of eliminations that begins at a leaf L. In that case, S must be a subformula of L.

Therefore, if a proof tree is in normal form, both ends of any chain of substitutions are subformulas of a leaf or the root of the proof tree. It follows from property (7.15) that if the instruction PROCLAIM P_i has been executed for each i in {1 ... k}, and there is a substitution-free deduction of Q from P_1, P_2, ... P_k that is in normal form, then the instruction REQUIRE Q will succeed.

Therefore we are interested in determining whether every substitution-free deduction can be put into normal form. If a proof tree uses Rule (2), it is a simple matter to transform it so that it uses Rule (2) only once to conclude the root. The difficult part of converting a

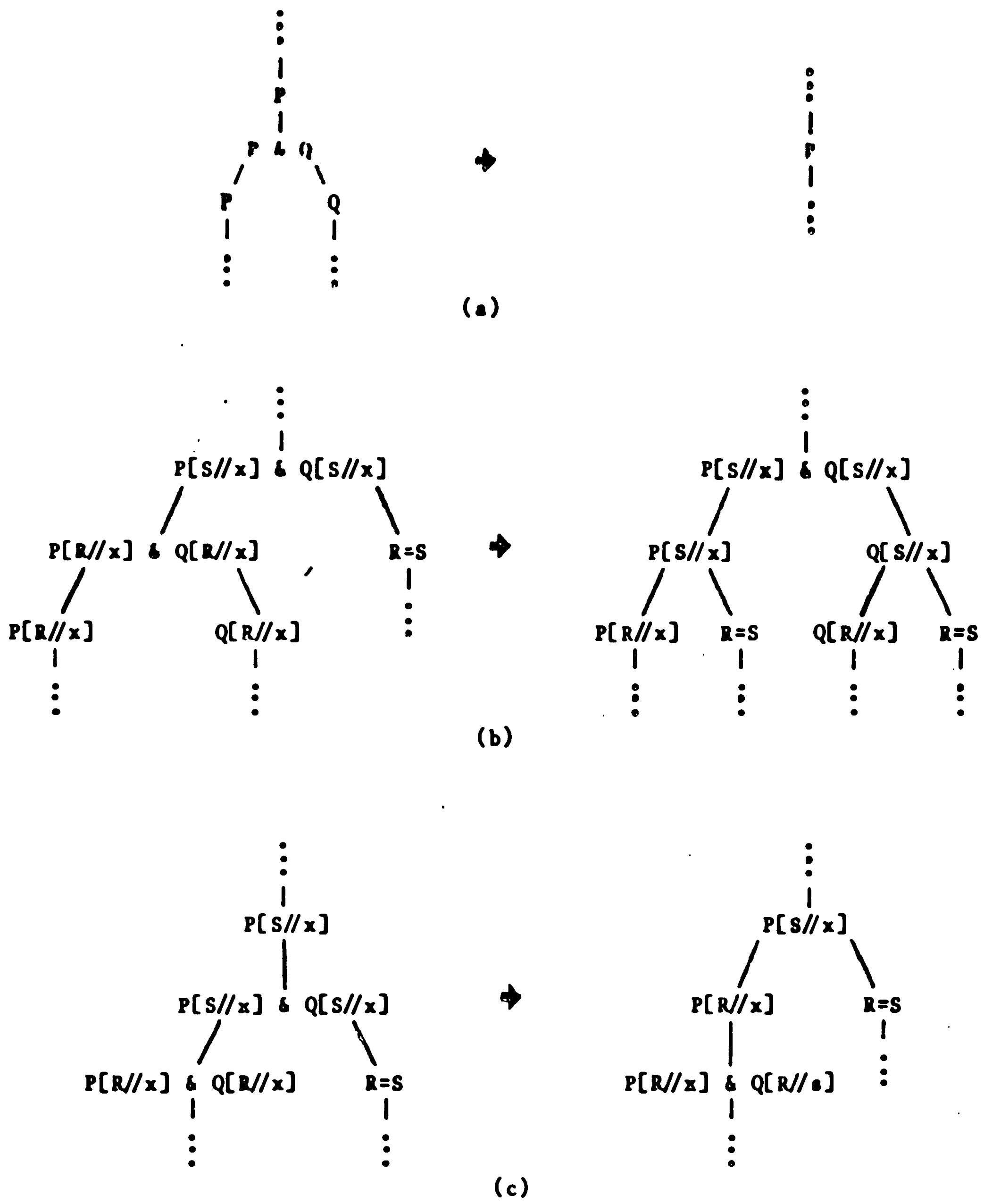


Figure 7.5 The elimination of maximum formulas

proof tree to normal form is the elimination of maximum formulas. However, most maximum formulas can be eliminated from proof trees. Some examples of how this is done are shown in Figure 7.5. In parts (a), (b),

```
          P | (Q | R)
           /       \
(P | Q) | R     (P | Q) | R <=> P | (Q | R)
     |
  (P | Q)
```

Figure 7.6 A proof tree that cannot be put into normal form

and (c) of the figure, maximum formulas of types (1), (2), and (3) are being eliminated.

Unfortunately, not every proof tree can be put into normal form; Figure 7.6 contains one such tree. Although the V-code Interpreter can automatically deduce the formula:

(P | Q) | R (7.16)

from the formula:

P | Q (7.17)

and it can deduce the formula:

P | (Q | R) (7.18)

from formula (7.16), it cannot deduce formula (7.18) from formula (7.17) in a single step. The intermediate step (7.16) must be written in the proof.

More work needs to be done on the problem of cleanly incorporating transformations into the V-code Interpreter. The preceding example would seem to indicate that not enough automatic deductions are being performed. Yet in another sense, too many deductions are being performed. We have just seen that V-code Interpreter cannot in general deduce formula (7.18) from formula (7.17) in a single step. However, in the presence of the completely vacuous formula:

'0'B => (P | Q) | R

formula (7.18) can be deduced from formula (7.17). All that is necessary for the deduction to be made is that (7.16) be a subformula of some formula in the proof that precedes formula (7.18). A fairly high price is paid for these spurious deductions. For each formula P of size N that appears on a PROCLAIM or REQUIRE instruction, a collection of normal forms whose sizes total $O(N^2)$ is put into the Normalization Table.

