ARPA ORDER NO. 2223

ISI/RR-78-70
January 1978

Donald Scott Lynn

Interactive Compiler Proving Using Hoare Proof Rules

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ISI/RR-78-70
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: Interactive Compiler Proving Using Hoare Rules Proof
5. TYPE OF REPORT & PERIOD COVERED: Research rept.
6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Donald S. Lynn
8. CONTRACT OR GRANT NUMBER(s): DAHC 15 72 C 0308 ARPA Order 2223
9. PERFORMING ORGANIZATION NAME AND ADDRESS: USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ARPA Order No. 2223
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209
12. REPORT DATE: Jan 78
13. NUMBER OF PAGES: 181 (13) 182 p
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Donald Scott/Lynn
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

This document approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

compiler proofs, Hoare proof rules, interactive proving, Lisp compiler, McCarthy-Painter compiler

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

(OVER)

407 952

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. ABSTRACT

This report demonstrates the proofs by structural induction of two compilers. The larger is a compiler from a subset of pure LISP into a machine language, and is similar to a compiler proved by London. The smaller is the compiler proved by McCarthy and Painter. In addition a portion of another compiler is proved. It is the compiler given by Wirth to compile a simple Pascal-like language (PL/0) into code for a hypothetical stack-oriented machine. We here employ several methods of proof, different from the previous methods, which are amenable to machine-aided proofs. These new methods include the use of Hoare proof rules to describe the semantics of the source and target languages, as well as the language in which the compiler was written, a new formalization of substitution, and axiomatic definition of functions, both in the compiler and in the assertion language. The machine assistance was provided by the Xivus program verification system, which is a later version of the system described by Good, London, and Bledsoe. We believe the methods of compiler proof given here are of general application in the proving of compilers including such source language features as assignment, conditional, repetition (WHILE), and go to statements, recursive and non-recursive function calls, and statement- or expression-oriented languages.

ARPA ORDER NO. 2223

ISI/RR-78-70
January 1978

Donald Scott Lynn

Interactive Compiler Proving Using Hoare Proof Rules

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308. ARPA ORDER NO. 2223.

ACKNOWLEDGMENTS

The research described here was supported by the Defense Advanced Research Projects Agency under contract DAHC-15-72-C-0308. The views expressed are those of the author.

Thanks are due my dissertation committee, Professor Ellis Horowitz, Professor Harry Andrews, and especially to my chairman Professor Ralph London for his continued guidance throughout this research. I have had helpful discussions with Dr. Donald Good and Dr. David Musser. My thanks go to the staff of the University of Southern California Information Sciences Institute for providing the research environment in which the work reported here was conducted.

Those who have made the Xivus system more accommodating to my needs include Martin Yonke, Lawrence Fagan, Peter Bruell, and Raymond Bates. Ray has also helped greatly with the mechanics of preparing this document. Betty Randall has provided secretarial assistance throughout the research.

My gratitude is extended to my management at McDonnell Douglas who have allowed me the time off to pursue the degree, and have given some tuition assistance.

My wife Marlys and daughter Erika have been extremely tolerant of the unreasonable sacrifices that are expected of a student's family.

CONTENTS

1 OVERVIEW

This dissertation reports the proofs by structural induction of two compilers. The larger is a compiler from a subset of pure Lisp into a machine language, and is similar to a compiler proved by London [London72]. The smaller is the compiler proved by McCarthy and Painter [McCarthy67]. In addition a portion of another compiler is proved. It is the compiler given by Wirth [Wirth76] to compile a simple Pascal-like language (PL/0) into code for a hypothetical stack-oriented machine. We here employ several methods of proof, different from the previous methods, which are amenable to machine-aided proofs. These new methods include the use of Hoare proof rules [Hoare69] to describe the semantics of the source and target languages, as well as the language in which the compiler was written, a new formalization of substitution, and axiomatic definition of functions, both in the compiler and in the assertion language. The machine assistance was provided by the Xivus program verification system, which is a later version of the system described by Good, London, and Bledsoe [Good75]. We believe the methods of compiler proof given here are of general application in the proving of compilers including such source language features as assignment, conditional, repetition (WHILE), and go to statements, recursive and non-recursive function calls, and statement- or expression-oriented languages.

In Chapter 2 the problem of proving correctness of compilers is outlined. The methods of proof used in these compiler proofs are given in detail in Chapter 3. The statement of precisely what was proved about the compilers is developed in Chapter 4, along with the shorter of the proofs and the portion of the PL/0 compiler proof. The major portions of the longer proof are left to the appendix. Previous proofs of compilers are described in Chapter 5, concluding the chapter with a discussion of the significance of our approach in relation to previous work. Chapters 6 and 7 present the conclusions reached in completing the compiler proofs and areas for future research, respectively. References cited in the text follow the last chapter.

2 PROBLEM STATEMENT

2.1 Need for Proving Programs

Recent experience with computing systems has shown a startling portion of time and cost to be expended on efforts to make software work correctly. The case has been presented several times, for example in London [London75], that program proving is a valuable tool to aid this effort. We will therefore restrict the remainder of this chapter to examining the motivation for applying this tool to the compiler correctness problem.

2.2 Importance of Compiler Correctness

This dissertation involves the application of correctness proofs to compilers. Why compilers? First, compilers are heavily used. Indeed in many installations nearly every program run is processed by a compiler. So the existence of errors in compilers will likely cost more than errors in less-used programs.

Second, an error in a compiler is difficult for the user to distinguish from his own programming errors, thereby increasing his debugging costs. Often the user's only means of remedying a problem caused by a faulty compiler is to complicate his own program unnecessarily. This can come about because debug time and funds are no longer available for a "working" compiler, or because of insufficient local support for a compiler that was written elsewhere. In any case, the user often must program around compiler problems and end up with a degraded program in terms of understandability and maintainability. The problem is compounded because most computer users will not or can not become familiar with the internal workings of a compiler, or with the target code produced. Yet they must use the compiler in order to write programs in the compiler's source language. So users must turn to a compiler expert, if one is available, thus requiring at least two persons rather than one to correct compiler errors. While this problem is shared by any program used by other than the author, compilers are probably the most prominent example of it (perhaps second to operating systems).

Further, the fact that many programmers usually work on creating a compiler means that a compiler is subject to the subtle kinds of problems caused by bad communication between parts of the compiler written by different persons. These interfacing problems could be found and eliminated by proving compilers, or possibly by just designing compilers with proving in mind. While it may be pointed out that compilers proved in the past have been of much smaller scale than this, we believe that continuing work on proving compilers, particularly with machine aid, will eventually allow much larger ones to be proved.

Any proof that a user's program is correct can be invalidated if an error exists in the compiler he is using. This is so because such proofs are usually based on a specification of what the source language of the compiler is intended to do, not of what the compiler actually does with source language programs.

2.3 Contrast with Non-compiler Proofs

What makes compiler proofs a different problem from proving nearly all other programs? There is not just a single correct (target code) result of compiling a program. Thus the statement of the compiler's task must state not that the correct answer results from the compiler, but rather that the result from the compiler has certain properties. Further, these properties relate to the results of executing the compiler output, thereby placing a second level of execution into the proof. Those properties will be the equivalent (in target language terms) of the properties that the source language program has. Another complication is determining exactly what the properties are that must be preserved during compiling. In some languages, such as pure Lisp, this includes only that the same value is returned, while in other languages, such as Algol, the correspondence between all variable names and values must be preserved, as well as the values of certain hidden entities, such as I/O files, recursion stacks, and returned values of functions.

This, of course, means that the proof of correctness of the compiler must include a description of the meaning or semantics of both the source and target languages in addition to the usual semantics of the programming language (of the compiler). We will need to express by symbolic means part of the target language code that corresponds to any non-terminal source language syntactic type (one that may contain other syntactic types as parts). For example, a piece of target code may be referred to as "the compilation of the first argument." This arises naturally from the general plan of the proof, which uses structural induction [Burstall69a], or induction on the source language structural parts. In other types of program proofs that involve the application of Hoare rules, we generally apply the rules to assertions and code, both of which are given completely without the use of names or symbols to represent parts of assertions or code. But here we will apply Hoare rules to symbolically represented code and assertions. We must use a notation to show the operations of the Hoare rule (usually substitution of expressions for names) being applied to symbolically expressed pieces of code. This is then a further level of symbolism which complicates compiler proofs.

Further problems with compilers arise because they are basically non-numeric programs; they are generally oriented toward character string handling, since most computer languages are expressed in character strings, while much previous program proving work has been on numeric programs. Also, we have the recursive nature of compiler source languages. Even the simplest of languages defines allowable expressions in recursive terms, while many languages also have recursive definitions of the allowable statement structure. Thus parts of the input to a compiler can be of any finite size, requiring the use of induction on that size to prove that the compiler will handle the input, no matter how large.

To make matters worse, most target languages do not have recursion, or at best have only a stack to accomplish all forms of recursion. Thus, compilers often have an undoing or transforming of recursion within them, complicating their proof of correctness.

2.4 Correct Input Assumed

We will always assume that a source language program is a legally executable program. Finding and preventing compile-time and run-time errors in the source program is a problem which we will not address. We simply wish to show that a compiled program will have the same effect as its assumed correct source would.

The style of these proofs could, however, be used to prove a compiler which did error checking of the source code at compile time and produce error-checking target code at run time. In order to accomplish compile-time checking it would be necessary to prove a separate case of input syntactic type, that of "none of the above types." An indication of error would then be specified as the meaning of such a type. For run-time checking--for example, division by zero--it would be necessary to define (within the source language Hoare rules) the error conditions and their results. Then it would be proved during the compiler proof that the proper error indications were produced under appropriate error conditions at target language run time.

3 METHODS OF PROOF

3.1 Introduction

The larger compiler proof, which is for the most part given in the appendix, uses many methods and techniques, some new and others previously applied in program proving. In the following sections is given a description of each method and where it is applied in this compiler proof. The significance of the new methods and their relation to previous work are presented in Section 5.4.

3.2 Comparison to C0

The compiler proved here is a modification of one called C0, which was written by John McCarthy and proved by London [London72]. We will refer to it as MC0 (for modified C0). The following changes were made to make Pascal the language in which the compiler is written. Compiler C0 was written in Rlisp, but the Xivus program verifying system, which provided the machine aid, requires programs to be written in (slightly extended) Pascal. The translation was intended to perform exactly the same functions as the original.

1. Conditional expressions were removed. Thus A := IF B THEN C ELSE D becomes IF B THEN A:=C ELSE A:=D.
2. All gensym function calls were made into procedure calls, returning the value as a new variable parameter. The reasons for this are fully explained in Section 3.17.
3. All lambda expressions were expanded fully. Since gensyms were previously removed from expressions, this expansion did not result in any single gensym calls being duplicated into multiple calls. Again, see Section 3.17 for further explanation.
4. Add sufficient parentheses on single argument functions because Rlisp allows such parentheses to be omitted.
5. Change the assignment operator from ← to := .
6. Declare the argument FLG as Boolean type and set it to FALSE and TRUE rather than NIL and non-NIL.
7. Express Lisp list notation within quoted constants as CONSes of quoted constants. Thus '(A B) becomes CONS('A,'B).
8. Expand n argument functions into nested two argument functions. Thus LIST(A,B,C) becomes CONS(A,CONS(B,CONS(C,NIL))) and APPEND(A,B,C) becomes APPEND(A,APPEND(B,C)). For consistency, even one or two argument LIST calls were expanded to CONSes. Some simplifications were applied, such as APPEND(CONS(A,NIL),B) becomes CONS(A,B).
9. Appropriate Pascal function headers and argument lists with argument types were added. Also BEGIN END; was placed around function bodies.
10. Infix dot is translated to prefix CONS.
11. EQ becomes = .

The following changes were made for reasons other than the change of compiler language, and such reasons are explained under each change. Again these changes were intended to preserve the functions of the original compiler, that is, not change the output of the compiler, except for the item involving the removal of an optimization.

1. All quotes on constants were replaced by the letter Q. Our Pascal parser does not recognize any form of alphanumeric constants. Similarly NIL becomes QNIL and, for consistency, T becomes QT. In every case the Q resulted in a new identifier.
2. Various expressions were grouped as separate functions for clarity in reading the code and to separate logically separate tasks. The resulting new functions are RETRIEVE, RPOP, and ADDIDS.
3. LOCTABLE was used as a more descriptive name than VPR for the variable holding the location table.
4. C0 compiles Boolean expressions nested inside other Booleans differently than if not so nested. This optimization, which jumps to a given label according to the Boolean value instead of producing a result to be used by a conditional jump, was removed. Thus COMPEXP (the function that compiles an expression) is called and a conditional jump statement is used in all places that formerly called COMBOOL (the function for optimized compiling of a Boolean) except the call inside COMPEXP. Then COMBOOL is never called with an atomic expression, so the first few lines of code therein may be discarded. Similarly the final ELSE clause of COMBOOL is never satisfied and may be omitted. The argument FLG is always FALSE in the remaining calls to COMBOOL, so FLG may be deleted and the code of COMBOOL simplified.
5. A new variable N was introduced to shorten the form resulting from expanded lambda expressions.
6. Function definitions being compiled are broken into their syntactic types inside compiler function COMP rather than before the call to COMP. We thought the compiler was more consistently broken into tasks this way.
7. When adding assertions to MC0 it was found necessary to refer in some places to target code already produced. But often some target code was an argument to a call to APPEND in a higher level routine, and thus not available in the routine being asserted. The solution to this is to pass a new argument (the output file, or briefly "OUTFILE") into nearly every routine in order to have newly produced target code appended to its end (the right-hand end if viewed as a Lisp list). In fact we want a new type here that resembles Lisp lists except that it has only one operation available in the compiler code, namely CONSing onto the "wrong" end. We will call this type FILE and allow RIGHTCONS as the only operation in the code. Assertions may, however, dissect FILEs. Since we now have a variable argument, many of the compiler functions must now be expressed as procedures. Furthermore, all appends of code are now incorrectly typed and unnecessary, and are replaced by either a call of a procedure that RIGHTCONSes the code onto the OUTFILE, or else an assignment to OUTFILE of a RIGHTCONS expression.
8. THEN and ELSE clauses are reversed and the IF conditions negated to allow dropping null ELSE clauses in MKPUSH, COMPLIS, and LOADAC.

Although this compiler was reasonably well structured in its original form, there were many changes made to increase its understandability, separation of tasks, and ease of expressing assertions. More could have been made. This again lends credence to two often expressed program proving philosophies:

1. The better situation for proving programs is to create the program and its proof together, not to add on proving afterwards as was done here.
2. Creating a program with the intention of adding assertions to it and proving it results in a more understandable program (many would hold this true even if proving is not carried out).

3.3 Inductive Assertions

The inductive assertion method of proving programs was introduced by Naur [Naur66] and Floyd [Floyd67]. Naur proposed the term "snapshots" for what Floyd and subsequent authors term assertions. Their method involves making assertions which state what is to be true each time execution passes through given points in a program. Then for all pairs of assertions connected by an executable path of statements, it is proved that the final assertion must be true after execution of those statements, provided the initial assertion was true before. The assertion reached at the conclusion of execution (called the output or Exit assertion) is then a specification of what the program accomplishes. We here assume that the program does indeed terminate. Proving what is true of a program making this assumption of termination is called a partial correctness proof. Partial correctness by using inductive assertions is the method used in proving MC0. A good review of the method and other aspects of program proving is Elspas et al.[Elspas72].

3.4 Hoare Rule Semantics

Hoare [Hoare69] introduced a method, now called Hoare proof rules, of precisely defining semantics of program statements. Appearing in Hoare proof rules is Hoare's notation P{S}Q, which means that if assertion P is true before a program part S is executed, and if execution of S terminates, then the assertion Q is true afterward. Another notation used by Hoare is

$$\frac{A,\ B}{C}$$

which is a rule of inference that allows us to deduce C if A and B have been proved. The Hoare proof rules, written in these notations, then take the form of axioms or rules of inference for the various programming constructs. We can then define the semantics of a statement just by stating what can be proved about the statement using its Hoare proof rule. The idea that specification of proof techniques provides a programming language definition is a main point of Floyd's paper significantly titled "Assigning meanings to programs" [Floyd67]. The Hoare proof rule method is closely related to the inductive assertion method, and is sometimes referred to as the Floyd-Hoare approach.

Nearly all of the languages Pascal [Hoare73] and Euclid [London77] have been defined in terms of such rules. For programs written in a language so described, it is a mechanical

procedure to transform a program containing assertions into a set of logical theorems. The theorems are called verification conditions, and the mechanical process is known as verification condition generation. The proof of these verification conditions shows that the program is consistent with its assertions, which are taken to be the definition of correctness for the program.

To prove a compiler we must specify the semantics of not only the language in which it is written, but also the source language and the target language. We have chosen to use Hoare proof rules to express the semantics of all three languages. The reasoning behind these three choices is as follows.

We wish to show what the effects are of executing the compiler. Toward this end we will use verification condition generation, applying Hoare proof rules to the program statements of the compiler. Therefore we will express the semantics of the language in which the compiler is written by means of Hoare proof rules, making those semantics directly and easily applicable in the proof.

What we wish to prove about a compiler is that the target language code that it produces has the same effect as the source language does (or would have if source language were directly executed). We will do this by proving, for each Hoare proof rule P{S}Q in the source language (rules of inference will be handled similarly), a Hoare formula in target language terms roughly of the form

compilation(P) {compilation(S)} compilation(Q) . (*)

The compilation of S is simply the target code output by the compiler when it compiles the source language code S, and it usually consists of several target language instructions. The compilation of assertions P and Q means the result of translating those assertions into target language terms in a manner similar to what the compiler does to source language code. This translation involves changing variable names into locations and source language constants into their target language representation while retaining the function names and structure of the assertion expressions.

A natural way to express source language semantics is with Hoare rules when we use the formulation of the correctness of the compiler as expressed by formula (*) above. Any other way would, of course, require translation to Hoare rule form for use in the formula (*). Then we may prove the target language formula resulting from this statement of correctness by applying the Hoare rules (for the various target language instructions in compilation(S)) to produce a logical theorem which we will then prove. Such proof for every possible form of compilation(S), corresponding to all possible syntactic forms of S, will constitute proof of the compiler. This points to the use of Hoare rules as the choice to express the semantics of the target language also. We will give the target language Hoare rules used to prove MC0 after giving some further notation involving lists and substitutions.

3.5 List Notation

We believe that the following expression, which arises in the proof of MC0, is unreadable.

```
  OUTFILE
= APPEND(OUTFILE',
         APPEND(FCOMPANDOR(CDR(EXP), M, L1, FALSE, LOCTABLE),
                CONS(CONS('MOVEI,
                          CONS(1, CONS(CONS('QUOTE, CONS('T, 'NIL)),
                                       'NIL))),
                     CONS(CONS('JRST, CONS(0, CONS(L2, 'NIL))),
                          CONS(L1,
                               CONS(CONS('MOVEI,
                                         CONS(1, CONS(0, 'NIL))),
                                    CONS(L2, 'NIL)))))))
```

Similar expressions appear to be even more unreadable under any of the conditions:

1. A longer expression is used (and many longer ones occur in the proof of MC0),
2. Straight Lisp notation for functions is used instead of arguments separated by commas, or
3. Careful indentation is not used.

Consequently it was decided that a better notation must be used to make more readable the assertions about lists. A notation was developed that borrows much from the clisp list construction notation [Teitelman75, p. 23.10]. Functions which are not considered intrinsic to the source language are expressed in conventional prefix with arguments separated by commas, while the intrinsic list building functions use the clisp style of notation. Briefly, that notation builds lists that are begun and ended with angle brackets: < >. Then each item in the brackets is a member of that list, except items prefixed by ! are lists of items to be placed in the list.

As in Lisp, unquoted items are to be evaluated while a prefixed single quote means not to do so. Assume F(X,Y) returns the Lisp value '(A B). Then in the list notation used here, < F(X,Y) 'C > means '((A B) C) while < !F(X,Y) 'C > means '(A B C).

When Lisp dotted pairs are expressed, we will use < 'A . 'B > to mean '(A . B).

The original example expression in this section may be expressed now as:

```
OUTFILE =
< ! OUTFILE'
  ! FCOMPANDOR(CDR(EXP),M,L1,FALSE,LOCTABLE)
  < 'MOVEI 1 < 'QUOTE 'T > >
  < 'JRST 0 L2 >
  L1
  < 'MOVEI 1 0 >
  L2
>
```

This notation was used in proving MC0 for writing assertions which were then mechanically translated to a form like the original example in this section for machine use.

3.6 Formalizing Substitution

The Hoare proof rule for an assignment statement is (ignoring such complications as function calls that might be in the statement):

$$Q\,\Big|_{b}^{a} \quad \{\ a := b\ \}\ Q$$

The vertical line notation means to substitute b for all free occurrences of a in the formula Q. A standard caution we note here is that the expression b must not have free uses of any variables that become bound when introduced into Q. Of course, renaming of the bound variable can be used to avoid this problem.

When applying similar Hoare rules during a compiler proof, it will often be the case that Q will be symbolically expressed. In that case we will not be able to write the result of such a substitution, but will leave it in the vertical line notation and apply further Hoare rules involving substitutions on it. Therefore we have extended this notation for various types of multiple substitutions.

$$Q\,\Big|_{b}^{a}\Big|_{d}^{c}$$

means to first apply the a-b substitution, then apply the c-d substitution to that result. This is sequential substitution. We will also need notation for simultaneous substitution.

$$Q\,\Big|_{b\ d}^{a\ c}$$

means to simultaneously substitute b and d for a and c, respectively. This form is often used when b contains an occurrence of c which we do not want further changed to d.

$$Q\,\Big|_{b\ \Big|_{d}^{c}}^{a}$$

means to first apply the c-d substitution to b, then use that result for all free occurrences of a in Q.

The rules which we will present regarding substitutions will be valid only when the item substituted for (for instance a in the above examples) is an identifier. The item substituted in (that is, b in the first example above) may, however, be an expression. Occasionally we will have a list of identifiers which we will represent with a single name. When we substitute such a name, we will use a double vertical line to remind us that it is a multiple substitution and the single substitution rules may not apply. For example, if x represents the list <a c e> and y the list <b d f>, then the form

$$Q\,\Big\|_{y}^{x}$$

means

$$Q \begin{vmatrix} a & c & e \\ b & d & f \end{vmatrix}$$

Note that this substitution is sequential. When we have occasion to express a multiple simultaneous substitution, we will give the items individual names, and the substitution will look something like this:

$$Q \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix}$$

We will occasionally resort to parentheses to clear up any unusual orders of substitution not covered by these forms.

In order to simplify expressions involving substitutions, we wish to express the condition that a certain name could not possibly exist (as a free occurrence) in an expression. We will use the notation a $\neg\epsilon$ e to describe this condition. Again, a must be atomic while e may be an expression. Both a and e will often be symbolically expressed in a proof. Precisely what is meant by a $\neg\epsilon$ e is that for all possible asserted source language programs from which e may be derived, a will never appear as a free variable name in e when e is written out fully in source or target language variable names (no code or assertions left symbolically named). For example, if e represents a source language expression and a represents a target language register, we would then know that a $\neg\epsilon$ e.

To assure that such statements involving $\neg\epsilon$ are true, we assume that different names are used for source language objects and target language objects. This could be easily accomplished by renaming to unique names when a conflict would occur, but no actual conflicts will occur in the proof of MC0 because we never refer to source variables by their names.

There are four distinct classes of source and target variables in a compiler organized like MC0. Knowing that they are distinct will aid us in simplifying expressions containing substitutions.

1. Source program variables. We will usually refer to these by expressions which select lists of variables from the source code or from the location table of the compiler.
2. Target program registers. We will refer to the registers by R1, R2, ... Rn.
3. Target program stack pointer P.
4. Target program stack locations. We will always refer to the stack locations with an array-like subscript notation, calling the stack memory m. For example, m[P+1]. Substitutions involving stack locations will always be done as a replacement of the entire stack m by an alpha expression, where α(m,i,y) means the result of changing the ith element of m to the value y.

The fact that the registers, pointer, and stack must be distinct may impose restrictions on certain implementations of the target language system. For example, if P, the stack pointer of MC0, is actually implemented as register fourteen, then proofs of correctness are not valid if register fourteen is used otherwise in executing a target program. Since registers are used in the target code produced by this particular compiler to hold arguments to functions, this

restriction would mean that user programs containing over thirteen arguments to any function could not be correctly compiled and run.

The rules describing simplifications which may be done, henceforth called subrules, are given in figure 3-1 at the end of this section. All D's, a's, and X's, numbered or not, are atomic names, while the other letters may be expressions. The first two rules express the distinctness of the four variable classes. Subrule 3 allows us to state that a variable is not-in ($\neg\epsilon$) an expression if it is not-in any of the expression's component substitutions. Subrule 4 states that substituting for an item not there produces no change. Subrules 5, 8, 9, and 20 show us the equality of certain forms that will appear in the proof of MC0. Subrule 6 allows us to distribute substitution over source language functions. It might be noted that *quantification is not a function*, and that passing functions as arguments to other functions (not allowed in the source language of MC0) may also invalidate subrule 6. Neither does this subrule apply to the (non-source language) functions such as substitution that are applied to assertions rather than appearing in assertions as source language functions do. Subrule 7 gives exactly the conditions under which we may change the order of multiple substitutions. Subrule 10 tells us when simultaneous and sequential substitution produce the same result. Subrule 11 tells us that $\neq$ and $\neg\epsilon$ are the same for names (actually we only give the implication one direction; it is true the other direction, but is not needed here). Subrule 12 states that an item substituted for is gone (unless it is put back in by that substitution). Subrule 13 allows us to distribute not-in ($\neg\epsilon$) over source language functions and the converse. The caveats of subrule 6 apply here also. Subrule 14 states that substituting something for itself results in no change.

In a few places in the proof of MC0, we have simultaneous substitutions which do not satisfy the criteria of subrule 10, and so may not be treated as sequential. Subrule 15 is a rather complex rule which allows us to sequentialize such substitutions under rather specific conditions. Those conditions are that a further substitution was to be performed after the simultaneous one which would change all names introduced in the simultaneous substitutions *into names with a certain distinctness*. The way we sequentialize then is to apply the further substitution to each item of the simultaneous substitution.

Subrules 16, 17, 18 and 19 define how substitution and $\neg\epsilon$ interact with universal quantification. They are easily understood by recalling that quantification causes variables not to be free, and thus affects substitution for free occurrences. Note that in subrule 18 we see the caution mentioned earlier that we may not introduce a variable during substitution that gets bound. Thus H in subrule 18 must not contain X, the variable being bound by the quantifier.

Subrule 21 is simply the carrying out of a substitution. In fact use of this subrule will often be referred to as doing or carrying out a substitution rather than being referred to by the number 21. Application of subrules 1 and 2 will occur frequently in the proof of MC0. When such application is quite obvious, the proof will omit mention of those rules. Subrule 6 will often be referred to by the term distribution rather than by number.

A more rigorous justification for some of the subrules, along with the flavor of how the others may be proved, is given in Section A.10.

FIGURE 3-1

Subrule 1:

a) Ri's $\neg\epsilon$ source language expressions
b) P $\neg\epsilon$ source language expressions
c) m $\neg\epsilon$ source language expressions
d) source variables $\neg\epsilon$ non-source expressions

Subrule 2:

a) source variable $\neq$ Ri
b) source variable $\neq$ P
c) source variable $\neq$ m
d) Ri $\neq$ P
e) Ri $\neq$ m
f) P $\neq$ m

Subrule 3:

a) $D \neg\epsilon G \wedge D \neg\epsilon H1 \rightarrow D \neg\epsilon G \begin{vmatrix} D1 \\ H1 \end{vmatrix}$

b) $D \neg\epsilon G \wedge D \neg\epsilon Hi \text{ (for } 1 \leq i \leq n) \rightarrow D \neg\epsilon G \begin{vmatrix} D1 & \ldots & Dn \\ H1 & \ldots & Hn \end{vmatrix}$

Subrule 4:

$D \neg\epsilon G \rightarrow G \begin{vmatrix} D \\ H = G \end{vmatrix}$

Subrule 5:

$D2 \neg\epsilon G \rightarrow G \begin{vmatrix} D1 & D2 \\ H1 & H2 = G \end{vmatrix} \begin{vmatrix} D1 & \\ H1 & D2 \\ & H2 \end{vmatrix}$

Subrule 6 (distribution of substitutions):

$(f\ G1\ \ldots\ Gn) \begin{vmatrix} D \\ H \end{vmatrix} = (f\ G1 \begin{vmatrix} D & \ldots \\ H & \ldots\ Gn \end{vmatrix} \begin{vmatrix} D \\ H \end{vmatrix})$

where f is any source language function.

Subrule 7:

a) $D1 \neg\epsilon H2 \wedge D2 \neg\epsilon H1 \wedge D1 \neq D2 \rightarrow G \begin{vmatrix} D1 & D2 \\ H1 & H2 = G \end{vmatrix} \begin{vmatrix} D2 & D1 \\ H2 & H1 \end{vmatrix}$

b) $Di \neg\epsilon Hj \wedge Di \neq Dj \text{ (for } 1 \leq i \leq n,\ 1 \leq j \leq n,\ i \neq j) \rightarrow$

$G \begin{vmatrix} D1 & \ldots & Dn \\ H1 & \ldots & Hn \end{vmatrix}$ can be permuted to any order.

Subrule 8:

$$D1 \neq D2 \wedge D1 \neg\epsilon H2 \rightarrow G \begin{vmatrix} D1 \\ H1 \end{vmatrix} \begin{vmatrix} D2 \\ H2 \end{vmatrix} = G \begin{vmatrix} D2 \\ H2 \end{vmatrix} \begin{vmatrix} D1 \\ H1 \end{vmatrix} \begin{vmatrix} D2 \\ H2 \end{vmatrix}$$

Subrule 9:

$$D1 = D2 \rightarrow G \begin{vmatrix} D1 \\ H1 \end{vmatrix} \begin{vmatrix} D2 \\ H2 \end{vmatrix} = G \begin{vmatrix} D1 \\ H1 \end{vmatrix} \begin{vmatrix} D2 \\ H2 \end{vmatrix}$$

Subrule 10:

a) $D2 \neg\epsilon H1 \rightarrow G \begin{vmatrix} D1 & D2 \\ H1 & H2 \end{vmatrix} = G \begin{vmatrix} D1 \\ H1 \end{vmatrix} \begin{vmatrix} D2 \\ H2 \end{vmatrix}$

b) $Di \neg\epsilon Hj \ (\text{for } 1 \leq j < i \leq n) \rightarrow G \begin{vmatrix} D1 & \dots & Dn \\ H1 & \dots & Hn \end{vmatrix} = G \begin{vmatrix} D1 & \dots \\ H1 & \dots \end{vmatrix} \begin{vmatrix} Dn \\ Hn \end{vmatrix}$

Subrule 11:

$D1 \neq D2 \rightarrow D1 \neg\epsilon D2$

Subrule 12:

$$D \neg\epsilon H \rightarrow D \neg\epsilon G \begin{vmatrix} D \\ H \end{vmatrix}$$

Subrule 13 (distribution of $\neg\epsilon$):

a) $D \neg\epsilon (f\ G1 \dots Gn) \rightarrow D \neg\epsilon Gi \ (\text{for } 1 \leq i \leq n)$

b) $D \neg\epsilon Gi \ (\text{for } 1 \leq i \leq n) \rightarrow D \neg\epsilon (f\ G1 \dots Gn)$

where f is any source language function.

Subrule 14:

$$G \begin{vmatrix} D \\ D = G \end{vmatrix}$$

Subrule 15:

$$X \begin{vmatrix} a1 & \dots & an \\ b1 & \dots & bn \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} = X \begin{vmatrix} a1 & & \dots & an \\ b1 & & & bn \end{vmatrix} \begin{vmatrix} v \\ w \dots \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix}$$

if all atoms in the b's appear in v at least once, a's are distinct from each other, and all items in w are distinct from those in v.

Subrule 16:

a) $X \neg\epsilon G1 \rightarrow \forall X \ (G1 \rightarrow G2) = G1 \rightarrow \forall X \ (G2)$

b) $X \neg\epsilon G1 \rightarrow \forall X \ (G1 \wedge G2) = G1 \wedge \forall X \ (G2)$

c) $X \neg\epsilon G \rightarrow \forall X \ (G) = G$

Subrule 17:

a) $\forall X\ (\forall X\ (G)) = \forall X\ (G)$

b) $\forall X1,\ldots,Xn\ (\forall X1,\ldots,Xm\ (G)) = \forall X1,\ldots,Xmax(m,n)\ (G)$

Subrule 18:

a) $X \neq D \wedge X \neg\epsilon H \rightarrow \forall X\ (G) \begin{vmatrix} D \\ H \end{vmatrix} = \forall X\ (G \begin{vmatrix} D \\ H \end{vmatrix})$

b) $Xi \neq Dj \wedge Xi \neg\epsilon Hj$ (for $1 \leq i \leq n$, $1 \leq j \leq m$) $\rightarrow$

$$\forall X1,\ldots,Xn\ (G) \begin{vmatrix} D1 & \ldots & Dm \\ H1 & \ldots & Hm \end{vmatrix} = \forall X1,\ldots,Xn\ (G \begin{vmatrix} D1 & \ldots & Dm \\ H1 & \ldots & Hm \end{vmatrix})$$

Subrule 19:

$$\forall X\ (G) \begin{vmatrix} X \\ H \end{vmatrix} = \forall X\ (G)$$

Subrule 20:

$$D2 \neg\epsilon H2 \rightarrow G \begin{vmatrix} D1 & \begin{vmatrix} D2 \\ H2 \end{vmatrix} \\ H1 & \end{vmatrix} \begin{vmatrix} D2 \\ H2 \end{vmatrix} = G \begin{vmatrix} D1 & \begin{vmatrix} D2 \\ H2 \end{vmatrix} \\ H1 & \end{vmatrix}$$

Subrule 21:

$$D \begin{vmatrix} D \\ H \end{vmatrix} = H$$

3.7 Hoare Rules Defining Target Language

We here give the Hoare rules defining the semantics of each of the target language statement types that appear in the output of the compiler MC0.

MOVEI:

$$Q \begin{vmatrix} Ri \\ y \end{vmatrix} \{ < \text{'MOVEI i y} > \} Q$$

This rule can be viewed as an assignment statement Hoare rule. For example, the instruction < 'MOVEI 1 0 > can be viewed as the assignment statement R1 := 0. The following standard Hoare rule for assignment would then apply.

$$Q \begin{vmatrix} R1 \\ 0 \end{vmatrix} \{ R1 := 0 \} Q$$

MOVE:

$$Q \begin{vmatrix} Ri \\ m[P+j] \end{vmatrix} \{ < \text{'MOVE i j 'P} > \} Q$$

This rule is similar to the MOVEI rule except that we must use j as a pointer to the memory location that holds the value assigned to the register Ri rather than using j itself as the value. In addition we have an index register being used, which in MC0 always happens to be the stack pointer P.

JRST:

$$\text{assertion(l)} \{ < \text{'JRST 0 l} > \} Q$$

The JRST rule may be understood by viewing JRST as a go to. The rule states that regardless of what is so at the point in the program located after the go to l, the assertion at the label l must be true immediately before. It should be noted that the zero in the statement is optional.

JUMPE:

$$(Ri=0 \rightarrow \text{assertion(l)}) \wedge (Ri \neq 0 \rightarrow Q) \{ < \text{'JUMPE i l} > \} Q$$

JUMPE is the equivalent of the higher level language statement: IF Ri=0 THEN GO TO l. The standard Hoare rules for such a compound statement are:

$$\frac{P \wedge R \{ A \} Q,\ P \wedge \neg R \rightarrow Q}{P \{ \text{IF R THEN A} \} Q}$$

and

$$\text{assertion(l)} \{ \text{GO TO l} \} Q$$

We may combine these to get the Hoare rule:

$$\frac{P \wedge R \rightarrow \text{assertion(l)},\ P \wedge \neg R \rightarrow Q}{P \{ \text{IF R THEN GO TO l} \} Q}$$

The rule for JUMPE is the axiomatic form of this rule of inference.

JUMPN:

$$(Ri \neq 0 \rightarrow \text{assertion(l)}) \wedge (Ri=0 \rightarrow Q) \{ < \text{'JUMPN i l} > \} Q$$

The JUMPN instruction is the same as JUMPE except that the register is checked to be non-zero rather than zero.

CALL:

$$\text{Entry}(f) \quad \begin{array}{|l|l|l|l} \text{NIL} & \text{a1} & \ldots & \text{an} \\ 0 & \text{R1} & \ldots & \text{Rn} \wedge \end{array}$$

$$\text{(Exit}(f) \quad \begin{array}{|l|l|l|l} \text{NIL} & \text{h} & \text{a1} & \ldots \text{an} \\ 0 & \text{<f a1 ... an>} & \text{R1} & \ldots \text{Rn} \rightarrow \end{array}$$

$$\forall R2,\ldots,RN(f)\ (Q) \quad \begin{array}{|l} \text{R1} \\ \text{<f R1 ... Rn>)} \end{array}$$

$$\{ < \text{'CALL n <'E f> > } \} \ Q$$

where N(f) is the maximum number of registers modified during execution of function f, h is the designation in Exit(f) for the result of the function call to f, and a1 ... an are the formal arguments of f. This rule depends on the function linkage conventions used by compiled code. What the rule says is that any compiled function has the same Entry and Exit conditions as the corresponding source function, except the formal arguments are replaced by the registers and the constant NIL is replaced by 0. Further, register R1 will contain the function result, and the other registers used are quantified as variable parameters to the function. Quantification is one of the standard ways of handling variable parameters, as shown by the following higher level language Hoare rule for a call of procedure p with variable parameter a.

$$\frac{P(a)\ \{\ p(a)\ \}\ R(a)}{P(a) \wedge \forall a\ (R(a) \rightarrow S(a))\ \{\ \text{CALL } p(a)\ \}\ S(a)}$$

The upper line of this rule of inference is the proof of the procedure body with respect to the Entry P and Exit R. Note that the quantification makes the variable a appearing in S and the Exit into a different variable than the a in the Entry. The quantification in our CALL rule does not include the Exit because the source language does not have variable parameters. Hence all references to parameters in the Exit refer to the values of the parameters before the CALL is performed.

It might be questioned how the target code can treat the parameters of a function as if they were variable parameters when the source language does not allow parameters to be changed. Before the call target language statement is executed, the target code has placed the values of the parameters into temporary locations (the registers). The target code may then procede to overwrite the registers during evaluation of the function just as if the registers were variable parameters. This has no effect on the original actual parameters in the calling function.

SUB:

$$Q \quad \begin{array}{|l} \text{P} \\ \text{P-n } \{ < \text{'SUB 'P <'C 0 0 n n> > } \} \ Q \end{array}$$

The strange appearance of this instruction is due to the use of P as two half words.

The right half is actually the stack pointer, while the left is used for a hardware check on the stack size. We assume the stack size is within bounds in the proof of MC0 (otherwise the program would not terminate normally), and we completely ignore the left half. Thus the instruction may be viewed simply as the assignment P := P-n.

POPJ:

$$\begin{array}{l|l|l|l|l} & h & NIL & a1 & \ldots & an \\ Exit(f) & R1 & 0 & R1' & \ldots & Rn' \end{array} \quad \{ < \text{'POPJ 'P} > \} \; Q$$

where f is the function in which we find the POPJ instruction and h and the a's are as previously. The POPJ instruction may be viewed as a return statement, which explains this rule's resemblance to a return Hoare rule. As with the CALL statement, the locations in the registers of the arguments and function result must be taken into account, and the source constant NIL must be translated to 0.

PUSH:

$$Q \begin{array}{|l|l|} \hline m & P \\ \alpha(m,P,Ri) & P+1 \\ \hline \end{array} \quad \{ < \text{'PUSH 'P i} > \} \; Q$$

This instruction accomplishes the same as the assignments P := P+1; m[P] := Ri. The rule is seen to be the composition of the standard assignment rules for these two statements. As before, the notation α(m,P,Ri) means the array which is the same as m except m[P] has been replaced by Ri.