Although it is unreasonable to expect the V-code Interpreter to automatically deduce all consequences of substitution and transformation rules, it would be nice if it were possible for the two forms of reasoning to be automatically combined in some limited fashion. A common approach used in PL/CV proofs to show that a formula P holds after the assignment statement v := e is to prove that the formula Q holds before the assignment statement, where Q is obtained by normalizing P[e//v]. This combination of normalization and substitution is exactly the type of automatic reasoning the V-code Interpreter cannot perform.

Another problem with transformations is that they can only be used to prove equivalences. To prove that the assignment statement X := X + 1 leaves the assertion X > 0 invariant, it is necessary to prove that X + 1 > 0 follows from X > 0. This is a simple deduction, but it cannot be proved using normalization, because the two formulas are not equivalent. One possible method for attacking this problem is to look for a partial ordering ≤ defined over normal forms that has the property that N⟦P⟧ ≤ N⟦Q⟧ implies P => Q.

7.4 The NEW instruction

A straightforward method of implementing the NEW instruction would involve modifying the data structures discussed in sections 7.2 and 7.3. This approach is not used in the V-code Interpreter.

Suppose we have implementations of two operations: Assert(P), which adds P to the Assertion Table, and Query(P), which determines whether P can be deduced via automatic rules from formulas in the Assertion Table. The algorithm for processing PROCLAIM and REQUIRE instructions discussed in sections 7.2 and 7.3 is one possible implementation of Assert and Query. For the sake of simplicity, we will assume in this section that Assert is implemented as described above, and Query is implemented by consulting an oracle. Given any implementation of Assert and Query, there is a simple technique by which a V-code interpreter that will process PROCLAIM, REQUIRE, and NEW instructions can be constructed.

Definition 7.4. If S is a set of formulas, and P is a formula, define the relation $S \vdash P$ to be the result of the operation Query(P) when the Assertion Table contains S.

The substitution notation is extended to sets of formulas and V-code instructions in the obvious fashion.

Definition 7.5. If S is a set of formulas, define $S[v'//v]$ to be $\{P[v'//v] \mid P \in S\}$. If I is a V-code instruction, define $I[v'//v]$ to be:

(1) PROCLAIM $P[v'//v]$ if I is PROCLAIM P,

(2) REQUIRE $P[v'//v]$ if I is REQUIRE P,

(3) NEW w if I is NEW w and w is distinct from v, and

(4) NEW v' if I is NEW v.

The automatic rules do not depend on names of variables. That is, if v' is a variable that does not occur in S or P, then $S \vdash P$ if and only if $S[v'//v] \vdash P[v'//v]$. This fact can be used to prove the following theorem.

<u>Theorem</u> 7.6. Suppose the execution of a sequence of V-code instructions is interrupted at some point. Let S_0 be the contents of the Assertion Table, and $I_1, I_2, \ldots I_n$ be the remaining V-code instructions. Let v' be a variable that does not occur in any of the formulas in S or in any of the V-code instructions $I_1, I_2, \ldots I_n$. If the Assertion Table is replaced by $S_0[v'//v]$, and for each j in $\{1 \ldots n\}$, I_j is replaced by $I_j[v'//v]$, the output (that is, the sequence of successes and failures of REQUIRE instructions) of this modified V-code sequence will be the same as the output of the original sequence.

<u>Proof</u>. For j in $\{1 \ldots n\}$, let S_j be the contents of the Assertion Table after the execution of I_j, and let T_j be the contents of the assertion table after the execution of $I_j[v'//v]$. Let T_0 be $S_0[v'//v]$. Clearly the success or failure of REQUIRE instructions does not depend on the names of the variables chosen by NEW instructions to substitute into the Assertion Table, as long as each variable chosen does not appear in the Assertion Table or in any subsequent V-code instruction. Therefore, assume without loss of generality that if I_j and $I_j[v'//v]$ are NEW instructions the two instructions choose the same variable to substitute into the Assertion Table, and neither v nor v' is chosen.

For all j in $\{0 \ldots n\}$, $T_j = S_j[v'//v]$ can be established using induction on j. The base case, that $T_0 = S_0[v'//v]$, holds by definition of T_0. For the induction step, suppose $T_{j-1} = S_{j-1}[v'//v]$. There are three cases, depending on the what kind of instruction I_j is.

If I_j is PROCLAIM P, then $S_j = S_{j-1} \cup \{P\}$ and $T_j = T_{j-1} \cup \{P[v'//v]\}$. We therefore have:

$$\begin{aligned} T_j &= T_{j-1} \cup \{P[v'//v]\} \\ &= S_{j-1}[v'//v] \cup \{P[v'//v]\} \\ &= (S_{j-1} \cup \{P\})[v'//v] \\ &= S_j[v'//v] \quad . \end{aligned}$$

If I_j is REQUIRE P, then $S_j = S_{j-1}$ and $T_j = T_{j-1}$. Therefore $T_j = T_{j-1} = S_{j-1}[v'//v] = S_j[v'//v]$.

If I_j is NEW w, and w and v are distinct variables, then $S_j = S_{j-1}[w'//w]$ and $T_j = T_{j-1}[w'//w]$. Since $v \neq w$, $v' \neq w$, and $w' \neq v$, for any P, $P[v'//v][w'//w] = P[w'//w][v'//v]$. Therefore:

$$\begin{aligned} T_j &= T_{j-1}[w'//w] \\ &= S_{j-1}[v'//v][w'//w] \\ &= S_{j-1}[w'//w][v'//v] \\ &= S_j[v'//v] \quad . \end{aligned}$$

The final case is that I_j is a NEW v instruction. Then $I_j[v'//v]$ is the instruction NEW v', so that $S_j = S_{j-1}[v''//v]$ and $T_j = T_{j-1}[v''//v']$. If the formula P contains no occurrences of v', then:

$$P[v'//v][v''//v'] = P[v''//v]$$

since the set of occurrences of v' in $P[v'//v]$ is exactly the set of occurrences introduced by substituting v' for v. Since the variable v' was chosen so that it did not appear in S_0 or any of the instructions $I_1, I_2, \ldots I_n$, and no NEW instructions selects v', there can be no occurrences of v' in S_{j-1}. Therefore we have:

$$
\begin{aligned}
T_j &= T_{j-1}[v''//v'] \\
&= S_{j-1}[v'//v][v''//v'] \\
&= S_{j-1}[v''//v] \\
&= S_j \quad .
\end{aligned}
$$

Since $S_j = S_{j-1}[v''//v]$, there can be no occurrences of v in S_j. Therefore $T_j = S_j = S_j[v'//v]$.

The induction is now complete. We have established that for each j in {0 ... n}, $T_j = S_j[v'//v]$. If the instruction I_j is REQUIRE P, it will succeed if and only if:

$$S_j \vdash P \quad . \tag{7.19}$$

But then the instruction $I_j[v'//v]$ is REQUIRE $P[v'//v]$, and it it will succeed if and only if:

$$S_j[v'//v] \vdash P[v'//v] \quad . \tag{7.20}$$

Since v' cannot occur in S_j or P, formulas (7.19) and (7.20) are equivalent. Therefore the two REQUIRE instructions must have the same result. □

The following strategy can be used to implement the NEW instruction. Suppose a sequence of V-code instructions has been partially executed, and the instruction NEW v is about to be executed. Interrupt the execution at this point, select some variable v' that does not appear in the Assertion Table or subsequent V-code, and replace every occurrence of v in the Assertion Table and subsequent V-code instructions by v'. By Theorem 7.6, this operation will not change the output of the subsequent REQUIRE instructions.

The above global replacement operation will change the NEW v instruction to NEW v'. This instruction selects a variable v" that does not appear anywhere in the Assertion Table or subsequent V-code and replaces every occurrence of v' in the Assertion Table with v". Because the global replacement has just gotten rid of every occurrence of v in the Assertion Table and V-code (and replaced each one with an occurrence of v'), the NEW instruction is free to choose v" = v. Then executing the NEW v' will put the Assertion Table back the way it was before the global replacement.

The net result of the global replacement and the NEW instruction is to leave the Assertion Table unchanged, and to replace every occurrence of v in the subsequent V-code with an occurrence of v'.

The NEW instruction will be implemented so that the unique names selected by each NEW v instruction will be of the form v$k, where k is an integer. The first NEW v instruction will use the name v$1, the second will used v$2, and so on.

For every sequence of V-code instructions containing NEW instructions there is an equivalent sequence of instructions that does not have any NEW instructions. To obtain this sequence, modify each PROCLAIM P or REQUIRE P instruction as follows. For each program variable v in P, if k is the number of NEW v instructiona that precedes the PROCLAIM or REQUIRE instruction replace every occurrence of v in P by v$k. After all the PROCLAIM and REQUIRE instructions are so modified, the NEW instructions are deleted.

The V-code Interpreter could operate in two passes. In the first pass, the NEW instructions would be eliminated as just described. The resulting sequence of PROCLAIM and REQUIRE instructions would be executed in the second pass.

However, the effect of these two passes can be combined as follows.

For each program variable v, an integer Instance[v] is maintained in a table known as the "Instance Table." Initially, Instance[v] is zero for every v. The instruction NEW v is implemented by:

```
Instance[v] := Instance[v] + 1  .
```

When building representations of formulas, the parser used by the Local Context Checker represents each program variable v by v$0. The function SubstInst(P) locates every program variable of the form v$j in P, and replaces it by v$k, where k = j + Instance[v]. The PROCLAIM P instruction is then implemented by:

```
Assert(SubstInst(P))
```

and REQUIRE P is implemented by:

```
Query(SubstInst(P))  .
```

In section 4.8 the V-code generated for the assignment statement v := t was presented as:

```
DOMAIN    t
NEW       v
PROCLAIM  v = (t[v'//v])
```

where v' is the variable selected by the NEW v instruction. Let us consider what the implementation of NEW that updates the V-code rather than the Assertion Table would do for the above V-code instructions. Suppose the NEW v generated for the assignment statement is the j'th NEW v instruction. At the time the above V-code is executed, the previous j-1 NEW v instructions will have changed it to:

```
DOMAIN    t[v$(j-1)//v]
NEW       v$(j-1)
PROCLAIM  v$(j-1) = (t[v'//v])
```

Just before the above NEW instruction is executed, the execution is interrupted, and every occurrence of v$(j-1) in the Assertion Table and in the subsequent V-code is changed to v$j:

```
DOMAIN    t[v$(j-1)//v]
NEW       v$j
PROCLAIM  v$j = (t[v'//v])  .
```

Then the NEW instruction is executed. The variable v$(j-1) is selected as v', so that substituting v' for every occurrence of v$j in the Assertion Table cancels out the previous substitution. Therefore, the effect of the PROCLAIM instruction should be to add the formula:

```
v$j = (t[v$(j-1)//v])                                    (7.21)
```

to the Assertion Table. Intuitively, what is going on here is that the instance of v on the left side of formula (7.21) matches instances of v that will appear in subsequent PROCLAIM instructions (the "new" value of v), whereas the instances of v on the right side of formula (7.21) match the instances of v that are currently in the Assertion Table (the "old" value of v).

When using the Instance Table, Instance[v] has the value j-1 before the NEW v is executed, and Instance[v] has the value j after the NEW v instruction is executed. Since the function SubstInst(P) replaces v$k by v$n, where n = k + Instance[v], the two V-code sequences:

```
DOMAIN    t
NEW       v
PROCLAIM  v$0 = t[v$(-1)//v]
```

and:

```
DOMAIN    t
PROCLAIM  v$1 = t[v$0//v]
NEW       v
```

each result in formula (7.21) being added to the Assertion Table. Because the parser substitutes v$0 for v, the second sequence is easier to generate, and therefore it is used. Similar techniques are used when generating the V-code for procedure calls.

7.5 The OPEN, CLOSE, and CLOSE-NEW Instructions

A straightforward implementation of OPEN and CLOSE would be for OPEN to push a copy of the data structure used to implement the other instructions onto a stack, and for CLOSE to restore the data structure from the top of the stack, and pop the stack. The components of the data structure that would have to be saved and restored are:

(1) the graph maintained by EqInsert and EqnAssert, which is a collection of nodes and a hash table used to locate those nodes,

(2) the variables TrueNode and FalseNode,

(3) the Normalization Table, and

(4) the Instance Table.

Saving and restoring the entire data structure would be an inefficient use of both time and space. The following strategy is used instead. When an OPEN instruction is executed, a special marker is pushed onto a "backup stack." Then as changes are made to the data structure, these changes are recorded in a "backup record", and this record is pushed onto the backup stack. When a CLOSE or CLOSE-NEW instruction is executed, all the backup records are popped off the backup stack until the marker is reached. As each record is popped off the stack, the indicated change is undone.