During the course of generating verification conditions we will assume the standard consequence and composition Hoare rules that apply to essentially all target languages.

$$\frac{P \rightarrow Q,\ Q \{ A \} R}{P \{ A \} R}$$

$$\frac{P \{ A \} Q,\ Q \rightarrow R}{P \{ A \} R}$$

$$\frac{P \{ A \} Q,\ Q \{ B \} R}{P \{ A\ B \} R}$$

where A B signifies that the instructions A and B are concatenated.

3.8 Two Part Proof

We divide the proof of a compiler into two distinct parts. The first proves what target code the compiler produces for each source language syntactic type. The second proves that the target language produced has the same effect as the source language. We find this division convenient because it divides the proof into parts that are easier to handle. It seems to be a natural division in that the two proof parts are independent of each other in the sense that the second part uses the result of the first part, but essentially none of the same methods or proof parts need be repeated.

The semantics of the language in which the compiler is written are needed in the first proof part to transform into verification conditions the assertions stating what the compiler produces. The first proof part, however, does not deal with the semantics of the source and target language (although the syntax of each is used).

The second proof part involves showing that the Hoare formula of the form compilation(P) {compilation(S)} compilation(Q) is satisfied. The value of compilation(S) has been established by the part one proof. To prove the target language Hoare formula, we will apply target language Hoare rules to it to produce a logical theorem which we will then prove. Thus the second proof part must use the semantics of the target and source language, but does not involve code in the language in which the compiler is written, nor its semantics.

3.9 Division of Labor

Our belief is that program proving should be done interactively. We believe that the human should be involved for the difficult and insightful portions of a proof, while the machine should handle routine and repetitive parts.

As a result of the division of the proof into two parts as described above, the first part is within the capabilities of present program proving systems. In a manner consistent with our philosophy, the first part of the proof of MC0 was in fact accomplished on the Xivus interactive program proving system.

The second proof part of the MC0 proof was tedious and could have used machine assistance. Unfortunately, it must be carried out at a level above that of existing program proving systems. The target code produced by the compiler is expressed in terms of the source code being compiled and the state of the compiler (symbol table, etc.). Thus we cannot use ordinary verification condition generators for part two, because they must operate on actual code, not an expression representing the code. The consequence of this and other features (a list of such features is in Section 7.2) missing from present proving systems is that part two of the MC0 proof was carried out as a hand proof rather than a machine-assisted proof. The proof of the McCarthy-Painter compiler was simple enough to coerce the Xivus system through the part two proof in order to show that the methods employed were mechanizable. In Section 7.2 we suggest ways that a mechanized system could be built to allow more complex part two proofs to be done interactively.

3.10 Brief Description of Xivus Verification System

The Xivus program verification (proving) system has the following major components:

1. A text editor
2. A parser for asserted programs written in the language Pascal [Wirth71,Jensen74]
3. A verification condition generator
4. A logical and arithmetic simplifier
5. A substitution package
6. An interactive theorem prover
7. An interactive top level program to direct and aid the proof

The system is the one described by Good, London, and Bledsoe [Good75] with further evolutionary changes. The verification condition generator is essentially that of Igarashi, Luckham, and London [Igarashi75], and the theorem prover was derived from the Bledsoe-Bruell prover [Bledsoe73].

Typically an asserted program is entered into the system via the editor and is then parsed, and its verification conditions are generated. The simplifier and substituter usually prove many of the verification conditions and shorten the remaining ones to more manageable size. Those remaining ones are then submitted to the theorem prover. Upon discovery of a false or unprovable verification condition, the problem must be located in either the program code or assertions, and the steps repeated. The system has some abilities to recognize portions of the proof which remain unchanged and thus need not be reproved.

3.11 Structural Induction

Structural induction has been used in practically all the proofs of operationally expressed compilers, including that of MC0. Structural induction may be stated as: If we have proved that a program works for each possible structural type of data while assuming the program works on the smaller pieces of data, then we have proved the program works. The basis for the induction lies in the proof of the structural types which do not contain other types. Structural induction is well suited to proving compilers because source languages are usually defined in terms of structural or syntactic types. Further, source languages are usually inductively defined, since some syntactic types may contain arbitrarily complex other syntactic types.

3.12 Optimizing Compilers

In treating separately each syntactic type of the source language, we assume that the results of compiling are truly separate. That is, the target code produced from compiling syntactic type X must not depend on whether that occurrence of X contains, for instance, type Y. Similarly the target code from type X should not depend on whether that occurrence of X is contained in, for instance, type Z. This independence may in fact be given as a definition of a non-optimizing compiler.

If the target code produced depends on what is contained in the syntactic type, it can be treated by the techniques here. For example, we could have a syntactic type X compile into

different code when its arguments are constants. This could be treated simply by defining a new syntactic type X-with-constants, and applying the techniques used here.

Optimizations in which the target code produced depends upon the context of its source would not be proved by the techniques demonstrated here, but as extensions to these techniques or as a separate proof of equivalence to the non-optimized version. More will be said about this in Section 7.3.

3.13 Axiomatic Description of Lisp

MC0 deals extensively with list structures, for recognizing source language syntactic types, for extracting parts from the source language, for building and using the location table, and for building the pieces of target language. The compiler was written in a modified Pascal which allows functions to return values of complex type. Dealing with lists was accomplished by functions with the same names and properties as basic Lisp functions, except having fixed numbers of arguments. Those Lisp-like functions were never supplied as Pascal code, nor were they given assertions. Therefore no properties of these functions appeared in the resulting verification conditions. A standard set of properties was entered directly into the theorem prover as rewrite rules to be used in the proofs of the verification conditions. Those rules may be found in Section A.3. In using this technique, we recognize that such rewrite rules can define functions as well as their code could, and further that in the environment of the proof of MC0 definition by rewrite rules is more easily used than the code would be. This technique can be used for functions which are to be considered intrinsic to the source language. In the verification strategy used in Alphard [Wulf76], this technique may be used on both intrinsic functions, such as those describing sequences, and on functions for which code is given by the programmer. In the latter case, though, the programmer must establish that the code is consistent with the axiomatic specifications.

3.14 Axiomatic Definition of Source Language Syntax

We use in the compiler assertions a set of functions (predicates) which tell us if any given source language expression is a certain syntactic type. For instance, in the proof of MC0, the function ISAND(S) is true if and only if source language expression S is an AND type. The definitions of such functions are not supplied as code, but as axioms to be used by the theorem prover in proving the verification conditions. For example, the following axioms are needed for the AND syntactic type. The dot after the X indicate universal quantification of the X over the entire axiom. The prefixed single quote, as in Lisp, indicates a constant.

```
1. ISAND(X.) → CAR(X.) = 'AND

2. ISAND(X.) →   NOT NULL(X.)
                ∧ NOT (X. = 'T)
                ∧ NOT NUMBERP(X.)
                ∧ NOT ATOM(X.)
```

Note that the first axiom says what the type is, and the second axiom says what the type is not. In fact the four conclusions of the second axiom represent exactly the four syntactic types which are checked by the compiler before checking the AND type. Most of the "what the type is not" properties could be derived from properties of Lisp functions involved, and therefore we could have used Lisp properties in the compiler proof instead. For example, our knowledge of Lisp would allow us to conclude that any form that has a CAR (such as AND) is not any of the atomic forms NIL, T, a number, or a general atom. However, we felt that the properties as expressed above are easily written by referring to the source language syntax definition, and that the properties as given are directly applicable in a compiler proof.

In order to express the fact that the arguments to an AND type consist of a list of valid expressions, we also need the axiom:

3. ISEXPRESSION(X.) ∧ CAR(X.) = 'AND → ISLISTOFEXP(CDR(X.))

There are also some axioms expressing that a LISTOFEXP expression consists of legal expressions. To accommodate the recursion on the number of arguments of AND, we need one further axiom:

4. ISLISTOFEXP(Z.) → ISAND(CONS('AND,Z.))

Similar sets of axioms are needed for the other syntactic types of the source language of MC0. A list of all such axioms is given in Section A.2.

3.15 Axiomatic Stack Proof

The proof of the compiler MC0 in Hoare rule terms makes certain assumptions about the run-time stack during the execution of certain strings of target code. These assumptions are explained precisely in Chapter 4, but may be roughly described by: the contents of the stack remain unchanged during execution of a string t of target code. That is, any items added to the stack by executing t must be removed, but none of the original items in the stack may be removed. When this property is true of a string t, we denote it by stackok(t). To discharge the proof of these assumptions we wish to describe (with axioms) whether certain strings of target language statements will, when executed, modify the stack, and if so, how the stack will change. The axioms will then be applied to the theorems to be proved to produce subgoals, to which further axioms are applied until all goals and subgoals are proved. The following is a brief description of the stack axioms used in the proof of MC0.

First we wish to give axioms describing how objects are pushed onto and popped off of the stack. The popping of the stack will be done with a single SUB instruction (even if several elements are to be popped), while the pushing onto the stack will be done by several PUSH instructions spread throughout the target code. In order to pop off the stack exactly the number of items earlier pushed onto it, we need a function that tells us how many PUSH instructions are contained in certain strings of target code. So we define containspushes(t,n) by:

C1: containspushes(<<'PUSH ... >>,1)

```
C2: containspushes(<<a ...>>,0) when a ≠ 'PUSH

C3: containspushes(t1,n1) ∧ containspushes(t2,n2) →
    containspushes(< !t1 !t2 >, n1 + n2 )
```

where t1 and t2 are lists of statements. Note that < !t1 !t2 > is the list containing all the statements in the list t1 followed by all the statements in the list t2. C1 and C2 give the containspushes property for strings that contain a single instruction, and C3 allows us to combine strings to arbitrary numbers of target instructions.

With the containspushes property we may then axiomatize the stackok property we described above. For reasons of function linkage explained more fully in Chapter 4 we also define stackokreturns, which describes a string of target code which has the stackok property with the exception of an additional concluding POPJ statement. POPJ returns to a location which is taken from the stack.

The following axioms then describe stackok and stackokreturns. Explanations of the axioms are found below.

```
S1: stackok(t) → stackokreturns(< !t '<POPJ P>>)

S2: stackok(t1) ∧ containspushes(t2,n) →
    stackok(< !t2 !t1 <'SUB 'P <'C 0 0 n n>>>)

S3: stackok(t1) ∧ stackok(t2) → stackok(< !t1 !t2 >)

S4: stackok(<<'CALL ...>>)

S5: stackok(<<'MOVE ...>>)

S6: stackok(<<'MOVEI ...>>)

S7: stackok(< l >)
```

where l is a label

```
S8: stackok(< < 'JRST 0 l > !t l >)
```

where t is a list of zero or more statements and l is a label

```
S9: stackok(t1) ∧ stackok(< !t2 l !t1 >) →
     stackok(< < 'JUMPx i l > !t2 l !t1 >)
```

where 'JUMPx means 'JUMPE or 'JUMPN, and t1 and t2 are lists of zero or more statements.

S1 is the definition of stackokreturns in terms of stackok. In S2 we state that n pushes to the stack must be balanced by a later pop of n items, with the intervening instructions

leaving the stackok. Concatenation of stackok strings is allowed by S3. We list certain target instructions (including labels) that are stackok in S4 through S7. In S8 we state that a string of statements is stackok if we execute an unconditional jump over all of them. S9 requires that both paths which a conditional jump might take must be stackok for the entire assembly of statements to be stackok.

It might be surmised that S9 could have been more simply stated as:

```
S9S: stackok(t2) → stackok(< < 'JUMPx i l > !t2 l >)
```

While this simpler axiom is true and would be useful in our proofs, it is not strong enough to prove some cases that arise. The shortcomings of S9S become obvious when it is realized that stackok(t2) may be undefined. For example, t2 can contain a jump to a point in t1. Then none of our axioms will allow us to define stackok(t2), but use of axiom S8 (possibly in combination with S3 and others) will allow us to define stackok of the combination of t1, the label l, and t2.

It may be noted that the simple forms of jumps output by the compiler are always forward jumps, and hence there are no loops. Also, all execution paths that separate at a conditional jump eventually rejoin below. It is this simplicity of target code that allows comparatively simple axioms defining stackok with respect to jump statements. We do not need to resort to finding execution paths or loops.

The fact that we will complete the stackok proofs of MC0 with only these stackok axioms will ensure that, as we have casually assured the reader, there are no backward jumps in the target code produced by MC0. If there were backward jumps (at least in code that is required to satisfy the stackok property), no axioms would apply to such code, and we would be unable to continue a stackok proof. Only S8 and S9 have jumps in them, and both have the forward direction of the jump built into them. Similar reasoning using axiom S9 ensures that paths which split must rejoin below.

3.16 The F Functions

In several places in the assertions we must speak just of the new target code produced by a specific call to a compiler procedure. The arrangement of the compiler MC0 does not result in any compiler variable holding just the code resulting from a given routine, so we are not able to use a compiler variable name for such references. Instead we pass an output file as a variable parameter to many of the procedures of the compiler, to which those procedures add statements of target code. Rather than try to extract from this output file the target code which was added to the file since some earlier time, we will arbitrarily give that added code a symbolic name. The name "F function" comes about because we have prefixed the letter F to associated procedure names to create a unique new name by which new target code may be referenced.

These F functions are an extension of Hoare and Wirth's method for variable parameters [Hoare73]. In their method, the existence of a function is assumed in order to express the final value of a variable parameter to a procedure as a function of the initial values of the parameters. These functions are given arbitrary names for use in program proving, and, except in very simply analyzable situations, these functions (or at least certain

important properties of them) must be supplied by the prover of the program. The F functions differ from the Hoare and Wirth functions in that ours refer only to the part added to the parameter rather than the entire final value.

The F functions are actually formally defined by the use of the assume function in the theorem prover when proving the resulting verification conditions. In other words, we use a new function name in the assertions about the compiler without supplying code or assertions describing that function. Then in the proof of the resulting verification conditions we may assume certain properties about those functions, and that assumption constitutes the definition of those functions.

An example of this technique occurs in proving the case involving the AND syntactic type of the compiler MC0. The following assertion is needed in procedure COMPEXP in order to carry out the proof of other assertions.

```
OUTFILE =
< ! OUTFILE'
  ! FCOMPEXP(EXP,M,LOCTABLE)
>
```

where the suffixed single quote mark indicates initial value of a variable parameter.

One clause of one of the verification conditions resulting from that assertion requires that we prove the following conclusion under the conditions of this subcase (syntactic type is AND, more than zero arguments).

```
< ! OUTFILE'
  ! FCOMPEXP(EXP,M,LOCTABLE)
>

=

< ! OUTFILE'
  ! FCOMPANDOR(CDR(EXP),M,L1,FALSE,LOCTABLE)
  < 'MOVEI 1 < 'QUOTE 'T > >
  < 'JRST 0 L2 >
  L1
  < 'MOVEI 1 0 >
  L2
>
```

Recalling that FCOMPEXP has not yet been formally defined, we see that assuming this subgoal in the theorem prover simply defines FCOMPEXP for this subcase as:

```
< ! FCOMPANDOR(CDR(EXP),M,L1,FALSE,LOCTABLE)
  < 'MOVEI 1 < 'QUOTE 'T > >
  < 'JRST 0 L2 >
  L1
  < 'MOVEI 1 0 >
  L2
>
```

The full set of definitions for FCOMPEXP and other F functions for all cases in the compiler MC0 is given in Section A.1. We assume the value of FCOMPEXP (or any other F function) only once for each case or subcase. The mutually exclusive definitions of source language syntactic types assures us that no two definitions apply for any given piece of source language, thus eliminating the possibility of this method producing an inconsistent definition of an F function.

3.17 The Gensym Problem

Gensym is a Lisp function of no arguments that produces a unique identifier every time it is called. The compiler MC0 uses it to produce unique labels in the target code. Unfortunately it is not a mathematical function because it has the side effect of causing the next call to it to produce a different value. It is the opposite of the aliasing problem (multiple names for one value), because gensym has one name that may represent multiple values. It complicates proving terribly if we cannot depend on F(X) = F(X), i.e., F=F.

One solution to this would be to write a gensym procedure complete with a variable parameter that is passed everywhere throughout the program and incremented at every gensym call. Its ever-changing value would present a base from which to construct an ever-changing identifier. We think this solution is too low-level and messy for easy proof. Instead we chose to use the verification condition generator mechanism for variable parameters to procedures, which assigns unique identifiers to represent them after each procedure call within a program unit. Therefore gensym was made a procedure with its former function value passed back as a variable parameter instead. Then each call to gensym within a program unit got its parameter renamed uniquely and consistently in the verification conditions produced.

Another problem associated with gensym is the violation of the assumption that procedure Exit assertions are expressed in terms of only the procedure parameters (and possibly their initial values). A reference in an Exit assertion to a variable existing only inside the procedure (i.e., a local variable, not a parameter) could clash with another variable of the same name in any other procedure calling it. For this reason, the Exit assertion must be considered to be outside the scope of locally declared variables. However, it is often necessary in the compiler MC0 to refer in an Exit assertion to the variable produced by a gensym call within a procedure. For example, the Exit assertion of COMPEXP (the main compiling routine for expressions) for the case of a COND syntactic type with no arguments is:

```
ISCOND(EXP) ∧ NULL(CDR(EXP)) →
OUTFILE =
< ! OUTFILE'
  L5
>
```

The suffixed single quote mark indicates initial value of a variable parameter. The L5 is a compiler variable into which gensym returns a unique name for use as a target code label.

When COMPEXP is called by a routine, say X, the verification condition being generated in X will have added to it the properties from COMPEXP's Exit assertion. Thus L5 from within COMPEXP could clash with an L5 in X; that is, we could have L5 referring to two different objects in the same verification condition. This would invalidate the proof, assuming we actually use those clashing uses of L5 in the proof of the verification condition. However, if we proved a verification condition, for instance, by means of its hypothesis being false, the proof would remain valid even if clashing variable uses occurred in the conclusion of that verification condition, since we did not actually use the conclusion.

To prevent clashes from even occurring in most verification conditions, we have named uniquely the arguments to gensym on all the calls throughout the compiler. Thus, in the above example from COMPEXP, no other routine will contain a variable name that could clash with gensym argument L5 introduced when COMPEXP is called. Even with the unique argument naming, two possible sources of clash remain that could invalidate a proof. One is the case of a recursive call. For example if COMPEXP calls itself, the L5 from the lower call will be introduced into the verification condition that may already refer to L5 in the top level instance of executing COMPEXP. The other possibility is that a verification condition could involve two calls to the same routine, and therefore introduce the gensym argument variable twice to mean two different scopes of that variable.

The part one proofs were examined for cases where a gensym variable name was introduced into a verification condition by a procedure call so that a gensym variable clashed with a variable name already used to mean something else. In every case where this did occur, the proofs were carried out without making any use of the clauses in which the clashing reference occurred. Thus the proofs may be considered valid. The problem of searching for such clashes was simplified greatly by the separate asserting and proving of each syntactic type. Had assertions for all cases been entered at once, a great many more clashes would have occurred, but all in clauses of the verification conditions that pertained to other cases and thus would not be used in the proof of that case.

An alternative method of avoiding such clashes would have been to pass all gensym arguments as variable parameters to every procedure that either used them or called a procedure that used them throughout MC0. Then the variable parameter mechanism of the verification condition generator would rename them to prevent clashes. Unfortunately it would also greatly increase the size of the verification conditions resulting. It appears that the examination to determine that clashes did not occur was indeed easier than it would have been to wade through more complex verification conditions at nearly all stages of the part one proof.

As mentioned above, all references in an Exit assertion to local variables must normally be eliminated. A local reference is eliminated by using an expression of the input parameters

that is equal to the local variable. But the gensym variable must be referred to in an Exit assertion by its name, not as a function of the procedure's input parameters, because the gensym variable, by the design of gensym, bears no relationship to previously known values. That is, the only requirement on gensym is that its output be different from previous output. This is why the gensym variables are different than other local variables in that they cannot be expressed as functions of the input parameters and therefore must appear in the Exit assertion.

4 RESULTS

4.1 Derivation and Statement of the Main Result

The function of a compiler is to translate a program from a language for which we do not have a suitable means of executing it (the source language) into another language for which we do (the target language). Therefore the statement of correctness of a compiler is that for all source programs the target program produced by the compiler produces the same results when run as the source program would if it were run. We will precisely define the effects of executing a statement by the use of Hoare type proof rules. Thus our first attempt at the statement of correctness for compiler MC0 would be:

For all programs A ((P{A}Q) → (P{COMPILATION(A)}Q))

We will always assume that the source program A is a legally executable program. Preventing compile-time and run-time errors is a problem which we will not address. We simply wish to show that if a source program can be proved to accomplish a given effect by use of the Hoare formalism, then its compilation will accomplish the same effect.

In order to tailor our statement of correctness to structural induction, we will express that statement in terms of the source language structural units, expressions and statements, rather than in program terms. The statement of correctness becomes:

For all statements or expressions S ((P{S}Q) → (P{COMPILATION(S)}Q))

In the proof of compiler MC0 we will first treat the one syntactic type of our source language that is a statement, the function definition. For it we have in the source language

Entry(f) {S} Exit(f)

where S is of the form < 'DE f <a1 ... an> exp >.

The Entry and Exit represent the only P and Q for which P{S}Q holds when S is a function definition. Thus the attempt at a statement of correctness is:

ISFUNCTIONDEF(S) → (Entry(f) {COMPILATION(S)} Exit(f))

This statement of correctness suffers from having made the assumption that the assertions may be stated in the same terms for both source and target languages. This is typically not the case. The source language will usually use symbolic names for variables, while the target language will usually use memory locations, or at least indirect memory references, such as locations relative to a stack pointer.

A similar problem occurs for constants in the source language. For compiler MC0, only

one constant is represented differently in the source and target languages. So we must substitute 0 (the target representation) for NIL (the source representation) in assertions.

The function calling convention of a compiler generally defines a fixed group of locations in the target machine in which we will expect to find the arguments to a called function. For MC0, the arguments will be passed in the registers, which we designate as R1, R2, ... , Rn. Similarly the result of a function will be passed back in register one (R1).

Thus for a function definition statement S, we will state the correctness of the compiler MC0 as (using the substitution notation of Section 3.6):

$$\text{Entry}(f)\ \begin{vmatrix}\text{NIL} & \text{a1} & \ldots & \text{an}\\ 0 & \text{R1'} & \ldots & \text{Rn'}\end{vmatrix}\ \{\text{COMPILATION}(S)\}$$

$$\text{Exit}(f)\ \begin{vmatrix}\text{h} & \text{NIL} & \text{a1} & \ldots & \text{an}\\ \text{R1} & 0 & \text{R1'} & \ldots & \text{Rn'}\end{vmatrix}$$

where S is of form < 'DE f <a1 ... an> exp >, the identifier h is the designation used in Exit(f) for the function value returned by the function f, and Ri' is the initial value of register Ri. Since the initial and present values of the registers are the same at the time of the Entry assertion, the single quotes signifying initial values are unnecessary for the Entry. However, they will make clearer the fact that we are referring to initial values during the proof, so they will be used here. It might be noted that h must be an identifier, not an expression, because the substitution formalization is valid only when substituting for identifiers.

Note that since the registers (Ri's) are distinct from the formal arguments (aj's), we may make the substitution of registers for arguments either sequentially or simultaneously. We choose sequentially because more of the substitution simplifications apply to sequential forms. Note also that Entry and Exit can be functions of only formal arguments; Exit may also include the returned function value. There are no global or free variables inherited in this source language. Therefore we have renamed into target language all variable names in Entry and Exit.

Now the question arises, by what name in the MC0 compiler code is the compilation of S returned? The answer is almost COMP(S,OUTFILE). What COMP(S,OUTFILE) actually returns is the initial value of the sequential output file OUTFILE with the compilation of S appended to the end. Therefore we will define a new name to describe exactly the compilation of S, rather than almost what we want. FCOMP(S) is (by definition) the target code added to OUTFILE to obtain the returned value of COMP(S,OUTFILE). Because of the way we wrote the Exit assertion for COMP, it is obvious what portion of that assertion is represented by FCOMP.

Having arrived at the statement of correctness for MC0 in the case of the function definition, we now turn to the case of expressions. The source language Hoare rule for expressions must account for the Entry and Exit conditions of the functions contained in the expression, and the way in which they are nested. We will define below a precondition Pre and a postcondition Post to accomplish this. The source language Hoare rule is:

$$\text{Pre}(S) \wedge (\text{Post}(S) \rightarrow Q)\ \{S\}\ Q$$

where S is of form <f b1 ... bn>, f has been defined (by the user or the basic Lisp language definition) with formal arguments a1, ..., an, and

$$Pre(S) = Pre(b1) \wedge \ldots \wedge Pre(bn) \wedge$$

$$(Post(b1) \wedge \ldots \wedge Post(bn) \rightarrow Entry(f) \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix},$$

and

$$Post(S) = Post(b1) \wedge \ldots \wedge Post(bn) \wedge Exit(f) \begin{vmatrix} a1 & \ldots & an & h \\ b1 & \ldots & bn & S \end{vmatrix}$$

Note that the substitutions of actual for formal arguments must be simultaneous. The substitution for the result h may be done after the argument substitution (rather than simultaneously) only by assuming h is distinct from the a's and b's. This can be made so by renaming h for purposes of this proof.

The Pre(S) then represents the collection of all necessary preconditions nested in the expression, while Post(S) is the collection of all results in the expression. There will be shortened forms of these definitions of Pre and Post in cases (AND, OR, COND) where not all arguments are necessarily evaluated. Shortening of these forms may also occur for certain functions whose Entry or Exit is TRUE, allowing logical simplification. A full accounting of Pre and Post for all cases of MC0 is given in figure 4-1.

It should be noted that this form of Hoare rule nests the Entry and Exit conditions in exactly the order that they are nested in the expression S. This allows the source language to contain dependencies between the Entries and Exits. The dependencies might be of the kind where the Entry condition of a function is implied by the Exit condition of its argument. For example, an Exit condition of the Lisp function CONS is that its result is not atomic, which would satisfy the Entry condition of CAR (that its argument not be atomic) in the expression CAR(CONS(X,Y)). It is clear by the duplication of the original nesting of the source expression into the Pre and Post conditions that this Hoare rule formulation will properly account for these dependencies.

Our first attempt at a statement of correctness for expressions is:

$$ISEXPRESSION(S) \rightarrow (Pre(S) \wedge (Post(S) \rightarrow Q)\ \{COMPILATION(S)\}\ Q\ )$$

But now we must return to the old problem of renaming variables to target language locations. An expression may use any variable names declared by a containing function definition or lambda expression.

Thus we will define a list v of variable names declared and a list w of memory locations. The list v will have all newly declared variables added to its beginning, and the corresponding location assigned will be added to the beginning of w simultaneously. Similarly items will be removed when we leave their scope. Thus this pair of lists, dynamic during compilation of a program, will have in v the name of any variable that may be used in an expression, and will have in the corresponding position in w the target language location assigned. If a variable is declared more than once, the most recent one is valid, so we must use the first time that variable appears in v.

Fortunately the concept of such a pair of lists has already been invented and is called a

symbol table. MC0, like most compilers, uses a symbol table, so we will be able to extract v and w from LOCTABLE, the variable within the MC0's data that holds the current symbol table. Because MC0 uses locations relative to a stack pointer, we will have to make a relative-to-absolute adjustment to put actual locations in w. But other than that, extracting v and w from LOCTABLE will be simply the extraction of a pair of lists from a list of pairs.

LOCTABLE holds Lisp dotted pairs of associated variable names and locations in the form:

< <NAME1 . LOC1> ... <NAMEr . LOCr> >

Thus v and w will be given by:

v = < NAME1 ... NAMEr >

w = < m[M+P+LOC1] ... m[M+P+LOCr] >

where m designates an array of memory used as a stack. When variables are declared, MC0 allocates space for them in m beginning at m[P+1], since P is a pointer to the last used location in m. The LOCi locations are relative to the run-time value of P at entry to the present function. Since P will be updated as execution proceeds, we will keep track of how far different the present value of P is from the initial (function entry) value with a variable called M. It is minus the number of stack locations locally used, so that M+P is the initial value of P.

Thus the statement of correctness of MC0 requires the substitution of locations from w for the variable names of v in source assertions to get target assertions. Since the notation we will use for locations will not be the same as that used for source variables, the substitution may be simultaneous or sequential. Again we will choose sequential to ease the simplification of substitutions. The statement of correctness for expressions is now:

ISEXPRESSION(S) →

$$(Pre(S) \left|\begin{matrix} v \\ w \end{matrix}\right| \begin{matrix} NIL \\ 0 \end{matrix} \wedge (Post(S) \left|\begin{matrix} v \\ w \end{matrix}\right| \begin{matrix} NIL \\ 0 \end{matrix} \rightarrow Q \left|\begin{matrix} v \\ w \end{matrix}\right| \begin{matrix} NIL \\ 0 \end{matrix})$$

$$\{COMPILATION(S)\}\ Q \left|\begin{matrix} v \\ w \end{matrix}\right| \begin{matrix} NIL \\ 0 \end{matrix}$$

However, this ignores the fact that expression values are accessed by the expression itself in source language, but by use of register one (R1) in target language. Thus we must substitute R1 for all occurrences in Q of the expression in question. We do not have a notation for substitution for an expression. The substitution notation and its simplification rules used here are only valid for substitution for an atom. But we will avoid the need to express or simplify such substitution by letting the variable T (not to be confused with the Lisp constant T) represent the result of substituting R1 for the expression in question. Further, we will assume that T has already had w substituted for v and 0 for NIL. Thus T is the target language equivalent of the source language assertion Q.

Then before the curly brackets of the Hoare formula we can say that T holds if we substitute in it the expression (translated to target language terms) for R1. Thus we get:

ISEXPRESSION(S) →

$$(\mathrm{Pre}(S)\ \Big|\Big|{}^{v}_{w}\Big|{}^{\mathrm{NIL}}_{0}\ \wedge\ (\mathrm{Post}(S)\ \Big|\Big|{}^{v}_{w}\Big|{}^{\mathrm{NIL}}_{0}\ \rightarrow T\ \Big|{}^{R1}_{S}\ \Big|\Big|{}^{v}_{w}\Big|{}^{\mathrm{NIL}}_{0}\ )$$

{COMPILATION(S)} T)

Note the strong resemblance of this formula to the Hoare rule that would result from assigning

$$S\ \Big|\Big|{}^{v}_{w}\Big|{}^{\mathrm{NIL}}_{0}$$

(the target language equivalent of S) to a variable called R1, where the substituted S is a function call with a precondition and postcondition.

One further problem remains. The action in target language of an expression evaluation involving a function call resembles a procedure call with variable arguments rather than a function call with constant arguments. That is, registers R2 ... Rm may have their values changed during evaluation of the function call. Of course R1 will be changed also, but its value will be set to the value returned by the expression. Note that m is not necessarily the same as n, the number of arguments, because nested function calls may wipe out more or fewer than n registers.

Thus in our proof of MC0 we must not allow the code to use the previous value of the registers R2 ... Rm. We ensure this with the same technique used for variable procedure arguments, that of quantifying the registers that are subject to new values during function execution.

We define N(S) as the number of such registers. It is precisely defined for various syntactic types in figure 4-2. Note that N(S) < 2 means no registers (other than R1) are subject to change and no quantification is then intended by the notation ∀R2,...,RN(S).

To avoid confusion between the argument of N and the expression quantified, we will always leave a space before the expression quantified. For example, ∀R2,...,RN(S) (T) means ∀R2,...,Rm (T) where m = N(S).

In a manner similar to FCOMP, we define a new function FCOMPEXP(S,M,LOCTABLE) to be the target code added to OUTFILE to obtain the returned value of COMPEXP(S,M,LOCTABLE,OUTFILE). The final version of the Hoare formulas we must prove for the correctness of MC0 is then:

1) ISFUNCTIONDEF(S) →

$$(\text{Entry}(f)\ \begin{vmatrix}\text{NIL} & \text{a1} & \dots & \text{an}\\ 0 & \text{R1'} & \dots & \text{Rn'}\end{vmatrix}\ \{\text{FCOMP}(S)\}$$

$$\text{Exit}(f)\ \begin{vmatrix}\text{h} & \text{NIL} & \text{a1} & \dots & \text{an}\\ \text{R1} & 0 & \text{R1'} & \dots & \text{Rn'}\end{vmatrix}\ )$$

where S is of form < 'DE f <a1 ... an> exp >, h is the designation used in Exit(f) for the function value returned by f, and Ri' is the initial value of register Ri.

2) ISEXPRESSION(S) →

$$(\text{Pre}(S)\ \begin{vmatrix}\text{v} & \text{NIL}\\ \text{w} & 0\end{vmatrix}\ \wedge\ (\text{Post}(S)\ \begin{vmatrix}\text{v} & \text{NIL}\\ \text{w} & 0\end{vmatrix}\ \rightarrow$$

$$\forall R2,\dots,RN(S)\ (T\ \begin{vmatrix}\text{R1} & \\ \text{S} & \begin{vmatrix}\text{v} & \text{NIL}\\ \text{w} & 0\end{vmatrix}\end{vmatrix}\ ))\ \{\text{FCOMPEXP}(S,M,\text{LOCTABLE})\}\ T\ )$$

where LOCTABLE is of form < < NAME1 . LOC1 > ... < NAMEr . LOCr > >, v = < NAME1 ... NAMEr >, w = < m[M+P+LOC1] ... m[M+P+LOCr] >, m is the array of memory used as a stack, M is minus the size of that portion of the stack containing locally declared variables, P is the stack pointer, Pre(S) is the collection of preconditions nested in S (this is precisely defined in figure 4-1), Post(S) is the collection of postconditions nested in S (see figure 4-1), and N(S) is the maximum number of registers that are modified during execution of the compilation of S (this is precisely defined in figure 4-2).

It might be pointed out that for MC0 these formulations of correctness (conditions 1) and 2)) require the target language program to terminate if the source language program did. The reason for this is that Luckham and Suzuki have shown [Luckham77] that Hoare rule proving systems are adequate to show termination (with the use of recursion and loop counters), coupled with the fact that MC0 does not introduce into the target program any sources of nontermination that were not in the source program. That is, there are no loops or recursions in the compilation of any syntactic types except the same recursive calls that appear in the source language. This assumes that all target language instructions involved always terminate. It is often true of compilers that no new sources of nontermination are introduced, except when source language operations are implemented by loops or recursion using simpler operations--for example, implementing exponentiation by a loop containing multiplication. In such a case, preservation of termination between the source and target language programs would require a proof of the termination of any new (not appearing in the source language program) loops or recursion that the compiler could produce in a target program.

There are some assumptions made about the stack in this formulation which we must identify and justify. No distinction has been made in the Hoare formulas between the values of P, the stack pointer, before and after the curly brackets containing target code. Similarly the distinction is not made about m, the stack array. This implies that m has the same value before execution of this target code as it has after, and similarly that P has the same value

before and after. But; in fact, both P and m are subject to change. So in order to use the given Hoare formulas as statements of correctness, we must show that the values of P and m after evaluation of each expression have been restored to the values before. Because all values within the stack m that we may subsequently use lie somewhere in the segment of m from m[1] to m[P], it is sufficient to show that this segment of m remains unchanged rather than all of m. It is often the case during execution of an expression evaluation that items will be added to the stack m beyond m[P] and that P will be raised. But we must show that at the end of that expression evaluation the value in P is returned to its former value, thus removing (or more precisely abandoning) items beyond m[P], and show that the segment of m up to m[P] was not changed.

The only two instructions produced by MC0 that set stack locations are PUSH and CALL. They set only the location after the one to which P is pointing before the instruction is executed. Thus we can see that a sufficient condition to accept the previously given statement of compiler correctness is to show that P never drops below its beginning value during evaluation of an expression, and that it returns with exactly the initial value. We denote as stackok(z) this property of target code z.

One further property of the compiled code is required. The generally accepted understanding of functions, as well as the Hoare formalization of functions, implies that a function, when its execution is completed, always returns to the point in the code from which it was called. The target code must explicitly perform this returning as a go to type of statement. Since the returning is to be accomplished by leaving the return point on the stack, we will combine its proof with the stackok proof. The mechanism used by target code is that a CALL instruction not only jumps to the code for the called instruction, but pushes onto the stack a location to which the called function is to return. Thus every compiled function must end its execution with a POPJ instruction, which removes one item from the stack and jumps to where that item points. Target code which is stackok except for a concluding POPJ instruction will be called stackokreturns.

These properties stackok and stackokreturns are defined by a set of axioms. The proof of MC0 then requires that the following two conditions be proved for each syntactic type:

3) ISFUNCTIONDEF(S) → stackokreturns(FCOMP(S))

4) ISEXPRESSION(S) → stackok(FCOMPEXP(S,M,LOCTABLE))

The conditions 1) through 4) then constitute the statement of correctness of the compiler MC0 and will be proved for all syntactic types of source code.

FIGURE 4-1

Definitions of Pre(S) and Post(S) for the syntactic cases of S:

case of NIL:
    Pre(S) = TRUE
    Post(S) = TRUE

case of T:

```
Pre(S) = TRUE
Post(S) = TRUE
```

ISNUMBER case:

```
Pre(S) = TRUE
Post(S) = TRUE
```

ISIDENTIFIER case:

```
Pre(S) = TRUE
Post(S) = TRUE
```

AND (no arguments) subcase:

```
Pre(S) = TRUE
Post(S) = (S = <'QUOTE 'T>)
```

where <'QUOTE 'T> is the Lisp constant for TRUE.

AND (n > 0 arguments) subcase:

```
Pre( < 'AND b1 b2 ... bn > ) =
  Pre(b1) ∧ (b1 ≠ NIL → Pre( < 'AND b2 ... bn > )

Post( < 'AND b1 b2 ... bn > ) =
  (b1 = NIL → S = NIL) ∧
  (b1 ≠ NIL → S = < 'AND b2 ... bn >) ∧
  Post(b1) ∧
  (b1 ≠ NIL → Post( < 'AND b2 ... bn > ))
```

OR (no arguments) subcase:

```
Pre(S) = TRUE
Post(S) = (S = NIL)
```

OR (n > 0 arguments) subcase:

```
Pre( < 'OR b1 b2 ... bn > ) =
  Pre(b1) ∧ (b1 = NIL → Pre( < 'OR b2 ... bn > )

Post( < 'OR  b1 b2 ... bn > ) =
  (b1 ≠ NIL → S = <'QUOTE 'T>) ∧
  (b1 = NIL → S = < 'OR b2 ... bn >) ∧
  Post(b1) ∧
  (b1 = NIL → Post( < 'OR b2 ... bn > ))
```

NOT case:

```
Pre( < 'NOT b1 > ) = Pre(b1)

Post( < 'NOT b1 > ) = (b1 = NIL → S = <'QUOTE 'T>) ∧
                      (b1 ≠ NIL → S = NIL) ∧
                      Post(b1)
```

COND (no arguments) subcase:

$$Pre(S) = TRUE$$
$$Post(S) = (S = UNDEFINED)$$

COND (n > 0 arguments) subcase:

$$Pre(S) = Pre(c1) \wedge (c1 \neq NIL \rightarrow Pre(d1)) \wedge (c1 = NIL \rightarrow Pre(\langle\ 'COND\ \langle c2\ d2\rangle\ \ldots\ \langle cn\ dn\rangle\ \rangle))$$

$$Post(S) = Post(c1) \wedge (c1 \neq NIL \rightarrow Post(d1) \wedge S = d1) \wedge (c1 = NIL \rightarrow Post(\langle\ 'COND\ \langle c2\ d2\rangle\ \ldots\ \langle cn\ dn\rangle\ \rangle) \wedge S = \langle\ 'COND\ \langle c2\ d2\rangle\ \ldots\ \langle cn\ dn\rangle\ \rangle)$$

where S is of form < 'COND <c1 d1> <c2 d2> ... <cn dn> >.

QUOTE case:

$$Pre(S) = TRUE$$
$$Post(S) = TRUE$$

case of a function call:

$$Pre(S) = Pre(b1) \wedge \ldots \wedge Pre(bn) \wedge (Post(b1) \wedge \ldots \wedge Post(bn) \rightarrow Entry(f) \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix})$$

$$Post(S) = Post(b1) \wedge \ldots \wedge Post(bn) \wedge Exit(f) \begin{vmatrix} a1 & \ldots & an & h \\ b1 & \ldots & bn & S \end{vmatrix}$$

where S is of form <f b1 ... bn>

case of a lambda:

$$Pre(S) = Pre(b1) \wedge \ldots \wedge Pre(bn) \wedge (Post(b1) \wedge \ldots \wedge Post(bn) \rightarrow Pre(exp) \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix})$$

$$Post(S) = Post(b1) \wedge \ldots \wedge Post(bn) \wedge Post(exp) \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix}$$

where S is of form < <'LAMBDA <a1 ... an> exp> b1 ... bn >

FIGURE 4-2

Definitions of N(S) for the syntactic cases of S:

case of NIL:

$$N(S) = 1$$

case of T:

$$N(S) = 1$$

ISNUMBER case:

```
N(S) = 1
```

ISIDENTIFIER case:

```
N(S) = 1
```

AND (no arguments) subcase:

```
N(S) = 1
```

AND (n > 0 arguments) subcase:

```
N( < 'AND b1 b2 ... bn > ) =
   If b1 = NIL then N(b1) else max(N(b1),N( < 'AND b2 ... bn > ))
```

OR (no arguments) subcase:

```
N(S) = 1
```

OR (n > 0 arguments) subcase:

```
N( < 'OR b1 b2 ... bn > ) =
   If b1 ≠ NIL then N(b1) else max(N(b1),N( < 'OR b2 ... bn > ))
```

NOT case:

```
N( < 'NOT b1 > ) = N(b1)
```

COND (no arguments) subcase:

```
N(S) = 1
```

COND (n > 0 arguments) subcase:

```
N( < 'COND <c1 d1> <c2 d2> ... <cn dn> > ) =
   If c1 ≠ NIL then max(N(c1),N(d1))
   else max(N(c1),N( < 'COND <c2 d2> ... <cn dn> > ))
```

QUOTE case:

```
N(S) = 1
```

case of a function call:

```
N( <f b1 ... bn> ) = max(N(f),N(b1),...,N(bn))
```

case of a lambda:

```
N( < <'LAMBDA <a1 ... an> exp> b1 ... bn > ) =
   max(N(exp),N(b1),...,N(bn))
```

FIGURE 4-3

Assertions about the compilation of code EXP for each syntactic case:
case of a function definition:

```
ISFUNCTIONDEF(EXP) →
OUTFILE =
< ! OUTFILE'
  < 'LAP CADR(EXP) 'SUBR >
  ! FMKPUSH(LENGTH(CADDR(EXP)),1)
  ! FCOMPEXP(CADDDR(EXP),
             -LENGTH(CADDR(EXP)),
             PRUP(CADDR(EXP),1),
            )
  < 'SUB 'P < 'C 0 0 LENGTH(CADDR(EXP)) LENGTH(CADDR(EXP)) > >
  < 'POPJ 'P >
  'NIL
>
```

case of NIL:

```
ISNIL(EXP) →
OUTFILE =
< ! OUTFILE'
  < 'MOVEI 1 0 >
>
```

case of T:

```
IST(EXP) →
OUTFILE =
< ! OUTFILE'
  < 'MOVEI 1 < 'QUOTE 'T > >
>
```

ISNUMBER case:

```
ISNUMBER(EXP) →
OUTFILE =
< ! OUTFILE'
  < 'MOVEI 1 < 'QUOTE EXP > >
>
```

ISIDENTIFIER case:

```
ISIDENTIFIER(EXP) →
OUTFILE =
< ! OUTFILE'
  < 'MOVE 1 RETRIEVE(EXP,M,LOCTABLE,OUTFILE') 'P >
>
```

AND (no arguments) subcase:

```
ISAND(EXP) ∧ NULL(CDR(EXP)) →
OUTFILE =
< ! OUTFILE'
  < 'MOVEI 1 < 'QUOTE 'T > >
  < 'JRST 0 L2 >
  L1
  < 'MOVEI 1 0 >
  L2
>
```

AND (n > 0 arguments) subcase:

```
OUTFILE =
< ! OUTFILE'
  ! FCOMPEXP(EXP,M,LOCTABLE)
>

ISAND(EXP) →
OUTFILE =
< ! OUTFILE'
  ! FCOMPANDOR(CDR(EXP),M,L1,FALSE,LOCTABLE)
  < 'MOVEI 1 < 'QUOTE 'T > >
  < 'JRST 0 L2 >
  L1
  < 'MOVEI 1 0 >
  L2
>

ISAND(EXP) ∧ NOT NULL(CDR(EXP)) →
OUTFILE =
< ! OUTFILE'
  ! FCOMPEXP(CADR(EXP),M,LOCTABLE)
  < 'JUMPE 1 L1 >
  ! FCOMPEXP(< 'AND ! CDDR(EXP) >,M,LOCTABLE)
>
```

OR (no arguments) subcase:

```
ISOR(EXP) ∧ NULL(CDR(EXP)) →
OUTFILE =
< ! OUTFILE'
  < 'JRST 0 L1 >
  L4
  < 'MOVEI 1 < 'QUOTE 'T > >
  < 'JRST 0 L2 >
  L1
  < 'MOVEI 1 0 >
  L2
>
```

OR (n > 0 arguments) subcase:

```
OUTFILE =
< ! OUTFILE'
  ! FCOMPEXP(EXP,M,LOCTABLE)
>

ISOR(EXP) →
OUTFILE =
< ! OUTFILE'
  ! FCOMPANDOR(CDR(EXP),M,L4,TRUE,LOCTABLE)
  < 'JRST 0 L1 >
  L4
  < 'MOVEI 1 < 'QUOTE 'T > >
  < 'JRST 0 L2 >
  L1
  < 'MOVEI 1 0 >
  L2
>

ISOR(EXP) ∧ NOT NULL(CDR(EXP)) →
OUTFILE =
< ! OUTFILE'
  ! FCOMPEXP(CADR(EXP),M,LOCTABLE)
  < 'JUMPN 1 L4 >
  ! FCOMPEXP(< 'OR ! CDDR(EXP) >,M,LOCTABLE)
>
```

NOT case:

```
OUTFILE =
< ! OUTFILE'
  ! FCOMPEXP(EXP,M,LOCTABLE)
>
```

```
ISNOT(EXP) →
OUTFILE =
< ! OUTFILE'
  ! FCOMPEXP(CADR(EXP),M,LOCTABLE)
  < 'JUMPN 1 L1 >
  < 'MOVEI 1 < 'QUOTE 'T > >
  < 'JRST 0 L2 >
  L1
  < 'MOVEI 1 0 >
  L2
>
```

COND (no arguments) subcase:

```
ISCOND(EXP) ∧ NULL(CDR(EXP)) →
OUTFILE =
< ! OUTFILE'
  L5
>
```

COND (n > 0 arguments) subcase:

```
OUTFILE =
< ! OUTFILE'
  ! FCOMPEXP(EXP,M,LOCTABLE)
>

ISCOND(EXP) →
OUTFILE =
< ! OUTFILE'
  ! FCOMCOND(CDR(EXP),M,L5,LOCTABLE)
>

ISCOND(EXP) ∧ NOT NULL(CDR(EXP)) →
OUTFILE =
< ! OUTFILE'
  ! FCOMPEXP(CAADR(EXP),M,LOCTABLE)
  < 'JUMPE 1 L3 >
  ! FCOMPEXP(CADADR(EXP),M,LOCTABLE)
  < 'JRST L5 >
  L3
  ! FCOMPEXP(< 'COND ! CDDR(EXP) >,M,LOCTABLE)
>
```

QUOTE case:

```
ISQUOTE(EXP) →
OUTFILE =
< ! OUTFILE'
  < 'MOVEI 1 EXP >
>
```

case of a function call:

```
ISFUNCTIONCALL(EXP) →
OUTFILE =
< ! OUTFILE'
  ! FCOMPLIS(CDR(EXP),M,LOCTABLE)
  ! FLOADAC(1-LENGTH(CDR(EXP)),1)
  < 'SUB 'P < 'C 0 0 LENGTH(CDR(EXP)) LENGTH(CDR(EXP)) > >
  < 'CALL LENGTH(CDR(EXP)) < 'E CAR(EXP) > >
>
```

case of a lambda:

```
OUTFILE =
< ! OUTFILE'
  ! FCOMPEXP(EXP,M,LOCTABLE)
>

ISLAMBDA(EXP) →
OUTFILE =
< ! OUTFILE'
  ! FCOMPLIS(CDR(EXP),M,LOCTABLE)
  ! FCOMPEXP(CADDAR(EXP),
             M-LENGTH(CDR(EXP)),
             ADDIDS(LOCTABLE,CADAR(EXP),1-M)
            )
  < 'SUB 'P < 'C 0 0 LENGTH(CDR(EXP)) LENGTH(CDR(EXP)) > >
>
```

4.2 A Simple Example Proof

As further evidence (beyond the proof of MC0) of the applicability of these techniques for compiler proving, we will use them in a proof of the compiler first proved by McCarthy and Painter [McCarthy67]. The proof is expected to serve also as a more easily comprehended example of how the collection of compiler proof techniques fits together into an integral whole. The simplicity of the McCarthy-Painter compiler allowed us to carry out nearly the entire proof interactively rather than by hand. The McCarthy-Painter compiler

produces few types of target language instructions, and small numbers of instructions. The verification conditions were correspondingly small, increasingly so because of the simple statement of correctness. These facts allowed the human to overcome many of the features required for compiler proving (such as those described in Section 7.2) lacking in the Xivus system. The proof serves as evidence that the approach taken to proving MC0, particularly the part two proof, is well adapted to an interactive program proving environment. The proof was done on a slightly modified version of the Xivus program verification system, with the exception of parsing the target code by hand. The modifications made to the system were to cause the verification condition generator to indicate where substitutions were to be made rather than actually doing the substitutions. This was required (for the part two proof only) because the assertions on which the substitutions were to be made were of course represented symbolically rather than being the assertions themselves.

Following is a listing of the McCarthy-Painter compiler after translating it into slightly extended Pascal. It has been made a procedure in order to have a variable parameter (OUTFILE) onto which target code is added as it is produced, as described in Section 3.2.

```
PROCEDURE COMPILE(E : LIST; T : INTEGER; MAP : LIST;
                  VAR OUTFILE : FILE);
ENTRY ISEXPRESSION(E);
EXIT [Use assertion for each syntactic type case here] ;
BEGIN
IF ISCONST(E) THEN OUTFILE := RIGHTCONS(OUTFILE,MKLI(VAL(E)))
ELSE IF ISVAR(E) THEN OUTFILE := RIGHTCONS(OUTFILE,MKLOAD(LOC(E,MAP)))
ELSE IF ISSUM(E) THEN BEGIN
                      COMPILE(S1(E),T,MAP,OUTFILE);
                      OUTFILE := RIGHTCONS(OUTFILE,MKSTO(T));
                      COMPILE(S2(E),T+1,MAP,OUTFILE);
                      OUTFILE := RIGHTCONS(OUTFILE,MKADD(T));
                      END;
END;
```

ISEXPRESSION is a predicate which holds if and only if its argument is a legal source language expression.

The functions MKLI, MKLOAD, MKSTO, and MKADD represent the abstractions of four types of target language instructions. The part two proof will use Hoare proof rules that work directly with these abstractions, unlike MC0 in which we dealt with the instructions themselves. The function VAL(E) gives the numerical value of a constant E expressed in source language, while LOC(E,MAP) gives the target machine location of a source language variable E by referring to the symbol table MAP. These functions are equivalent to the substitutions

$$E \;||\; c \;||\; k$$

and

$$E \left|\left| \begin{matrix} getv(MAP) \\ getw(MAP) \end{matrix} \right.\right.$$

respectively, where c is the list of acceptable source language constants, k is the corresponding list of values (that is, the ith item in k is VAL of the ith item in c), getv extracts the list of all source language variables in MAP (in the order that LOC searches MAP if redeclaration is important), and getw extracts the corresponding list of their locations. In order that the substitution may be sequential rather than just simultaneous, we assume that constant values are distinguishable from source language constants (c and k have no intersection), and variables and their locations are distinguishable. Further we will assume that no source constants can occur in the notation used for target locations, no source variables can occur in the notation used for constant values, and that source variables and constants are distinguishable. These assumptions will allow us to interchange certain of these substitutions easily during the part two proof. This distinguishability could of course be accomplished by use of some sort of special symbols to express any sets that would otherwise violate these assumptions.

As explained in Section 3.13, the Lisp-like function RIGHTCONS will be defined by axioms introduced into the theorem prover describing how RIGHTCONS relates to other Lisp-like functions which will arise from the assertions. The predicates ISCONST, ISVAR, and ISSUM are the definitions of the source language syntactic types, and so will be defined by axioms used in the theorem prover, as described in Section 3.14. Similarly the analytic syntactic functions S1 and S2 (used in the compiler) will be described by axioms later.

We now write the assertions for the part one proof (proving what the compiler produces) in the list notation described in Section 3.5. The compiler has no loops and therefore needs no internal assertions. For the reasons given above, all the subsidiary functions will be later defined, and thus will simply be given TRUE Entry and Exit assertions now. Therefore the only assertions needed for the compiler are the Entry and Exit of the main procedure. Because we will carry out each case separately, we will require three Exit assertions.

```
1)  ISCONST(E) → OUTFILE = < ! OUTFILE'
                             MKLI(VAL(E))
                           >

2)  ISVAR(E) → OUTFILE = < ! OUTFILE'
                           MKLOAD(LOC(E,MAP))
                         >

3)  ISSUM(E) → OUTFILE = < ! OUTFILE'
                           ! FCOMPILE(S1(E),T,MAP)
                           MKSTO(T)
                           ! FCOMPILE(S2(E),T+1,MAP)
                           MKADD(T)
                         >
```

As before the concluding prime indicates initial value.

In addition we will have the assertion involving the F function FCOMPILE, as explained in Section 3.16.

```
OUTFILE = < ! OUTFILE'
            ! FCOMPILE(E,T,MAP)
          >
```

This assertion shows the relation that we wish to exist between the final value of OUTFILE and FCOMPILE; that relation is that FCOMPILE is exactly the target code added to OUTFILE during an execution of compile. During the part one proof we will assume certain theorems involving FCOMPILE, and those assumptions will constitute the definition of FCOMPILE.

We now mechanically translate the list notation assertions to Pascal using Lisp-like functions to construct the lists. The assertions for each syntactic case are inserted in turn into the compiler code, and each case is then submitted to the Xivus program verification system to complete the part one proof of the compiler. Four verification conditions were produced for the ISCONST case (because there are four execution paths in the compiler), and the following axiom was required for their proof.

```
LISSUM1:
      ISSUM(X.)
  →    ISEXPRESSION(S1(X.))
   ∧ ISEXPRESSION(S2(X.))
```

LISSUM1 is a name for the axiom which is used by the theorem prover. A dot after a variable name indicates universal quantification of that variable over the entire axiom. The indentation is used to indicate the precedence of the operations. That is, in the axiom above, the principal operator is → (implies), while the ∧ (and) simply joins two conclusions.

The ISVAR case produced four new verification conditions which required LISSUM1 and the following axiom for their proof.

```
LISVAR1:
     ISVAR(X.) → NOT ISCONST(X.)
```

The ISSUM case produced four new verification conditions which required LISSUM1 and the following axioms and rewrite rules for their proof.

```
LISSUM2:
     ISSUM(X.) → NOT ISCONST(X.)

LISSUM3:
     ISSUM(X.) → NOT ISVAR(X.)
```

```
LISEXPRESSION1:
          NOT ISCONST(X.)
        ∧ NOT ISVAR(X.)
        ∧ NOT ISSUM(X.)
      → NOT ISEXPRESSION(X.)

RCONS1:
           APPEND(XX.,CONS(YY.,ZZ.))
      -->  APPEND(RIGHTCONS(XX.,YY.),ZZ.)

RCONS2:
      APPEND(XX.,'NIL)  -->  XX.

RCONS3R:
           APPEND(XX.,APPEND(YY.,ZZ.))
      -->  APPEND(APPEND(XX.,YY.),ZZ.)
```

The axioms, whose labels begin with the letter L, are those required to define the source language syntax as explained in Section 3.14. The rewrite rules represent the axiomatic description of the Lisp-like functions as explained in Section 3.13. The dashed right arrow (-->) indicates that the pattern to its left is replaced by the form to the right. Variables in rewrite rules followed by dots are ones that may be bound to any expression to cause a match of the left side.

During the part one proof we have assumed, in the theorem prover, several theorems about the F function FCOMPILE, as explained in Section 3.16. The assumptions state that FCOMPILE is exactly the target code added to the initial value of OUTFILE, as specified in the Exit assertions for the part one proof. Those assumptions, written in the list notation, are as follows.

```
ISCONST(E):

  < ! OUTFILE'
    ! FCOMPILE(E,T,MAP)
  >

=

  < ! OUTFILE'
    MKLI(VAL(E))
  >

ISVAR(E):

  < ! OUTFILE'
    ! FCOMPILE(E,T,MAP)
  >
```

```
=

  < ! OUTFILE'
    MKLOAD(LOC(E,MAP))
  >

ISSUM(E):

  < ! OUTFILE'
    ! FCOMPILE(E,T,MAP)
  >

=

  < ! OUTFILE'
    ! FCOMPILE(S1(E),T,MAP)
    MKSTO(T)
    ! FCOMPILE(S2(E),T+1,MAP)
    MKADD(T)
  >
```

The statement of correctness of this compiler is:

$$A\ \left|\begin{matrix} R1 \\ E \end{matrix}\right|\left|\begin{matrix} c \\ k \end{matrix}\right|\left|\begin{matrix} v \\ w \end{matrix}\right| \{\ \text{FCOMPILE}(E,T,\text{MAP})\ \}\ A$$

where v stands for getv(MAP), w stands for getw(MAP), c and k are as above, and R1 is the accumulator. This is equivalent to the statement of correctness of MC0 when the following facts are taken into account. The Pre and Post conditions on the ISCONST and ISVAR syntactic types are TRUE, as they were for the corresponding types in MC0. In the ISSUM case the Pre and Post are conjunctions consisting of the Entry and Exit of the function plus (both TRUE) and the Pres and Posts of the arguments. By induction on the depth of nesting of syntactic types, the Pre and Post conditions of all expressions are TRUE. No registers other than the accumulator are wiped out, so N(S) is always one. Where MC0 had only one source language constant requiring translation to target language, the McCarthy-Painter compiler requires translation of all constants; hence we have the c-k substitution instead of the NIL-0 one.

This compiler uses a run-time stack for one of the two purposes that MC0 did, namely that of storing temporary values during evaluation of expressions. The other use of the stack, that of storing variables, is not needed in this compiler because its source language does not have any dynamic allocation of storage. As with MC0, the Hoare formula describing the correctness of this compiler is only valid if certain facts are true about the stack. Namely, we have assumed that the stack, which we will represent as array m, has not been changed across execution of the target code in FCOMPILE, at least up to (but not including) m[T]. Here T

is the compiler variable pointing to the next available location in the stack. Now only four target language instructions are produced by this compiler, and the one represented by MKSTO is the only one that can affect the stack m. Since all uses of MKSTO in the compiler use T as the location in m to be set, we need only show that T never holds a lower value than its initial value. The only place T is changed is by recursively calling the compiler with T+1 or T as T's new value. By induction on the depth of such calls, we can easily show that T is always greater than or equal to the initial value of T. Thus we have escaped the need to set up axioms describing the action of target language instructions on the stack to effect a stackok type of proof.

We may note here that T is the parallel of P, the stack pointer of MC0, except P changes at execution time (every time a function is entered, by means of the PUSH instructions placing the newly allocated arguments onto the stack). This allows MC0's compiled code to be re-enterable, which is required for recursive functions to be in the source language. But T may be (and is, in fact) computed at compile time, since the target code produced is not re-enterable.

We also note another difference between MC0 and the McCarthy-Painter compiler here. The latter uses ISCONST, ISVAR, and ISSUM to test the expression to be compiled to determine which syntactic type it is. MC0 however, uses easier tests to determine syntactic type. This is possible because we may allow certain non-expressions to pass a test or because we know several things that the expression is not by virtue of the types for which we have already tested earlier in the code of the compiler. Use of these simplified tests required us to state and use more source syntax axioms to describe what each given syntactic type of MC0 was in terms of the functions used in the compiler code tests.

We will now do the part two proof, that of showing that the results of compiling (FCOMPILE) always satisfy the Hoare formula statement of correctness given above, and therefore the target code produced by the compiler has the same semantics as the source code that was compiled. We will do this by symbolic verification condition generation back through the code of FCOMPILE (which we know from the part one proof), and then prove the resulting verification condition. To do this we will use the following Hoare rules describing the abstractions of the target language instructions.

$$Q \begin{vmatrix} R1 \\ V \end{vmatrix} \{ \text{MKLI}(V) \} Q$$

$$Q \begin{vmatrix} R1 \\ m[I] \end{vmatrix} \{ \text{MKLOAD}(I) \} Q$$

$$Q \begin{vmatrix} m \\ \alpha(m, I, R1) \end{vmatrix} \{ \text{MKSTO}(I) \} Q$$

$$Q \begin{vmatrix} R1 \\ m[I]+R1 \end{vmatrix} \{ \text{MKADD}(I) \} Q$$

These Hoare rules can be understood by viewing the target language instructions as assignment statements. MKLI assigns a value to R1, MKLOAD assigns the value in a given stack location to R1, MKSTO is the assignment of the value in R1 to a given stack location, and MKADD assigns to R1 the sum of a given stack location and itself.

We must also have a Hoare rule for FCOMPILE itself because we will encounter FCOMPILE of the components of the code that we are presently proving. These represent the recursive calls to the compiling procedure. By structural induction we may use the statement of correctness itself as a Hoare rule for these smaller components.

In order to carry out this proof interactively on the available Xivus system, the target code of each syntactic type case was parsed by hand to a Pascal statement that had an equivalent Hoare rule. The outline of an interactive system for carrying out more complex part two proofs is given in Section 7.2.

The part two proof of the ISCONST case required the use of a subrule and a rewrite rule. The former is a generalization of subrule 4 (which we shall call subrule 4b), and the latter (which we shall call MP1) expressing the relationship between the c-k substitution and the function val.

$$\text{Di} \neg\epsilon\ \text{G (for } 1 \le i \le n) \rightarrow \text{G} \begin{vmatrix} \text{D1} & \ldots & \text{Dn} \\ \text{H1} & \ldots & \text{Hn} \end{vmatrix} = \text{G}$$

MP1:
if ISCONST(X.)

$$\text{then X.} \begin{vmatrix} \text{c} \\ \text{k} \end{vmatrix} \text{ --> VAL(X.)}$$

For the ISVAR case, the part two proof required subrule 4b and the following rewrite rule that expresses the relationship between the getv-getw substitution and the function loc.

MP2:
if ISVAR(X.)

$$\text{then X.} \begin{vmatrix} \text{getv(MAP)} \\ \text{getw(MAP)} \end{vmatrix} \text{ --> m[LOC(X.,MAP)]}$$

In addition to subrules 1, 2, 3, 4, 5, 6, 8, 9, 12, 13, and 21, the following axioms and rewrite rules were required for the ISSUM case of the part two proof. Discussions of them follow.

MP3:

$$\text{A} \begin{vmatrix} \text{m} \\ \alpha\text{(m,T,X.)} \end{vmatrix} \text{ --> A}$$

MP4:

$$\text{S2(E)} \begin{vmatrix} \text{c} \\ \text{k} \end{vmatrix} \begin{vmatrix} \text{getv(MAP)} \\ \text{getw(MAP)} \end{vmatrix} \begin{vmatrix} \text{m} \\ \alpha\text{(m,T,X.)} \end{vmatrix}$$

$$\text{--> S2(E)} \begin{vmatrix} \text{c} \\ \text{k} \end{vmatrix} \begin{vmatrix} \text{getv(MAP)} \\ \text{getw(MAP)} \end{vmatrix}$$

LISSUM4:

ISSUM(X.) → X. = S1(X.) + S2(X.)

Subrule 6b:

$$(f\ G1\ \ldots\ Gn)\ \|\|^{DD}_{HH} = (f\ G1\ \|\|^{DD\ \ldots}_{HH\ \ldots\ Gn}\ \|\|^{DD}_{HH})$$

where DD is a list of D's, HH is a list of H's, and f is any source language function.

In MP3 the A is the A from the Hoare formula that is the statement of correctness of the compiler. That is, A represents the assertion about the target language program that we are using in the verification condition generation of the part two proof. Because T is defined as pointing to the next available location in the stack, we know that A contains only references to m with subscripts less than T. Therefore when the alpha substitution is applied to such references to m, the alpha will always simplify back to m, that is, remain unchanged, because of the alpha simplification rule: α(m,T,X)[I] becomes m[I] if I ≠ T.

In MP4 we know that all previously declared variables have locations less than T, and all locations to be used in compiling S2(E) are to be located in M[T+1] and beyond. As in the MP3 rule, no references will ever occur to m[T], so the alpha will simplify the same way. This suggests a possible further abstraction that could be defined, that of containing no references to m[T]. Then a general rewrite rule could be written for dropping the alpha substitution when that condition held.

We note that LISSUM4 is simply a definition of the type ISSUM in terms of its arguments selected by S1 and S2. We have called the last axiom above subrule 6b because it is the obvious generalization of subrule 6 to the case of a multiple substitution.

This concludes the description of the interactive proof of the McCarthy-Painter compiler.

4.3 An Example Case of a More Complex Compiler

Here we present the proof of one syntactic case of a more complex compiler. It is intended to serve as evidence that the techniques given here for proving compilers are applicable both to larger, less "toy-like" compilers, and to statement-oriented source languages. The example we have chosen is the compiler for the language PL/0 given by Wirth [Wirth76, pp. 337-347]. The language PL/0 contains such features as assignment statements, begin-end statement grouping, conditional (if) statement, and the WHILE repetition statement. Block structured declaration of variables and recursive procedures are included, though without formal parameters. The only data type is integer, and the usual arithmetic and relational operators are supplied. The compiler produces target code for a hypothetical stack-oriented machine. The stack is used for declared variables upon each entry to a recursive procedure and also for temporary storage used during evaluation of expressions. More complete descriptions of PL/0 and the compiler (in fact, a derivation of the compiler) are found in the above reference [Wirth76, pp. 307-336].

The WHILE statement is the syntactic type case selected. Both the part one and part two proofs were done by hand rather than interactively, part two for the same reasons as the MC0 proof, and part one for the reasons given below.

The following list of problems prevents a part one proof of the PL/0 compiler from

being carried out interactively on the Xivus system. Each is accompanied by an explanation of how this problem could be overcome, often by rewrite of the compiler code, but never by actually changing the output of the compiler.

1. The compiler contains gotos that jump out of the procedure in which they lie. This violates the assumption in proving the calling procedure that the called procedure returns to the statement immediately after the call. It also causes the proof of the called procedure to use an assertion (the one at the label outside the procedure) that may use variable names out of their declared scope. Since all such gotos are used to "bail out" of a procedure in case of an error, we may rewrite the compiler eliminating the gotos if we assume correct source language input (ignore source language error detection) for our compiler proof. We could also rewrite the compiler with an error flag as a var parameter that would be checked by a conditional goto in the calling procedure upon return of the called procedure.
2. The compiler uses inherited variables rather than passing them as parameters to called procedures. Although it would result in lengthy parameter lists, the compiler could be rewritten with all such variables passed as parameters. Use of a system using the Euclid style of procedure call rules treating global variables would also solve this problem.
3. Declared constants, WITH, and CASE statements are used. These features of Pascal have not been implemented on Xivus, but can all be expanded easily in terms of other Pascal features that are implemented.
4. A variable named CX is used by the compiler to count the target code locations as they are filled. Exit assertions must speak of several intermediate values of this counter in cases where several addresses are significant in expressing the target code produced. The Xivus system has no way to express values of a var parameter other than the initial and final values. We could invent a function and define it as the amount by which CX is incremented in compiling various pieces of code in order to express intermediate values of CX in terms of the initial or final value, but we feel this would result in cumbersome notation. Our solution for the proof we present here is to append a period and digit to express successive values of a variable. This notation is similar to that used in such works as Ragland [Ragland73].
5. Value parameters to procedures are modified. The Xivus system treats non-var parameters as constant, that is incapable of being modified, while the version of Pascal in which the PL/0 compiler was written treats them as changeable, though the change is not reflected in the calling procedure. Thus we must rewrite the compiler to use local variable copies of non-var parameters which we wish to change.
6. Some variables are used only by the lowest level character reading routines (procedures GETCH and GETSYM) in order to buffer a line of input at a time. In order to simplify notation these variables should be treated as own variables or Euclid module variables, that is, ones that are internal only to the lower level routines, but whose values must be preserved from one call to the next. This would eliminate the appearance of those variables in any of the higher level routines that call GETSYM. The interface (which must be expressed in Entry and Exit assertions) between the higher and lower level routines would be simplified nearly to the point of simply saying that GETSYM gets the next symbol from the input file. However Xivus has no way of treating an own variable.

The above reasons convinced us that it was impractical to use the available machine

assistance for the part one proof of the PL/0 compiler, although it probably would have been worthwhile if the entire compiler were to be proved rather than just one syntactic type case. In essence the part one proof was carried out, however, by symbolically executing the code of the compiler with symbols representing the syntactic parts contained within the WHILE statement. It was assumed during this execution that no errors were found and that GETSYM correctly sets the variables SYM, ID, and NUM with the next symbol and identifier or number, respectively. Thus procedures GETSYM and GETCH have not been proved in any manner in this proof.

The following list highlights the differences between the PL/0 compiler and the others to which we have applied our proving methods. In all cases we discuss how these differences may be accommodated into our proving methods.

1. The compiler will not successfully compile programs if certain limits are exceeded, for example identifier table size, depth of nesting of procedures, and size of resulting code. We may assume for purposes of our proof that these limits are not exceeded, but should supply a separate proof (which should be a relatively straightforward inductive assertion proof) that if these limits are exceeded then the compiler notifies the user that the compilation is not to be accepted as correct. We could then state that the compiler was proved correct under the condition that no compilation error messages were produced.
2. Actual numeric addresses are used in the target language rather than symbols for labels. This means that forward gotos often have to be created without the proper address in them, and then have to be "fixed" later when the address has been determined. Fortunately the compilation of a complete source language syntactic type is never produced with an address still awaiting the "fixing" operation. Therefore we have no problem defining what the compilation of a given syntactic type is for the part one or part two proofs. However, we will have to mark points in the target code produced that correspond to locations to which gotos may jump. Then our part two proof will treat these marks as labels, even though the target code itself does not use labels. It might be noted that this treatment by the PL/0 compiler avoids the need for gensym to create labels.
3. There is no "top-level" to this compiler that checks all (or nearly all) pieces of source language to determine their syntactic types, then calls the appropriate parts of the compiler for those types. It is a more efficient compiler design in that it can determine the syntactic type of any piece by one symbol look-ahead (this is possible by clever source language design), given the knowledge of what immediate context that piece is in. Then the appropriate compiler procedure is called without returning to a top-level, where the immediate context would probably be lost. This means that the F function (which we will call FCOMPILE) that represents the target code produced by compiling the various syntactic types will be defined not in terms of one procedure (as FCOMPEXP was) but in terms of the code produced by different major compiler procedures for each different syntactic type.
4. The one symbol look-ahead mechanism requires several extra parameters in the compiler, such as the last character read, the last symbol read, the last identifier read, etc. In order to carry out the part one proof it will be necessary to assert that upon beginning and ending the compilation of each source language syntactic type, all such variables have been carefully kept current with the proper values. Thus the assertions for part one will contain somewhat more information than in the MC0 proof.

5. Several loops occur in this compiler, while MC0 had none. This simply means that in addition to Entry and Exit assertions, some internal assertions must be made to "break" the loops.
6. There are some errors in the compiler as published. In the course of symbolically executing the code, two misspellings and the use of an undeclared (and apparently unnecessary) variable were found. The use of the previously proved compilers MC0 and the McCarthy-Painter compiler reduced the chances of finding errors in those instances. However, the errors found in the PL/0 compiler underscore the need in interactive program proving systems for capabilities to reprove a slightly changed program in a manner that makes efficient use of all previous work.
7. Output of the target code is done by placing the code into an array with the use of a pointer to the array instead of using a file in which to write the code. This is required because the already written portion of the array must be accessible later for goto address fixing. The only effect this has on our proof is that the F function FCOMPILE must be defined as the code lying in the output array (CODE) between the initial and final value of the array pointer variable (CX).
8. Input is accomplished as a hybrid of file, array, and variables because of the need to look-ahead one symbol and the desire to buffer a line ahead. This complicates our denoting "the present statement" (or expression) in our assertions. We will solve this by inventing a function, which we will call PRESENT, that combines either the variable holding the last identifier read or the variable holding the last number read or a source language keyword or symbol (depending on the last symbol indicator SYM) with the remainder of the line buffer and the remainder of the input file to give us the entire present statement (or expression). Because the present statement may be nested inside another statement or be followed by another statement, we may get more statements or tails of statements from the remaining part of the input file. So PRESENT will also remove any such tails to give us exactly the present statement or expression. The F function FCOMPILE will then be defined in terms of the present statement or expression as selected by PRESENT.
9. In compiler function POSITION a programming shortcut is used that will unnecessarily complicate the proof. Instead of having two conditions for exiting the loop, that of searching the entire table or of finding the desired item in the table, only the finding condition is used after the desired item has been placed at the end (last in the direction of search) of the table. This has the effect of requiring all assertions about items being in the table or not to note the exception of the last position. It also adds a var parameter (the table) to the function, which requires it to be rewritten as a procedure, and which complicates all verification conditions involving POSITION. In short, the abstraction that POSITION represents had an unnecessarily messy interface because this programming trick was used. We would therefore rewrite the function before proving.

By symbolically executing the compiler we have determined that:

```
FCOMPILE( < 'WHILE C 'DO STMT > ,TABLE,TX,CX,LEV,DX) =
< CX.0: ! FCOMPILE( C ,TABLE,TX,CX.0,LEV,DX)
  CX.1: < 'JPC 0 CX.4 >
  CX.2: ! FCOMPILE( STMT ,TABLE,TX,CX.2,LEV,DX)
  CX.3: < 'JMP 0 CX.0 >
  CX.4: >
```

Note that the implied labels have been shown, as promised. The WHILE statement given represents the value of PRESENT for purposes of the proof of this syntactic type case, while the value of PRESENT will be C at the time that a compiler routine is invoked to compile the condition C, and will be STMT at the time that a routine is invoked to compile the statement STMT. The context of the compiler is passed to FCOMPILE by the arguments TABLE (the symbol table), TX (the symbol table size), CX (the index to the next available target code location), LEV (the nesting level of the present procedure), and DX (the size of stack that is filled with locally declared variables).

The statement of correctness of the compiler in the form of a Hoare formula will be similar to that of MC0. For statements, we have corresponding to every source language Hoare rule the target language Hoare formula which is exactly the same except that it uses FCOMPILE of each piece of code, and it applies the * operation to all assertions. For example, the source language Hoare rule

```
P { S } Q
```

would require the proof of the corresponding target language Hoare formula

```
P * { FCOMPILE(S,TABLE,TX,CX,LEV,DX) } Q *
```

where the * notation is the shorthand for the two multiple substitutions that translate source language constants to the corresponding target language constants, and source language variables to the corresponding target language locations. Note that the Hoare formula that is the statement of correctness will be of different form for each different statement type because the source language Hoare rules are different for each type. This makes it appear more complicated than MC0, which had only one statement type. However, the form of each Hoare formula will be simpler than in MC0 in that there need be no mention of registers (there are none in the PL/0 target machine), Pre and Post conditions of expressions (for reasons explained below), or function values (statements in PL/0 do not return values).

One slight complication introduced by PL/0 is the use of symbolic names for constants. As these are declared, they are placed, along with their values, into the symbol table. If we expand the definition of v (the list of variables obtained from the symbol table) to include the constant names, and expand w (the corresponding list of variables' locations) to include constants' values, then the multiple substitution of w for v will still translate all source language identifiers into their target language counterparts. Note that we cannot do this translation with a separate list for the constants because the source language allows redeclaration of names as either variables or constants, and so the order of declaration must be strictly preserved to obtain the most recent one. Note that this requires that getv and getw

(the functions to extract the lists v and w from the symbol table) must extract the lists in reverse order, since newly declared variables are added to the end of the table in this compiler. Also note that addresses in this compiler are two-dimensional; that is, they have a level and a location within that level.

The definitions of c and k as lists to translate source language constants (not having symbolic names) will remain the same as in the proof of the McCarthy-Painter compiler. Thus the * notation will represent the substitutions:

$$\left|\begin{matrix} c \\ k \end{matrix}\right| \left|\begin{matrix} getv(TABLE,TX) \\ getw(TABLE,TX) \end{matrix}\right|$$

The Hoare formulas that are statements of correctness for expressions are slightly more complicated than for statements because we must deal with the returned value of the expression. We will henceforth refer to PL/0 expressions, including the syntactic types condition, term, and factor in addition to expression, as general expressions to distinguish them from the narrowly defined syntactic type expression. Since there are no registers in the target machine, general expression values are returned in the next available location in the stack. The stack is denoted by S and the index to the last used location is T. With this notation, the effect of executing a compiled general expression is equivalent to executing the statements: S[T+1] := general-expression; T := T+1.

All general expressions are composed of constants, variables, and the arithmetic, relational, and odd (as opposed to even) operators. User defined functions are not allowed in PL/0. The Pre and Post conditions of constants and variables, as defined for the MC0 compiler in Figure 4-1, are TRUE. The Entry and Exit conditions for all the operators used are also TRUE. By a simple induction argument (on the depth of nesting) we can show that the Pre and Post conditions of any general expression are TRUE. This allows us to drop the Pre and Post conditions from the Hoare formulas that are the statements of correctness for general expressions and statements, as mentioned above. The result for general expressions is:

$$Q \left|\begin{matrix} T \\ T+1 \end{matrix}\right| \left|\begin{matrix} S \\ \alpha(S,T+1,E\ *\ )\ \{\ FCOMPILE(E,TABLE,TX,CX,LEV,DX)\ \}\ Q \end{matrix}\right|$$

As with MC0, we assume that the stack is preserved (at least up to the T position) during the execution of the target code representing any source language syntactic type. But in the case of general expression, the stack pointer T is incremented by one to point to the location used for the expression value. Therefore the definition of stackok remains the same (with the change of notation for the stack and its pointer), but in addition we need to define a new property. We will call it stackplus1 and use it to mean stackok except that T is one greater upon returning. Because the axioms about stackok are specific to the target language, we will have to redefine the axioms. We will also need a new axiom to relate stackplus1 to stackok. The fact that the target code has loops in it means further change to the axioms, but the stackok proof still remains quite similar to that of MC0.

We are now ready to tackle the part two proof of a WHILE statement. The source language Hoare rule for WHILE is:

$$\frac{Q \wedge C \{ STMT \} Q,\ Q \wedge \neg C \rightarrow R}{Q \{ < \text{WHILE C DO STMT} > \} R}$$

The target language Hoare formula which we must prove is then:

$$\frac{Q* \wedge C* \{ FCOMPILE(STMT,TABLE,TX,CX1,LEV,DX) \} Q*,\quad Q* \wedge \neg(C*) \rightarrow R*}{Q* \{ FCOMPILE(< \text{WHILE C DO STMT} >,TABLE,TX,CX,LEV,DX) \} R*}$$

What we must show for a rule of inference such as this is that the formula below the line is provable given the formulas above. It might be noted that the compiler context (TABLE, etc.) of two different FCOMPILEs appearing in one Hoare formula need not be the same. Careful examination of the compiler will show, however, that TABLE, TX, LEV, and DX are constant over the code of a given procedure. Therefore we need name only CX differently in the Hoare formula.