When an OPEN instruction is encountered, the value of the variable FalseNode is put into a backup record and pushed onto the backup stack. Subsequent modifications of this variable need not be recorded.

All the modifications that are made to the graph and the variable TrueNode by EqInsert and EqnAssert are collected into a number of "transactions." A transaction is a function f that is applied to the graph to yield a new graph. The backup record for the transaction f contains whatever information is necessary to compute the inverse of f.

An important design decision is how to define the transactions. There is a general approach that works no matter what algorithms are used used to update the graph. Just two transactions are used: "Create" and "Modify." The Create transaction is used to allocate a new node. To invert this transaction, it is necessary to save the address and size of the node. The Modify transaction is used to update a node. To invert this transaction, it is necessary to save the address of the node and a copy of the node before it was updated.

This technique was used in an early version of the V-code Interpreter, but it was abandoned because too much storage was needed for the backup stack. A better pair of transactions is "Create" and "Merge," where Create is the operation performed by Critical Region I in EqInsert, and Merge is the operation performed by lines (13-32) of Figure 7.2.

To invert a Create transaction, it is only necessary to have a pointer to the created node. It is important to realize that when the backup record is being popped off the stack, before the inverse of the transaction is applied, the graph will be in whatever state it was in just after the transaction being inverted was performed.

The Create transaction has three steps:

(1) A new node n is created with operand fields a[1], a[2], ... a[k].

(2) A bucket in the hash table is selected, and the node n is placed in that bucket.

(3) For each i in {1 ... k}, a pointer to the i'th operand field of n is put at the beginning of the container list of a[i].

To invert the transaction, the following steps are performed:

(1) For each i in {1 ... k}, delete the first element of the container list of a[i]. Note that there may be duplicates among the list a[1], a[2], ... a[k]. If a[j] appears in the list m times, m elements are deleted from the front of its container list.

(2) The node n is removed from the bucket it is in. Since the bucket lists are doubly linked, this operation is simple to perform.

(3) The node n is returned to free storage.

To invert a Merge, which is performed by lines (13-32) of Figure 7.2, it is it is necessary to have:

(1) a pointer to the node y,

(2) the value of TrueNode before the Merge transaction, and

(3) a list of addresses of the "eligible" and "equiv" fields that were changed by the transaction.

Given a pointer to y, a pointer to x may be found in y.reprlink. The reprlink and reprcount fields that were modified by the transaction can be restored by executing:

```
( y.reprlink := nil
; x.reprcount := x.reprcount - y.reprcount
) .
```

When a Merge is executed, y.containers is appended to x.containers, and y.reprlist is appended to x.reprlist. It is possible to implement these operations so that they can be easily inverted.

Before y.containers is appended to x.containers in a Merge transac-

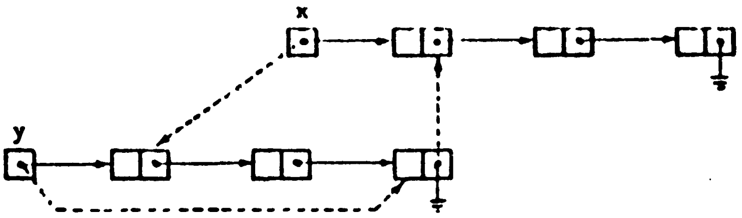


Figure 7.7 Appending container lists

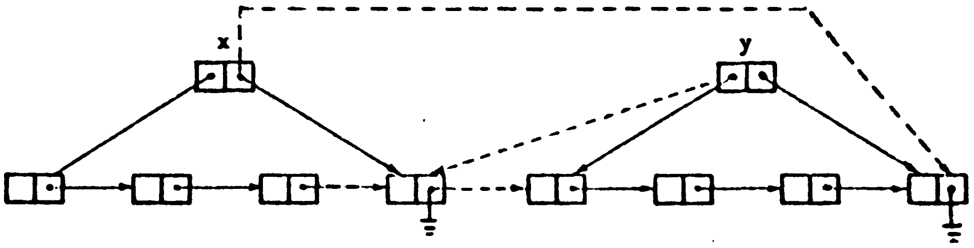


Figure 7.8 Appending reprlists

tion, the container list is traversed, so the last element in the list will be known. The append operation is performed by changing the pointers as shown in Figure 7.7; the dotted lines indicate the new values of pointer fields. The field of y that formerly contained a pointer to the head of its container list is changed to contain a pointer to the tail of that list. There is no harm in this, because y is no longer a representative, and its container list is no longer used. The updated fields of x and y contain sufficient information to locate and restore all the fields that need to be changed to invert the concatenation operation.

Both the head and the tail of the reprlist are stored in each representative node. The reprlist of y is appended to the reprlist of x by changing the pointers shown as in Figure 7.8. The head of the reprlist of y is replaced by the tail of the reprlist of x so that the concatenation operation can be inverted. As was the case with the container lists, there is no harm in changing this field because y is no longer a representative.

Once the reprlists of x and y have been restored, the variable TrueNode is restored from the backup record. If either x or y is TrueNode, the reprlist of the other list is traversed, and the "established" bits of the nodes on that list are set back to _false_.

Finally, the operation performed by ConLoop must be inverted. Every time ConLoop makes a node ineligible or sets its "equiv" bit to _true_, the address of the modified bit field is saved in the backup record for the transaction. When the transaction is inverted, these bit fields are restored. The reconstructed container list of y is then traversed. Every operand field to which there is a pointer on the reconstructed container list of y is changed from x back to y, and if the node containing that operand field is eligible, the hash function is applied to the node and the node is moved to the proper bucket.

The inversion of the Merge transaction is then complete.

When a new entry is added to the Normalization Table, it is also recorded in a backup record. When that backup record is popped off the stack, the corresponding entry in the Normalization Table is deleted.

The processing discussed so far is performed for both the CLOSE and the CLOSE-NEW instructions. The difference between the two instructions is their effect on the Instance Table. The CLOSE instruction must restore the Instance Table to the state it was in when the matching OPEN instruction was performed. That is, the intervening NEW instructions must be undone.

The specification for the CLOSE-NEW instruction is to undo all the NEW instructions performed since the matching OPEN, perform a CLOSE, and then perform the NEW instructions again. But there is no point in undoing the NEW instructions just to perform them again. But suppose, however, there is a matching pair of OPEN and CLOSE instructions nested between the OPEN and matching CLOSE-NEW instructions. Any NEW instructions between the OPEN and CLOSE will get undone when the CLOSE instruction is encountered, and must be performed again when the CLOSE-NEW instruction is encountered.