To prove the Hoare formula for WHILE we will generate a verification condition from the R * back through the target code of FCOMPILE. First note that assertion(CX.4) = R *. Similarly assertion(CX.0) = Q *. We now apply the following Hoare rule for a JMP (unconditional jump) statement.

$$assertion(L) \{ < \text{'JMP 0 L} > \} Q$$

The first premise of the Hoare formula we are proving allows us to proceed generating across the FCOMPILE of STMT. The result is $Q* \wedge C*$. The JPC target language statement is a conditional jump with the added twist that the tested item is removed from the stack (or more precisely abandoned by backing up the pointer). Therefore the Hoare rule for JPC looks like the combination of the Hoare rule for an assignment to the stack pointer T and that of a conditional jump.

$$(S[T] \rightarrow Q \Big|_{T-1}^{T}) \wedge (\neg S[T] \rightarrow assertion(L) \Big|_{T-1}^{T})$$

$$\{ < \text{'JPC 0 L} > \} Q$$

The result is:

$$(S[T] \rightarrow (Q* \wedge C*) \Big|_{T-1}^{T}) \wedge (\neg S[T] \rightarrow R* \Big|_{T-1}^{T})$$

The inductive assumption on smaller pieces of code allows us to generate across the FCOMPILE of the condition C by applying the substitutions:

$$\begin{vmatrix} T \\ T+1 \end{vmatrix} \begin{vmatrix} S \\ \alpha(S,T+1,C*) \end{vmatrix}$$

Then complete the verification condition generation by adding the initial assertion $Q*$ as a hypothesis. Distribute the substitutions over and, implies, not, and subscripting to get:

$$Q* \rightarrow$$

$$(S \begin{vmatrix} T \\ T+1 \end{vmatrix} \begin{vmatrix} S \\ \alpha(S,T+1,C*) \end{vmatrix} [T \begin{vmatrix} T \\ T+1 \end{vmatrix} \begin{vmatrix} S \\ \alpha(S,T+1,C*)] \rightarrow$$

$$Q* \begin{vmatrix} T \\ T-1 \end{vmatrix} \begin{vmatrix} T \\ T+1 \end{vmatrix} \begin{vmatrix} S \\ \alpha(S,T+1,C*) \end{vmatrix} \wedge C* \begin{vmatrix} T \\ T-1 \end{vmatrix} \begin{vmatrix} T \\ T+1 \end{vmatrix} \begin{vmatrix} S \\ \alpha(S,T+1,C*) \end{vmatrix}) \wedge$$

$$(\neg S \begin{vmatrix} T \\ T+1 \end{vmatrix} \begin{vmatrix} S \\ \alpha(S,T+1,C*) \end{vmatrix} [T \begin{vmatrix} T \\ T+1 \end{vmatrix} \begin{vmatrix} S \\ \alpha(S,T+1,C*)] \rightarrow$$

$$R* \begin{vmatrix} T \\ T-1 \end{vmatrix} \begin{vmatrix} T \\ T+1 \end{vmatrix} \begin{vmatrix} S \\ \alpha(S,T+1,C*) \end{vmatrix})$$

Subrules 2f (remembering that S corresponds to m and T to P), 11, and 4 allow us to drop the T+1 substitution on S in both places. Then subrule 21 is applied to S and to T. Distribution of S substitution over T+1, then subrules 2f, 11, and 4 results in the hypotheses of the latter two implications becoming:

$$\alpha(S,T+1,C*) \ [T+1]$$

and

$$\neg \alpha(S,T+1,C*) \ [T+1]$$

respectively. Apply subrules 9 and 21 and arithmetic simplification to reduce

$$\begin{vmatrix} 1 \\ T-1 \end{vmatrix} \begin{vmatrix} T \\ T+1 \end{vmatrix}$$

to the form

$$\begin{vmatrix} T \\ 1 \end{vmatrix}$$

which is dropped by subrule 14. The result is:

$$Q\,* \rightarrow$$

$$(\alpha(S,T+1,C\,*)\ [T+1] \rightarrow Q\,* \Big|^{S}_{\alpha(S,T+1,C\,*)} \wedge C\,* \Big|^{S}_{(S,T+1,C\,*)})$$

$$\wedge\ (\neg\ \alpha(S,T+1,C\,*)\ [T+1] \rightarrow R\,* \Big|^{S}_{\alpha(S,T+1,C\,*)})$$

But by simplification rules for α, we have α(S,T+1,C *) [T+1] = C *. Q is a source language assertion and the * operation translates source variables into locations in stack S that are at S[T] or before. The same applies to R and C. Changing the (T+1)st location of S will have no effect on Q *, so the α for S substitutions may be dropped on Q *, R *, and C *. The result is:

$$Q\,* \rightarrow (C\,* \rightarrow Q\,* \wedge C\,*) \wedge (\neg\ C\,* \rightarrow R\,*)$$

Simplify this logically to obtain:

$$Q\,* \wedge \neg\ C\,* \rightarrow R\,*$$

This may be assumed by the second premise of the Hoare formula.

That concludes the proof of the Hoare formula, but we must still demonstrate the stackok property. To accommodate the loops in the target code we may either add more complex axioms, such as

```
stackplus1(t1) ∧ stackok(t2) →
  stackok( < L1:
             ! t1
             < 'JPC 0 L2 >
             ! t2
             < 'JMP 0 L1 >
             L2:
           > )
```

or else find all paths between assertions and prove the stackok property for each path. By induction on the number of assertions passed during an execution, we know any path through the code from the entrance to the exit has the stackok property. We will choose the latter because the above axiom is so messy as to make it difficult to understand.

We will unravel the paths of the target code for a WHILE statement, discarding the jumps, and find two paths. One caution is that we cannot discard the part of a JPC instruction that affects the stack. We will denote that part by a JPC' instruction. The resulting paths are:

```
1) < CX.0: ! FCOMPILE( C ,TABLE,TX,CX.0,LEV,DX)
     CX.1: < 'JPC' 0 CX.4 >
     CX.2: ! FCOMPILE( STMT ,TABLE,TX,CX.2,LEV,DX)
   >
```

```
2) < CX.0: ! FCOMPILE( C ,TABLE,TX,CX.0,LEV,DX)
     CX.1: < 'JPC' 0 CX.4 >
   >
```

We now introduce the new axiom required to relate the stackplus1 property to stackok.

```
S10: stackplus1(t1) → stackok( < ! t1 < 'JPC' 0 L > > )
```

This axiom says that a JPC' instruction removes one item (the location it may conditionally jump to) from the stack and so "balances" a stackplus1 string of code.

We may assume as the inductive assumption the following on any smaller pieces of code.

```
stackplus1( FCOMPILE(general-expression, ... ) )

stackok( FCOMPILE(statement, ... ) )
```

With the use of stackok axioms S3 and S7 (which do not change, since they do not mention specific target language instructions), we immediately show the stackok property on both of the above paths. This concludes the PL/0 WHILE case proof.

5 PREVIOUS RELEVANT WORK

5.1 Introduction

In the following two sections are listed the published compiler proofs with a brief description of each. The sections are divided according to the manner in which the compiler is specified, as this has a distinct effect on the way in which the proof is carried out. Discussion of their relation to this dissertation appears in Section 5.4.

5.2 Proofs of Compilers Expressed Operationally

Proving a program correct requires a specification of what the program is supposed to do. In the case of a compiler, this usually amounts to a statement that the compiled program, when run, has the same effect as the source program would have had if it could have been run. Thus the specification must somehow include the meaning or semantics of both the source and target language of the compiler we wish to prove.

McCarthy [McCarthy63,McCarthy66] was the first to suggest a method of expressing the syntax and semantics of a language in a manner useful to compiler proofs. He proposed describing the syntax of a language by the use of predicates and functions which would recognize syntactic types of the language and break down syntactic constructions into their component parts, respectively. These abstractions help us to define the syntax and semantics of the language as well as to translate from the language to another. The abstractions make the proof independent of the particular forms used to express the source language. He then defined semantics of expressions in terms of their values, which are defined recursively as functions of the values of the component syntactic parts. Semantics of statements are similarly expressed in terms of their effects on the program's state vector. McCarthy and Painter [McCarthy67] then used these concepts to prove a compiler to translate very simple arithmetic expressions into a simple machine language containing load, store, add, and load-immediate instructions. Their compiler was basically a case statement, with recursive compiler calls for subexpressions, each case corresponding to an allowable syntactic structure. Ideas first used in their paper are found in much of the later compiler correctness work. It is their compiler that is used as an example (in Section 4.2) of our compiler proof methods.

Still using similar concepts, Painter [Painter67] proved a compiler that would accept assignment statements, conditional gotos, and I/O statements. About the same time Kaplan [Kaplan67] produced a proof using recursion induction, again based on McCarthy's work, of a compiler similar to Painter's, but without the I/O statements. Burstall [Burstall69a] presented a proof of a simple expression compiler in the McCarthy-Painter vein to demonstrate the usefulness of structural induction. A paper by Wada et al [Wada73] applies the McCarthy-Painter methods to prove a rather simple compiler.

London [London71,London72] proved two compilers which take a subset of pure Lisp

into an assembly language with, among other features, push and pop stack instructions. This was the first example of a compiler proof where either the language in which the compiler was written, in this case Rlisp, or the source language was a computer language in existence for purposes other than a proof. In London's work the task of specifying the effect of executing a statement is simplified by the fact that in the source language the effect is only to return a single value. The first of these two compilers is, in slightly modified form, the compiler used as the principal example (presented in the appendix) of the compiler proof techniques presented here. The second of the compilers proved in London's work provides us with the example used here to show proof of optimized versions of compilers (see Section A.9).

Although Ragland [Ragland73] proved a verification condition generator rather than a compiler, his work is relevant because verification condition generators possess some of the same problems as compilers. The verification condition generator proof resembles a compiler proof in the important respect that the correct operation of either of those types of programs depends on the semantic meaning of its source language (that language in which programs must be written to be processed by the verification condition generator or compiler). The verification condition generator then implements those semantics rather than producing target language with equivalent semantics as a compiler does. Thus proof of the generator requires the specification of semantics for more than just the language in which it was written. Ragland chose the use of Hoare proof rules to describe the source language, a choice which has been found useful in compiler proofs, and was in fact used in this present work.

Newey [Newey75] proved a Lisp interpreter. He then described how to prove the first of the compilers London proved, but Newey was not able to carry it out in the automated Scott logic system he was using. Newey's method of proof was to give an interpreter for the source language and an interpreter for the target language, and then to show that the result of running the first interpreter on a source language function is always the same as running the second interpreter on the compilation of the function. This proof, as did London's, makes use of the fact that the only effect of a source language function is to return a single value. The methods presented in this present work do not need this restriction.

Chirica and Martin [Chirica75] first clearly divided a compiler proof into two parts: "what is produced" and "what it means." Their source language has expressions, assignments, conditionals, while loops, and block structure, but not procedures or functions. They stated that the method could be extended to certain kinds of procedures, however. They did not prove the parsing part of the compiler, but dealt only with proving the code generation part. Their compiler was written in a simple language with such features as assignment and whiles. They state that their proof method will work with a variety of methods of expressing semantics, including the Hoare proof rules they used. Chirica's subsequent dissertation [Chirica76] announced a change in approach from the Chirica-Martin paper. The new direction is toward an algebraic approach, and so is described in the next section.

Samet [Samet75] proves the equivalence of target code produced by an optimizing compiler to that produced by a non-optimizing compiler. Thus, he was not addressing our problem of proving the equivalence of source and target code. However, his work could be used to prove an optimizing compiler by showing equivalence to a simpler compiler. This approach appears to be more easily carried out than proving an optimizing compiler directly. Thus Samet's work could be used to advantage for equivalence proofs as described in Section 7.3.

Boyer and Moore [Boyer77] have recently proved a simple optimizing compiler written in Lisp to compile certain Lisp expressions. The proof was carried out on a more powerful version of the Boyer-Moore Lisp Theorem Prover [Boyer75]. It was accomplished entirely without human guidance with the exceptions of stating the correctness theorem about the compiler, and asking that a few lemmas be established before the main result. Their system uses previously stated and established lemmas in making further proofs. The statement of correctness was a Lisp expression stating that executing the target code produces a value equal to the value of the source expression. The semantics of the source and target languages were expressed by interpreting functions. The size of the compiler proved was much smaller than MC0, but the lack of the need for human intervention during the proof is impressive.

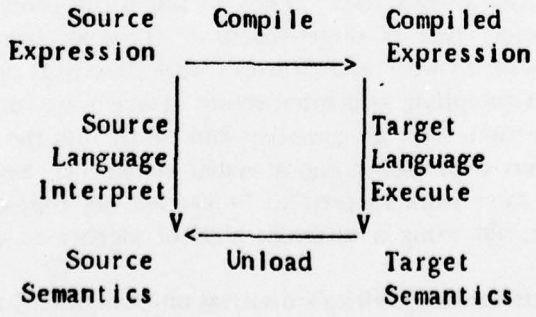
5.3 Proofs of Compilers Expressed As Mathematical Functions

In what we here call the mathematical function approach, the compiler is specified as a mathematical function rather than as a program. Because actual compilers are written in computer languages, not as functions, we view this specification as a shortcoming, unless auxiliary proof is given to show the equivalence of the compiler code and the compiling function. Of the papers mentioned below, only Chirica addresses this problem. This functional approach does, however, add to our understanding of compiling by identifying some of the structure underlying computer languages and compiling.

Burge [Burge68] gave a proof of a compiler for lambda calculus expressions. It flattened the tree structure of the expressions and accomplished some binding of variables.

Blum [Blum69] gave a proof of a functionally expressed compiler to take recursive functions to Turing code. Such choices of languages seem to be of little practical importance.

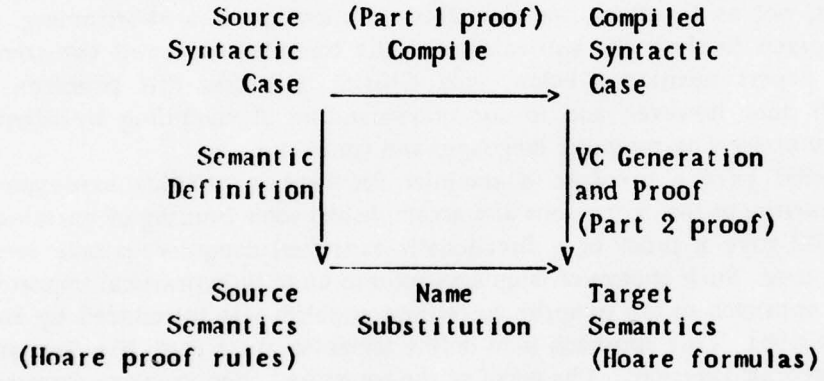
An algebraic approach to the compiler correctness problem was introduced by Burstall and Landin [Burstall69b]. This approach is to define semantics (in a form like interpreting) and compiling as algebraic functions. The proof of the correctness then involves showing that the result of the source interpreting function is the same as the result of the target interpreting function applied to the compiled source, modulo some method (a function of course) of translating target semantic terms into source semantic terms. Their method may be diagrammed (and was by them similarly) as follows:



The unload operation is the inverse of load, which is to load the value (semantics) into the target language stack. Unload is then the method of translating target semantic terms into source semantic terms. By using properties of the algebras and functions representing the

corners and sides of this diagram, they showed that either way around the box (that is, by interpreting or by compiling, executing, and unloading) results in the same value.

A similar diagram could be drawn for the methods of compiler proof given here. The source language at the corner of our diagram would represent a single syntactic case, though, since we have not pushed the induction on the source language syntactic structure into the algebraic representation. Further, the semantics are represented by Hoare proof rules and Hoare formulas. The process of showing that the semantics are true of the program is that of verification condition (VC) generation and proof of the resulting theorems. The verification process on the left is unnecessary, however, because the Hoare proof rules are exactly the definitions of the semantics of the various source syntactic cases. Therefore we have labelled this process as semantic definition. The translation between source and target language semantics proceeds in the other direction from the unload operation. This process, which we have called name substitution in the diagram, is that of adding the v-w and c-k (NIL-0 in MC0) substitutions to the Hoare proof rules, as well as the quantification, formal argument, and result substitutions derived in Chapter 4.



Again we show that two ways around the box arrive at the same semantics, this time target semantics, though. Note that we have marked the part one and part two proofs of our method.

To return to Burstall and Landin, they carried out a proof for a compiler for expressions whose target machine used a stack. Then in stages the proof was extended to a conventional accumulator-memory type of target machine. This algebraic proof apparently served as the inspiration for Morris's work and Chirica's work described below.

Morris [Morris72] treats compiling and interpreting as algebraic functions to prove the correctness of a compiling function. Not all compilers can be fit into the algebraic model he used. His source language has expressions, concatenated statements, assignment statements, while loops, and gotos. In a later work [Morris73] he carried out part of a compiler proof similar in concept to his first, but using a different kind of algebra as the structure of the languages and semantics.

As noted in the previous section, Chirica's dissertation [Chirica76] represents a change in approach from Chirica-Martin [Chirica75]. The new approach followed Burstall and Landin's, but Chirica applied new types of algebras. The problem still arises of finding the algebraic function to translate from target semantic terms to source. The shortcoming mentioned above of proving a compiling function rather than a compiling program is

addressed by showing how assertions may be generated for applying the inductive assertion method to a program to show its equivalence to a compiling function. The source language of the compiler proved has such features as assignment, conditionals, while, block structured declarations, read, write, integer arithmetic, and logical expressions. Although the language lacks functions or procedures, it is stated that no difficulties were found in applying these techniques to a source language with recursive procedures with no global variables (a common restriction; for instance, MC0 has no global variables).

Using a mechanical first order predicate calculus proof checker, W. Diffie checked the McCarthy-Painter proof about 1971 (this unpublished work is mentioned by Igarashi, London, and Luckham [Igarashi75, p. 179]). With this possible exception, Milner and Weyhrauch [Milner72] were the first to apply machine aid to the compiler proof process by submitting much of a compiler proof to a mechanical proof checker. Their compiler took a simple Algol-like language to an assembly language with stacks. The compiler was described (written), as well as proved, in Scott's system of logic [Scott70].

Germano and Maggiolo-Schettini [Germano75] use Markov algorithms for the compiler and for the target language. These language choices are also of little practical importance.

5.4 Significance of this Work

The principal contributions of this dissertation are to present a method of proving compilers with machine assistance, and to give example proofs using this method to demonstrate its utility. The principal features of the method are the use of Hoare proof rules to describe the semantics of the target, source, and compiler languages, and the use of a formalism to describe and simplify the substitutions of variables that result from the use of Hoare rules. The method also involves the use of many other techniques of lesser importance or originality. However the practicality of the method is greatly enhanced by the combination of these techniques.

Machine assistance that has appeared in previous compiler proofs includes the generation of verification conditions in the Igarashi, London, and Luckham example [Igarashi75], Newey's use of an automated Scott logic system [Newey75], the machine checking of a hand-generated proof by Milner and Weyhrauch [Milner72], and the nearly completely automatic proof of Boyer and Moore [Boyer77]. Thus this dissertation is one of only a few efforts that use interactive machine assistance in the proof of a compiler. The significance of this work, though, is that it gives methods that we believe will allow larger and more complex compilers to be proved with machine assistance than was possible with past methods. MC0, the compiler proved here, though by hand application of our methods, is much larger than the compilers in the papers cited above, except Newey's. It is essentially the same compiler as was not completely proved in Newey's work. We think these facts are evidence that our methods represent a step toward proving larger compilers. Further evidence is given by the fact that our methods have been applied to part of a still larger compiler, the PL/0 one, and we see no reason why a program-proving system such as described in Section 7.2 would not be able to prove all of it.

This proof of MC0, while using a compiler quite similar to one previously proved by London [London72], employs several techniques that in general formalize the proof methods, and should help in applying machine aid to future compiler proofs. Probably the largest

problem in trying to mechanize London's type of proof is having to symbolically execute the symbolically expressed target code. Existing symbolic execution systems are made to execute completely specified code, not symbolically expressed code. We avoid this problem by restructuring the problem to require only generation of verification conditions across the symbolically expressed target code. Then we give the formalization of substitution to allow us to accomplish the generation and proof of the resulting verification conditions. A further difference between our proof and London's is that our semantic definition method is not restricted to an expression language which returns a single value, but instead applies to languages whose semantics can be expressed by Hoare proof rules. Clarke [Clarke77] has obtained some interesting results defining which programming features may be included in a language whose semantics are to be completely expressed (complete in a sense described in Clarke's paper) by Hoare proof rules.

This dissertation represents the first complete published example of how Hoare type proof rules can be used not only on the compiler code being proved, but also for the description of the semantics of the source and target languages. Chirica and Martin [Chirica75] gave a general outline of such a proof, but did not present all the formal details. It is interesting that in Chirica's later work [Chirica76] the use of Hoare rule semantics was abandoned in favor of an initial algebra approach to semantics. Chirica comments [Chirica76, p. 273] that a proof directly from the Hoare rules "is simply unmanageable!" While the length of the part two proofs in Section A.8 might at first lend credence to his view, we feel that application of machine assistance overcomes this problem. Our methods of proof were developed with the thought of running them on a mechanized proving system, and have in fact been run for the proof of the McCarthy-Painter compiler. With a system such as that discussed in Section 7.2, we believe compilers at least as complex as MC0 could be interactively proved.

A new technique used in the proof of MC0 is in the formalization of substitution and its properties. The Hoare proof rule for an assignment statement requires that a substitution of an expression for the free uses of a variable be carried out upon an assertion. This substitution produces the assertion that must be true before execution of a statement, from the assertion that is true after execution. In the proof of MC0 we are often working with assertions containing symbolic names for parts of target code and target expressions. Thus we cannot actually perform substitutions prescribed by Hoare rules in many cases. So we develop a notation to indicate where a substitution is to be performed and a series of simplification rules to simplify such notation. That this formalization may be used on a mechanized program proving system was shown by its use in the Xivus system for the part two proof of the McCarthy-Painter compiler. We believe that a program proving system like that described in Section 7.2 would in fact make this formalization quite easily used in compiler proofs.

A serious omission of the Chirica and Martin work [Chirica75] is its lack of treatment of user-defined functions. This present dissertation is, we believe, the only proof of a compiler using any machine assistance that treats such functions, which are a major tool in the use of nearly every computer language. Thus the use of Hoare proof rules to define the source language semantics has now been shown to work for this important concept of functions in source languages. In fact it was found that the treatment of functions, including the similar concept of Lisp lambda expressions, was at least as complex to prove as the combination of all the other syntactic types of the source language.

In Igarashi, London, and Luckham [Igarashi75] the Hoare rules were given for the various types of statements assuming each contained no nested calls to user-defined functions. It was further stated:

> The definition of P [precondition] and R [postcondition] as conjunctions means some loss of generality if nested function calls occur such as in Y ← G(H(X)). A more complicated definition of P and R is known for such cases but it is not implemented.

Such expanded rules for nested calls were developed for an early version of the proof rules for Euclid, but were not used in the final version [London77]. In this dissertation we borrow those nested function rules and pioneer in their use in compiler proofs. Such rules use a precondition and postcondition recursively defined for a series of nested functions. Use of them in a compiler proof makes precise the expression of the effects of executing nested functions in addition to the already demonstrated facility of Hoare rules for statements. Thus the method of compiler proof in this work can be applied to either a statement type or expression type of source language.

Few of the other works have applied the inductive assertion technique to compiler code. Many, such as Milner and Weyhrauch [Milner72] and Morris [Morris72], have defined the compiler as a mathematical function rather than actual code. Since a compiler must be written as code to be used, we feel the distinction is important. London's proof of two Lisp compilers [London71] uses compilers expressed as code, but hand executes it to determine results. A mechanical procedure that formally turns code and assertions about the code into verification conditions has been applied in one example, that being the final example in Igarashi, London, and Luckham [Igarashi75]; but the formal proof of those verification conditions was not carried out. Chirica and Martin [Chirica75] use the inductive assertion method on the code generation portion of a compiler expressed as code. In our method applied to MC0, an entire compiler is given as code, the assertions about the compiler are given, a mechanical method is applied to produce verification conditions, and they are proved. Thus this is one of only two relatively complete applications of the inductive assertion method on a compiler.

If we are to prove actual compilers, not just abstract compiling methods, we must apply the method of inductive assertions to the compiler code and complete the proof of the entire compiler. The only alternative we see is an auxiliary proof that the compiler is equivalent to the abstract compiling method. We cannot believe that a proof of a mathematically defined compiler alone is adequate proof of a corresponding actual compiler because there are too many ways in which the actual compiler may subtly differ. Computer programming has produced countless examples of programs with subtle differences between the code and the intent. Chirica [Chirica76] seems to be the only one among those expressing compilers as functions (rather than code) who has addressed this problem. He applies an auxiliary inductive assertion proof after the proof of his compiling function to show that the compiling function actually represents what the compiler code does. Our approach is more straightforward in the sense that it applies the inductive assertion methods for the entire proof.

A technique used in this proof which has appeared in only one other work [Chirica75] is that of dividing the compiler proof into two distinct parts. First it is proved what the compiler produces for each source language syntactic type, then it is proved that the target language produced has the same effect as the source language does (or would have if source language were directly executed). The division into two parts simplifies the rather complex

problem of proving a compiler. It allows us to deal with the compiler code in part one, keeping it separate from the use of source and target language semantics, which enter only the part two proof. The first part can be fairly simply handled with well known program proving techniques. It requires only the addition of a good way of speaking about the structures of the source and target languages (lists in MC0) to a typical program proving system. The second part here was carried out interactively on a small example, and a system is discussed in Section 7.2 that would prove more complex compilers.

Proof of the properties of the target machine stack during execution, which is required to validate the Hoare rule formulation, is carried out by an axiomatic approach similar to the subgoaling techniques used by Suzuki [Suzuki75]. We felt that such properties could be more understandably expressed by this approach than by dealing directly with the messy details of the stack implementation, and the ease with which the stack proofs were carried out bears out this feeling. This is the first application of such techniques to a compiler as a whole, although Guttag, Horowitz, and Musser [Guttag76] have given a symbol table example proved axiomatically.

Further use of axiomatic definitions is made in defining the source language syntax functions and the Lisp-like functions used in the compiler to build or take apart lists. We consider these uses important because they reduce the complexity of the verification conditions by postponing the entry of information into the verification conditions until and unless it is needed in their proofs.

The techniques used to address the gensym problem, that of handling in a proof a Lisp function that is not a mathematical function, were described in Section 3.17. Gensym or something like it is needed in a compiler to generate unique labels. The only other approach to the gensym problem we have seen is that of Milner and Weyhrauch [Milner72]. We believe our approach is a higher level of abstraction than Milner and Weyhrauch's, and results in a less messy proof.

The use of the list notation described in Section 3.5 is important because it helps reduce the complexity of the assertions that must be written for the part one proof. One of the criticisms of program proving that has often been presented is that assertion writing is too error-prone. To counter this criticism we must use tools such as the list notation to simplify the assertion writing process.

6 CONCLUSIONS

6.1 Introduction

The conclusions reached from proving MC0, the McCarthy-Painter compiler, and part of the PL/0 compiler are that certain of the methods employed here in the construction of compiler proofs are of benefit in proving compilers. Secondarily, we believe that certain methods used in constructing compilers are of benefit in their proofs. Further, these methods should help produce more reliable, understandable, and maintainable compilers, particularly if they are proved, but also to a lesser degree if they are only kept in mind during compiler construction. The following sections list such methods in the order of importance we assign them.

6.2 Hoare Proof Rule Semantics

One of the goals of defining the language Pascal by the use of Hoare proof rules [Hoare73] was to communicate the language designer's intent to the language implementers, that is, the compiler writers. It is also stated there that such a definition serves as "an axiomatic basis for formal proofs of properties of programs." We feel that the proofs given here demonstrate that such language definitions serve not just as the basis for proving properties of a program (the compiler), but further serve to define the properties of all three languages involved in the compiler proof. The proofs here presented worked directly from the Hoare proof rule description of the semantics of the compiler's source language.

We also demonstrate in these proofs the use of Hoare proof rules to describe the target language. It has been argued that proving machine language programs by such techniques as Hoare proof rules and verification condition generation is impractical because machine language is able to operate at bit levels, modify code, jump to code anywhere in memory, etc. But MC0, as well as most other compilers, produces machine language code that intentionally uses separate areas for code and data, uses a very small subset of the available operations, and has a strictly specified interface between functions. In fact MC0 and the McCarthy-Painter compiler (but not the PL/0 compiler) produce target code without any loops. In other words, the target code produced is well-behaved. For instance, it is the strict function interface definition specifying where arguments are, where the result is returned, where the return point is left, and how the stack is to be preserved that allow us to write the Hoare rule for the CALL statement. Thus for such well-behaved use of machine language, we believe Hoare proof rules are an appropriate method of describing machine language semantics.

6.3 Substitution Formalization

A fairly lengthy description of the substitution formalization used in this proof is given in Section 3.6. We will briefly review here why this formalization was needed. The use of Hoare rules, as described above, requires substitution to be performed on assertions. The use of symbolically expressed assertions and code, which occurs in the part two proof, prevents the substitutions from being performed. Therefore the substitutions are indicated symbolically, and the need arises for a method of manipulating and simplifying them. The complexity of formulas involved in the MC0 part two proof was simply too great to be dealt with without such a method. The properties of substitution given here met that need, and further was found to be easily mechanized when it was applied in the automated part two proof of the McCarthy-Painter compiler. The ability to be mechanized is important because, as is discussed below, we feel it is necessary to apply machine assistance to the problem of proving compilers.

6.4 Machine Assistance for Proofs

We believe that program proving should be done interactively. The human should be involved for the difficult and insightful portions of a proof, while the machine should handle routine and repetitive parts. The proof of MC0 contained far too much routine detail (see the lengthy Section A.8) to reasonably expect humans to prove compilers this way, or to take the time to understand the proof once completed. Further, the amount of detail will grow as we attempt proofs of larger compilers. Yet we know of no automatic proving system capable of proving MC0 without human intervention. In fact the part two proof of MC0 was, we believe, beyond the capability of any computer-assisted proving system, with or without human help (see Section 3.9 for justification of this claim), although the system of Boyer and Moore [Boyer77] has done quite well with proving a simple compiler with practically no user intervention. We envision (in Section 7.2) an interactive compiler proving system using the approach given here that could be built in the immediate future. The point is, we believe compiler proofs to be too tedious to expect to do them by hand, and to be beyond the capability of completely automatic theorem provers; therefore they must be done interactively.

6.5 Axiomatic Descriptions

As explained in Chapter 3, rewrite rules and axioms were used to describe the Lisp-like functions, the source language syntax, the substitution formalization, and stackok (the property of run-time stack preservation). The ease with which we completed portions of the proof of MC0 dealing with these aspects leads us to recommend their use. Such methods tend to be mechanizable (for example, the DTVS system [Guttag76, p. 35] mechanizes such methods) and fit well into our philosophy of proving compilers interactively.

Introducing axiomatic properties at the time of theorem proving helps reduce the accumulation of data into verification conditions, which is invariably a problem in constructing and understanding large program proofs. If all properties of functions (such as the source syntax predicates or the Lisp-like functions) were introduced into the code of the

compiler, then all such properties would appear in all verification conditions involving those functions, whether the properties were needed in that particular verification condition or not. This would substantially reduce the readability of the verification conditions.

Guttag, Horowitz, and Musser [Guttag76] give further justification for such techniques, and in fact use a symbol table as an example of where the methods may be applied.

6.6 Abstract Data Types

The case has been made many places, among them the Alphard language work [Wulf76], that structuring data and the operations upon them using abstractions will aid in producing better programs. It is noted there [Wulf76, p. 254] that grouping the associated data and functions of an abstraction has the following advantages.

1) The places where modifications must be made are more likely to be close together.

2) A smaller portion of the program will be likely to require reverification when a change is made.

3) The user of the abstraction may ignore the details of the implementation.

4) It becomes possible to make *absolute* statements about certain things (e.g., data structures) which are independent of even perverse programmers.

5) The implementation of the abstraction may (sometimes) ignore the complexity of the environment in which the abstraction will be used.

We find such concepts to be applicable to compilers. In particular, the source language, target language, location table (symbol table), and run-time stack each have their own structure and operations. In fact, the function of a compiler may be viewed as rendering the source language structure into its component parts, then building a target language structure, the constraint being to preserve semantics. Thus these data structures play a very important role in the construction of a compiler, and abstractions of them are likely to have a large effect on the quality of the code of a compiler and the ease of its proof.

The source and target languages may be viewed as each containing further structures. In the case of Lisp, the source language for MC0, function definitions are strung together linearly to form larger units. All else in the language is nested calls to functions.

The target language also deals with different levels of structure. The statements of target language were constructed with the Lisp-like functions, but were strung together to form functions and programs with what we called files. These files are strictly linear lists built strictly in order. The code of MC0 has only an operation to add a statement to a file, none to rearrange or take apart a file. Such limitation of the means of accessing a data structure are typical of the data abstraction approach.

Other areas of MC0 where data abstraction was in evidence were the following. The location table of MC0 was built as a Lisp ASSOC list, that is, a list of pairs, each pair being a name and its associated location. The run-time stack was treated at the lowest levels as an array with a top-of-stack pointer in parts of the proof. This was necessitated by the need to access items not at the top during execution. What was really needed was a better abstraction to describe stacks with access into them. Some abstraction was done in the stackok proofs (proofs that the stack was preserved over certain operations).

It became obvious during the proof of MC0 that isolating these various data structures

so that a set of axioms or assertions represents the only interface with the rest of the program is essential to making large programs understandable. We believe that understandability is a prerequisite to correctness and maintainability. What data abstractions we found (or put) in MC0 helped make its proof simpler; more would have been even better. Reaping the benefits cited above from the Alphard paper in proving the properties of these structures within compilers would indeed ease the proving task.

6.7 Other Useful Methods

The case was made for a list describing language in Section 3.5. We believe that lack of good notation severely restricts the complexity of programs that can be proved. Because a compiler is essentially tearing down source language and building up target language, we must have notations for these objects (languages) in order to describe the actions of the compiler. Computer languages are basically character strings, so we must have at least a notation for character strings. But most languages have further structure to them to indicate nested parts, and thus a list (as in Lisp lists) notation seems suitable. Hence we claim a good notation to describe the structures of the source and target languages, often a list notation, is essential to proving compilers. Because many expressions that appear in the assertions are also in the code, we recommend the inclusion of that notation in the language in which a compiler is written.

The proofs here used the two-part proof method, that is, proving first what the compiler produces, then that what was produced was semantically correct (as described in detail in Section 3.8). It divided the proof into two parts that required different kinds of proving systems and different parts of the compiler description. The interface between the parts was relatively small. Therefore we feel this method is an important aid in reducing the complexity of compiler proofs and in managing them.

Nearly all the proofs of operationally expressed compilers have been made using structural induction. In other words, the structure of the proof reflects the source language syntactic structure. From carrying out a number of program proofs (including that of MC0) we believe that a program proof is more easily carried out if the structure of the program's code parallels the structure of the proof. This is one reason we believe that a program and its proof should be constructed together. Then that parallel between code and proof structure can be accomplished without the need to force the structure of the proof, which so often happens when a proof is attempted on an existing program.

If compiler code is made to reflect the syntactic structure of the source language, it will then parallel the structure of the proof, and most likely ease that proof. Thus the compiler should be built to reflect the source language structure, such as MC0 does as a case statement, each case compiling a syntactic type, or such as the PL/0 compiler does in having a procedure to compile each major syntactic type.

7 FUTURE RESEARCH

7.1 Introduction

The work presented here has enabled a very simple compiler (the McCarthy-Painter one) to be proved almost completely interactively, while only the part one proof of the more complex MC0 was done with machine assistance. One case of a compiler with a different organizational structure (the PL/0 compiler) was also proved with the same methods (though proved without machine assistance, for the reasons given before the proof). The motivation for proving compilers given in Chapter 2 combined with the success we have had with the methods of compiler proof presented here lead us to recommend further research in applying these methods. We feel the most important step toward applying them is to construct and use a program proving system with the facilities required for an automated part two proof of a more complex compiler. The other research which we will recommend would be difficult without it. Such a system is described in the next section.

Next we would recommend research toward applying these methods to the proof of optimizing compilers. This is motivated by the fact that nearly all working compilers produce optimizations in their target code of kinds not found in MC0. Therefore we include in this chapter a section on what directions such research should take.

Also included in this chapter are sections about further theorem proving developments that would aid compiler proofs, and about some ideas that may simplify the part one compiler proofs.

7.2 Automating the Part Two Proof

The two parts of the compiler proof are: part one to prove what the compiler produces, and part two to prove that the target language produced has the same effect as the source language. Part two is carried out at a meta level above that of ordinary verification condition generators. Some of the target code produced by the compiler is expressed as a function of the source code being compiled and the state of the compiler (symbol table, etc.). Existing verification condition generators operate on actual code, not expressions representing code. This, and the lack of other necessary features described below, prevented us from carrying out the part two proof of MC0 with machine assistance. Some of the features were actually added to the Xivus system in order to carry out the interactive proof of the McCarthy-Painter compiler.

We believe a meta-level verification condition generator could be constructed which would allow the part two proof to be done interactively. The major requirements of such a system would be:

1. The verification condition generator must indicate substitutions symbolically rather than actually doing them.

2. The verification condition generator should be able to generate from an assertion somewhat after a label back to the label to establish what the assertion is at that label (at least where this is theoretically possible) to avoid having the user do the same by hand.
3. The system must keep and use a great deal of type information about variables, functions, and expressions.
4. The system should allow the language being processed and its Hoare rules to be user specified.
5. The system should gather the results (target code produced) of the part one proof, the statement of correctness of the compiler, and the syntactic case designations into a procedure whose proof constitutes the part two proof of the compiler.
6. The verification condition generator should apply some simplification as it operates.
7. The system must print substitutions and the not-in ($\neg\epsilon$) property (see Section 3.6) in an easily read form.

As mentioned above, some of the target code produced by the compiler is expressed in terms of the source code being compiled and the state of the compiler (symbol table, etc.), which means that we cannot use ordinary verification condition generators for the part two proof. The solution to this problem is to have the verification condition generator indicate substitutions symbolically rather than doing the substitutions.

The type information is required in such a system because the variables, functions, and even constants must be treated differently depending on whether they are source language, target language, compiler code, or were introduced by the verification condition generator itself. For example the target code produced by MC0 for the case of a function call is:

```
< ! FCOMPLIS(CDR(EXP),M,LOCTABLE)
  ! FLOADAC(1-LENGTH(CDR(EXP)),1)
  < 'SUB 'P < 'C 0 0 LENGTH(CDR(EXP)) LENGTH(CDR(EXP)) > >
  < 'CALL LENGTH(CDR(EXP)) < 'E CAR(EXP) > >
>
```

The functions CDR, LENGTH, and subtract (-) were all actually called at compile time. That is, they are simply expressions from the compiler code standing for some part of the target code that was produced by the compiler. FCOMPLIS and FLOADAC are functions used only in the assertions introduced into the compiler for the part one proof, but similarly stand for pieces of target code produced by the compiler. Normally during verification condition generation we must add into the verification condition some form of the Entry assertion of each function call we encounter in the code being processed. Similarly we may use the Exit assertion of such a function as a hypothesis in the verification condition. But that is only true of functions actually executed in the code being processed by the verification condition generator. Therefore our meta-level generator must distinguish compiler functions from target language functions (by use of type information) to properly treat them.

Another place where we treat compiler functions differently is in the application of subrules 6 and 13. They may only be applied to source language functions. Again we must keep track of type in our part two proving system. It might be noted here that compiled functions in target code have the same name as the source language function that was compiled. Not having to translate function names from source to target language is inherent

in our compiler proof method. To require such translation would unduly complicate compiler proofs. However, in our proving system we will keep carefully separated the properties of source language and target language entities, and the context in verification conditions should allow us to distinguish source from target language when necessary. There is nothing to prevent the compiler function names from overlapping source language function names either. Thus our system must be prepared to accommodate this complication, perhaps by the standard trick of adding extensions to function names where necessary to distinguish the type.

Another place where we must distinguish type is in separating the target language variables from the source language variables. Then a simple type check will suffice to apply (at least to variables) subrules 1a, 1b, 1c, 2a, 2b, and 2c. These rules are simply statements of the fact that source language variables and target language variables are different. If we further distinguish the stack (m), the stack pointer (P), and the registers (Ri) as different types within the target language, subrules 2d, 2e, and 2f become type checks. Obviously this type information will be of value to the theorem prover in applying the simplification rules (including some of the subrules), as well as of value to the verification condition generator.

We may get more benefit out of this type system by being able to determine types of expressions as well as of variables. For instance, getv(MAP) in the proof of the McCarthy-Painter compiler is used to extract a list of declared source language variables from the symbol table MAP. Hence getv requires a type compiler-variable-standing-for-symbol-table as its argument and produces a type source-language-variables-expression as its result. Using this type information would immediately tell us that items of certain other types are not contained in (-c) a getv(MAP), rather than going through an arduous application of subrules to establish this fact. Typing of expressions will also allow us to apply all parts of subrule 1 to expressions as well as variables by means of a type check.

A problem that arises with typing expressions is that we may get a mixed type. For instance, if we have the expression m[P] we have both the stack type and the stack pointer type. So our system needs to be able to keep track of mixed types such as the union of target-stack type and target-pointer type. Such unions of types have been treated in language design before, so they should present no major problems.

This handling of mixed types would bring us more flexibility in dealing with another problem. That problem is that in many cases during the part two proof we have expressions representing mixtures of target and source language terms. For example, after applying the c-k substitution in the McCarthy-Painter proof, we have translated source language constants to target language, but the expression still contains source language variables. A union of the types source-variable and target-constant would allow us to type this expression.

The type mechanism should also distinguish between items that will be atomic variable names and those that will be expressions in the source or target language assertions. For example, in the MC0 proof, the variables EXP and M are both compiler variables that appear in the expressions representing target code that will be processed by the verification condition generator. However, EXP will be a source language expression, while M will always be an integer. Therefore we can say that M will never contain any source language variables, but EXP probably will. That fact affects the way in which subrules may be applied to either EXP or M.

The theorem prover of course must be able to access this type information in defining and applying the rewrite rules and axioms. For instance, the system user must be able to require when defining the subrules that those items that we have designated by the letter D in

the subrule list must be atomic identifiers, not expressions (in target or source language, that is; they could perfectly well be expressions from the compiler code that always represent a target or source variable). Since we have not defined substitution for an expression, nor are the subrules valid if applied to such, any system that did not check that substitutions were made only for identifiers would allow us to construct an erroneous proof.

The requirement of user definition of language and Hoare rules comes about for several reasons. The language to be processed through the verification condition generator for a part two proof is of course the target language of the compiler. Because of the great variety of target languages, one would like to be able to change the language. This could be done by using the techniques of parser generators, in which a description of the language to be parsed is entered and a parser for that language is produced. A more pressing reason for desiring this capability is that the language which the system must process is not simply the target language. It is the target language plus the symbolic expressions from the compiler representing target language. Therefore the language actually depends on the way the compiler is written.

The user must be able to specify the Hoare rules that are used by the verification condition generator to operate on the target language. This is required because some of the Hoare rules depend on the way the compiler is written, not just on the target language. The function interface, that is, where arguments are passed, where the result is returned, and how execution control returns to the calling procedure, must show up in both the Hoare rule for the call statement and the Hoare formula for the statement of correctness (which is used as a Hoare proof rule during processing of the result of a recursive call to the compiling function).

For these reasons we believe that the system to automate part two proofs will have to allow the user to specify each Hoare rule and the Hoare formula for the statement of correctness of the compiler. The verification condition generator must then work directly from these Hoare rules rather than having the Hoare rules built into its code. Otherwise a new verification condition generator would have to be written for each new compiler, or even for a change in the statement of what is to be proved about the compiler.

The part two proof consists of generating verification conditions from the right hand side of the Hoare formula that is the statement of compiler correctness back (assuming a backwards generator such as was used in this research) through the target code produced for each case, and the proof of each resulting verification condition. The system for automating the part two proof should set up a procedure containing these pieces such that proof of the procedure constituted a part two proof. The right hand side of the Hoare formula that is the statement of correctness becomes the Exit assertion. The left-hand side becomes the Entry assertion. The code is then a case (or nested conditional) statement where the conditional test for each case is the predicate denoting the various source language syntactic cases (ISSUM, etc.), and the code executed for each case is the result of the part one proof. The verification condition generation then produces one (or possibly more) verification condition to represent each of the source language structural types required to make this structural induction proof.

A problem arises when two different representations exist for the same code. An example of this occurs in the MC0 proof in the AND case (subcase with arguments). There we have one representation which shows the target code in recursive terms, that is, as FCOMPEXP of the first argument and of the remaining arguments. This form readily proves the part two proof for this subcase except for the problem of containing a goto to the label L1, which is buried inside the FCOMPEXP of the remaining arguments. The second

representation of exactly the same code shows in detail where the labels lie. It is then possible to run the verification condition condition generator back through the latter representation to establish the assertions at the labels, discard the verification condition(s) produced, then run the generator through the other representation of the same code to actually produce the desired verification condition. The ability to accept two different representations of the same code is one which could either be built into the verification condition generator or could be handled during the part two set up.

Another problem that arises in setting up all syntactic cases of the part two proof as a single procedure is that a label can appear in two different places. An example of this appears in the MC0 proof in the AND case. The same label L1 is used in both the subcase with no arguments and the subcase with arguments. This is to be expected, since the code produced is the same for the two subcases except for additional code for additional arguments. The solution here is to relabel each subcase with unique sets of labels during part two set up.

The part two set up will also have to accept Hoare formulas defining the action of certain other compiler procedures in addition to the main compiling procedure. In the MC0 proof these other procedures were COMPLIS, LOADAC, and MKPUSH. The reason for the special treatment of these procedures is that the target code they produced depended on the number of arguments allowed for some source functions (AND, etc.). In other words, proof of the code from these functions involved induction on the number of source language arguments, which is a different induction than the structural induction on nested source language functions. Hoare formulas were therefore written to express the effects of the procedures COMPLIS, LOADAC, and MKPUSH in the proof of MC0. Whenever a piece of target code was encountered during verification condition generation that was expressed as the result of COMPLIS, LOADAC, or MKPUSH, the appropriate Hoare formula was applied as a Hoare proof rule. This required, however, that these Hoare formulas be proved correct. The same method was used as for the statement of correctness of the compiler; that is, verification conditions were generated from the right-hand side of a recursive form of each Hoare formula (for COMPLIS, LOADAC, and MKPUSH) back through the target code of the respective procedure, and then the verification conditions were proved.

Because expressions tend to grow faster without being able to carry out substitutions, the verification condition generator would have to simplify as it generated to keep expressions to manageable size. This same complexity of the verification conditions is what requires the system to have a good system of printing verification conditions for viewing by the interactive user. Because of the extreme frequency with which substitutions and not-ins ($\neg\epsilon$) appear in the verification conditions, special attention must be paid to printing them out in a readable manner if the user is expected to interact during the proof in an intelligent manner.

The register notation used in the hand part two proof of MC0 should be changed if it were done by machine. The problem with it arises when applying the statement of correctness as a Hoare proof rule during verification condition generation. Then it is required to quantify an undetermined number (depending on what is in the syntactic types contained in arguments) of registers. Actually all registers (except R1) could be quantified, since the target code never depends on any higher registers to be saved. So in a machine proof we would refer to all registers (except R1) collectively by use of an R array, and just quantify the entire array.

Another problem area is that of applying the subrules 6 and 13 in a machine proof. Careful examination of them shows that they contain quantification of function names. This

at first appears to require a higher-order logic in the theorem prover. However, it would always instantiate the function to some given function in the theorem at the time of applying the subrule. Thus no such quantification would ever enter the theorems being proved. The provision must be made to quantify functions in the subrules when they are introduced and when they are applied; otherwise subrules 6 and 13 would have to be entered into the system again every time they were to be applied to a different function.

7.3 Proving Optimizing Compilers

In Section 3.12 it was pointed out that the methods used here would not work for optimizations in the target code which depend on the context in which a particular source syntactic type is found. One approach to working with such optimizations would be to assign a name and predicate to each such context. For example, in C0, the compiler on which MC0 is based, boolean syntactic types (AND, OR, NOT) that lie immediately inside another boolean type are compiled into shorter code than normal. We could name this context Inbool, and the predicate which tells us if Inbool exists at a given point could be called Isinbool. Isinbool would have to be a function of the program and where within that program we were discussing.

Then in the part one proof we would specify what the target code would be for compiling the syntactic type in question for both the case where the context predicate is true and where the context predicate is false. The statement of correctness would have to be prefixed by the hypothesis that the context predicate is false, and another statement of correctness would be added that is prefixed by the context predicate being true. The additional statement of correctness portion must then state exactly the conditions (in the form of a Hoare formula in target language terms) that hold upon entry and exit of the optimized target code.

It then becomes necessary during the part one proof to establish whether the context predicate holds or not at each point that the compiler recurses. This information allows us to know which part of the statement of correctness (the context predicate true or false part) is to be assumed as the inductive assumption at each point in the part two proof at which we encounter a recursive call to the compiler. The value of the context predicate could actually be passed as an argument to the compiler at each call. However, compilers often do not recurse and pass in the information that an optimization is to be made, but instead call a different point in the compiler that does the optimization. In that case the part one proof must show that what results is the same as would occur if the compiler recursed with the information that the optimization was to take place. It would of course be easier to prove the compiler if the recursion actually took place and the optimization information were passed straightforwardly. Perhaps it would also add understandability to compilers if they were written that way.

Another approach would be to prove semantic equivalence of the optimized form in its special context to the unoptimized form without benefit of the context. This would allow us to build on the proof we have already done rather than doing it all again with a few aspects changed. An example of an equivalence proof between code produced by an optimized version and an unoptimized version of a compiler is included in Section A.9. It serves as an example that these techniques can be extended to prove optimizing compilers.

7.4 More Powerful Theorem Prover

Much of the theorem prover work in the proof of MC0 involved proof by contradiction in the hypotheses of a theorem (verification condition). The same contradictions kept appearing, which meant that execution of the compiler could not have followed that execution path for the same reason it could not have followed similar paths. A command to "prove this theorem like theorem X" would have saved a great amount of time. This could probably be accomplished if a history of commands entered to the theorem prover were kept and interrogated when asked to repeat proofs. It could even be done without the human asking if a search were made on new theorems to see if they had the same or stronger hypotheses and the same or weaker conclusions than those that were actually used in previous proofs.

The interactive part of the proof of MC0 was not done without errors. Several times proofs had to be redone with an error corrected in either the program or assertions. The Xivus system recognizes when a verification condition is regenerated exactly as before, and if it has been proved before, the user is not asked to reprove it. However, many times the verification condition is changed in parts that were not even used in the proof, such as a conclusion in a proof by contradiction in the hypotheses. Other times the new version was weaker than the old, and so the same proof applied. A powerful redo feature would help greatly in the correction process, and could also be used to prove different versions of a compiler, such as optimizing versions. Then only the portions of the compiler producing the different or optimized code would need to be proved.

Another area where machine help would be welcome is in suggesting axioms, rewrite rules, or equality substitutions to apply. Quite often a simple pattern match would find the axiom or rewrite rule that bridges the gap between what we have (hypotheses) and what we need (conclusions). Particularly when the expression of the theorem is quite large, such a facility would save the human much time in looking through it. Many of the theorems in the MC0 proof were of the form hypotheses imply expression1 = expression2. Pattern matching techniques to find the similarities in the two expressions and then pinpoint the differences could easily produce good suggestions as to what axioms, rewrite rules, or substitutions to apply.

7.5 Part One Proof Reduction

The stating of assertions for the part one proof of MC0 was, for the most part, tedious and without the need for insight. Once a good notation was used, the job became an easy one of writing what target language the compiler produced for each source language syntactic case. The only difficult assertions were the ones involving functions with a variable number of arguments (AND, OR, and COND), since those assertions had to reflect the induction on the number of arguments. Many of the simpler assertions were written by putting a sample of a given syntactic type into the compiler and examining the compiled result. Such a compiled result or a simple generalization of it was invariably the correct assertion.

This suggested a method of having the computer write most of the part one assertions, thereby greatly reducing the size of the job done by the human. The idea is to symbolically execute the code of the compiler on a symbolically expressed source syntactic type. The idea of a symbolic executer is not new; King [King75,King76] and others have applied it with

essentially the features described below. Such a symbolic executer would have to be able to backtrack down each possible execution path. Of course it would have to do theorem proving at conditional statements to determine if the true condition or false condition branches would always or never be taken. It would have to stop itself from descending too far on a series of recursive calls. Perhaps this could be under user control since the user would know where he was breaking the recursion in the proof for which he was trying to generate assertions.

Another idea which might entirely eliminate the part one proof is to write the compiler in a pattern matching language. Each source language syntactic type would be represented by a pattern, and the output of the compiler could be specified as a construction made from the parts that matched the pieces of the pattern. A simple example of a function written in such a manner is found in Hoare's paper on recursive data structures [Hoare75, p. 110]. Such a compiler would look very much like a case statement (or the equivalent nested if-elseif statement) with each case being a rewrite rule, that is, a pattern to be matched followed by the construction to be output. Except for handling such matters as the symbol table or variable numbers of arguments, the compiler written in such a language would appear almost exactly like the assertions that had to be proved in the part one proof.

REFERENCES

[Bledsoe73] Bledsoe, W. W. and Bruell, P., "A man-machine theorem proving system," *Third International Joint Conference on Artificial Intelligence, Advance Papers on the Conference*, 1973, pp. 56-65.

[Blum69] Blum, E. K., "Towards a theory of semantics and compilers for programming languages," *Journal of Computer and System Sciences* 3, 3 (August 1969), pp. 248-275.

[Boyer75] Boyer, R. S. and Moore, J S., "Proving theorems about LISP functions," *Journal of the ACM* 22, 1 (January 1975), pp. 129-144.

[Boyer77] Boyer, R. S. and Moore, J S., *A computer proof of the correctness of a simple optimizing compiler for expressions*, Stanford Research Institute Technical Report 5 (January 1977), 59 pp.

[Burge68] Burge, W. H., *Proving the correctness of a compiler*, Report RC-2111(10695), IBM Research Division, Yorktown Heights, New York (June 1968), 19 pp.

[Burstall69a] Burstall, R. M., "Proving properties of programs by structural induction," *Computer Journal* 12, 1 (February 1969), pp. 41-48.

[Burstall69b] Burstall, R. M. and Landin, P. J., "Programs and their proofs: an algebraic approach," *Machine Intelligence 4*, B. Meltzer and D. Michie [Eds.], American Elsevier, New York, 1969, pp. 17-43.

[Chirica75] Chirica, L. M. and Martin, D. F., "An approach to compiler correctness," *Proceedings International Conference on Reliable Software*, April 1975, pp. 96-103.

[Chirica76] Chirica, L. M., *Contributions to compiler correctness*, UCLA Computer Science Report UCLA-ENG-7697 (October 1976), 304 pp.; also PhD Thesis, UCLA, 1976.

[Clarke77] Clarke, E. M., "Programming language constructs for which it is impossible to obtain good Hoare-like axiom systems," *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, January 1977, pp. 10-20.

[Elspas72] Elspas, B., Levitt, K. N., Waldinger, R. J., and Waksman, A., "An assessment of techniques for proving program correctness," *ACM Computing Surveys*, 4, 2 (June 1972), pp. 97-147.

[Floyd67] Floyd, R. W., "Assigning meanings to programs," *Proceedings of a Symposium in Applied Mathematics, Vol. 19--Mathematical Aspects of Computer Science*, J. T. Schwartz [Ed.], Providence, Rhode Island, 1967, pp. 19-32.

[Germano75] Germano, G. and Maggiolo-Schettini, A., "Proving a compiler correct: a simple approach," *Journal of Computer and System Sciences* 10, 3 (June 1975), pp. 370-383.

[Good75] Good, D. I., London, R. L., and Bledsoe, W. W., "An interactive program verification system," *IEEE Transactions on Software Engineering* SE-1, 1 (March 1975), pp. 59-67.

[Guttag76] Guttag, J. V., Horowitz, E., and Musser, D. R., *Abstract data types and software validation*, USC Information Sciences Institute Report ISI/RR-76-48 (August 1976), 46 pp.

[Hoare69] Hoare, C. A. R., "An axiomatic basis for computer programming," *Communications of the ACM* 12, 10 (October 1969), pp. 576-580,583.

[Hoare73] Hoare, C. A. R. and Wirth, N., "An axiomatic definition of the programming language PASCAL," *Acta Informatica* 2, 4 (1973), pp. 335-355.

[Hoare75] Hoare, C. A. R., "Recursive data structures," *International Journal of Computer and Information Sciences* 4, 2 (June 1975), pp. 105-132.

[Igarashi75] Igarashi, S., London, R. L., and Luckham, D. C., "Automatic program verification I: a logical basis and its implementation," *Acta Informatica* 4, 2 (1975), pp. 145-182.

[Jensen74] Jensen, K. and Wirth, N., "Pascal user manual and report," *Lecture Notes in Computer Science, Vol. 18*, G. Goos and J. Hartmanis [Eds.], Springer-Verlag, Berlin, Heidelberg, and New York, 1974, 170 pp.

[Kaplan67] Kaplan, D. M., *Correctness of a compiler for Algol-like programs*, Stanford Artificial Intelligence Memo No. 48 (July 1967), 34 pp. + appendix.

[King75] King, J. C., "A new approach to program testing," *Proceedings International Conference on Reliable Software*, April 1975, pp. 228-233.

[King76] King, J. C., "Symbolic execution and program testing," *Communications of the ACM* 19, 7 (July 1976), pp. 385-394.

[London71] London, R. L., *Correctness of two compilers for a Lisp subset*, Stanford Artificial Intelligence Project Memo AIM-151 (October 1971), 41 pp.

[London72] London, R. L., "Correctness of a compiler for a LISP subset," *Proceedings of an*

ACM Conference on Proving Assertions about Programs, SIGPLAN Notices 7, 1 (January 1972), pp. 121-127; also *SIGACT News*, 14 (January 1972), pp. 121-127.

[London75] London, R. L., "A view of program verification," *Proceedings International Conference on Reliable Software*, April 1975, pp. 534-545.

[London77] London, R. L., Guttag, J. V., Horning, J. J., Lampson, B. W., Mitchell, J. G., and Popek, G. J., "Proof rules for the programming language Euclid," Technical Report, May 1977, 24 pp.

[Luckham77] Luckham, D. C. and Suzuki, N., "Proof of termination within a weak logic of programs," *Acta Informatica* 8, 1 (1977), pp. 8-21.

[McCarthy63] McCarthy, J., "Towards a mathematical science of computation," *Information processing; Proceedings of IFIP Congress 1962*, C. M. Popplewell [Ed.], North Holland Publishing Co., Amsterdam, 1963, pp. 21-28.

[McCarthy66] McCarthy, J., "A formal description of a subset of ALGOL," *Formal Language Description Languages for Computer Programming*, T. B. Steel, Jr. [Ed.], North-Holland Publishing Co., Amsterdam, 1966, pp. 1-12.

[McCarthy67] McCarthy, J. and Painter, J. A., "Correctness of a compiler for arithmetic expressions," *Proceedings of a Symposium in Applied Mathematics, Vol. 19--Mathematical Aspects of Computer Science*, J. T. Schwartz [Ed.], Providence, Rhode Island, 1967, pp. 33-41.

[Milner72] Milner, R. and Weyhrauch, R., "Proving compiler correctness in a mechanized logic," *Machine Intelligence 7*, B. Meltzer and D. Michie [Eds.], Edinburgh University Press, Edinburgh, and Halsted Press, New York, 1972, pp. 51-70.

[Morris72] Morris, F. L., *Correctness of translations of programming languages -- an algebraic approach*, Stanford Artificial Intelligence Project Memo AIM-174 (August 1972), 126 pp.; also PhD Thesis, Stanford University, 1972.

[Morris73] Morris, F. L., "Advice on structuring compilers and proving them correct," *Conference Record of ACM Symposium on Principles of Programming Languages*, October 1973, pp. 144-152.

[Naur66] Naur, P., "Proof of algorithms by general snapshots," *BIT* 6, 4 (1966), pp. 310-316.

[Newey75] Newey, M. C., *Formal semantics of LISP with applications to program correctness*, Stanford Artificial Intelligence Laboratory Memo AIM-257 (January 1975), 173 pp.; also PhD Thesis, Stanford University, 1975.

[Painter67] Painter, J. A., *Semantic correctness of a compiler for an Algol-like language*, Stanford Artificial Intelligence Memo No. 44 (March 1967), 130 pp.; also PhD Thesis, Stanford University, 1967.

[Ragland73] Ragland, L. C., *A verified program verifier*, University of Texas at Austin Department of Computer Sciences Technical Report No. 18, May 1973, 153 pp.; also PhD thesis, University of Texas at Austin, 1973.

[Samet75] Samet, H., *Automatically proving the correctness of translations involving optimized code*, Stanford Artificial Intelligence Laboratory Memo AIM-259, May 1975, 214 pp.; also PhD Thesis, Stanford University, 1975.

[Scott70] Scott, D., "Outline of a mathematical theory of computation," *Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems*, 1970.

[Suzuki75] Suzuki, N., "Verifying programs by algebraic and logical reduction," *Proceedings International Conference on Reliable Software*, April 1975, pp. 473–481.

[Teitelman75] Teitelman, W., *Interlisp reference manual*, Xerox Palo Alto Research Center, Palo Alto, December, 1975.

[Wada73] Wada, K., Masunaga, Y., Noguchi, S., and Oizumi, J., "Correctness of a syntax directed compiler," *Record of Electrical and Communication Engineering Conversazione Tohoku University (Sendai, Japan)* 42, 2 (May 1973), pp. 103–112 (Japanese Language).

[Wirth71] Wirth, N., "The programming language Pascal," *Acta Informatica* 1, 1 (1971), pp. 35-63.

[Wirth76] Wirth, N., *Algorithms + data structures = programs*, Prentice-Hall, Englewood Cliffs, N.J., 1976.

[Wulf76] Wulf, W. A., London, R. L., and Shaw, M., "An introduction to the construction and verification of Alphard programs," *IEEE Transactions on Software Engineering* SE-2, 4 (December 1976), pp. 253–265.

APPENDIX

A.1 Definitions of F Functions

The F function FCOMPEXP was for every source language syntactic case or subcase defined by an assumption of the form:

```
  < ! OUTFILE'
    ! FCOMPEXP(EXP,M,LOCTABLE)
  >
=
  [The expression found in figure 4-3 detailing the target language
   instructions to be found in OUTFILE for this case or subcase]
```

For example, for the case of syntactic type AND, subcase more than zero arguments, we will assume the following formula:

```
< ! OUTFILE'
  ! FCOMPEXP(EXP,M,LOCTABLE)
>

=

< ! OUTFILE'
  ! FCOMPANDOR(CDR(EXP),M,L1,FALSE,LOCTABLE)
  < 'MOVEI 1 < 'QUOTE 'T > >
  < 'JRST 0 L2 >
  L1
  < 'MOVEI 1 0 >
  L2
>
```

Assuming this subgoal in the theorem prover defines FCOMPEXP for this subcase as:

```
< ! FCOMPANDOR(CDR(EXP),M,L1,FALSE,LOCTABLE)
  < 'MOVEI 1 < 'QUOTE 'T > >
  < 'JRST 0 L2 >
  L1
  < 'MOVEI 1 0 >
  L2
>
```

The other F functions are given here with their defining assumptions in order of source language syntactic type case.

function definition case:

```
  < ! OUTFILE'
    ! FMKPUSH(N, M)
  >
= OUTFILE'
```

where N < M.

```
  < ! OUTFILE'
    ! FMKPUSH(N, M)
  >

=

  < ! OUTFILE'
    < 'PUSH 'P M >
    ! FMKPUSH(N, M+1)
  >
```

where N ≥ M.

AND (no arguments) subcase:

```
  < ! OUTFILE'
    ! FCOMPANDOR(U, M, L, FLG, LOCTABLE)
  >
= OUTFILE'
```

AND (n > 0 arguments) subcase (FLG is FALSE):

```
  < ! OUTFILE'
    ! FCOMPANDOR(U, M, L, FLG, LOCTABLE)
  >

=

  < ! OUTFILE'
    ! FCOMPEXP(CAR(U), M, LOCTABLE)
    < 'JUMPE 1 L >
    ! FCOMPANDOR(CDR(U), M, L, FLG, LOCTABLE)
  >
```

OR (no arguments) subcase:

The same form applies is this subcase as did for AND (no arguments).

OR (n > 0 arguments) subcase (FLG is TRUE):

```
< ! OUTFILE'
  ! FCOMPANDOR(U, M, L, FLG, LOCTABLE)
>
```

=

```
< ! OUTFILE'
  ! FCOMPEXP(CAR(U), M, LOCTABLE)
  < 'JUMPN 1 L >
  ! FCOMPANDOR(CDR(U), M, L, FLG, LOCTABLE)
>
```

COND (no arguments) subcase:

```
< ! OUTFILE'
  ! FCOMCOND(U, M, L, LOCTABLE)
>
```

=

```
< ! OUTFILE'
  L
>
```

COND (n > 0 arguments) subcase:

```
< ! OUTFILE'
  ! FCOMCOND(U, M, L, LOCTABLE)
>
```

=

```
< ! OUTFILE'
  ! FCOMPEXP(CAAR(U), M, LOCTABLE)
  < 'JUMPE 1 L3 >
  ! FCOMPEXP(CADAR(U), M, LOCTABLE)
  < 'JRST L >
  L3
  ! FCOMCOND(CDR(U), M, L, LOCTABLE)
>
```

function call case:

```
  < ! OUTFILE'
    ! FCOMPLIS(U, M, LOCTABLE)
  >
= OUTFILE'
```

where U is a null argument list.

```
  < ! OUTFILE'
    ! FCOMPLIS(U, M, LOCTABLE)
  >

=

  < ! OUTFILE'
    ! FCOMPEXP(CAR(U), M, LOCTABLE)
    < 'PUSH 'P 1 >
    ! FCOMPLIS(CDR(U), M-1, LOCTABLE)
  >
```

where U is a non-null argument list.

```
  < ! OUTFILE'
    ! FLOADAC(N, K)
  >
- OUTFILE'
```

where N > 0.

```
  < ! OUTFILE'
    ! FLOADAC(N, K)
  >

-

  < ! OUTFILE'
    < 'MOVE K N 'P >
    ! FLOADAC(N+1, K+1)
  >
```

where N ≤ 0.

lambda case:
The same forms apply for FCOMPLIS in this case as did for the function call case.

A.2 Axioms Describing Source Language Syntax

These axioms are given, along with labels for use by the theorem prover, in the order of the syntactic cases. A dot after a variable name indicates universal quantification of that variable over the entire axiom. ISEXPRESSION is a predicate which holds if and only if its argument is a legal source language expression. A description of the axioms for the AND syntactic type is given in Section 3.14. The axioms for the other syntactic types follow similar patterns.

case of a function definition:

LISFUNDEF3:

ISFUNCTIONDEF(X.) → ISEXPRESSION(CADDDR(X.))

case of NIL:

LISNIL1:

ISNIL(X.) → NULL(X.)

case of T:

LIST1:

IST(X.) → X. = 'T

LIST2:

IST(X.) → NOT NULL(X.)

ISNUMBER case:

LISNUMBER1:

ISNUMBER(X.) → NUMBERP(X.)

LISNUMBER2:

ISNUMBER(X.) → NOT NULL(X.)

ISIDENTIFIER case:

LISIDENTIFIER1:

ISIDENTIFIER(X.) → ATOM(X.)

LISIDENTIFIER2:

```
        ISIDENTIFIER(X.)
→         NOT NULL(X.)
        ∧ (X. ≠ 'T)
        ∧ NOT NUMBERP(X.)
```

AND case:

LISAND1:

ISAND(X.) → CAR(X.) = 'AND

LISAND2:

```
                ISAND(X.)
            →     NOT NULL(X.)
                ∧ (X. ≠ 'T)
                ∧ NOT NUMBERP(X.)
                ∧ NOT ATOM(X.)

        LISAND3:
                  ISEXPRESSION(X.)
                ∧ (CAR(X.) = 'AND)
            → ISLISTOFEXP(CDR(X.))

        LISAND4:
            ISLISTOFEXP(Z.) → ISAND(CONS('AND,Z.))
```

OR case:

```
        LISOR1:
            ISOR(X.) → CAR(X.) = 'OR

        LISOR2:
                ISOR(X.)
            →     NOT NULL(X.)
                ∧ (X. ≠ 'T)
                ∧ NOT NUMBERP(X.)
                ∧ NOT ATOM(X.)
                ∧ (CAR(X.) ≠ 'AND)

        LISOR3:
                  ISEXPRESSION(X.)
                ∧ (CAR(X.) = 'OR)
            → ISLISTOFEXP(CDR(X.))

        LISOR5:
            ISLISTOFEXP(Z.) → ISOR(CONS('OR,Z.))
```

NOT case:

```
        LISNOT1:
            ISNOT(X.) → CAR(X.) = 'NOT

        LISNOT2:
                ISNOT(X.)
            →     NOT NULL(X.)
                ∧ (X. ≠ 'T)
                ∧ NOT NUMBERP(X.)
                ∧ NOT ATOM(X.)
                ∧ (CAR(X.) ≠ 'AND)
                ∧ (CAR(X.) ≠ 'OR)
```

```
      LISNOT3:
                  ISEXPRESSION(X.)
               ∧ (CAR(X.) = 'NOT)
            → ISEXPRESSION(CADR(X.))

COND case:
      LISCOND1:
            ISCOND(X.) → CAR(X.) = 'COND

      LISCOND2:
                ISCOND(X.)
            →     NOT NULL(X.)
               ∧ (X. ≠ 'T)
               ∧ NOT NUMBERP(X.)
               ∧ NOT ATOM(X.)
               ∧ (CAR(X.) ≠ 'AND)
               ∧ (CAR(X.) ≠ 'OR)
               ∧ (CAR(X.) ≠ 'NOT)

      LISCOND3:
                  ISEXPRESSION(X.)
               ∧ (CAR(X.) = 'COND)
            → ISCONDLIST(CDR(X.))

      LISCOND4:
            ISCONDLIST(X.) ∧ NOT NULL(X.) → ISCONDLIST(CDR(X.))

      LISCOND5:
                  ISCONDLIST(X.)
               ∧ NOT NULL(X.)
            →     ISEXPRESSION(CAAR(X.))
               ∧ ISEXPRESSION(CADAR(X.))

      LISCOND6:
            ISCONDLIST(Z.) → ISCOND(CONS('COND,Z.))

QUOTE case:
      LISQUOTE1:
            ISQUOTE(X.) → CAR(X.) = 'QUOTE

      LISQUOTE2:
```

```
                ISQUOTE(X.)
            →     NOT NULL(X.)
                ∧ (X. ≠ 'T)
                ∧ NOT NUMBERP(X.)
                ∧ NOT ATOM(X.)
                ∧ (CAR(X.) ≠ 'AND)
                ∧ (CAR(X.) ≠ 'OR)
                ∧ (CAR(X.) ≠ 'NOT)
                ∧ (CAR(X.) ≠ 'COND)
```

case of a function call:

```
    LISFUNCALL1:
        ISFUNCTIONCALL(X.) → ATOM(CAR(X.))

    LISFUNCALL2:
                ISFUNCTIONCALL(X.)
            →     NOT NULL(X.)
                ∧ (X. ≠ 'T)
                ∧ NOT NUMBERP(X.)
                ∧ NOT ATOM(X.)
                ∧ (CAR(X.) ≠ 'AND)
                ∧ (CAR(X.) ≠ 'OR)
                ∧ (CAR(X.) ≠ 'NOT)
                ∧ (CAR(X.) ≠ 'COND)
                ∧ (CAR(X.) ≠ 'QUOTE)

    LISFUNCALL3:
                  ISEXPRESSION(X.)
                ∧ ATOM(CAR(X.))
                ∧ (CAR(X.) ≠ 'COND)
            → ISLISTOFEXP(CDR(X.))
```

case of a lambda:

```
    LISLAMBDA1:
        ISLAMBDA(X.) → CAAR(X.) = 'LAMBDA

    LISLAMBDA2:
```

```
            ISLAMBDA(X.)
        →     NOT NULL(X.)
            ∧ (X. ≠ 'T)
            ∧ NOT NUMBERP(X.)
            ∧ NOT ATOM(X.)
            ∧ (CAR(X.) ≠ 'AND)
            ∧ (CAR(X.) ≠ 'OR)
            ∧ (CAR(X.) ≠ 'NOT)
            ∧ (CAR(X.) ≠ 'COND)
            ∧ (CAR(X.) ≠ 'QUOTE)
            ∧ NOT ATOM(CAR(X.))

    LISLAMBDA3:
            ISEXPRESSION(X.)
          ∧ (CAAR(X.) = 'LAMBDA)
        → ISEXPRESSION(CADDAR(X.))

    LISLAMBDA4:
            ISEXPRESSION(X.)
          ∧ (CAAR(X.) = 'LAMBDA)
        → ISLISTOFEXP(CDR(X.))
```

general argument list axioms:

```
    LISLISTOFEXP1:
            ISLISTOFEXP(X.)
          ∧ NOT NULL(X.)
        → ISLISTOFEXP(CDR(X.))

    LISLISTOFEXP2:
            ISLISTOFEXP(X.)
          ∧ NOT NULL(X.)
        → ISEXPRESSION(CAR(X.))
```

A.3 Rewrite Rules Describing Lisp Functions

These rules are given along with labels for use by the theorem prover. The dashed right arrow (-->) indicates that the pattern to its left is replaced by the form to the right. *Variables followed by dots are ones that may be bound to any expression to cause a match of the left side.*

CONS, APPEND rules:

```
    RCONS1:
            APPEND(XX.,CONS(YY.,ZZ.))
        --> APPEND(RIGHTCONS(XX.,YY.),ZZ.)
```

```
RCONS2:
     APPEND(XX.,'NIL)  -->  XX.

RCONS3R:
          APPEND(XX.,APPEND(YY.,ZZ.))
     -->  APPEND(APPEND(XX.,YY.),ZZ.)

RCONS4:
          APPEND(XX.,RIGHTCONS(YY.,ZZ.))
     -->  RIGHTCONS(APPEND(XX.,YY.),ZZ.)
```

CDR of a CONS rule:

```
RCDRCONS:
     CDR(CONS(XX.,YY.))  -->  YY.
```

CAR-CDR combination rules:

```
RCAAR:
     CAAR(XX.)  -->  CAR(CAR(XX.))

RCADR:
     CADR(XX.)  -->  CAR(CDR(XX.))

RCDDR:
     CDDR(XX.)  -->  CDR(CDR(XX.))

RCAADR:
     CAADR(XX.)  -->  CAR(CAR(CDR(XX.)))

RCADAR:
     CADAR(XX.)  -->  CAR(CDR(CAR(XX.)))

RCADADR:
     CADADR(XX.)  -->  CAR(CDR(CAR(CDR(XX.))))
```

A.4 Basis for Part One Proofs

Following is a list of the source language syntactic description axioms and the rewrite rules describing the Lisp-like functions which were required to prove the part one proof of each syntactic type case. The axioms have names beginning with the letter L, and the rewrite rules with R. The name before the colon in each list is the procedure or function name whose proof requires the basis immediately following the colon. The full axioms and rewrite rules are found in Section A.2 and Section A.3.

A.4.1 ISNIL Case

COMPEXP: RCONS1, RCONS2, LISNIL1, LISCOND3, LISFUNCALL3, LISLAMBDA4, LISLAMBDA3.

A.4.2A IST Case

COMPEXP: LIST2, LIST1, RCONS1, RCONS2, LISCOND3, LISFUNCALL3, LISLAMBDA4, LISLAMBDA3.

A.4.2B ISNUMBER Case

COMPEXP: LISNUMBER2, RCONS1, RCONS2, LISNUMBER1, LISCOND3, LISFUNCALL3, LISLAMBDA4, LISLAMBDA3.

A.4.3 ISIDENTIFIER Case

COMPEXP: LISIDENTIFIER2, RCONS1, RCONS2, LISIDENTIFIER1, LISCOND3, LISFUNCALL3, LISLAMBDA4, LISLAMBDA3.

A.4.4-0 ISAND No Arguments Subcase

COMPEXP: LISAND2, RCONS1, RCONS2, LISCOND3, LISAND1, LISFUNCALL3, LISLAMBDA4, LISLAMBDA3.

COMBOOL: LISAND3, LISAND1, LISOR3, LISNOT3.

COMPANDOR: LISLISTOFEXP1, LISLISTOFEXP2.

A.4.4-N ISAND With Arguments Subcase

COMPEXP: LISAND2, LISAND1, LISAND3, LISLISTOFEXP1, LISAND4, RCONS1, RCONS2, RCONS4, RCDDR, RCADR, RCDRCONS, LISCOND3, LISFUNCALL3, LISLAMBDA4, LISLAMBDA3.

COMBOOL: LISAND3, LISAND1, LISOR3, LISNOT3.

COMPANDOR: LISLISTOFEXP1, LISLISTOFEXP2, RCONS1, RCONS4, RCONS3R.

A.4.5-0 ISOR No Arguments Subcase

COMPEXP: LISOR2, RCONS2, RCONS2, LISCOND3, LISOR1, LISFUNCALL3, LISLAMBDA4, LISLAMBDA3.

COMBOOL: LISAND3, LISOR2, LISOR3, RCONS1, RCONS2, LISNOT3, LISOR1.

COMPANDOR: LISLISTOFEXP1, LISLISTOFEXP2.

A.4.5-N ISOR With Arguments Subcase

COMPEXP: LISOR2, LISOR3, LISLISTOFEXP1, LISOR1, LISOR5, RCONS1, RCONS2, RCONS4, RCDDR, RCADR, RCDRCONS, LISCOND3, LISFUNCALL3, LISLAMBDA4, LISLAMBDA3.

COMBOOL: LISAND3, LISOR2, LISOR3, RCONS1, RCONS2, RCONS4, LISNOT3, LISOR1.

COMPANDOR: LISLISTOFEXP1, LISLISTOFEXP2, RCONS1, RCONS4, RCONS3R.

A.4.6 ISNOT Case

COMPEXP: LISNOT2, RCONS1, RCONS2, RCADR, RCONS3R, LISNOT1, LISCOND3, LISFUNCALL3, LISLAMBDA4, LISLAMBDA3.

COMBOOL: LISAND3, LISNOT2, LISOR3, LISNOT3, RCONS1, RCONS2, RCADR, RCONS3R, LISNOT1.

A.4.7-0 ISCOND No Arguments Subcase

COMPEXP: LISCOND2, LISCOND3, LISCOND1, LISFUNCALL3, LISLAMBDA4, LISLAMBDA3.

COMCOND: RCONS1, RCONS2, LISCOND4, LISCOND5.

A.4.7-N ISCOND With Arguments Subcase

COMPEXP: LISCOND2, LISCOND3, LISCOND4, LISCOND6, RCONS1, RCDDR, RCDRCONS, RCONS3R, RCAADR, RCAAR, RCAADR, RCADADR, RCADAR, LISCOND1, LISFUNCALL3, LISLAMBDA4, LISLAMBDA3.

COMCOND: LISCOND4, LISCOND5, RCONS1, RCONS3R, RCAAR, RCADAR.

A.4.8 ISQUOTE Case

COMPEXP: LISQUOTE2, LISCOND3, RCONS1, RCONS2, LISFUNCALL3, LISQUOTE1, LISLAMBDA4, LISLAMBDA3.

A.4.9 ISFUNCTIONCALL Case

COMPEXP: LISFUNCALL2, LISCOND3, LISFUNCALL3, RCONS1, RCONS2, RCONS3R, LISLAMBDA4, LISLAMBDA3, LISFUNCALL1.

A.4.10 ISLAMBDA Case

COMPEXP: LISLAMBDA2, LISCOND3, LISFUNCALL3, LISLAMBDA4, LISLAMBDA3, RCONS1, RCONS2, RCONS3R, RCADAR, LISLAMBDA1.

A.4.11 ISFUNCTIONDEF Case

COMP: LISFUNDEF3, RCONS1, RCONS2, RCADR, RCONS3R.

A.5 Sketch of Part One Proofs

For each source language syntactic type case or subcase (except function definition, which is compiled by COMP) the part one proof consists of proving the verification conditions for procedure COMPEXP. The Xivus system produces nine verification conditions because there are nine execution paths through COMPEXP. Eight of these paths will not be taken for a given syntactic type case. The proof of each corresponding verification condition is simply finding a contradiction in the hypothesis. For instance, we may have hypotheses of ISNIL(EXP) and NOT NULL(EXP). In light of the syntactic description axiom LISNIL1, this is a contradiction. Some of these verification conditions will, however, have a conclusion or two which discharge the necessity of satisfying Entry assertions of functions or procedures called along that path. These conclusions are proved by straightforward application of equality substitutions, rewrite rules, and syntactic description axioms.