Recall that a "nest" is defined as a region of V-code delimited by matching OPEN and CLOSE(-NEW) instructions. Define a C-nest to be a nest that is terminated by a CLOSE instruction, and an N-nest to be a nest that it terminated by a CLOSE-NEW instruction. When an OPEN instruction is generated by the Local Context Checker it is marked to indicate whether it begins a C-nest or an N-nest. In the following discussion, the two different kinds of OPEN instructions will be referred to as C-OPEN and N-OPEN instructions. Note that implicit instructions inserted by incrementing the count field of an OPEN instruction are always N-OPEN instructions.

Two stacks of lists of variable names, the C-stack and the N-stack, are maintained. The empty list is pushed onto C-stack when a C-OPEN instruction is encountered, and the empty list is pushed onto the N-stack when an N-OPEN instruction is encountered. When a CLOSE or CLOSE-NEW instruction is encountered, the list on the top of C-stack or N-stack (respectively) is processed, and the stack is popped. Because the V-code generated for each routine begins with an OPEN instruction and ends with a CLOSE instruction, the C-stack is never empty when processing the V-code for a routine.

When a NEW v instruction is performed, it must be recorded so that it can be undone when the smallest C-nest containing the NEW instruction is closed. The variable v is added to the list on the top of the C-stack, and Instance[v] is incremented by one.

When a CLOSE instruction is performed, the NEW instructions recorded on the top of the C-stack must be undone; for each variable v on this list, Instance[v] is decreased by one. However, if the C-nest is contained in an N-nest, the NEW instructions that are being undone must be performed again when the smallest N-nest containing the C-nest is closed. Therefore if the N-stack is not empty, the list on top of the C-stack is appended to the list on top of the N-stack. In either event, the C-stack is popped.

When a CLOSE-NEW instruction is performed, the NEW instructions that have been accumulated on the top of the N-stack must be performed. For every variable v on the list on top of the N-stack, a NEW v instruction is performed. That is, Instance[v] is increased by one, and v is added to the list on top of the C-stack. The N-stack is then popped.

Because of the CLOSE-NEW instruction, the Local Context Checker is almost freed from the task of keeping track of which variables can be modified by compound statements. However, the LCC must still keep track of which variable can be modified in a loop body so that it can insert NEW instructions for those variables at the top of the loop body. If a new V-code instruction, NEW-OPEN, were introduced, the bookkeeping that is required of the LCC could be simplified. The NEW-OPEN instruction is defined to be the reverse of the CLOSE-NEW instruction. The effect of the NEW-OPEN instruction is to first perform a NEW v operation for every NEW instruction appearing in the nest begun by the NEW-OPEN, and then perform an OPEN operation.

If the NEW-OPEN instruction were added, the V-code interpreter would have to make two passes through the V-code. The extra pass would be required to compute the list of NEW operations that would have to be performed for each NEW-OPEN instruction. But the use of this instruction would simplify the LCC sufficiently to justify adding the extra pass.

7.6 The cost of OPEN, CLOSE and CLOSE-NEW Instructions

The amount of extra storage required to implement the OPEN, CLOSE, and CLOSE-NEW instructions is O(N + T + P + W), where N the number of nodes in the graph. T is the number of entries in the Normalization Table, and P is the number of OPEN instructions in the V-code, and W is the number of NEW instructions in the V-code.

One backup record for a Create transaction is put on tbe backup stack for each node in the graph. The stack will also contain one backup record for every formula stored in the Normalization Table, and one record for every OPEN statement for which the matching CLOSE(-NEW) has not been encountered. The backup record for a Merge transaction contains a pointer to a node that is made a nonrepresentative by the transaction, and to nodes for which the "eligible" bit is set to false or the "equiv" bit is set to true by the transaction. The only way a nonrepresentative node can become a representative, an ineligible node can become eligible, or the "equiv" bit can become false is by means of a CLOSE(-NEW) instruction. Therefore, at most O(N) space is used on the backup stack to record Merge transactions. The total size of all the lists on the N-stack and C-stack cannot exceed the number of NEW instructions in the V-code.

Let S be the state of the data structure when an OPEN instruction encountered, and T be the state of the data structure when the matching CLOSE(-NEW) instruction is encountered, and let N be the nest delimited by these two instructions. Only as much time is spent by a CLOSE instruction changing the data structure from state T to state S as was spent changing the data structure from state S to state T by the PROCLAIM, REQUIRE, and NEW instructions that are contained in the nest N but no smaller nest. The CLOSE-NEW instruction uses no more time than corresponding the CLOSE and NEW instructions would require, and usually uses less time.

If the amount of time spent on each PROCLAIM instruction were uniform, OPEN and CLOSE would only increase the running time of the V-code Interpreter by a linear factor. Unfortunately, the time spent on the PROCLAIM instructions is not uniform. In the absence of the OPEN and CLOSE instructions, the execution of any one PROCLAIM instruction I_0 can take a comparatively long time, but the the succeeding PROCLAIM instructions I_1, I_2, ... I_n will each be executed quickly to compensate for the time spent processing I_0. The sequence of instructions:

OPEN
I_0
CLOSE
OPEN
I_0
CLOSE
⋮

can force the V-code interpreter to execute the lengthy instruction I_0, then spend just as much time undoing the effect of the instruction, and repeat the process over and over without getting the diluting effect of the instructions I_1, I_2, ... I_n.

It is for this reason that path compression, a method for speeding up the UNION-FIND algorithm mentioned in section 6.15, is not used in the V-code Interpreter. If path compression were used, extra work would be done when processing some PROCLAIM and REQUIRE instructions to allow subsequent instructions to be executed faster. But the beneficial effect of this extra work would last only until a CLOSE(-NEW) instruction is executed, and then extra work would have to be performed to undo the path compression. Because the addition of path compression would increase complexity of the already complex backup mechanism, it is not used in the V-code Interpreter.

CHAPTER 8

SOME FINAL REMARKS

8.1 A Summary of the Major Results

The notion of formula graphs presented in Chapter 2 provides a simple model for quantified formulas that eliminates the problem of capture. The quantifier count representation for bound variables is an implementation of this model. Formulas represented in this fashion are easy to copy, instantiate, combine, and decompose.