The remaining one verification condition of each case or subcase consists of a conclusion representing the FCOMPEXP assertion, which may be assumed in order to define FCOMPEXP, and one or more conclusions which are usually proved by straightforward application of equality substitutions, rewrite rules, and axioms. The proofs of subsidiary functions or procedures called by COMPEXP turn out to be quite similar to those of COMPEXP in the ways mentioned above. The following is a sketch of how the one type of part one proof that was not straightforward was carried out.

In the AND, OR, and COND (more than zero arguments) subcases, the following technique was used. First the conclusions of the verification condition resulting from the longest of the three assertion clauses and from the shortest are proved by straightforward means. Then those conclusions are entered in slightly modified form into the theorem prover as an axiom to be used to prove the medium length conclusion. For example, in the AND subcase, the verification condition conclusions resulting from these assertions are proved first.

```
OUTFILE =
< ! OUTFILE'
  ! FCOMPEXP(EXP,M,LOCTABLE)
>

ISAND(EXP) →
OUTFILE =
< ! OUTFILE'
  ! FCOMPANDOR(CDR(EXP),M,L1,FALSE,LOCTABLE)
  < 'MOVEI 1 < 'QUOTE 'T > >
  < 'JRST 0 L2 >
  L1
  < 'MOVEI 1 0 >
  L2
>
```

Then the following newly proved theorem is entered into the theorem prover as an axiom.

```
ISAND(EXP.) →
FCOMPEXP(EXP., M., LOCTABLE.) =
< ! FCOMPANDOR(CDR(EXP.),M.,L1,FALSE,LOCTABLE.)
  < 'MOVEI 1 < 'QUOTE 'T > >
  < 'JRST 0 L2 >
  L1
  < 'MOVEI 1 0 >
  L2
>
```

The other of the three conclusions then proves easily by instantiating the arguments of FCOMPEXP in this axiom to the arguments of the recursive use of FCOMPEXP on the AND form with one less argument (it may be recalled that the syntax and semantics of the form <'AND a1 a2 ... an> are recursively defined in terms of the form <'AND a2 ... an>; hence the use of the AND with one less argument that is being proved in the medium length conclusion).

A.6 Compiler Listing

Following is a listing of the compiler proved. The assertions are in the clisp-like notation (before translation to the form acceptable to the proving system). The assertion from figure 4-3 appropriate for a particular source language syntactic case must be added to procedure COMPEXP in the place marked. The assertion for the function definition case, however, goes in the place marked in procedure COMP.

```
PROCEDURE COMP (EXP : LIST; VAR OUTFILE : FILE);
ENTRY ISFUNCTIONDEF(EXP);
EXIT [Use assertion for function definition case.];
VAR N : INTEGER;
```

```
BEGIN
N := LENGTH(CADDR(EXP));
OUTFILE := RIGHTCONS(OUTFILE,
                     CONS(QLAP,CONS(CADR(EXP),CONS(QSUBR,QNIL))));
MKPUSH(N,1,OUTFILE);
COMPEXP(CADDDR(EXP),-N,PRUP(CADDR(EXP),1),OUTFILE);
OUTFILE :=
 RIGHTCONS(OUTFILE,
           CONS(QSUB,
                CONS(QP,
                     CONS(CONS(QC,
                               CONS(0,CONS(0,CONS(N,CONS(N,QNIL))))),
                          QNIL))));
OUTFILE := RIGHTCONS(OUTFILE,CONS(QPOPJ,CONS(QP,QNIL)));
OUTFILE := RIGHTCONS(OUTFILE,QNIL);
END;

PROCEDURE MKPUSH (N , M : INTEGER; VAR OUTFILE : FILE);
ENTRY TRUE;
EXIT OUTFILE = < ! OUTFILE'
                 ! FMKPUSH(N,M)
               >
   & N < M → OUTFILE = OUTFILE'
   & N ≥ M → OUTFILE = < ! OUTFILE'
                         < 'PUSH 'P M >
                         ! FMKPUSH(N, M+1)
                       > ;
BEGIN
IF N GE M
THEN BEGIN
     OUTFILE := RIGHTCONS(OUTFILE,CONS(QPUSH,CONS(QP,CONS(M,QNIL))));
     MKPUSH(N,M+1,OUTFILE);
     END;
END;

FUNCTION PRUP (VARS : LIST; N : INTEGER) : LIST;
ENTRY TRUE;
EXIT NULL(VARS) → PRUP = 'NIL
   & NOT NULL(VARS) → PRUP = < <CAR(VARS) . N>
                               ! PRUP(CDR(VARS), N+1)
                             > ;
BEGIN
IF NULL(VARS)
THEN PRUP:=QNIL
ELSE PRUP:=CONS(CONS(CAR(VARS),N),PRUP(CDR(VARS),N+1));
END;
```

```
FUNCTION RETRIEVE(EXP : LIST; M : INTEGER; LOCTABLE : LIST;
                  OUTFILE : FILE) : INTEGER;
ENTRY TRUE;
EXIT  RETRIEVE(EXP, M, LOCTABLE, OUTFILE)
    = M + CDR(ASSOC(EXP, LOCTABLE)) ;
BEGIN
RETRIEVE:=M + CDR(ASSOC(EXP,LOCTABLE));
END;

PROCEDURE COMPEXP (EXP : LIST; M : INTEGER; LOCTABLE : LIST;
                   VAR OUTFILE : FILE);
ENTRY ISEXPRESSION(EXP);
EXIT OUTFILE = < ! OUTFILE'
                 ! FCOMPEXP(EXP,M,LOCTABLE)
               >
   & [Use assertion for each syntactic type case here] ;
VAR L1, L2, L5 : LIST;
BEGIN
IF NULL(EXP)
THEN OUTFILE := RIGHTCONS(OUTFILE,CONS(QMOVEI,CONS(1,CONS(0,QNIL))))
ELSE IF (EXP = QT) OR NUMBERP(EXP)
THEN OUTFILE :=
      RIGHTCONS(OUTFILE,
                CONS(QMOVEI,
                     CONS(1,
                          CONS(CONS(QQUOTE,CONS(EXP,QNIL)),
                               QNIL))))
ELSE IF ATOM(EXP)
THEN OUTFILE :=
      RIGHTCONS(OUTFILE,
                CONS(QMOVE,
                     CONS(1,
                          CONS(RETRIEVE(EXP,M,LOCTABLE,OUTFILE),
                               CONS(QP,QNIL)))))
```

```
ELSE IF (CAR(EXP) = QAND) OR (CAR(EXP) = QOR) OR (CAR(EXP) = QNOT)
THEN BEGIN
     GENSYM(L1);
     GENSYM(L2);
     COMBOOL(EXP,M,L1,LOCTABLE,OUTFILE);
     OUTFILE :=
      RIGHTCONS(OUTFILE,
                CONS(QMOVEI,
                     CONS(1,
                          CONS(CONS(QQUOTE,CONS(QT,QNIL)),
                               QNIL))));
     OUTFILE := RIGHTCONS(OUTFILE,CONS(QJRST,CONS(0,CONS(L2,QNIL))));
     OUTFILE := RIGHTCONS(OUTFILE,L1);
     OUTFILE := RIGHTCONS(OUTFILE,CONS(QMOVEI,CONS(1,CONS(0,QNIL))));
     OUTFILE := RIGHTCONS(OUTFILE,L2);
     END
ELSE IF CAR(EXP) = QCOND
THEN BEGIN
     GENSYM(L5);
     COMCOND(CDR(EXP),M,L5,LOCTABLE,OUTFILE);
     END
ELSE IF CAR(EXP) = QQUOTE
THEN OUTFILE := RIGHTCONS(OUTFILE,
                          CONS(QMOVEI,CONS(1,CONS(EXP,QNIL))))
ELSE IF ATOM(CAR(EXP))
THEN BEGIN
     COMPLIS(CDR(EXP),M,LOCTABLE,OUTFILE);
     LOADAC(1-LENGTH(CDR(EXP)),1,OUTFILE);
     OUTFILE := RIGHTCONS(OUTFILE,RPOP(LENGTH(CDR(EXP))));
     OUTFILE :=
      RIGHTCONS(OUTFILE,
                CONS(QCALL,
                     CONS(LENGTH(CDR(EXP)),
                          CONS(CONS(QE,CONS(CAR(EXP),QNIL)),
                               QNIL))));
     END
ELSE IF CAAR(EXP) = QLAMBDA
THEN BEGIN
     COMPLIS(CDR(EXP),M,LOCTABLE,OUTFILE);
     COMPEXP(CADDAR(EXP),M-LENGTH(CDR(EXP)),
             ADDIDS(LOCTABLE,CADAR(EXP),1-M),OUTFILE);
     OUTFILE := RIGHTCONS(OUTFILE,RPOP(LENGTH(CDR(EXP))));
     END;
END;
```

```
PROCEDURE COMPLIS (U : LIST; M : INTEGER; LOCTABLE : LIST;
                   VAR OUTFILE : FILE);
ENTRY ISLISTOFEXP(U);
EXIT OUTFILE = < ! OUTFILE'
                 ! FCOMPLIS(U,M,LOCTABLE)
               >
   & NULL(U) → OUTFILE = OUTFILE'
   & NOT NULL(U) → OUTFILE = < ! OUTFILE'
                               ! FCOMPEXP(CAR(U), M, LOCTABLE)
                               < 'PUSH 'P 1 >
                               ! FCOMPLIS(CDR(U), M-1, LOCTABLE)
                             > ;
BEGIN
IF NOT NULL(U)
THEN BEGIN
     COMPEXP(CAR(U),M,LOCTABLE,OUTFILE);
     OUTFILE := RIGHTCONS(OUTFILE,CONS(QPUSH,CONS(QP,CONS(1,QNIL))));
     COMPLIS(CDR(U),M-1,LOCTABLE,OUTFILE);
     END
END;

PROCEDURE LOADAC (N, K : INTEGER; VAR OUTFILE : FILE);
ENTRY TRUE;
EXIT OUTFILE = < ! OUTFILE'
                 ! FLOADAC(N,K)
               >
   & N > 0 → OUTFILE = OUTFILE'
   & N ≤ 0 → OUTFILE = < ! OUTFILE'
                         < 'MOVE K N 'P >
                         ! FLOADAC(N+1, K+1)
                       > ;
BEGIN
IF N LE 0
THEN BEGIN
     OUTFILE :=
      RIGHTCONS(OUTFILE,CONS(QMOVE,CONS(K,CONS(N,CONS(QP,QNIL)))));
     LOADAC(N+1,K+1,OUTFILE);
     END
END;
```

```
PROCEDURE COMCOND (U : LIST; M : INTEGER; L, LOCTABLE : LIST;
                   VAR OUTFILE : FILE);
ENTRY ISCONDLIST(U);
EXIT OUTFILE = < ! OUTFILE'
                 ! FCOMCOND(U,M,L,LOCTABLE)
               >
   & NULL(U) → OUTFILE = < ! OUTFILE'
                           L
                         >
   & NOT NULL(U) → OUTFILE = < ! OUTFILE'
                               ! FCOMPEXP(CAAR(U), M, LOCTABLE)
                               < 'JUMPE 1 L3 >
                               ! FCOMPEXP(CADAR(U), M, LOCTABLE)
                               < 'JRST L >
                               L3
                               ! FCOMCOND(CDR(U), M, L, LOCTABLE)
                             > ;
VAR L3:LIST;
BEGIN
IF NULL(U) THEN OUTFILE := RIGHTCONS(OUTFILE,L)
ELSE BEGIN
     GENSYM(L3);
     COMPEXP(CAAR(U),M,LOCTABLE,OUTFILE);
     OUTFILE :=
      RIGHTCONS(OUTFILE,CONS(QJUMPE,CONS(1,CONS(L3,QNIL))));
     COMPEXP(CADAR(U),M,LOCTABLE,OUTFILE);
     OUTFILE := RIGHTCONS(OUTFILE,CONS(QJRST,CONS(L,QNIL)));
     OUTFILE := RIGHTCONS(OUTFILE,L3);
     COMCOND(CDR(U),M,L,LOCTABLE,OUTFILE);
     END
END;

PROCEDURE COMBOOL (P : LIST; M : INTEGER; L : LIST;
                   LOCTABLE : LIST; VAR OUTFILE : FILE);
ENTRY ISEXPRESSION(P);
EXIT ISAND(P) ∧ NULL(CDR(P)) → OUTFILE = OUTFILE'
   &   ISAND(P) ∧ NOT NULL(CDR(P))
     → OUTFILE = < ! OUTFILE'
                   ! FCOMPEXP(CAR(CDR(P)), M, LOCTABLE)
                   < 'JUMPE 1 L >
                   ! FCOMPANDOR(CDR(CDR(P)), M, L, FALSE, LOCTABLE)
                 >
   &   ISAND(P)
     → OUTFILE = < ! OUTFILE'
                   ! FCOMPANDOR(CDR(P), M, L, FALSE, LOCTABLE)
                 >
```

```
   & ISOR(P) ∧ NULL(CDR(P)) → OUTFILE = < ! OUTFILE'
                                          < 'JRST 0 L >
                                          L4
                                        >
   &   ISOR(P) ∧ NOT NULL(CDR(P))
     → OUTFILE = < ! OUTFILE'
                   ! FCOMPEXP(CAR(CDR(P)), M, LOCTABLE)
                   < 'JUMPN 1 L4 >
                   ! FCOMPANDOR(CDR(CDR(P)), M, L4, TRUE, LOCTABLE)
                   < 'JRST 0 L >
                   L4
                 >
   &   ISOR(P)
     → OUTFILE = < ! OUTFILE'
                   ! FCOMPANDOR(CDR(P), M, L4, TRUE, LOCTABLE)
                   < 'JRST 0 L >
                   L4
                 >
   & ISNOT(P) → OUTFILE = < ! OUTFILE'
                            ! FCOMPEXP(CADR(P), M, LOCTABLE)
                            < 'JUMPN 1 L >
                          > ;
VAR L4:LIST;
BEGIN
IF CAR(P) = QAND
THEN COMPANDOR(CDR(P),M,L,FALSE,LOCTABLE,OUTFILE)
ELSE
IF CAR(P) = QOR
THEN BEGIN
     GENSYM(L4);
     COMPANDOR(CDR(P),M,L4,TRUE,LOCTABLE,OUTFILE);
     OUTFILE := RIGHTCONS(OUTFILE,CONS(QJRST,CONS(0,CONS(L,QNIL))));
     OUTFILE := RIGHTCONS(OUTFILE,L4);
     END
ELSE IF CAR(P)=QNOT
     THEN BEGIN
          COMPEXP(CADR(P),M,LOCTABLE,OUTFILE);
          OUTFILE :=
           RIGHTCONS(OUTFILE,CONS(QJUMPN,CONS(1,CONS(L,QNIL))));
          END
END;
```

```
PROCEDURE COMPANDOR (U : LIST; M : INTEGER; L : LIST; FLG : BOOLEAN;
                     LOCTABLE : LIST; VAR OUTFILE : FILE);
ENTRY ISLISTOFEXP(U);
EXIT OUTFILE = < ! OUTFILE'
                 ! FCOMPANDOR(U,M,L,FLG,LOCTABLE)
               >
   & NULL(U) → OUTFILE = OUTFILE'
   &   NOT NULL(U) ∧ FLG
     → OUTFILE = < ! OUTFILE'
                   ! FCOMPEXP(CAR(U), M, LOCTABLE)
                   < 'JUMPN 1 L >
                   ! FCOMPANDOR(CDR(U), M, L, FLG, LOCTABLE)
                 >
   &   NOT NULL(U) ∧ NOT FLG
     → OUTFILE = < ! OUTFILE'
                   ! FCOMPEXP(CAR(U), M, LOCTABLE)
                   < 'JUMPE 1 L >
                   ! FCOMPANDOR(CDR(U), M, L, FLG, LOCTABLE)
                 > ;
BEGIN
IF NOT NULL(U)
THEN IF FLG
     THEN BEGIN
          COMPEXP(CAR(U),M,LOCTABLE,OUTFILE);
          OUTFILE :=
           RIGHTCONS(OUTFILE,CONS(QJUMPN,CONS(1,CONS(L,QNIL))));
          COMPANDOR(CDR(U),M,L,FLG,LOCTABLE,OUTFILE);
          END
     ELSE BEGIN
          COMPEXP(CAR(U),M,LOCTABLE,OUTFILE);
          OUTFILE :=
           RIGHTCONS(OUTFILE,CONS(QJUMPE,CONS(1,CONS(L,QNIL))));
          COMPANDOR(CDR(U),M,L,FLG,LOCTABLE,OUTFILE);
          END
END;

FUNCTION RPOP(I : INTEGER) : LIST;
ENTRY TRUE;
EXIT RPOP(I) = < 'SUB 'P < 'C 0 0 I I > > ;
BEGIN
RPOP := CONS(QSUB,
            CONS(QP,
                 CONS(CONS(QC,CONS(0,CONS(0,CONS(I,CONS(I,QNIL))))),
                      QNIL)));
END;
```

```
FUNCTION ADDIDS(LOCTABLE, IDLIST : LIST; LASTLOC : INTEGER) : LIST;
ENTRY TRUE;
EXIT  ADDIDS(LOCTABLE, IDLIST, LASTLOC)
    = < ! PRUP(IDLIST, LASTLOC)
        ! LOCTABLE > ;
BEGIN
ADDIDS:=APPEND(PRUP(IDLIST,LASTLOC),LOCTABLE);
END;

BEGIN END.
```

A.7 Intrinsic Lisp-like Routines

Following is an alphabetic list of the Lisp-like functions (and one procedure) which were treated as intrinsic to the source language in the sense that their code was not supplied. Their properties were given to the theorem prover as rewrite rules. The types of the functions and their arguments are given in Pascal format.

```
FUNCTION APPEND (X1, X2 : LIST) : LIST;
FUNCTION ASSOC (X, Y : LIST) : LIST;
FUNCTION ATOM (X : LIST) : BOOLEAN;
FUNCTION CAAR (X : LIST) : LIST;
FUNCTION CADAR (X : LIST) : LIST;
FUNCTION CADDAR (X : LIST) : LIST;
FUNCTION CADDDR (X : LIST) : LIST;
FUNCTION CADDR (X : LIST) : LIST;
FUNCTION CADR (X : LIST) : LIST;
FUNCTION CAR (X : LIST) : LIST;
FUNCTION CDR (X : LIST) : LIST;
FUNCTION CONS (X, Y : LIST) : LIST;
PROCEDURE GENSYM (VAR X : LIST) ;
FUNCTION LENGTH (X : LIST) : INTEGER;
FUNCTION NULL (X : LIST) : BOOLEAN;
FUNCTION NUMBERP (X : LIST) : BOOLEAN;
FUNCTION RIGHTCONS (OUTFILE : FILE; STUFF : LIST) : FILE;
```

A.8 Part Two Proofs

The following subsections contain the part two proofs for compiler MC0.

A.8.1 ISNIL Case

We here prove the statement of correctness of the compiler for the case where ISNIL(S):

$$\text{ISEXPRESSION}(S) \rightarrow$$

$$\left(\text{Pre}(S)\left|\begin{matrix}v\\w\end{matrix}\right|\left|\begin{matrix}\text{NIL}\\0\end{matrix}\right. \wedge \left(\text{Post}(S)\left|\begin{matrix}v\\w\end{matrix}\right|\left|\begin{matrix}\text{NIL}\\0\end{matrix}\right. \rightarrow VR2,\ldots,RN(S)\ \left(T\left|\begin{matrix}R1\\S\end{matrix}\right|\left|\begin{matrix}v\\w\end{matrix}\right|\left|\begin{matrix}\text{NIL}\\0\end{matrix}\right.\right)\right)$$

$$\{\text{FCOMPEXP}(S,M,\text{LOCTABLE})\}\ T\ )$$

$$\wedge\ \text{stackok}(\text{FCOMPEXP}(S,M,\text{LOCTABLE}))$$

where LOCTABLE is of form $< < \text{NAME1} . \text{LOC1} > \ldots < \text{NAMEr} . \text{LOCr} > >$, $v = < \text{NAME1} \ldots \text{NAMEr} >$, $w = < m[M+P+\text{LOC1}] \ldots m[M+P+\text{LOCr}] >$, and N(S) is a function giving the maximum number of registers that are modified during execution of the compilation of S.

To prove the Hoare rule portion we will apply Hoare rules to T for the statements of FCOMPEXP for the particular case in question to form a verification condition. We will apply simplifications to the verification condition during and after generation to reduce it to TRUE.

The code produced by the compiler for this case is:

$$\text{FCOMPEXP}(S,M,\text{LOCTABLE}) = < < \text{'MOVEI 1 0} > >$$

Using the MOVEI Hoare rule:

$$Q\left|\begin{matrix}Ri\\y\end{matrix}\right.\ \{ < \text{'MOVEI i y} > \}\ Q$$

gives us

$$T\left|\begin{matrix}R1\\0\end{matrix}\right.$$

The verification condition is then:

$$\text{Pre}(S)\left|\begin{matrix}v\\w\end{matrix}\right|\left|\begin{matrix}\text{NIL}\\0\end{matrix}\right. \wedge \left(\text{Post}(S)\left|\begin{matrix}v\\w\end{matrix}\right|\left|\begin{matrix}\text{NIL}\\0\end{matrix}\right. \rightarrow VR2,\ldots,RN(S)\ \left(T\left|\begin{matrix}R1\\S\end{matrix}\right|\left|\begin{matrix}v\\w\end{matrix}\right|\left|\begin{matrix}\text{NIL}\\0\end{matrix}\right.\right)\right)$$

$$\rightarrow T\left|\begin{matrix}R1\\0\end{matrix}\right.$$

We now expand Pre(S), Post(S), N(S), and S by the formulas which apply for the case of NIL. This results in:

$$\text{TRUE}\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right.\ \wedge\ (\text{TRUE}\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right.\ \rightarrow T\ \left|\begin{matrix}R1\\\text{NIL}\end{matrix}\right.\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right.\ )\ \rightarrow T\ \left|\begin{matrix}R1\\0\end{matrix}\right.$$

Since TRUE and NIL represent constants distinct from the source language variables in v, we may drop all v substitutions by subrule 4. Similarly we may drop the NIL substitutions on TRUE. Then carry out the NIL substitution on NIL, and simplify logically. The result is:

$$T\ \left|\begin{matrix}R1\\0\end{matrix}\right.\ \rightarrow T\ \left|\begin{matrix}R1\\0\end{matrix}\right.$$

which is obviously TRUE.

To prove the stackok term of the statement of correctness we simply apply S6. This completes the proof of the compiler for the ISNIL case.

A.8.2A IST Case

We here prove the statement of correctness of the compiler for the case where IST(S).

To prove the Hoare rule portion we will again apply Hoare rules to the assertion T for the statements of FCOMPEXP for the particular case in question to form a verification condition. We will apply simplifications to the verification condition during and after generation to reduce it to TRUE.

The code produced by the compiler for this case is:

```
FCOMPEXP(S,M,LOCTABLE) = < < 'MOVEI 1 <'QUOTE 'T> > >
```

Using the MOVEI Hoare rule:

$$Q\ \left|\begin{matrix}Ri\\y\end{matrix}\right.\ \{\ <\ \text{'MOVEI i y}\ >\ \}\ Q$$

gives us

$$T\ \left|\begin{matrix}R1\\<\text{'QUOTE 'T}>\end{matrix}\right.$$

The verification condition is then:

$$\text{Pre}(S)\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right.\ \wedge\ (\text{Post}(S)\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right.\ \rightarrow \text{VR2},\ldots,\text{RN}(S)\ (T\ \left|\begin{matrix}R1\\S\end{matrix}\right.\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right.\ ))$$

$$\rightarrow T\ \left|\begin{matrix}R1\\<\text{'QUOTE 'T}>\end{matrix}\right.$$

We now expand Pre(S), Post(S), N(S), and S by the formulas which apply for the case of T. We may use <'QUOTE 'T> for S in place of the constant T, since the two forms mean the same in both the source and target language. We choose the longer form here to avoid confusion with the variable T used in the statement of correctness.

This results in:

$$\text{TRUE} \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right. \wedge (\text{TRUE} \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right. \rightarrow T \left|\begin{matrix}\text{RI}\\ <\text{'QUOTE 'T}>\end{matrix}\right. \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right.) \rightarrow$$

$$T \left|\begin{matrix}\text{RI}\\ <\text{'QUOTE 'T}>\end{matrix}\right.$$

In a manner similar to the proof of the NIL case, we simplify the verification condition to:

$$T \left|\begin{matrix}\text{RI}\\ <\text{'QUOTE 'T}>\end{matrix}\right. \rightarrow T \left|\begin{matrix}\text{RI}\\ <\text{'QUOTE 'T}>\end{matrix}\right.$$

which is obviously TRUE.

To prove the stackok term of the statement of correctness we simply apply S6. This completes the proof of the compiler for the IST case.

A.8.2B ISNUMBER Case

We here prove the statement of correctness of the compiler for the case where ISNUMBER(S).

We will apply Hoare rules to the assertion T for the statements of FCOMPEXP for the particular case in question to form a verification condition. We will apply simplifications to the verification condition during and after generation to reduce it to TRUE.

The code produced by the compiler for this case is:

FCOMPEXP(S,M,LOCTABLE) = < < 'MOVEI 1 <'QUOTE S> > >

Using the MOVEI Hoare rule:

$$Q \left|\begin{matrix}\text{Ri}\\ y\ \{ <\text{'MOVEI i y}> \}\ Q\end{matrix}\right.$$

gives us

$$T \left|\begin{matrix}\text{RI}\\ <\text{'QUOTE S}>\end{matrix}\right.$$

The verification condition is then:

$$\text{Pre(S)}\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right. \wedge\ (\text{Post(S)}\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right. \rightarrow \text{VR2,...,RN(S)}\ (T\ \left|\begin{matrix}R1\\S\end{matrix}\right|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right))$$

$$\rightarrow T\ \left|\begin{matrix}R1\\\text{<'QUOTE S>}\end{matrix}\right.$$

We now expand Pre(S), Post(S), and N(S) by the formulas which apply for the ISNUMBER case. This results in:

$$\text{TRUE}\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right. \wedge\ (\text{TRUE}\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right. \rightarrow T\ \left|\begin{matrix}R1\\S\end{matrix}\right|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right) \rightarrow$$

$$T\ \left|\begin{matrix}R1\\\text{<'QUOTE S>}\end{matrix}\right.$$

In a manner similar to the proof of the NIL case, we simplify the verification condition to:

$$T\ \left|\begin{matrix}R1\\S\end{matrix}\right. \rightarrow T\ \left|\begin{matrix}R1\\\text{<'QUOTE S>}\end{matrix}\right.$$

For the case S being a constant, S and <'QUOTE S> mean the same thing, and thus the verification condition is TRUE.

To prove the stackok term of the statement of correctness we simply apply S6. This completes the proof of the compiler for the ISNUMBER case.

A.8.3 ISIDENTIFIER Case

We here prove the statement of correctness of the compiler for the case where ISIDENTIFIER(S).

We will apply Hoare rules to the assertion T for the statements of FCOMPEXP for the particular case in question to form a verification condition. We will apply simplifications to the verification condition during and after generation to reduce it to TRUE.

The code produced by the compiler for this case is:

```
FCOMPEXP(S,M,LOCTABLE) =
 < < 'MOVE 1 RETRIEVE(S,M,LOCTABLE,OUTFILE') 'P > >
```

We can see that whenever variables are declared (function definition or lambda cases), the names of the new variables are added to LOCTABLE. In this case S is a variable, and thus will be in LOCTABLE. Further, the most recent declaration of a variable name will be the leftmost occurrence in LOCTABLE, since names are always added on the left. Thus LOCTABLE will be of form < <NAME1 . LOC1> ... <NAMEi . LOCi> ... <NAMEr . LOCr> > where NAMEi is the first NAMEj such that S = NAMEj. Now RETRIEVE(S,M,LOCTABLE,OUTFILE') is (by

examination of its code) M+CDR(ASSOC(S,LOCTABLE)). This is (by executing the Lisp functions CDR and ASSOC) M+LOCi.

Using the MOVE Hoare rule:

$$Q \begin{vmatrix} Ri \\ m[P+j] \end{vmatrix} \{ < \text{'MOVE i j 'P} > \} \; Q$$

gives us

$$T \begin{vmatrix} R1 \\ m[P+M+LOCi] \end{vmatrix}$$

The complete verification condition is then:

$$\text{Pre(S)} \left|\left| \begin{matrix} v \\ w \end{matrix} \right| \begin{matrix} NIL \\ 0 \end{matrix} \wedge (\text{Post(S)} \left|\left| \begin{matrix} v \\ w \end{matrix} \right| \begin{matrix} NIL \\ 0 \end{matrix} \rightarrow VR2,\ldots,RN(S) \; (T \begin{vmatrix} R1 \\ S \end{vmatrix} \left|\left| \begin{matrix} v \\ w \end{matrix} \right| \begin{matrix} NIL \\ 0 \end{matrix}))$$

$$\rightarrow T \begin{vmatrix} R1 \\ m[P+M+LOCi] \end{vmatrix}$$

We now expand Pre(S), Post(S), and N(S) by the formulas which apply for the ISIDENTIFIER case. This results in:

$$\text{TRUE} \left|\left| \begin{matrix} v \\ w \end{matrix} \right| \begin{matrix} NIL \\ 0 \end{matrix} \wedge (\text{TRUE} \left|\left| \begin{matrix} v \\ w \end{matrix} \right| \begin{matrix} NIL \\ 0 \end{matrix} \rightarrow T \begin{vmatrix} R1 \\ S \end{vmatrix} \left|\left| \begin{matrix} v \\ w \end{matrix} \right| \begin{matrix} NIL \\ 0 \end{matrix}) \rightarrow T \begin{vmatrix} R1 \\ m[P+M+LOCi] \end{vmatrix}$$

In a manner similar to the proof of the NIL case, we simplify the verification condition to:

$$T \begin{vmatrix} R1 \\ S \end{vmatrix} \left|\left| \begin{matrix} v \\ w \end{matrix} \right| \begin{matrix} NIL \\ 0 \end{matrix} \rightarrow T \begin{vmatrix} R1 \\ m[P+M+LOCi] \end{vmatrix}$$

Now $v = < NAME1 \ldots NAMEi \ldots NAMEr >$ and $w = < m[M+P+LOC1] \ldots m[M+P+LOCi] \ldots m[M+P+LOCr] >$ where NAMEi is the first such NAMEj such that S = NAMEj. Thus

$$S \left|\left| \begin{matrix} v \\ w = m[M+P+LOCi] \end{matrix} \right.$$

We see $NIL \neg\epsilon\ m[M+P+LOCi]$, and thus by subrule 4, we get:

$$T \begin{vmatrix} R1 \\ m[M+P+LOCi] \end{vmatrix} \rightarrow T \begin{vmatrix} R1 \\ m[P+M+LOCi] \end{vmatrix}$$

which is obviously TRUE.

To prove the stackok term of the statement of correctness we simply apply S5. This completes the proof of the compiler for the ISIDENTIFIER case.

A.8.4 ISAND Case

We here prove the statement of correctness of the compiler for the case where ISAND(S). We will prove it in two subcases: that of no arguments, and that of one or more.

The code produced by the compiler for the first subcase is:

```
FCOMPEXP(S,M,LOCTABLE) = < < 'MOVEI 1 <'QUOTE 'T> >
                           < 'JRST 0 L2 >
                           L1
                           < 'MOVEI 1 0 >
                           L2
                         >
```

The assertion T is to have Hoare rules applied for these statements. First we note that assertion(L2) = T. Then we apply the MOVEI Hoare rule:

$$Q \begin{vmatrix} Ri \\ y \end{vmatrix} \{ < \text{'MOVEI i y} > \} Q$$

to obtain

$$T \begin{vmatrix} R1 \\ 0 \end{vmatrix}$$

Then note that

$$\text{assertion(L1)} = T \begin{vmatrix} R1 \\ 0 \end{vmatrix}$$

Next we apply the JRST Hoare rule:

$$\text{assertion(1)} \{ < \text{'JRST 0 1} > \} Q$$

to obtain T. Now apply the MOVEI Hoare rule, resulting in

$$T \begin{vmatrix} R1 \\ <\text{'QUOTE 'T}> \end{vmatrix}$$

Completing the verification condition generation we get:

$$\text{Pre(S)} \begin{vmatrix} v & \text{NIL} \\ w & 0 \end{vmatrix} \wedge (\text{Post(S)} \begin{vmatrix} v & \text{NIL} \\ w & 0 \end{vmatrix} \rightarrow \text{VR2},\ldots,\text{RN(S)}\ (T \begin{vmatrix} \text{R1} \\ S \end{vmatrix} \begin{vmatrix} v & \text{NIL} \\ w & 0 \end{vmatrix}))$$

$$\rightarrow T \begin{vmatrix} \text{R1} \\ \text{<'QUOTE 'T>} \end{vmatrix}$$

We now expand Pre(S), Post(S), and N(S) by the formulas which apply for this subcase, and we let S = <'QUOTE 'T>. Note that the value for the AND with no arguments is the Lisp equivalent of TRUE. This results in:

$$\text{TRUE} \begin{vmatrix} v & \text{NIL} \\ w & 0 \end{vmatrix}$$

$$\wedge\ ((\text{<'QUOTE 'T>=<'QUOTE 'T>}) \begin{vmatrix} v & \text{NIL} \\ w & 0 \end{vmatrix} \rightarrow T \begin{vmatrix} \text{R1} \\ \text{<'QUOTE 'T>} \end{vmatrix} \begin{vmatrix} v & \text{NIL} \\ w & 0 \end{vmatrix})$$

$$\rightarrow\ T \begin{vmatrix} \text{R1} \\ \text{<'QUOTE 'T>} \end{vmatrix}$$

In a manner similar to the proof of the NIL case, we simplify the verification condition to:

$$T \begin{vmatrix} \text{R1} \\ \text{<'QUOTE 'T>} \end{vmatrix} \rightarrow T \begin{vmatrix} \text{R1} \\ \text{<'QUOTE 'T>} \end{vmatrix}$$

which is obviously TRUE.

The code produced by the compiler for the second subcase is:

```
FCOMPEXP(S,M,LOCTABLE) = < ! FCOMPEXP(b1,M,LOCTABLE)
                           < 'JUMPE 1 L1 >
                           ! FCOMPEXP(<'AND b2 ... bn>,M,LOCTABLE)
                         >
```

where S is of form < 'AND b1 b2 ... bn >.

Assuming the FCOMPEXP properties on < 'AND b2 ... bn >, a smaller portion of source code (the inductive assumption), we get:

$$\text{Pre(s)} \begin{vmatrix} v & \text{NIL} \\ w & 0 \end{vmatrix} \wedge (\text{Post(s)} \begin{vmatrix} v & \text{NIL} \\ w & 0 \end{vmatrix} \rightarrow \text{VR2},\ldots,\text{RN(s)}\ (T \begin{vmatrix} \text{R1} \\ s \end{vmatrix} \begin{vmatrix} v & \text{NIL} \\ w & 0 \end{vmatrix}))$$

where s = < 'AND b2 ... bn >. Next we wish to apply the JUMPE Hoare rule:

$$(\text{Ri}=0 \rightarrow \text{assertion(1)}) \wedge (\text{Ri}\neq 0 \rightarrow Q)\ \{ < \text{'JUMPE 1 1} > \}\ Q$$

but we don't have assertion(L1) because L1 lies inside FCOMPEXP of s. From the assertions proved in the part one proof we have:

```
FCOMPEXP(S,M,LOCTABLE) = < ...
                           L1
                           < 'MOVEI 1 0 >
                           L2
                         >
```

Thus assertion(L1) is always the same as in the subcase of n=0. The verification condition thus becomes:

$$(R1{=}0 \rightarrow T \begin{vmatrix} R1 \\ 0 \end{vmatrix}) \wedge (R1{\neq}0 \rightarrow$$

$$Pre(s) \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix} \wedge (Post(s) \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix} \rightarrow VR2,\ldots,RN(s) (T \begin{vmatrix} R1 \\ s \end{vmatrix} \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix}))$$

We again assume the FCOMPEXP properties on a smaller portion of source code, this time on b1. The result is:

$$Pre(b1) \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix} \wedge (Post(b1) \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix} \rightarrow VR2,\ldots,RN(b1)$$

$$(((R1{=}0 \rightarrow T \begin{vmatrix} R1 \\ 0 \end{vmatrix}) \wedge (R1{\neq}0 \rightarrow$$

$$Pre(s) \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix} \wedge (Post(s) \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix} \rightarrow VR2,\ldots,RN(s) (T \begin{vmatrix} R1 \\ s \end{vmatrix} \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix}))))$$

$$\begin{vmatrix} R1 \\ b1 \end{vmatrix} \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix})$$

Since this formula is getting rather large, we introduce a new notation. The substitutions

$$\begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix}$$

will be abbreviated by *. The form

$$\begin{vmatrix} x \\ y & * \end{vmatrix}$$

will be understood to mean

$$\begin{vmatrix} x \\ & \| \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix} \\ y \end{vmatrix}$$

The formula using this notation is then:

$$Pre(b1)\ *\ \wedge\ (Post(b1)\ *\ \rightarrow\ \forall R2,\ldots,RN(b1)\ (((R1{=}0\ \rightarrow\ T\ \begin{vmatrix} R1 \\ 0 \end{vmatrix})\ \wedge$$

$$(R1{\neq}0\ \rightarrow\ Pre(s)\ *\ \wedge\ (Post(s)\ *\ \rightarrow\ \forall R2,\ldots,RN(s)\ (T\ \begin{vmatrix} R1 \\ s\ * \end{vmatrix}))))\ \begin{vmatrix} R1 \\ b1\ * \end{vmatrix}))$$

Distribute the last R1 substitution as far as possible. Then apply subrule 12 and subrule 4 to drop the b1 substitution on the first T term. Do the b1 substitution on R1=0 and R1≠0. R1 $\neg\epsilon$ Pre(s), Post(s), b1, s, w, and 0, so by subrule 3 we get R1 $\neg\epsilon$ Pre(s) *, Post(s) *, b1 *, and s *. So drop the R1 substitution on Pre(s) * and Post(s) * by subrule 4. Apply subrule 18b to move the outer R1 substitution inside ∀R2,...,RN(s). Then drop the outer R1 substitution on the last T term by subrule 12 and subrule 4. The result is:

$$Pre(b1)\ *\ \wedge\ (Post(b1)\ *\ \rightarrow\ \forall R2,\ldots,RN(b1)\ ((b1\ *\ {=}0\ \rightarrow\ T\ \begin{vmatrix} R1 \\ 0 \end{vmatrix})\ \wedge$$

$$(b1\ *\ {\neq}0\ \rightarrow\ Pre(s)\ *\ \wedge\ (Post(s)\ *\ \rightarrow\ \forall R2,\ldots,RN(s)\ (\ T\ \begin{vmatrix} R1 \\ s\ * \end{vmatrix})))))$$

The same argument as used on R1 above established for all i Ri $\neg\epsilon$ b1 *, Pre(s) *, and Post(s) *. Thus we can apply subrule 16 repeatedly, then regroup the AND-IMPLY structure, and finally apply subrule 17b to get:

$$Pre(b1)\ *\ \wedge\ (Post(b1)\ *\ \wedge\ b1\ *\ {=}0\ \rightarrow\ \forall R2,\ldots,RN(b1)\ (\ T\ \begin{vmatrix} R1 \\ 0 \end{vmatrix}))$$

$$\wedge\ (Post(b1)\ *\ \wedge\ b1\ *\ {\neq}0\ \rightarrow\ Pre(s)\ *\ \wedge\ (Post(s)\ *\ \rightarrow$$

$$\forall R2,\ldots,Rmax(N(b1),N(s))\ (\ T\ \begin{vmatrix} R1 \\ s\ * \end{vmatrix})))$$

Completing the verification condition generation we get:

$$\mathrm{Pre}(S)\ * \wedge (\mathrm{Post}(S)\ * \rightarrow \forall R2,\ldots,RN(S)\ (\ T\ \Big|_{S\ *}^{R1}\ )) \rightarrow$$

$$\mathrm{Pre}(b1)\ * \wedge (\mathrm{Post}(b1)\ * \wedge b1\ * = 0 \rightarrow \forall R2,\ldots,RN(b1)\ (\ T\ \Big|_{0}^{R1}\ ))$$

$$\wedge\ (\mathrm{Post}(b1)\ * \wedge b1\ * \neq 0 \rightarrow \mathrm{Pre}(s)\ * \wedge (\mathrm{Post}(s)\ * \rightarrow$$

$$\forall R2,\ldots,\mathrm{Rmax}(N(b1),N(s))\ (\ T\ \Big|_{s\ *}^{R1}\ )))$$

We now expand Pre(S) and Post(S) by the formulas which apply for this subcase. Distributing the substitutions we get:

$$\mathrm{Pre}(b1)\ * \wedge (b1\ * \neq 0 \rightarrow \mathrm{Pre}(s)\ *) \wedge$$

$$((b1\ * = 0 \rightarrow S\ * = 0) \wedge (b1\ * \neq 0 \rightarrow S\ * = s\ *) \wedge \mathrm{Post}(b1)\ * \wedge$$

$$(b1\ * \neq 0 \rightarrow \mathrm{Post}(s)\ *\ ) \rightarrow \forall R2,\ldots,RN(S)\ (\ T\ \Big|_{S\ *}^{R1}\ ))$$

$$\rightarrow$$

$$\mathrm{Pre}(b1)\ * \wedge (\mathrm{Post}(b1)\ * \wedge b1\ * = 0 \rightarrow \forall R2,\ldots,RN(b1)\ (\ T\ \Big|_{0}^{R1}\ ))$$

$$\wedge\ (\mathrm{Post}(b1)\ * \wedge b1\ * \neq 0 \rightarrow \mathrm{Pre}(s)\ * \wedge (\mathrm{Post}(s)\ * \rightarrow$$

$$\forall R2,\ldots,\mathrm{Rmax}(N(b1),N(s))\ (\ T\ \Big|_{s\ *}^{R1}\ )))$$

Because Pre(b1)* appears in the hypotheses, we may set it to TRUE and simplify logically. The Pre(s)* term is implied by a hypothesis and may be similarly treated. The result is:

$$\mathrm{Pre}(b1)\ * \wedge (b1\ * \neq 0 \rightarrow \mathrm{Pre}(s)\ *) \wedge$$

$$((b1\ * = 0 \rightarrow S\ * = 0) \wedge (b1\ * \neq 0 \rightarrow S\ * = s\ *) \wedge \mathrm{Post}(b1)\ * \wedge$$

$$(b1\ * \neq 0 \rightarrow \mathrm{Post}(s)\ *\ ) \rightarrow \forall R2,\ldots,RN(S)\ (\ T\ \Big|_{S\ *}^{R1}\ ))$$

$$\rightarrow$$

$$(Post(b1)\ *\ \wedge\ b1\ *\ =\ 0\ \rightarrow\ \forall R2,\ldots,RN(b1)\ (\ T\ \begin{vmatrix} R1 \\ 0\ ))\ \wedge$$

$$(Post(b1)\ *\ \wedge\ b1\ *\ \neq\ 0\ \wedge\ Post(s)\ *\ \rightarrow\ \forall R2,\ldots,Rmax(N(b1),N(s))$$

$$(\ T\ \begin{vmatrix} R1 \\ s\ *\ ))$$

We will break this into two cases: that of b1 = NIL, and that of b1 ≠ NIL.

Case 1: b1 = NIL. Then in the target language, b1 * = 0. We expand N(S) by the formula which holds for this subcase. Substituting TRUE or FALSE according to the case 1 information, then logically simplifying, the verification condition becomes:

$$Pre(b1)\ *\ \wedge\ (S\ *\ =\ 0\ \wedge\ Post(b1)\ *\ \rightarrow\ \forall R2,\ldots,RN(b1)\ (\ T\ \begin{vmatrix} R1 \\ S\ *\ ))\ \rightarrow$$

$$(Post(b1)\ *\ \rightarrow\ \forall R2,\ldots,RN(b1)\ (\ T\ \begin{vmatrix} R1 \\ 0\ ))$$

which is obviously TRUE.

Case 2: b1 ≠ NIL. Then b1 * ≠ 0. In a manner similar to case 1 we get:

$$Pre(b1)\ *\ \wedge\ Pre(S)\ *\ \wedge\ (S\ *\ =\ s\ *\ \wedge\ Post(b1)\ *\ \wedge\ Post(s)\ *\ \rightarrow$$

$$\forall R2,\ldots,Rmax(N(b1),N(s))\ (\ T\ \begin{vmatrix} R1 \\ S\ *\ ))$$

$$\rightarrow$$

$$(Post(b1)\ *\ \wedge\ Post(s)\ *\ \rightarrow\ \forall R2,\ldots,Rmax(N(b1),N(s))\ (\ T\ \begin{vmatrix} R1 \\ s\ *\ ))$$

which is obviously TRUE.

We now must prove the stackok term for this case. This we will also do by the two subcases. For the no argument subcase we apply S3 to obtain the subgoals stackok(< < 'MOVEI 1 <'QUOTE 'T> > >) and stackok(< < 'JRST 0 L2 > L1 < 'MOVEI 1 0 > L2 >). The first is disposed of by applying S6. For the second subgoal we use S8.

For the n argument subcase, we will first apply S3 to obtain the subgoals stackok(FCOMPEXP(b1,M,LOCTABLE)) and stackok(< < 'JUMPE 1 L1 > ! FCOMPEXP(s,M,LOCTABLE) >). The first is part of the *inductive assumption that these properties are true of smaller parts.* The second subgoal requires S9. This rule may be paraphrased as: if the stack is preserved both where you may jump to and where you fall through, then it is preserved by the conditional jump structure as a whole.

The two subgoals thus created are stackok(< < 'MOVEI 1 0 > L2 >) and

stackok(FCOMPEXP(s,M,LOCTABLE)). The first was derived by recalling that we previously proved that FCOMPEXP(s,M,LOCTABLE) always ended in L1, < 'MOVEI 1 0 >, and L2. We may prove this subgoal by applying S3 to get the subgoals stackok(< < 'MOVEI 1 0 > >) and stackok(< L2 >). The first of these is proved by S6 and the second by S7.

The remaining subgoal of stackok(FCOMPEXP(s,M,LOCTABLE)) is proved by appealing to the inductive assumption.

This completes the proof of the compiler for the case of AND.

A.8.5 ISOR Case

We here prove the statement of correctness of the compiler for the case where ISOR(S). The target code and proof of this case closely parallel the ISAND case. As with AND, we will prove the case in two subcases: that of no arguments, and that of one or more.

The code produced by the compiler for the first subcase is:

```
FCOMPEXP(S,M,LOCTABLE) = < < 'JRST 0 L1 >
                           L4
                           < 'MOVEI 1 <'QUOTE 'T> >
                           < 'JRST 0 L2 >
                           L1
                           < 'MOVEI 1 0 >
                           L2
                         >
```

The assertion T is to have Hoare rules applied for these statements. First we note that assertion(L2) = T. Then we apply the MOVEI Hoare rule:

$$Q \begin{vmatrix} Ri \\ y \end{vmatrix} \{ < \text{'MOVEI i y} > \} \; Q$$

to obtain

$$T \begin{vmatrix} R1 \\ 0 \end{vmatrix}$$

Then note that

$$\text{assertion(L1)} = T \begin{vmatrix} R1 \\ 0 \end{vmatrix}$$

Next we apply the JRST Hoare rule:

$$\text{assertion(1)} \; \{ < \text{'JRST 0 1} > \} \; Q$$

to obtain T. Now apply the MOVEI Hoare rule, resulting in

$$T \begin{vmatrix} R1 \\ <'QUOTE\ 'T> \end{vmatrix}$$

Now we see that

$$assertion(L4) = T \begin{vmatrix} R1 \\ <'QUOTE\ 'T> \end{vmatrix}$$

Then applying the JRST Hoare rule we get assertion(L1), which is

$$T \begin{vmatrix} R1 \\ 0 \end{vmatrix}$$

Completing the verification condition generation we get:

$$Pre(S) \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix} \wedge (Post(S) \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix} \rightarrow VR2,\ldots,RN(S)\ (T \begin{vmatrix} R1 \\ S \end{vmatrix} \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix}))$$

$$\rightarrow T \begin{vmatrix} R1 \\ 0 \end{vmatrix}$$

We now expand Pre(S), Post(S), and N(S) by the formulas which apply for this subcase, and we let S = NIL. Note that the value for the OR with no arguments is the Lisp equivalent of FALSE. This results in:

$$TRUE \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix} \wedge ((NIL=NIL) \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix} \rightarrow T \begin{vmatrix} R1 \\ NIL \end{vmatrix} \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix}) \rightarrow T \begin{vmatrix} R1 \\ 0 \end{vmatrix}$$

In a manner similar to the proof of the NIL case, we simplify the verification condition to:

$$T \begin{vmatrix} R1 \\ 0 \end{vmatrix} \rightarrow T \begin{vmatrix} R1 \\ 0 \end{vmatrix}$$

which is obviously TRUE.

The code produced by the compiler for the second subcase is:

```
FCOMPEXP(S,M,LOCTABLE) = < ! FCOMPEXP(b1,M,LOCTABLE)
                           < 'JUMPN 1 L4 >
                           ! FCOMPEXP(<'OR b2 ... bn>,M,LOCTABLE)
                         >
```

where S is of form < 'OR b1 b2 ... bn >.

Assuming the FCOMPEXP properties on < 'OR b2 ... bn >, a smaller portion of source code (the inductive assumption), we get:

$$Pre(s)\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ \wedge\ (Post(s)\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ \rightarrow VR2,\ldots,RN(s)\ (T\ \left|\begin{matrix}R1\\s\end{matrix}\right|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ ))$$

where s = < 'OR b2 ... bn >. Next we wish to apply the JUMPN Hoare rule:

$$(Ri\neq 0 \rightarrow assertion(1))\ \wedge\ (Ri=0 \rightarrow Q)\ \{\ <\ 'JUMPN\ i\ 1\ >\ \}\ Q$$

but we don't have assertion(L4) because L4 lies inside FCOMPEXP of s. From the assertions proved in the part one proof we have:

```
FCOMPEXP(S,M,LOCTABLE) = < ...
                           L4
                           < 'MOVEI 1 <'QUOTE 'T> >
                           < 'JRST 0 L2 >
                           L1
                           < 'MOVEI 1 0 >
                           L2
                         >
```

Thus assertion(L4) is always the same as in the subcase of n=0. The verification condition thus becomes:

$$(R1\neq 0 \rightarrow T\ \left|\begin{matrix}R1\\<'QUOTE\ 'T>\end{matrix}\right.\ )\ \wedge\ (R1=0 \rightarrow$$

$$Pre(s)\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ \wedge\ (Post(s)\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ \rightarrow VR2,\ldots,RN(s)\ (T\ \left|\begin{matrix}R1\\s\end{matrix}\right|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ )))$$

We again assume the FCOMPEXP properties on a smaller portion of source code, this time on b1. The result is:

$$Pre(b1)\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ \wedge\ (Post(b1)\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ \rightarrow VR2,\ldots,RN(b1)$$

$$(((R1\neq 0 \rightarrow T\ \left|\begin{matrix}R1\\<'QUOTE\ 'T>\end{matrix}\right.\ )\ \wedge\ (R1=0 \rightarrow$$

$$Pre(s)\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ \wedge\ (Post(s)\ \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ \rightarrow VR2,\ldots,RN(s)\ (T\ \left|\begin{matrix}R1\\s\end{matrix}\right|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ ))))$$

$$\left|\begin{matrix}R1\\b1\end{matrix}\right|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ ))$$

Using * for the substitutions

$$\left|\begin{array}{l} v \\ w \end{array}\right| \begin{array}{l} NIL \\ 0 \end{array}$$

we get:

$$Pre(b1)\ *\ \wedge\ (Post(b1)\ *\ \rightarrow\ \forall R2,\ldots,RN(b1)\ (((R1\neq 0\ \rightarrow\ T\ \Big|^{R1}_{<'QUOTE\ 'T>}\ )\ \wedge$$

$$(R1=0\ \rightarrow\ Pre(s)\ *\ \wedge\ (Post(s)\ *\ \rightarrow\ \forall R2,\ldots,RN(s)\ (T\ \Big|^{R1}_{s\ *}\ ))))\ \Big|^{R1}_{b1\ *}\ ))$$

Distribute the last R1 substitution as far as possible. Then apply subrule 12 and subrule 4 to drop the b1 substitution on the first T term. Do the b1 substitution on R1=0 and R1≠0. R1 $\neg\epsilon$ Pre(s), Post(s), b1, s, w, and 0, so by subrule 3 we get R1 $\neg\epsilon$ Pre(s) *, Post(s) *, b1 *, and s *. So drop the R1 substitution on Pre(s) * and Post(s) * by subrule 4. Apply subrule 18b to move the outer R1 substitution inside ∀R2,...,RN(s). Then drop the outer R1 substitution on the last T term by subrule 12 and subrule 4. The result is:

$$Pre(b1)\ *\ \wedge\ (Post(b1)\ *\ \rightarrow\ \forall R2,\ldots,RN(b1)\ ((b1\ *\ \neq 0\ \rightarrow\ T\ \Big|^{R1}_{<'QUOTE\ 'T>})\ \wedge$$

$$(b1\ *\ =0\ \rightarrow\ Pre(s)\ *\ \wedge\ (Post(s)\ *\ \rightarrow\ \forall R2,\ldots,RN(s)\ (\ T\ \Big|^{R1}_{s\ *}\ )))))$$

The same argument as used on R1 above established for all i Ri $\neg\epsilon$ b1 *, Pre(s) *, and Post(s) *. Thus we can apply subrule 16 repeatedly, then regroup the AND-IMPLY structure, and finally apply subrule 17b to get:

$$Pre(b1)\ *\ \wedge\ (Post(b1)\ *\ \wedge\ b1\ *\ \neq 0\ \rightarrow\ \forall R2,\ldots,RN(b1)\ (\ T\ \Big|^{R1}_{<'QUOTE\ 'T>}\ ))$$

$$\wedge\ (Post(b1)\ *\ \wedge\ b1\ *\ =0\ \rightarrow\ Pre(s)\ *\ \wedge\ (Post(s)\ *\ \rightarrow$$

$$\forall R2,\ldots,Rmax(N(b1),N(s))\ (\ T\ \Big|^{R1}_{s\ *}\ )))$$

Completing the verification condition generation we get:

$$\mathrm{Pre}(S)\ *\ \wedge\ (\mathrm{Post}(S)\ *\ \rightarrow\ \forall R2,\ldots,RN(S)\ (\ T\ \Big|^{R1}_{S\ *}\ ))\ \rightarrow$$

$$\mathrm{Pre}(b1)\ *\ \wedge\ (\mathrm{Post}(b1)\ *\ \wedge\ b1\ *\ \neq 0\ \rightarrow\ \forall R2,\ldots,RN(b1)\ (\ T\ \Big|^{R1}_{<'\mathrm{QUOTE}\ 'T>}\ ))$$

$$\wedge\ (\mathrm{Post}(b1)\ *\ \wedge\ b1\ *\ =0\ \rightarrow\ \mathrm{Pre}(s)\ *\ \wedge\ (\mathrm{Post}(s)\ *\ \rightarrow$$

$$\forall R2,\ldots,\mathrm{Rmax}(N(b1),N(s))\ (\ T\ \Big|^{R1}_{s\ *}\ )))$$

We now expand Pre(S) and Post(S) by the formulas which apply for this subcase. Distributing the substitutions we get:

$$\mathrm{Pre}(b1)\ *\ \wedge\ (b1\ *\ =\ 0\ \rightarrow\ \mathrm{Pre}(s)\ *)\ \wedge$$

$$((b1\ *\ \neq\ 0\ \rightarrow\ S\ *\ =\ <'\mathrm{QUOTE}\ 'T>)\ \wedge\ (b1\ *\ =0\ \rightarrow\ S\ *\ =\ s\ *)\ \wedge\ \mathrm{Post}(b1)\ *\ \wedge$$

$$(b1\ *\ =\ 0\ \rightarrow\ \mathrm{Post}(s)\ *\ )\ \rightarrow\ \forall R2,\ldots,RN(S)\ (\ T\ \Big|^{R1}_{S\ *}\ ))$$

$$\rightarrow$$

$$\mathrm{Pre}(b1)\ *\ \wedge\ (\mathrm{Post}(b1)\ *\ \wedge\ b1\ *\ \neq 0\ \rightarrow\ \forall R2,\ldots,RN(b1)\ (\ T\ \Big|^{R1}_{<'\mathrm{QUOTE}\ 'T>}\ ))$$

$$\wedge\ (\mathrm{Post}(b1)\ *\ \wedge\ b1\ *\ =0\ \rightarrow\ \mathrm{Pre}(s)\ *\ \wedge\ (\mathrm{Post}(s)\ *\ \rightarrow$$

$$\forall R2,\ldots,\mathrm{Rmax}(N(b1),N(s))\ (\ T\ \Big|^{R1}_{s\ *}\ )))$$

Because Pre(b1)* appears in the hypotheses, we may set it to TRUE and simplify logically. The Pre(s)* term is implied by a hypothesis and may be similarly treated. The result is:

$$\mathrm{Pre}(b1)\ *\ \wedge\ (b1\ *\ =\ 0\ \rightarrow\ \mathrm{Pre}(s)\ *)\ \wedge$$

$$((b1\ *\ \neq\ 0\ \rightarrow\ S\ *\ =\ <'\mathrm{QUOTE}\ 'T>)\ \wedge\ (b1\ *\ =0\ \rightarrow\ S\ *\ =\ s\ *)\ \wedge\ \mathrm{Post}(b1)\ *\ \wedge$$

$$(b1\ *\ =\ 0\ \rightarrow\ \mathrm{Post}(s)\ *\ )\ \rightarrow\ \forall R2,\ldots,RN(S)\ (\ T\ \Big|^{R1}_{S\ *}\ ))$$

$$\rightarrow$$

$$(Post(b1)\ *\ \wedge\ b1\ *\ \neq\ 0\ \rightarrow\ \forall R2,\ldots,RN(b1)\ (\ T\ \begin{vmatrix} R1 \\ <'QUOTE\ 'T>\ ))\ \wedge$$

$$(Post(b1)\ *\ \wedge\ b1\ *\ =\ 0\ \wedge\ Post(s)\ *\ \rightarrow\ \forall R2,\ldots,Rmax(N(b1),N(s))$$

$$(\ T\ \begin{vmatrix} R1 \\ s\ *\ ))$$

We will break this into two cases: that of b1 ≠ NIL, and that of b1 = NIL.

Case 1: b1 ≠ NIL. Then in the target language, b1 * ≠ 0. We expand N(S) by the formula which holds for this subcase. Substituting TRUE or FALSE according to the case 1 information, then logically simplifying, the verification condition becomes:

$$Pre(b1)\ *\ \wedge$$

$$(S\ *\ =\ <'QUOTE\ 'T>\ \wedge\ Post(b1)\ *\ \rightarrow\ \forall R2,\ldots,RN(b1)\ (\ T\ \begin{vmatrix} R1 \\ S\ *\ ))\ \rightarrow$$

$$(Post(b1)\ *\ \rightarrow\ \forall R2,\ldots,RN(b1)\ (\ T\ \begin{vmatrix} R1 \\ <'QUOTE\ 'T>\ ))$$

which is obviously TRUE.

Case 2: b1 = NIL. Then b1 * = 0. In a manner similar to case 1 we get:

$$Pre(b1)\ *\ \wedge\ Pre(S)\ *\ \wedge\ (S\ *\ =\ s\ *\ \wedge\ Post(b1)\ *\ \wedge\ Post(s)\ *\ \rightarrow$$

$$\forall R2,\ldots,Rmax(N(b1),N(s))\ (\ T\ \begin{vmatrix} R1 \\ S\ *\ ))$$

$$\rightarrow$$

$$(Post(b1)\ *\ \wedge\ Post(s)\ *\ \rightarrow\ \forall R2,\ldots,Rmax(N(b1),N(s))\ (\ T\ \begin{vmatrix} R1 \\ s\ *\ ))$$

which is obviously TRUE.

We now must prove the stackok term for this case. This we will also do by the two subcases. For the no argument subcase we apply S3 to obtain the subgoals stackok(< < 'JRST 0 L1 > ... L1 >) and stackok(< < 'MOVEI 1 0 > L2 >). The first is proved by applying S8. For the second subgoal we use S3 again, then prove the resulting subgoals with S6 and S7, respectively.

For the n argument subcase, we will first apply S3 to obtain the subgoals stackok(FCOMPEXP(b1,M,LOCTABLE)) and stackok(< < 'JUMPN 1 L4 > ! FCOMPEXP(s,M,LOCTABLE) >). The first is part of the inductive assumption that these properties are true of smaller parts. The second subgoal requires S9.

The two subgoals thus created are

```
stackok(< < 'MOVEI 1 <'QUOTE 'T> >
          < 'JRST 0 L2 >
          L1
          < 'MOVEI 1 0 >
          L2 >)
```

and stackok(FCOMPEXP(s,M,LOCTABLE)). The first was derived by recalling that we previously proved that FCOMPEXP(s,M,LOCTABLE) always ended in L4 followed by code in this subgoal. We may prove this subgoal by applying S3 to get the subgoals stackok(< < 'MOVEI 1 <'QUOTE 'T> > >) and stackok(< < 'JRST 0 L2 > ... L2 >). The first of these is proved by S6 and the second by S8.

The remaining subgoal of stackok(FCOMPEXP(s,M,LOCTABLE)) is proved by appealing to the inductive assumption.

This completes the proof of the compiler for the case of OR.

A.8.6 ISNOT Case

We here prove the statement of correctness of the compiler for the case where ISNOT(S). The code produced by the compiler for the case is:

```
FCOMPEXP(S,M,LOCTABLE) = < ! FCOMPEXP(b1,M,LOCTABLE)
                           < 'JUMPN 1 L1 >
                           < 'MOVEI 1 <'QUOTE 'T> >
                           < 'JRST 0 L2 >
                           L1
                           < 'MOVEI 1 0 >
                           L2
                         >
```

where S is of form < 'NOT b1 >.

The assertion T is to have Hoare rules applied for these statements. First we note that assertion(L2) = T. Then we apply the MOVEI Hoare rule:

$$Q \begin{vmatrix} Ri \\ y \end{vmatrix} \{ < \text{'MOVEI i y} > \} \; Q$$

to obtain

$$T \begin{vmatrix} RI \\ 0 \end{vmatrix}$$

Then note that

$$\text{assertion(L1)} = T \begin{vmatrix} RI \\ 0 \end{vmatrix}$$

Next we apply the JRST Hoare rule:

$$\text{assertion}(1)\ \{ < \text{'JRST 0 1} > \}\ Q$$

to obtain T. Now apply the MOVEI Hoare rule, resulting in

$$T\ \Big|^{R1}_{<\text{'QUOTE 'T}>}$$

We now apply the JUMPN Hoare rule:

$$(R1 \neq 0 \rightarrow \text{assertion}(1)) \wedge (R1 = 0 \rightarrow Q)\ \{ < \text{'JUMPN 1 1} > \}\ Q$$

to obtain:

$$(R1 \neq 0 \rightarrow T\ \Big|^{R1}_{0}) \wedge (R1 = 0 \rightarrow T\ \Big|^{R1}_{<\text{'QUOTE 'T}>})$$

We assume the FCOMPEXP properties on b1, a smaller portion of source code (the inductive assumption). Then distributing the substitutions on R1, we get:

$$Pre(b1)\ * \wedge (Post(b1)\ * \rightarrow VR2, \ldots, RN(b1)$$

$$((b1\ * \neq 0 \rightarrow T\ \Big|^{R1}_{0}\ \Big|^{R1}_{b1\ *}) \wedge (b1\ * = 0 \rightarrow T\ \Big|^{R1}_{<\text{'QUOTE 'T}>}\ \Big|^{R1}_{b1\ *}))$$

By subrule 12 and subrule 4 we may drop the second of each pair of R1 substitutions. We now complete the verification condition generation to obtain:

$$Pre(S)\ * \wedge (Post(S)\ * \rightarrow VR2, \ldots, RN(S)\ (T\ \Big|^{R1}_{S\ *})) \rightarrow$$

$$Pre(b1)\ * \wedge (Post(b1)\ * \rightarrow VR2, \ldots, RN(b1)$$

$$((b1\ * \neq 0 \rightarrow T\ \Big|^{R1}_{0}) \wedge (b1 * = 0 \rightarrow T\ \Big|^{R1}_{<\text{'QUOTE 'T}>}))$$

We now expand Pre(S), Post(S), and N(S) by the formulas which apply for this case, to get (after distributing substitutions):

$$Pre(b1)\ * \wedge ((b1\ * = NIL\ * \rightarrow S\ * = <\text{'QUOTE 'T}>\ *) \wedge$$

$$(b1\ * \neq NIL\ * \rightarrow S\ * = NIL\ *) \wedge Post(b1)\ * \rightarrow$$

$$VR2, \ldots, RN(b1)\ (T\ \Big|^{R1}_{S\ *})$$

→

Pre(b1) * ∧ (Post(b1) * → VR2,...,RN(b1)

((b1 * ≠ 0 → T |R1 0) ∧ (b1 * = 0 → T |R1 <'QUOTE 'T>)))

Now NIL * is 0 and <'QUOTE 'T> * is <'QUOTE 'T>. Since Pre(b1) * is a hypothesis, we may eliminate it as a conclusion. We will break this into two cases: that of b1 = NIL, and that of b1 ≠ NIL.

Case 1: b1 = NIL. Then b1 * = 0. We then substitute TRUE or FALSE according to the case 1 information, then logically simplify to get:

Pre(b1) * ∧

(S * = <'QUOTE 'T> ∧ Post(b1) * → VR2,...,RN(b1) (T |R1 S *))

→

(Post(b1) * → VR2,...,RN(b1) (T |R1 <'QUOTE 'T>))

which is obviously TRUE.

Case 2: b1 ≠ NIL. Then b1 * ≠ 0. In a manner similar to case 1 we get:

Pre(b1) * ∧ (S * = 0 ∧ Post(b1) * → VR2,...,RN(b1) (T |R1 S *)) →

(Post(b1) * → VR2,...,RN(b1) (T |R1 0))

which is obviously TRUE.

We now must prove the stackok term for this case. We apply S9 to obtain the subgoals stackok(FCOMPEXP(b1,M,LOCTABLE)) and

```
stackok(< < 'JUMPN 1 L1 >
          < 'MOVEI 1 <'QUOTE 'T> >
          < 'JRST 0 L2 >
          L1
          < 'MOVEI 1 0 >
          L2
        >)
```

The first subgoal is part of the inductive assumption that these properties are true of smaller parts. The second requires S9.

The two subgoals thus created are stackok(< < 'MOVEI 1 0 > L2 >)

```
stackok(< < 'MOVEI 1 <'QUOTE 'T> >
          < 'JRST 0 L2 >
          L1
          < 'MOVEI 1 0 >
          L2
        >)
```

We may prove the first subgoal by applying S3, then proving the resulting subgoals with S6 and S7 respectively. The long subgoal above is proved by applying S3 to obtain the subgoals stackok(< < 'MOVEI 1 <'QUOTE 'T> >) and stackok(< < 'JRST 0 L2 > ... L2 >). The first is proved by S6 and the second by S8.

This completes the proof of the compiler for the case of NOT.

A.8.7 ISCOND Case

We here prove the statement of correctness of the compiler for the case where ISCOND(S). We will prove it in two subcases: that of no arguments, and that of one or more.

The code produced by the compiler for the first subcase is:

FCOMPEXP(S,M,LOCTABLE) = < L5 >

The assertion T is to have the Hoare rule applied for this statement. First we note that assertion(L5) = T. Completing the verification condition generation we get:

$$Pre(S) * \wedge (Post(S) * \rightarrow \forall R2,\ldots,RN(S)\ (T \mid_{S *))}^{R1} \rightarrow T$$

We now expand Pre(S), Post(S), and N(S) by the formulas which apply for this subcase, and let S=UNDEFINED, to obtain:

$$TRUE * \wedge ((UNDEFINED=UNDEFINED) * \rightarrow T \mid_{UNDEFINED *)}^{R1} \rightarrow T$$

This verification condition would simplify to TRUE if R1 $\neg\epsilon$ T. Then subrule 4 would allow us to drop the substitution. Indeed this is what we mean by saying that the value of S is undefined. That is, the proof of any program cannot depend on the value of a no argument COND, which is exactly what R1 represents in this subcase. Now since COND is defined recursively in terms of a COND with fewer arguments, this means that no proof of a program can depend on the value of a COND that executes all of its arguments without finding a non NIL one.

The code produced by the compiler for the second subcase is:

```
FCOMPEXP(S,M,LOCTABLE) = < ! FCOMPEXP(c1,M,LOCTABLE)
                           < 'JUMPE 1 L3 >
                           ! FCOMPEXP(d1,M,LOCTABLE)
                           < 'JRST L5 >
                           L3
                           ! FCOMPEXP(<'COND <c2 d2> ... <cn dn>>,
                                      M,LOCTABLE)
                         >
```

where S is of form < 'COND <c1 d1> <c2 d2> ... <cn dn> >.

Assuming the FCOMPEXP properties on < 'COND <c2 d2> ... <cn dn> >, a smaller portion of source code (the inductive assumption), we get:

$$Pre(s) * \wedge (Post(s) * \rightarrow VR2,\ldots,RN(s)\ (T \begin{vmatrix} R1 \\ s * \end{vmatrix}))$$

where s = < 'COND <c2 d2> ... <cn dn> >. Now we note that

$$assertion(L3) = Pre(s) * \wedge (Post(s) * \rightarrow VR2,\ldots,RN(s)\ (T \begin{vmatrix} R1 \\ s * \end{vmatrix}))$$

We wish to apply the JRST Hoare rule:

$$assertion(1)\ \{ < \text{'JRST 0 1} > \}\ Q$$

but the label L5 lies inside the final FCOMPEXP. By induction on n, the number of arguments, we obtain:

```
FCOMPEXP(S,M,LOCTABLE) = < ...
                           L5
                         >
```

Thus assertion(L5) is always T, and the verification condition becomes T. We again assume the FCOMPEXP properties on a smaller piece of source code to obtain:

$$Pre(d1) * \wedge (Post(d1) * \rightarrow VR2,\ldots,RN(d1)\ (T \begin{vmatrix} R1 \\ d1 * \end{vmatrix}))$$

We now apply the JUMPE Hoare rule:

$$(Ri{=}0 \rightarrow assertion(1)) \wedge (Ri{\neq}0 \rightarrow Q)\ \{ < \text{'JUMPE i 1} > \}\ Q$$

to obtain:

$$(R1{=}0 \rightarrow Pre(s)\,{*} \wedge (Post(s)\,{*} \rightarrow \forall R2,\ldots,RN(s)\ (T\ \Big|^{R1}_{s\,*}\,))) \wedge$$

$$(R1{\neq}0 \rightarrow Pre(d1)\,{*} \wedge (Post(d1)\,{*} \rightarrow \forall R2,\ldots,RN(d1)\ (T\ \Big|^{R1}_{d1\,*}\,)))$$

Yet again we assume the FCOMPEXP properties on a smaller portion of source code to get:

$$Pre(c1)\,{*} \wedge (Post(c1)\,{*} \rightarrow \forall R2,\ldots,RN(c1)$$

$$(((R1{=}0 \rightarrow Pre(s)\,{*} \wedge (Post(s)\,{*} \rightarrow \forall R2,\ldots,RN(s)\ (T\ \Big|^{R1}_{s\,*}\,))) \wedge$$

$$(R1{\neq}0 \rightarrow Pre(d1)\,{*} \wedge (Post(d1)\,{*} \rightarrow \forall R2,\ldots,RN(d1)\ (T\ \Big|^{R1}_{d1\,*}\,))))$$

$$\Big|^{R1}_{c1\,*}\,))$$

Distributing and performing the last R1 substitution we get:

$$Pre(c1)\,{*} \wedge (Post(c1)\,{*} \rightarrow \forall R2,\ldots,RN(c1)$$

$$((c1\,{*}{=}0 \rightarrow Pre(s)\,{*}\ \Big|^{R1}_{c1\,*} \wedge$$

$$(Post(s)\,{*}\ \Big|^{R1}_{c1\,*} \rightarrow \forall R2,\ldots,RN(s)\ (T\ \Big|^{R1}_{s\,*}\,)\ \Big|^{R1}_{c1\,*}\,)) \wedge$$

$$(c1\,{*}{\neq}0 \rightarrow Pre(d1)\,{*}\ \Big|^{R1}_{c1\,*} \wedge$$

$$(Post(d1)\,{*}\ \Big|^{R1}_{c1\,*} \rightarrow \forall R2,\ldots,RN(d1)\ (T\ \Big|^{R1}_{d1\,*}\,)\ \Big|^{R1}_{c1\,*}\,))\,)$$

Now $R1 \neg\epsilon$ Pre(s), Post(s), Pre(d1), Post(d1), 0, and the stack elements of w. By subrule 3 $R1 \neg\epsilon$ Pre(s) *, Post(s) *, Pre(d1) *, and Post(d1) *. Thus subrule 4 allows us to drop the R1 substitutions on Pre(s) *, Post(s) *, Pre(d1) *, and Post(d1) *. A similar argument shows us that $Ri \neg\epsilon$ c1 * for $2 \leq i$. Thus we can apply subrule 18 to move the c1 * substitutions inside the inner ∀'s. Similarly we get $R1 \neg\epsilon$ s * and $R1 \neg\epsilon$ d1 *. Then subrule 12 and subrule 4 allow us to drop the substitutions we just moved in. The result is:

$$Pre(c1)\ *\ \wedge\ (Post(c1)\ *\ \rightarrow\ VR2,\ldots,RN(c1)$$

$$((c1\ *\ =0\ \rightarrow\ Pre(s)\ *\ \wedge\ (Post(s)\ *\ \rightarrow\ VR2,\ldots,RN(s)\ \ (T\ \begin{vmatrix} R1 \\ s\ * \end{vmatrix}\ ))\ \wedge$$

$$(c1\ *\ \neq 0\ \rightarrow\ Pre(d1)\ *\ \wedge\ (Post(d1)\ *\ \rightarrow\ VR2,\ldots,RN(d1)\ \ (T\ \begin{vmatrix} R1 \\ d1\ * \end{vmatrix}\ ))\ ))$$

We complete the verification condition generation, then expand Pre(S) and Post(S) by the formulas which apply for this subcase. After distributing the * substitutions and performing them where possible, we get:

$$Pre(c1)\ *\ \wedge\ (c1\ *\ \neq\ 0\ \rightarrow\ Pre(d1)\ *)\ \wedge\ (c1\ *\ =\ 0\ \rightarrow\ Pre(s)\ *)\ \wedge$$

$$(Post(c1)\ *\ \wedge\ (c1\ *\ \neq\ 0\ \rightarrow\ Post(d1)\ *\ \wedge\ S\ *\ =\ d1\ *)\ \wedge$$

$$(c1\ *\ =\ 0\ \rightarrow\ Post(s)\ *\ \wedge\ S\ *\ =\ s\ *)\ \rightarrow\ VR2,\ldots,RN(S)\ \ (T\ \begin{vmatrix} R1 \\ S\ * \end{vmatrix}))$$

$$\rightarrow$$

$$Pre(c1)\ *\ \wedge\ (Post(c1)\ *\ \rightarrow\ VR2,\ldots,RN(c1)$$

$$((c1\ *\ =0\ \rightarrow\ Pre(s)\ *\ \wedge\ (Post(s)\ *\ \rightarrow\ VR2,\ldots,RN(s)\ \ (T\ \begin{vmatrix} R1 \\ s\ * \end{vmatrix}\ ))\ \wedge$$

$$(c1\ *\ \neq 0\ \rightarrow\ Pre(d1)\ *\ \wedge\ (Post(d1)\ *\ \rightarrow\ VR2,\ldots,RN(d1)\ \ (T\ \begin{vmatrix} R1 \\ d1\ * \end{vmatrix}\ ))\ ))$$

Since Pre(c1) * is a hypothesis, we may drop it as a conclusion. We will now break this into two cases: that of c1 = NIL, and that of c1 ≠ NIL.

Case 1: c1 = NIL. Then c1 * = 0. We expand N(S) by the formula which holds for this subcase. Substituting TRUE or FALSE according to the case 1 information, then logically simplifying, the verification condition becomes:

$$Pre(c1)\ *\ \wedge\ Pre(s)\ *\ \wedge$$

$$(Post(c1)\ *\ \wedge\ Post(s)\ *\ \wedge\ S\ *\ =\ s\ *\ \rightarrow$$

$$VR2,\ldots,Rmax(N(c1),N(s))\ \ (T\ \begin{vmatrix} R1 \\ S\ * \end{vmatrix}))$$

$$\rightarrow$$

$$(Post(c1)\ *\ \rightarrow\ \forall R2,\ldots,RN(c1)$$

$$(Pre(s)\ *\ \wedge\ (Post(s)\ *\ \rightarrow\ \forall R2,\ldots,RN(s)\ (T\ \begin{vmatrix} R1 \\ s\ * \end{vmatrix}\ ))))$$

Since for all i ($2 \leq i$) Ri $\neg\epsilon$ Pre(s), Post(s), 0, and the stack elements of w, we may apply subrule 3 to conclude that Ri $\neg\epsilon$ Pre(s) * and Post(s) *. Thus we may apply subrule 16 repeatedly to move the outer ∀ inward to the inner one. Then apply subrule 17b. Since Pre(s) * is a hypothesis, we may replace it with TRUE in the conclusion and logically simplify. The result is:

$$Pre(c1)\ *\ \wedge\ Pre(s)\ *\ \wedge$$

$$(Post(c1)\ *\ \wedge\ Post(s)\ *\ \wedge\ S\ *\ =\ s\ *\ \rightarrow$$

$$\forall R2,\ldots,Rmax(N(c1),N(s))\ (T\ \begin{vmatrix} R1 \\ S\ * \end{vmatrix}))$$

$$\rightarrow$$

$$(Post(c1)\ *\ \wedge\ Post(s)\ *\ \rightarrow\ \forall R2,\ldots,Rmax(N(c1),N(s))\ (T\ \begin{vmatrix} R1 \\ s\ * \end{vmatrix}))$$

which is obviously TRUE.

Case 2: c1 ≠ NIL. Then c1 ≠ 0. We expand N(S) again, then replace the case information by TRUE or FALSE, and logically simplify.

$$Pre(c1)\ *\ \wedge\ Pre(d1)\ *\ \wedge$$

$$(Post(c1)\ *\ \wedge\ Post(d1)\ *\ \wedge\ S\ *\ =\ d1\ *\ \rightarrow$$

$$\forall R2,\ldots,Rmax(N(c1),N(d1))\ (T\ \begin{vmatrix} R1 \\ S\ * \end{vmatrix}))$$

$$\rightarrow$$

$$(Post(c1)\ *\ \rightarrow\ \forall R2,\ldots,RN(c1)$$

$$(Pre(d1)\ *\ \wedge\ (Post(d1)\ *\ \rightarrow\ \forall R2,\ldots,RN(d1)\ (T\ \begin{vmatrix} R1 \\ d1\ * \end{vmatrix}\ ))))$$

Since for all i ($2 \leq i$) Ri $\neg\epsilon$ Pre(d1), Post(d1), 0, and the stack elements of w, we may apply subrule 3 to conclude that Ri $\neg\epsilon$ Pre(d1) * and Post(d1) *. Thus we may apply subrule 16 repeatedly to move the outer ∀ inward to the inner one. Then apply subrule 17b. Since Pre(d1) * is a hypothesis, we may replace it with TRUE in the conclusion and logically simplify. The result is:

```
Pre(c1) * ∧ Pre(d1) * ∧

(Post(c1) * ∧ Post(d1) * ∧ S * = d1 * →

                                                    |R1
  ∀R2,...,Rmax(N(c1),N(d1)) (T                      |S *))

→

                                                                    |R1
(Post(c1) * ∧ Post(d1) * → ∀R2,...,Rmax(N(c1),N(d1)) (T             |d1 * ))
```

which is obviously TRUE.