The proof of equivalence of PL/CV predicate calculus and Gentzen's Natural Deduction in Chápter 3 assures us that the PL/CV predicate calculus rules are sound and complete. The tree structure attributed to PL/CV proofs is a useful notion in its own right. It provides a clear model that can be used to visualize the concept of accessibility, and forms the basis of Proof Checker.

The organization of the Proof Checker is independent of the PL/CS programming language. Different versions of the Proof Checker for different programming languages could all share the same V-code interpreter. Because all the automatic rules are built into the V-code interpreter, they can be changed without modifying the LCC.

Because the LCC is similar to a compiler, well-understood techniques could be used to construct it. It would be fairly simple to construct a version of the Proof Checker based on PASCAL, for example, by using a compiler compiler such as YACC [51] to build an LCC. (Unfortunately the predicate calculus rules, which would not change appreciably from one system to the next, have not been partitioned into their own module.) Theorem 4.15 could be used to derive the V-code that must be generated for various control constructs. The NEW instruction could be used to model various quirks in the language, such as different methods of parameter passing, in much the same way they are modeled by program verifiers based on symbolic execution [56].

The omnibus statement provides for multiple-exit loops of the type that can be written using Zahn's construct [53]. The statement has a relatively simple proof rule, but is flexible enough that it can be used for many different programming tasks. The methods of Hehner [40] and Manna and Waldinger [59] are well-suited for deriving programs written using the omnibus statement. One of the problems with the statement as it is presented in section 4.20 is that it does not have enough syntactic redundancy.

The algorithm presented in Chapter 5 is a formalization of the pencil-and-paper methods often used to apply such axioms as commutativity and distributivity. The algorithm makes use of a replacement system similar to that used by AFFIRM [60], but in the PL/CV system, tbe replacement rules are applied in a controlled fashion to ensure a quadratic running time. An advantage of the AFFIRM system is that additional replacement rules can added by the user, whereas the replacement rules used in the PL/CV system are built into the system. But PL/CV's the replacement rules are more general than those allowed by AFFIRM.

Algorithm Eq, the algorithm presented in Chapter 6 for deducing equalities, is similar to the Fast Congruence Closure Algorithm (FCCA) of Downey, Sethi, and Tarjan [30]. The proof of Algorithm Eq is much more complicated than the proof of FCCA, for two reasons:

(1) The input to FCCA is a single graph and a relation on the vertices of the graph, whereas Algorithm Eq processes a stream of instruc-

tions, each of which adds vertices to the graph or updates the relation.

(2) Downey, Sethi, and Tarjan's proof is more abstract. They prove that FCCA finds the congruence closure of a relation defined over the vertices of a graph. The proof in Chapter 6 makes explicit the relationship between the graph and the represented expressions.

The proof in Chapter 6 could serve as a guide to producing a variation on Algorithm Eq. When this modified algorithm determined the existence of a sequence of substitutions that transforms one expression to another, it would output one such sequence. Such an algorithm could be useful in an interactive program verification environment, such as the one being developed by Krafft [58] or in an LCF tactical [32], in which high-level proofs are monitored for correctness by a kernel.

Algorithm Eq may be used in applications other than program verification. It can be used in compilers to locate common subexpressions. Downey, Sethi, and Tarjan [30] show how FCCA can be used to test losslessness of joins in relational data bases.

Algorithm Eq is the foundation of the implementation of the V-code interpreter. The V-code interpreter uses that algorithm to deduce all consequences of substitutions and the PL/CV automatic propositional rules. Transformations can be combined with propositional reasoning, but not with complete generality. The NEW instruction and the OPEN, CLOSE, and CLOSE-NEW instructions can be efficiently implemented using an instance table mechanism and an audit trail.

8.2 The Utility of PL/CV

When evaluating PL/CV as a programming logic, people often ask the question, "what is the ratio of proof lines to program lines in a typical PL/CV program?" The answer to this question varies widely with type of program. In the program proofs that have been written to date, the ratio varies from 15:1 for a complex example that manipulates arrays to 1:1 for Euclid's division algorithm. The number of examples that have been written is too small to give any meaningful statistics.

But I claim that asking about the ratio of proof lines to program lines is asking the wrong question. Consider, for example, the problem of determining whether the vertices of a graph can be colored with k colors so that no edge connects vertices of the same color. Since this problem is NP-complete [4], there is interest in efficient algorithms to solve this problem. Recently, a quite simple polynomial time algorithm was discovered for a special case of this problem, the case in which the graph is planar and $k \leq 4$. The algorithm is very short, simply:

```
RETURN('1'B);
```

but the shortest known proof of the correctness of this algorithm is over 500 pages long. The proof was produced with the aid of a computer, and it is estimated that no human reader could possibly verify its correctness [8].

The point of this somewhat frivolous example is that some short programs are going to have long proofs, and no program verification system is going to alter this fact. Therefore, a more suitable metric to use to evaluate a program verification system is the ratio of size of the formal proof accepted by the Proof Checker to a corresponding informal proof.

By this metric, I feel the predicate calculus part of PL/CV has been quite successful. It is a very natural notation in which to express proofs. Proofs of simple programs can also be expressed fairly easily.

However, we have already seen one trouble spot: the array assignment statement. As discussed in section 4.9, the notion of accessibility is not powerful enough to correspond to programmer's intuition when reasoning about programs that modify arrays. And of course, the set of programs that modify arrays includes the vast majority of interesting programs.

Another stumbling block encountered when attempting to translate informal reasoning into PL/CV is the limited expressibility of the assertion language. Suppose we wish to prove a program that takes the sum of the first N elements of the array. An informal loop invariant would be "the variable S contains the sum of the elements A(1), A(2), ... A(J-1)." This loop invariant is not particularly easy to express in the PL/CV assertion language. The simplest method of expressing the notion of "the sum of the elements A(1), A(2), ... A(J-1)" in PL/CV is to write a program that calculates the sum.

In fact, there is a builtin SUM function in PL/I. This particular problem would be solved if the SUM function, and axioms to reason about it, were added to PL/CV. But it is impossible to add enough primitives to the assertion language to build in every concept anyone might want to use to prove a program. A better solution is to make the assertion language extensible. The DEFINE statement already provides a degree of extensibility, but the definitions allowed are not expressive enough. The assertion language at least ought to be extended to include finite sets, user-defined data types, and recursive definitions. These extensions are planned in PL/CV3 [23].

8.3 The Next Step

The PL/CV Proof Checker is an improvement over its predecessor in that abbreviations and automatic rules can be used to reduce the amount of proof text required to prove a program. How much more improvement of this type is possible?

The ultimate goal of such improvements would be a progrsm verification system for which one need only supply a specification and a program, and the system would indicate whether the program meets the specification. Such a system would solve what I call the CSTA (Computer Science Teaching Assistant) problem.

The CSTA problem is in general undecidable. Even if the problem is made easier by adding a few comments to the program, and we are willing to accept systems that cannot solve every instance of CSTA, we are still faced with an extremely difficult problem in artificial intelligence. If the reader doubts the accuracy of this statement, he is referred to any graduate student who has graded programs for a sophomore level Computer Science course. To attack CSTA it is necessary to make hypotheses, discard them, and in general bring as much intuition and experience to bear as one can muster. Traditionally computers have excelled at tasks requiring calculation and bookkeeping and have been rather inept at tasks requiring intuition.

Attempts to advance the state of the art of program verification by pushing it in the direction of solving CSTA are likely to be quite difficult. And intuitively I feel that approximations to CSTA are the wrong direction for program verification research to take. When I write a program, I know how it works. A program verification system should not have to waste computer resources guessing how my program works; it should be possible for me to explain in a formal way how the program works. Such an explanation ought to be useful to the human reader as well.

Yet all too often, formal proofs of programs degenerate into a vast sea of assertions in which program statements drift about like bits of flotsam. Proofs of this nature, as is eloquently pointed out by DeMillo, Lipton and Perlis [28], are of little value to the human reader. A common complaint about complex program proofs written in PL/CV is, "How can I understand this program when it is obscured by all these assertions?"

The most ambitious example to date of a PL/CV proof is a collection of number theory proofs, culminating in the proof of the fundamental theorem of arithmetic [16]. The proof is long (1477 lines), yet it is legible to a reader familiar with PL/CV because it is divided into a collection of small theorems, each of which can be individually digested.

No program proofs of the magnitude of the proof of the fundamental theorem of arithmetic have been attempted. Why is this? What is the difference between proofs of programs and proofs of mathematical theorems?

The proof of the fundamental theorem of arithmetic begins with some simple lemmas, such as "if a divides b and b divides c, then a divides c." These simple lemmas are used to prove more complex lemmas, such as "the logarithm function is monotonically increasing." When the fundamental theorem is finally proved, these lemmas are used. Moreover, it should be possible to use these low-level lemmas to prove other high-level theorems.

This hierarchical organization does not seem possible in program proofs. The state of a program is modified by a sequence of assignment statements; each assignment statement updates one word of the state. Backus [9] calls this phenomenon the "von Neumann bottleneck." Attempts at writing high-level proofs of programs fail because the proofs are dragged down to the low level of the program. An assertion must be written (or synthesized by the verification system) to describe the state before and after each word is pushed through the von Neumann bottleneck. And unlike the low-level lemmas in the proof of the fundamental theorem of arithmetic, these low-level assertions are of no value whatsoever when proving other programs.

Procedures are supposed to be a tool used to form high-level programs from low-level ones, but they do not seem to be general enough. Consider the following example. Let some program text containing a free variable x be denoted by S(x), and let P(z) be a predicate. Further, suppose P and S are related as follows:

(1) after the execution of S(x), P(x) must hold, and

(2) if y $\neq$ x, and P(y) holds before the execution of S(x), then P(y) still holds afterwards.

Then, after the execution of the following loop:

```
DO I := L TO U BY 1;
    S(I);
END;
```

the predicate:

```
ALL X FIXED. (L <= X <= U => P(X))                                (8.1)
```

must hold. The majority of PL/CV proofs of indexed loops have this flavor. And every time one of these proofs is written in PL/CV, the following theorem is reproved:

Theorem 8.1 If there is a program with properties (1) and (2), there is a program whose execution will make formula (8.1) true.

There is no way this theorem can be abstracted into a subroutine in PL/CV, because the program text S(x) cannot be passed as a parameter.

DeMillo, Lipton and Perlis argue that program proofs are worthless because they are detailed and tedious, and no one would ever want to read one. But one of the motivations for research in program verification is that programs (sans proof) are detailed and tedious, and no one would ever want to read one. A typical journal article describing a new algorithm does not contain a listing of a complete operational program; such a listing would contain far too much grubby detail. Instead the author of the article attempts to convince the reader that such a program can in principle be created.

I propose that program development systems that operate in the same fashion be developed and studied. Rather than exhibit a program in all its gory detail, and then prove that each gory detail has been correctly supplied, the user constructively argues that a program exists, using theorems like Theorem 8.1, and leaves details of actually producing a program to the verification system. One such system is described by Bates [10]. Because the proof does not have to be tied to assertions about the state of the machine at each step of the program, it should be possible to write high-level proofs similar in form to proofs of mathematical theorems.

I am not proposing a system that will magically produce a program from a specification; this problem is even harder than CSTA. The user of this proposed system must specify which theorems are to be used, and how they are combined to prove that an appropriate program exists. Because the proof is constructive, the program can be derived from the proof in a strictly mechanical fashion.

Let us not forget that a program is in actuality a sequence of bits. Once upon a time, people programmed computers by writing out such bit sequences. The invention of assemblers made it possible to program computers without paying any attention to the actual sequence of bits the program comprised. Instead, programs could be viewed as sequences of instructions. In the same manner, high level languages have made it possible to program computers without paying attention to individual instructions; programs are now viewed as sequences of statements. The next step in this progression is to stop paying attention to the individual statements in a program, and instead concentrate on the logic used to derive what those statements should be.

REFERENCES

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman.
<u>The Design and Analysis of Computer Algorithms</u>.
Addison-Wesley, 1974.
[1] Pages 77-84.
[2] Pages 84-86.
[3] Pages 124-139.
[4] Pages 377-393.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Code optimization and finite Church-Rosser Theorems.
In R. Rustin, editor, <u>Design and Optimization of Compilers</u>, pages 89-105. Prentice-Hall, 1972. Cited by Sethi [69].

[6] Alfred V. Aho and Jeffrey D. Ullman.
<u>Principles of Compiler Design</u>.
Addison-Wesley, 1977.

[7] American National Standards Institute.
<u>USA Standard FORTRAN</u>.
ANSI X3.9, 1966.

[8] Kenneth Appel and Wolfgang Haken.
The Four-Color Problem.
In Lynn Arthur Steen, editor, <u>Mathematics Today</u>, pages 153-180. Vintage Books, 1978.

[9] John Backus.
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
<u>Communications of the ACM</u> XXI(8):613-641, August 1978.

[10] Joseph L. Bates.
<u>A Logic for Correct Program Development</u>.
PhD thesis, Cornell University, August 1979.

[11] Thomas Bishop and Richard Conway.
User's Guide to Release 2. PL/CS.
Computer Science Technical Report 77-315, Cornell University, 1977.

[12] H. Boehm, A. Demers, and J. Donahue.
An Informal Introduction to Russell.
Computer Science Technical Report 80-430, Cornell University, September 1980.

[13] Robert Cartwright and Derek C. Oppen.
Unrestricted Procedure Calls in Hoare's Logic.
<u>Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages</u>, 1978, pages 131-140.
To appear in Acta Informatica as "The Logic of Aliasing."

[14] Tat-Hung Chan.
An Algorithm for Checking PL/CV Arithmetic Inferences.
Computer Science Technical Report 77-326, Cornell University, October 1977.

[15] Edmund Melson Clarke.
Proving Correctness of Coroutines Without History Variables.
Acta Informatica XIII(2):169-188, 1980.

[16] Robert L. Constable
The Fundamental Theorem of Arithmetic in PL/CV2.
Computer Science Technical Report 80-424, Cornell University, 1980.

[17] Robert L. Constable, Scott D. Johnson, and Carl Eichenlaub, Department of Computer Science, Cornell University.
PL/CV2 Reference Manual.
Second edition, in preparation, October 1980.

[18] Robert L. Constable and Michael J. O'Donnell.
A Programming Logic.
Winthrop, 1978.
[19] Pages 10-46.
[20] Pages 121-122.
[21] Pages 234-249.

[22] Robert Constable, Michael O'Donnell, Scott Johnson, and Carl Hauser.
User's Guide for the PL/CV Progrsm Verifier.
Unpublished memorandum, Department of Computer Science, Cornell University, 1976.

[23] Robert L. Constable and Daniel Zlatin, Department of Computer Science, Cornell University.
Report on the PL/CV3 Programming Logic.
In preparation, October 1980.

[24] Richard Conway and Robert L. Constable.
PL/CS, a Disciplined Subset of PL/C.
Computer Science Technical Report 76-273, Cornell University, 1976.

[25] Richard Conway and David Gries.
An Introduction to Programming: A Structured Approach using PL/I and PL/C, pages 117-118.
Winthrop, third edition 1979.

[26] Stephen A. Cook.
Soundness and Completeness of an Axiom System for Program Verification.
SIAM Journal on Computing VII(1):70-90, February 1978.

[27] John Corcoran, William Frank, and Michael Maloney.
String Theory.
Journal of Symbolic Logic XXXIX(4):625-637, December 1974.

[28] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis.
Social Processes and Proofs of Theorems and Programs.
Communications of the ACM XXII(5):271-280, May 1979.

[29] Edsger W. Dijkstra.
A Discipline of Programming.
Prentice-Hall, 1976.

[30] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan.
Variations on the Common Subexpression Problem.
To Appear in Journal of the Association for Computing Machinery.

[31] Herbert B. Enderton.
A Mathematical Introduction to Logic, pages 79-86.
Academic Press, 1972.

[32] Michael J. Gordon, Arthur J. Milner and Christopher P. Wadsworth.
Edinburgh LCF.
Lecture Notes in Computer Science 78, Springer-Verlag, 1979.

[33] David Gries.
Compiler Construction for Digital Computers.
John Wiley & Sons, 1971.
[34] Pages 49-71, 98.
[35] Pages 85-101.
[36] Pages 225-229.
[37] Pages 377-379.

[38] R. E. Griswold, J. F. Poage, and I. P. Polonsky.
The SNOBOL4 Programming Language.
Prentice-Hall, 1971.

[39] David Harel.
On Folk Theorems.
Communications of the ACM XXIII(7):379-389, July 1980.

[40] Eric C. R. Hehner.
do Considered od: A Contribution to the Programming Calculus.
Acta Informatica XI(4):287-304, 1979.

[41] C. A. R. Hoare.
An Axiomatic Basis for Computer Programming.
Communications of the ACM XII(10):576-581, October 1969.

[42] John E. Holm.
Floating-Point Arithmetic and Program Correctness Proofs.
PhD thesis, Cornell University, 1980.

[43] Shigeru Igarasshi, R. L. London, and D. C. Luckham.
Automatic Program Verification I: A Logical Basis and its Implementation.
Acta Informatica IV(2):145-182, 1975.

[44] International Business Machines Corporation.
OS PL/I Checkout and Optimizing Compilers: Language Reference Manual.
pages 462, 474-475.
Fifth edition, 1976.

[45] International Business Machines Corporation.
PL/I (F) Language Reference Manual.
Fourth edition, 1970.
[46] Page 164.
[47] Pages 205-213.
[48] Page 331.
[49] Pages 374-375.

[50] Kathleen Jensen and Niklaus Wirth.
PASCAL User Manual and Report.
Springer-Verlag, second edition 1974.

[51] S. C. Johnson and M. E. Lesk.
Language Development Tools.
The Bell System Technical Journal LVII(6):2155-2175, July-August 1978.

[52] Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, 1978.

[53] Donald E. Knuth.
Structured Programming with *go to* Statements.
ACM Computing Surveys VI(4):261-301, December 1974.

[54] Donald E. Knuth.
The Art of Computer Programming. Volume 1: *Fundamental Algorithms,*
pages 332-345.
Addison-Wesley, 1968.

[55] Donald E. Knuth.
The Art of Computer Programming. Volume 3: *Sorting and Searching,*
pages 506-549.
Addison-Wesley, 1973.

[56] James C. King.
Symbolic Execution and Program Testing.
Communications of the ACM XIV(7):385-394, July 1976.

[57] Dean B. Krafft.
The Assertion Table System for the PL/CV2 Program Verifier.
Computer Science Technical Report 78-337, Cornell University, April 1978.

[58] Dean B. Krafft, Department of Computer Science, Cornell University.
AVID: A System for the Interactive Development of Verifiably Correct Programs.
In preparation, October 1980.