We now must prove the stackok term for this case. This we will also do by the two subcases. For the no argument subcase we apply S7.

For the n argument subcase, we will first apply S3 to obtain the subgoals stackok(FCOMPEXP(c1,M,LOCTABLE)) and

```
stackok(< < 'JUMPE 1 L3 >
          ! FCOMPEXP(d1,M,LOCTABLE)
          < 'JRST L5 >
          L3
          ! FCOMPEXP(s,M,LOCTABLE)
        >)
```

The first subgoal is part of the inductive assumption that these properties are true of smaller parts. The second requires S9. The two subgoals thus created are stackok(FCOMPEXP(s,M,LOCTABLE)) and

```
stackok(< ! FCOMPEXP(d1,M,LOCTABLE)
          < 'JRST L5 >
          L3
          ! FCOMPEXP(s,M,LOCTABLE)
        >)
```

The first subgoal is proved by appealing to the inductive assumption. The second requires S3. The first subgoal so produced is proved by the inductive assumption, while the second requires S8. In order to apply S8, we have to recall that we previously proved that FCOMPEXP(s,M,LOCTABLE) will end with L5.

This completes the proof of the compiler for the case of COND.

A.8.8 ISQUOTE Case

We here prove the statement of correctness of the compiler for the case where ISQUOTE(S). The code produced by the compiler for this case is:

```
FCOMPEXP(S,M,LOCTABLE) = < < 'MOVEI 1 <'QUOTE b1> > >
```

where S is of form <'QUOTE b1>.

The assertion T is to have the Hoare rule applied for this statement. Applying the MOVEI Hoare rule:

$$Q \begin{vmatrix} Ri \\ y \end{vmatrix} \{ < \text{'MOVEI i y} > \} \; Q$$

we get

$$T \begin{vmatrix} R1 \\ <\text{'QUOTE b1}> \end{vmatrix}$$

Completing the verification condition generation we obtain:

$$\text{Pre(S)} \; * \wedge \; (\text{Post(S)} \; * \rightarrow \forall R2, \ldots, RN(S) \; (T \begin{vmatrix} R1 \\ S \; * \end{vmatrix}) \rightarrow T \begin{vmatrix} R1 \\ <\text{'QUOTE b1}> \end{vmatrix}$$

Now expand Pre(S), Post(S), N(S), and S by the formulas which apply for this case. We previously showed that TRUE * is TRUE. Then simplifying logically we get:

$$T \begin{vmatrix} R1 \\ <\text{'QUOTE b1}> \end{vmatrix} * = T \begin{vmatrix} R1 \\ <\text{'QUOTE b1}> \end{vmatrix}$$

Now a QUOTE expression is a constant distinct from NIL and thus substitution for variables and NIL in the * will have no effect on it by subrule 4. Thus the verification condition is TRUE.

To prove the stackok term of the statement of correctness we simply apply S6. This completes the proof of the compiler for the ISQUOTE case.

A.8.9 ISFUNCTIONCALL Case

We here prove the statement of correctness of the compiler for the case where ISFUNCTIONCALL(S):

$$\text{ISEXPRESSION(S)} \rightarrow$$

$$\left(\text{Pre(S)} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} \text{NIL} \\ 0 \end{vmatrix} \wedge \; (\text{Post(S)} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} \text{NIL} \\ 0 \end{vmatrix} \rightarrow \forall R2, \ldots, RN(S) \; (T \begin{vmatrix} R1 \\ S \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} \text{NIL} \\ 0 \end{vmatrix}) \right)$$

$$\{\text{FCOMPEXP(S,M,LOCTABLE)}\} \; T \;)$$

$$\wedge \; \text{stackok(FCOMPEXP(S,M,LOCTABLE))}$$

where LOCTABLE is of form $< < \text{NAME1} . \text{LOC1} > \ldots < \text{NAMEr} . \text{LOCr} > >$, $v = < \text{NAME1} \ldots \text{NAMEr} >$, $w = < m[M+P+LOC1] \ldots m[M+P+LOCr] >$, and N(S) is a

function giving the maximum number of registers that are modified during execution of the compilation of S.

We have that

```
ISFUNCTIONCALL(S) → N(S) = max(N(f),N(b1),...,N(bn))
```

where S is of form < f b1 ... bn >.

To prove the Hoare rule portion of the compiler correctness we will apply Hoare rules to the T for the statements of FCOMPEXP for the particular case in question to form a verification condition. We will apply simplifications to the verification condition during and after generation to reduce it to TRUE.

Let S be of form < f b1 b2 ... bn >. Then we may express the code produced by the compiler for this case by:

```
FCOMPEXP(S,M,LOCTABLE) = < ! FCOMPLIS(<b1 ... bn>,M,LOCTABLE)
                           ! FLOADAC(1-n,1)
                           < 'SUB 'P <'C 0 0 n n> >
                           < 'CALL n <'E f> >
                         >
```

We apply the following Hoare rule for a CALL in target language:

```
          |NIL |a1 ... |an
Entry(f)  |0   |R1 ... |Rn ∧

          |NIL |h              |a1 ... |an
(Exit(f)  |0   |<f a1 ... an> |R1 ... |Rn →

                          |R1
  ∀R2,...,RN(f) (Q)       |<f R1 ... Rn> )

{ < 'CALL n <'E f> > } Q
```

where N(f) is the maximum number of registers modified during execution of function f, h is the designation in Exit(f) for the result of the function call to f, and a1 ... an are the formal arguments of f.

We obtain:

```
          |NIL |a1 ... |an
Entry(f)  |0   |R1 ... |Rn ∧

          |NIL |h              |a1 ... |an
( Exit(f) |0   |<f a1 ... an> |R1 ... |Rn →

                          |R1
  ∀R2,...,RN(f) (T)       |<f R1 ... Rn> )
```

We apply the following SUB Hoare rule:

$$Q \begin{vmatrix} P \\ P\text{-}n \end{vmatrix} \{ < \text{'SUB 'P <'C 0 0 n n> > } \} \; Q$$

resulting in:

$$(\text{Entry}(f) \begin{vmatrix} NIL & | a1 \; \ldots & | an \\ 0 & | R1 \; \ldots & | Rn \end{vmatrix} \wedge$$

$$(\text{Exit}(f) \begin{vmatrix} NIL & | h & | a1 \; \ldots & | an \\ 0 & | <f \; a1 \; \ldots \; an> & | R1 \; \ldots & | Rn \end{vmatrix} \rightarrow$$

$$\forall R2,\ldots,RN(f) \; (T) \begin{vmatrix} R1 \\ <f \; R1 \; \ldots \; Rn> \end{vmatrix}))) \begin{vmatrix} P \\ P\text{-}n \end{vmatrix}$$

We distribute the P substitution to the three terms. Noting that Entry and Exit are in source language, and so contain no P, we apply subrules 3b and 4 to drop the P substitution on the first two terms. Applying the FLOADAC rule and distributing substitutions we obtain:

$$\text{Entry}(f) \begin{vmatrix} NIL & | a1 \; \ldots & | an & | R1 \; \ldots & | Rn \\ 0 & | R1 \; \ldots & | Rn & | m[P\text{-}n\text{+}1] \; \ldots & | m[P] \end{vmatrix} \wedge$$

$$(\text{Exit}(f) \begin{vmatrix} NIL & | h & | a1 \; \ldots & | an & | R1 \; \ldots & | Rn \\ 0 & | <f \; a1 \; \ldots \; an> & | R1 \; \ldots & | Rn & | m[P\text{-}n\text{+}1] \; \ldots & | m[P] \end{vmatrix} \rightarrow$$

$$\forall R2,\ldots,RN(f) \; (T) \begin{vmatrix} R1 & | P & | R1 \; \ldots & | Rn \\ <f \; R1 \; \ldots \; Rn> & | P\text{-}n & | m[P\text{-}n\text{+}1] \; \ldots & | m[P] \end{vmatrix})$$

Repeated application of subrule 7a allows us to reorder the substitutions on the Entry to the order:

$$\begin{vmatrix} NIL & | a1 & | R1 \; \ldots & | an & | Rn \\ 0 & | R1 & | m[P\text{-}n\text{+}1] \; \ldots & | Rn & | m[P] \end{vmatrix}$$

Subrule 5 simplifies this to:

$$\begin{vmatrix} NIL & | a1 \; \ldots & | an \\ 0 & | m[P\text{-}n\text{+}1] \; \ldots & | m[P] \end{vmatrix}$$

Similarly the substitutions on Exit become:

$$\begin{vmatrix} NIL & | h & | a1 \; \ldots & | an \\ 0 & | <f \; a1 \; \ldots \; an> & | m[P\text{-}n\text{+}1] \; \ldots & | m[P] \end{vmatrix}$$

Applications of subrules 12, 3, and 13 establish that R1 $\neg\epsilon$ the entire formula. We may therefore apply the FCOMPLIS rule to obtain:

$$\text{Pre(b1)} \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right. \wedge \text{(Post(b1)} \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right. \rightarrow \ldots$$

$$\text{Pre(bn)} \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right|\begin{matrix}m\\\beta(n-1)\end{matrix}\right. \wedge \text{(Post(bn)} \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right|\begin{matrix}m\\\beta(n-1)\end{matrix}\right. \rightarrow$$

$$\forall R2, \ldots, Rmax(N(b1), \ldots, N(bn))$$

$$\text{((Entry(f)} \left|\begin{matrix}\text{NIL}\\0\end{matrix}\right|\begin{matrix}a1 & \ldots & an\\m[P-n+1] & \ldots & m[P]\end{matrix}\right. \wedge$$

$$\text{(Exit(f)} \left|\begin{matrix}\text{NIL}\\0\end{matrix}\right|\begin{matrix}h\\\langle f\ a1\ \ldots\ an\rangle\end{matrix}\right|\begin{matrix}a1 & \ldots & an\\m[P-n+1] & \ldots & m[P]\end{matrix}\right. \rightarrow$$

$$\forall R2, \ldots, RN(f)\ \text{(T)} \left|\begin{matrix}R1\\\langle f\ R1\ \ldots\ Rn\rangle\end{matrix}\right|\begin{matrix}P\\P-n\end{matrix}\left|\begin{matrix}R1 & \ldots & Rn\\m[P-n+1] & \ldots & m[P]\end{matrix}\right)\left|\begin{matrix}P\\P+n\end{matrix}\right|\begin{matrix}m\\\beta(n)\end{matrix}$$

$$) \ ) \ \ldots \ )$$

We now distribute the P+n and the $\beta(n)$ substitutions, then apply subrule 8 to P+n to move it inward to the Entry and the Exit. The P+n substitutions may now be discarded by subrule 4. On the T term we may similarly move the P+n in as far as the P-n substitution. Then use subrule 9 on the P+n and P-n, and simplify arithmetically to obtain a substitution of P for P, which may be discarded by subrule 14. The portion of the formula in the first $\forall$ is now:

$$\text{(Entry(f)} \left|\begin{matrix}\text{NIL}\\0\end{matrix}\right|\begin{matrix}a1 & \ldots & an\\m[P+1] & \ldots & m[P+n]\end{matrix}\left|\begin{matrix}m\\\beta(n)\end{matrix}\right. \wedge$$

$$\text{(Exit(f)} \left|\begin{matrix}\text{NIL}\\0\end{matrix}\right|\begin{matrix}h\\\langle f\ a1\ \ldots\ an\rangle\end{matrix}\left|\begin{matrix}a1 & \ldots & an\\m[P+1] & \ldots & m[P+n]\end{matrix}\right|\begin{matrix}m\\\beta(n)\end{matrix}\right. \rightarrow$$

$$\forall R2, \ldots, RN(f)\ \text{(T)} \left|\begin{matrix}R1\\\langle f\ R1\ \ldots\ Rn\rangle\end{matrix}\right|\begin{matrix}R1 & \ldots & Rn\\m[P+1] & \ldots & m[P+n]\end{matrix}\left|\begin{matrix}m\\\beta(n)\ ))\end{matrix}\right.$$

Subrule 16 may be used to move the $\forall$ containing this expression inward to the inside $\forall$ term (with its substitutions).

Recall that the a's are the formal parameters of function f. The a's $\neg\epsilon$ w, which is a list of elements from the array m. Now for some ai, either ai $\neg\epsilon$ b's, or ai ϵ bj for one or more j's. In the first case, subrule 3 gives us

$$ai\ \neg\epsilon\ bk \left|\left|\begin{matrix}v\\w\end{matrix}\right|\begin{matrix}\text{NIL}\\0\end{matrix}\right.$$

and therefore the a's $\neg\epsilon$ $\beta(n)$. In the second case, a formal parameter has simply been named the same as a previously declared variable. Thus the previous declaration will cause it to appear as a name in v. So we can apply subrule 12 at that point, and subrule 3 on the substitutions of v made after that point to show that

$$ai\ \neg\epsilon\ bk \quad \left|\begin{matrix} v \\ w \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \end{matrix}\right.$$

and therefore the a's $\neg\epsilon$ $\beta(n)$. So in either case we may apply subrule 8 repeatedly to $\beta(n)$ on both the Entry and Exit terms. Use subrule 7 to move the $\beta(n)$ past the NIL and h substitutions. Subrule 4 allows us to drop the $\beta(n)$ substitution on both Entry and Exit. Recall that $\beta(n)[P+j]$ is

$$bj \quad \left|\begin{matrix} v \\ w \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \end{matrix}\right.$$

for $1 \leq j \leq n$. The R's $\neg\epsilon$ b's and R's $\neg\epsilon$ w, so subrule 9 gives us R's $\neg\epsilon$ any term of $\beta(n)$, and therefore $\neg\epsilon$ $\beta(n)$. Thus we may apply subrule 8 on $\beta(n)$ on the last term repeatedly. The above part of the formula (including the outside $\forall$) is now:

$$Entry(f) \quad \left|\begin{matrix} NIL \\ 0 \end{matrix}\right. \left|\begin{matrix} a1 & \ldots \\ b1 * & \ldots \end{matrix}\right. \left|\begin{matrix} an \\ bn * \end{matrix}\right. \wedge$$

$$(Exit(f) \quad \left|\begin{matrix} NIL \\ 0 \end{matrix}\right. \left|\begin{matrix} h \\ \langle f\ a1\ \ldots\ an \rangle \end{matrix}\right. \left|\begin{matrix} a1 & \ldots \\ b1 * & \ldots \end{matrix}\right. \left|\begin{matrix} an \\ bn * \end{matrix}\right. \rightarrow$$

$$\forall R2, \ldots, Rmax(N(b1), \ldots, N(bn))\ (\ $$

$$\forall R2, \ldots, RN(f)\ (T) \quad \left|\begin{matrix} m \\ \beta(n) \end{matrix}\right. \left|\begin{matrix} R1 \\ \langle f\ R1\ \ldots\ Rn \rangle \end{matrix}\right. \left|\begin{matrix} R1 & \ldots \\ b1 * & \ldots \end{matrix}\right. \left|\begin{matrix} Rn \\ bn * \end{matrix}\right.)\ )$$

Recall that the * notation means the substitutions

$$\left|\begin{matrix} v \\ w \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \end{matrix}\right.$$

The form

$$\left|\begin{matrix} x1 & \\ y1 * & \end{matrix}\right. \left|\begin{matrix} x2 \\ y2 \end{matrix}\right.$$

means

$$\left|\begin{matrix} x1 & & \\ & \left|\begin{matrix} v \\ w \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \end{matrix}\right. & \\ y1 & & y2 \end{matrix}\right. \left|\begin{matrix} x2 \\ \\ y2 \end{matrix}\right.$$

Apply subrule 9 to the R1 substitutions. Now apply subrule 8 followed by subrule 19 to each of the other Ri's to make the last term of the above formula become:

$$\forall R2,\ldots,Rmax(N(b1),\ldots,N(bn))\ (\ \forall R2,\ldots,RN(f)\ (T\ \begin{vmatrix} m \\ \beta(n)\ \end{vmatrix}\ \begin{vmatrix} R1 \\ <f\ b1\ *\ \ldots\ bn\ *\ >\ \end{vmatrix}$$

In a manner similar to showing Ri $\neg\epsilon$ β, we show Ri $\neg\epsilon$ the item substituted for R1. Subrule 18 will now move the R1 substitution inside the inner $\forall$, and then subrule 17b can combine the two $\forall$'s. The resulting $\forall$ includes R2,...,Rmax(N(f),N(b1),...,N(bn)).

Recall that $\beta(j)$ is the same as m up to and including the Pth element. Thus the β substitutions may be dropped (by subrule 14) if they are applied to an expression that refers only to the first P elements of m. Now the elements of m past the Pth represent the values of the formal parameters of f. Thus T and all the Pre's and Post's (after the substitutions that change source language names to target language elements of m) will indeed not use m past element P, since they cannot refer to the formal parameters of f.

To complete the verification condition generation, we now add the terms before the "{" in the FCOMPEXP Hoare rule as a hypothesis to the formula we have simplified. The result is:

$$Pre(S)\ *\ \wedge\ (Post(S)\ *\ \rightarrow\ \forall R2,\ldots,RN(S)\ (T\ \begin{vmatrix} R1 \\ S\ *\ \end{vmatrix})\ \rightarrow$$

$$Pre(b1)\ *\ \wedge\ (Post(b1)\ *\ \rightarrow\ \ldots\ Pre(bn)\ *\ \wedge\ (Post(bn)\ *\ \rightarrow$$

$$Entry(f)\ \begin{vmatrix} NIL & a1 & \ldots & an \\ 0 & b1\ * & \ldots & bn\ * \end{vmatrix}\ \wedge$$

$$(Exit(f)\ \begin{vmatrix} NIL & h & a1 & \ldots & an \\ 0 & <f\ a1\ \ldots\ an> & b1\ * & \ldots & bn\ * \end{vmatrix}\ \rightarrow$$

$$\forall R2,\ldots,Rmax(N(f),N(b1),\ldots,N(bn))\ (T\ \begin{vmatrix} R1 \\ <f\ b1\ *\ \ldots\ bn\ *\ > \end{vmatrix}$$

$$)\ \ldots\ )$$

We now expand Pre(S) and Post(S) in the hypothesis by the formulas which apply for the case of a function call. Let S = <f b1 ... bn>.

Distribute the v and NIL substitutions (the *) over the terms of Pre(S) and Post(S). Now substitute TRUE in the conclusion for occurrences of the hypotheses, then simplify the AND-IMPLY structure, resulting in:

$$Pre(b1)\ *\ \wedge\ \ldots\ \wedge\ Pre(bn)\ *\ \wedge$$

$$(Post(b1)\ *\ \wedge\ \ldots\ \wedge\ Post(bn)\ *\ \rightarrow\ Entry(f)\ \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix}\ \begin{vmatrix} v & NIL \\ w & 0\ \end{vmatrix}\ )\ \wedge$$

$$(Post(b1) * \wedge \ldots \wedge Post(bn) * \wedge$$

$$Exit(f) \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix} \begin{vmatrix} h \\ \langle f\ b1\ \ldots\ bn \rangle \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix} \rightarrow$$

$$\forall R2, \ldots, RN(S)\ (T \begin{vmatrix} R1 \\ \langle f\ b1\ \ldots\ bn \rangle * \end{vmatrix}))$$

$\rightarrow$

$$(Post(b1) * \wedge \ldots \wedge Post(bn) * \rightarrow Entry(f) \begin{vmatrix} NIL \\ 0 \end{vmatrix} \begin{vmatrix} a1 & \ldots & an \\ b1 * & \ldots & bn * \end{vmatrix}) \wedge$$

$$(Post(b1) * \wedge \ldots \wedge Post(bn) * \wedge$$

$$Exit(f) \begin{vmatrix} NIL \\ 0 \end{vmatrix} \begin{vmatrix} h \\ \langle f\ a1\ \ldots\ an \rangle \end{vmatrix} \begin{vmatrix} a1 & \ldots & an \\ b1 * & \ldots & bn * \end{vmatrix} \rightarrow$$

$$\forall R2, \ldots, Rmax(N(f), N(b1), \ldots, N(bn))\ (T \begin{vmatrix} R1 \\ \langle f\ b1 * \ldots\ bn * \rangle \end{vmatrix}))$$

Use subrule 8 to change the conclusion Exit term to:

$$Exit(f) \begin{vmatrix} NIL \\ 0 \end{vmatrix} \begin{vmatrix} a1 & \ldots & an \\ b1 * & \ldots & bn * \end{vmatrix} \begin{vmatrix} h \\ \langle f\ b1 * \ldots\ bn * \rangle \end{vmatrix}$$

Then use subrule 8 to change the hypothesis Exit term to:

$$Exit(f) \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix} \begin{vmatrix} h \\ \langle f\ b1 * \ldots\ bn * \rangle \end{vmatrix}$$

Apply subrule 15 to the Entry and Exit terms in the hypotheses. Since Entry(f) and *Exit(f) contain only a's and h (they must be expressed only in terms of the formal parameters* and result), we may apply subrule 12 and subrule 3 to establish that for any NAMEi in v,

$$NAMEi \neg\epsilon\ Entry(f) \begin{vmatrix} a1 & \ldots & an \\ b1 & & bn \end{vmatrix} \begin{vmatrix} v \\ w \ldots \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix}$$

which resulted from subrule 15. Thus subrule 4 allows us to drop the final w substitution on this term, and similarly drop it on the Exit term of the hypotheses. Then use subrule 8 repeatedly to move the NIL substitution in on both the Entry and Exit terms of the hypotheses. The first conclusion now exactly matches a hypothesis, and so may be dropped. The result is:

$$Pre(b1) * \wedge \ldots \wedge Pre(bn) * \wedge$$

$$(Post(b1) * \wedge \ldots \wedge Post(bn) * \rightarrow Entry(f) \begin{vmatrix} NIL & a1 & \ldots & an \\ 0 & b1 * & \ldots & bn * \end{vmatrix}) \wedge$$

$$(Post(b1) * \wedge \ldots \wedge Post(bn) * \wedge$$

$$Exit(f) \begin{vmatrix} NIL & a1 & \ldots & an & h \\ 0 & b1 * & \ldots & bn * & <f\ b1 * \ldots bn * > \rightarrow \end{vmatrix}$$

$$\forall R2, \ldots, RN(S)\ (T \begin{vmatrix} R1 \\ <f\ b1 \ldots bn> * \end{vmatrix}))$$

$\rightarrow$

$$(Post(b1) * \wedge \ldots \wedge Post(bn) * \wedge$$

$$Exit(f) \begin{vmatrix} NIL & a1 & \ldots & an & h \\ 0 & b1 * & \ldots & bn * & <f\ b1 * \ldots bn * > \rightarrow \end{vmatrix}$$

$$\forall R2, \ldots, Rmax(N(f), N(b1), \ldots, N(bn))\ (T \begin{vmatrix} R1 \\ <f\ b1 * \ldots bn * >) \end{vmatrix})$$

Use subrule 6 repeatedly to distribute the NIL and the items in v into <f b1 ... bn> in the last hypothesis. Since max(N(f),N(b1),...,N(bn)) is N(S), the conclusion matches the last hypothesis. The Hoare rule for the function call case is proved.

The other term which we must prove about this case is stackok(FCOMPEXP(S,M,LOCTABLE)). To prove it for the function call case, apply S3 and S4 to the target code produced in this case. This gets rid of the CALL statement. By induction on n, we may easily derive the containspushes property of FCOMPLIS:

containspushes(FCOMPLIS(< b1 ... bn >, M, LOCTABLE), n)

This allows us to apply S2 to get a subgoal of stackok(FLOADAC(1-n,1)). But this is obvious by S5 and the fact that FLOADAC always contains only MOVE statements.

This completes the proof of the compiler for the case of a function call.

A.8.10 ISLAMBDA Case

We here prove the statement of correctness of the compiler for the case where ISLAMBDA(S). This case is similar to the function call case because a lambda expression is essentially a call to an unnamed function. The major differences lie in passing the arguments in the stack rather than the registers, and in having the code of the lambda compiled in-line. The former difference avoids the use of the LOADAC sequence of instructions in the calling code and the MKPUSH sequence in the called code, while the latter difference avoids the use of the CALL and POPJ instructions. The statement of correctness is:

ISEXPRESSION(S) →

$$\left(\text{Pre}(S) \;\left|\begin{matrix} v \\ w \end{matrix}\right|\left|\begin{matrix} \text{NIL} \\ 0 \end{matrix}\right. \wedge \left(\text{Post}(S) \;\left|\begin{matrix} v \\ w \end{matrix}\right|\left|\begin{matrix} \text{NIL} \\ 0 \end{matrix}\right. \rightarrow VR2,\ldots,RN(S)\ \left(T \;\left|\begin{matrix} RI \\ S \end{matrix}\right|\left|\begin{matrix} v \\ w \end{matrix}\right|\left|\begin{matrix} \text{NIL} \\ 0 \end{matrix}\right.\right)\right)$$

$$\{\text{FCOMPEXP}(S,M,\text{LOCTABLE})\}\ T\ )$$

$$\wedge\ \text{stackok}(\text{FCOMPEXP}(S,M,\text{LOCTABLE}))$$

where LOCTABLE is of form < < NAME1 . LOC1 > ... < NAMEr . LOCr > >, v = < NAME1 ... NAMEr >, w = < m[M+P+LOC1] ... m[M+P+LOCr] >, and N(S) is a function giving the maximum number of registers that are modified during execution of the compilation of S.

To prove the Hoare rule portion we will apply Hoare rules to the T for the statements of FCOMPEXP for the particular case in question to form a verification condition. We will apply simplifications to the verification condition during and after generation to reduce it to TRUE.

Let S be of form < <'LAMBDA <a1 ... an> exp> b1 ... bn >. Then we may express the code produced by the compiler for this case by:

```
FCOMPEXP(S,M,LOCTABLE) = < ! FCOMPLIS(<b1 ... bn>,M,LOCTABLE)
                           ! FCOMPEXP(exp,M-n,ADDIDS(LOCTABLE,
                                                     <a1 ... an>,
                                                     1-M))
                         < 'SUB 'P < 'C 0 0 n n > >
                       >
```

Using the SUB Hoare rule:

$$Q \;\left|\begin{matrix} P \\ P-n \end{matrix}\right.\ \{\ <\ \text{'SUB 'P <'C 0 0 n n> >}\ \}\ Q$$

gives us

$$T \;\left|\begin{matrix} P \\ P-n \end{matrix}\right.$$

Assuming the FCOMPEXP properties on exp, a smaller portion of source code (the inductive assumption), we get:

$$\text{Pre}(exp) \;\left|\begin{matrix} v' \\ w' \end{matrix}\right|\left|\begin{matrix} \text{NIL} \\ 0 \end{matrix}\right. \wedge \left(\text{Post}(exp) \;\left|\begin{matrix} v' \\ w' \end{matrix}\right|\left|\begin{matrix} \text{NIL} \\ 0 \end{matrix}\right. \rightarrow VR2,\ldots,RN(exp)\right.$$

$$\left(T \;\left|\begin{matrix} P \\ P-n \end{matrix}\right|\left|\begin{matrix} RI \\ exp \end{matrix}\right|\left|\begin{matrix} v' \\ w' \end{matrix}\right|\left|\begin{matrix} \text{NIL} \\ 0 \end{matrix}\right.\right)$$

We prime the v and w simply to distinguish them from v and w used earlier. They are different than the earlier ones because the LOCTABLE appearing in the FCOMPEXP property we are applying is now to have the value of ADDIDS(LOCTABLE,<a1 ... an>,1-M), and M is now to have the value M-n. From the code of the compiler we have:

ADDIDS(LOCTABLE,<a1 ... an>,1-M) = APPEND(PRUP(<a1 ... an>,1-M),
LOCTABLE)

We show in the proof of COMP that PRUP(<a1 ... an>,k) =

< <a1 . k> <a2 . k+1> ... <an . k+n-1> >

Thus if LOCTABLE is of form

< < NAME1 . LOC1 > ... < NAMEr . LOCr > >

then ADDIDS(LOCTABLE,<a1 ... an>,1-M) =

< <a1 . 1-M> <a2 . 2-M> ... <an . n-M>
<NAME1 . LOC1> ... <NAMEr . LOCr> >

Thus v' = < a1 a2 ... an NAME1 ... NAMEr >, and w' = < m[P-n+1] m[P-n+2] ... m[P] m[P+M-n+LOC1] ... m[P+M-n+LOCr] >.

We can see that R1 $\neg\epsilon$ Pre's, R1 $\neg\epsilon$ Post's, R1 $\neg\epsilon$ w', R1 $\neg\epsilon$ exp, and R1 $\neg\epsilon$ 0. Then by subrule 3b, subrule 12, and subrule 13 we get R1 $\neg\epsilon$ the entire formula above. Thus we may apply the FCOMPLIS rule to it to get:

$$Pre(b1) * \wedge (Post(b1) * \rightarrow$$

$$Pre(b2) * \begin{vmatrix} m \\ \beta(1) \end{vmatrix} \wedge (Post(b2) * \begin{vmatrix} m \\ \beta(1) \end{vmatrix} \rightarrow$$

...

$$Pre(bn) * \begin{vmatrix} m \\ \beta(n-1) \end{vmatrix} \wedge (Post(bn) * \begin{vmatrix} m \\ \beta(n-1) \end{vmatrix} \rightarrow$$

$$\forall R2, \ldots, Rmax(N(b1), \ldots, N(bn))$$

$$((Pre(exp) \begin{vmatrix} \begin{vmatrix} v' \\ w' \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix} \end{vmatrix} \wedge (Post(exp) \begin{vmatrix} \begin{vmatrix} v' \\ w' \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix} \end{vmatrix} \rightarrow$$

$$\forall R2, \ldots, RN(exp) \; (T \begin{vmatrix} P \\ P-n \end{vmatrix} \begin{vmatrix} R1 \\ exp \end{vmatrix} \begin{vmatrix} \begin{vmatrix} v' \\ w' \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix} \end{vmatrix}))) \begin{vmatrix} P \\ P+n \end{vmatrix} \begin{vmatrix} m \\ \beta(n) \end{vmatrix})$$

) ...))

We now distribute the P+n and $\beta(n)$ substitutions, then apply subrule 8 to P+n to move it inward to the Pre(exp) and Post(exp). The P+n substitutions may now be discarded by subrule 4. Since all Ri's $\neg\epsilon$ w, bj, or 0, subrule 3 yields that Ri's $\neg\epsilon$ all elements of $\beta(n)$, and therefore Ri's $\neg\epsilon$ $\beta(n)$. Now subrule 18b allows us to move the P+n and $\beta(n)$ substitutions inside the inner ∀. Use subrule 8 to move the P+n in as far as the P-n. Then use subrule 9 on the P+n and P-n, and simplify arithmetically to obtain a substitution of P for P, which may be discarded by subrule 14. The portion of the formula in the first ∀ is now:

$$\text{Pre(exp)} \left|\left|\begin{matrix} v' \\ w'' \end{matrix}\right| \begin{matrix} \text{NIL} \\ 0 \end{matrix} \left| \begin{matrix} m \\ \beta(n) \end{matrix}\right. \wedge \text{(Post(exp)} \left|\left|\begin{matrix} v' \\ w'' \end{matrix}\right| \begin{matrix} \text{NIL} \\ 0 \end{matrix} \left| \begin{matrix} m \\ \beta(n) \end{matrix}\right. \rightarrow$$

$$\forall R2,\ldots,RN(\text{exp})\ (T \left|\begin{matrix} R1 \\ \text{exp} \end{matrix}\right| \left|\left|\begin{matrix} v' \\ w' \end{matrix}\right| \begin{matrix} \text{NIL} \\ 0 \end{matrix} \left| \begin{matrix} P \\ P+n \end{matrix}\right| \left| \begin{matrix} m \\ \beta(n) \end{matrix}\right.))$$

where $w'' = < m[P+1]\ m[P+2] \ldots m[P+n]\ m[P+M+LOC1] \ldots m[P+M+LOCr] >$. Similarly we may move the P+n inward to the exp, then discard P+n. The w' on the exp will of course become a w''. Subrule 16 may be used to move the ∀ containing this expression inward to the inside ∀ term. Now apply subrule 17b to make the above part of the formula (including the outside ∀):

$$\text{Pre(exp)} \left|\left|\begin{matrix} v' \\ w'' \end{matrix}\right| \begin{matrix} \text{NIL} \\ 0 \end{matrix} \left| \begin{matrix} m \\ \beta(n) \end{matrix}\right. \wedge \text{(Post(exp)} \left|\left|\begin{matrix} v' \\ w'' \end{matrix}\right| \begin{matrix} \text{NIL} \\ 0 \end{matrix} \left| \begin{matrix} m \\ \beta(n) \end{matrix}\right. \rightarrow$$

$$\forall R2,\ldots,R\max(N(\text{exp}),N(b1),\ldots,N(bn))\ (T \left|\begin{matrix} R1 \\ \text{exp} \end{matrix}\right| \left|\left|\begin{matrix} v' \\ w'' \end{matrix}\right| \begin{matrix} \text{NIL} \\ 0 \end{matrix} \left| \begin{matrix} m \\ \beta(n) \end{matrix}\right.))$$

Apply subrule 8 to move the β substitution inward to the T. Since for all elements X of v' we have X $\neg\epsilon$ $\beta(n)$, the latter being in target language, we can also use subrule 8 to move the β substitutions inward to Pre(exp) and Post(exp). They can then be dropped by subrule 4. Recall that $\beta(n)[P+j]$ is

$$bj \left|\left|\begin{matrix} v \\ w \end{matrix}\right| \begin{matrix} \text{NIL} \\ 0 \end{matrix}\right.$$

for $1 \le j \le n$, and $\beta(n)[P+j]$ is $m[P+j]$ for $-P \le j \le 0$. Since -M is the size of the stack, all M+LOCi's are between -P and 0. Thus the above portion of the formula is:

$$\text{Pre(exp)} \left|\begin{matrix} a1 & \ldots & an \\ b1 * & \ldots & bn * \end{matrix}\right| \left|\left|\begin{matrix} v \\ w \end{matrix}\right| \begin{matrix} \text{NIL} \\ 0 \end{matrix} \right. \wedge$$

$$\text{(Post(exp)} \left|\begin{matrix} a1 & \ldots & an \\ b1 * & \ldots & bn * \end{matrix}\right| \left|\left|\begin{matrix} v \\ w \end{matrix}\right| \begin{matrix} \text{NIL} \\ 0 \end{matrix} \right. \rightarrow$$

$\forall R2, \ldots, Rmax(N(exp), N(b1), \ldots, N(bn))$

$$(T \begin{vmatrix} m \\ \beta(n) \end{vmatrix} \begin{vmatrix} R1 \\ exp \end{vmatrix} \begin{vmatrix} a1 & \ldots & an \\ b1 * & \ldots & bn * \end{vmatrix} \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix}))$$

Recall that $\beta(j)$ is the same as m up to and including the Pth element. Thus the β substitutions may be dropped (by subrule 14) if they are applied to an expression that refers only to the first P elements of m. Now the elements of m past the Pth represent the values of the formal parameters of the lambda. Thus T and all the Pre's and Post's (after the substitutions that change source language names to target language elements of m) will indeed not use m past element P, since they cannot refer to the formal parameters of the lambda. T could have resulted from an assertion that mentions the lambda expression as a whole. But the formal parameters of that lambda expression are not free uses of the names, and so will not be substituted for to produce elements of m past P.

We will use N(S) for the max (they are equal for the case of S being a lambda). Then to complete the verification condition generation, we add as a hypothesis the terms before the "{" in the Hoare rule we are trying to prove. The result is:

$$Pre(S) * \wedge (Post(S) * \rightarrow \forall R2, \ldots, RN(S) \ (T \begin{vmatrix} R1 \\ S * \end{vmatrix})) \rightarrow$$

$$Pre(b1) * \wedge (Post(b1) * \rightarrow$$

$$Pre(b2) * \wedge (Post(b2) * \rightarrow$$

$$\ldots$$

$$Pre(bn) * \wedge (Post(bn) * \rightarrow$$

$$Pre(exp) \begin{vmatrix} a1 & \ldots & an \\ b1 * & \ldots & bn * \end{vmatrix} \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix} \wedge$$

$$(Post(exp) \begin{vmatrix} a1 & \ldots & an \\ b1 * & \ldots & bn * \end{vmatrix} \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix} \rightarrow$$

$$\forall R2, \ldots, RN(S) \ (T \begin{vmatrix} R1 \\ exp \end{vmatrix} \begin{vmatrix} a1 & \ldots & an \\ b1 * & \ldots & bn * \end{vmatrix} \begin{vmatrix} v & NIL \\ w & 0 \end{vmatrix}))$$

$$) \ \ldots \ ))$$

We now expand Pre(S) and Post(S) in the hypothesis by the formulas which apply for the case of a lambda. Let

$$S = exp \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix}$$

Distribute the v and NIL substitutions (the *) over the terms of Pre(S) and Post(S). Now substitute TRUE in the conclusion for occurrences of the hypotheses, then simplify the AND-IMPLY structure, obtaining:

$$\mathrm{Pre}(b1)\,{*}\,\wedge \ldots \wedge \mathrm{Pre}(bn)\,{*}\,\wedge$$

$$(\mathrm{Post}(b1)\,{*}\,\wedge \ldots \wedge \mathrm{Post}(bn)\,{*}\,\rightarrow \mathrm{Pre}(exp)\ \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix}\ ) \wedge$$

$$(\mathrm{Post}(b1)\,{*}\,\wedge \ldots \wedge \mathrm{Post}(bn)\,{*}\,\wedge \mathrm{Post}(exp)\ \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix} \rightarrow$$

$$\forall R2,\ldots,RN(S)\ (T\ \begin{vmatrix} R1 \\ exp \end{vmatrix} \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix}\ ))$$

$\rightarrow$

$$(\mathrm{Post}(b1)\,{*}\,\wedge \ldots \wedge \mathrm{Post}(bn)\,{*}\,\rightarrow \mathrm{Pre}(exp)\ \begin{vmatrix} a1 & \ldots & an \\ b1\,{*} & \ldots & bn\,{*} \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix}\ )$$

$$(\mathrm{Post}(b1)\,{*}\,\wedge \ldots \wedge \mathrm{Post}(bn)\,{*}\,\wedge \mathrm{Post}(exp)\ \begin{vmatrix} a1 & \ldots & an \\ b1\,{*} & \ldots & bn\,{*} \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix} \rightarrow$$

$$\forall R2,\ldots,RN(S)\ (T\ \begin{vmatrix} R1 \\ exp \end{vmatrix} \begin{vmatrix} a1 & \ldots & an \\ b1\,{*} & \ldots & bn\,{*} \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix}\ ))$$

We will now work with the series of substitutions

$$X\ \begin{vmatrix} a1 & \ldots & an \\ b1 & \ldots & bn \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix}\ .$$

First we apply subrule 15 to obtain:

$$X\ \begin{vmatrix} a1 & \ldots & an \\ b1 & & bn \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

Now apply subrule 7b to get:

$$X\ \begin{vmatrix} a1 & \ldots & an \\ b1 & & bn \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix}$$

Apply subrule 8 repeatedly yielding:

$$X\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}\ \begin{vmatrix} a1 \\ b1 \end{vmatrix}\ \begin{vmatrix} v \\ w \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}\ \ldots\ \begin{vmatrix} an \\ bn \end{vmatrix}\ \begin{vmatrix} v \\ w \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}\ \begin{vmatrix} v \\ w \end{vmatrix}$$

Now by subrule 12 we have

$$NIL \neg\epsilon\ bi\ \begin{vmatrix} v \\ w \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

So we can now apply subrule 7b to obtain:

$$X\ \begin{vmatrix} a1 \\ b1 \end{vmatrix}\ \begin{vmatrix} v \\ w \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}\ \ldots\ \begin{vmatrix} an \\ bn \end{vmatrix}\ \begin{vmatrix} v \\ w \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}\ \begin{vmatrix} v \\ w \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

or, using the * notation:

$$X\ \begin{vmatrix} a1 \\ b1\ * \end{vmatrix}\ \ldots\ \begin{vmatrix} an \\ bn\ * \end{vmatrix}\ \begin{vmatrix} v \\ w \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

By replacing the series of substitutions with this equivalent in the three places they occur in the verification condition, we obtain conclusions that exactly match hypotheses, so the Hoare rule for the lambda case is proved.

To prove the stackok part of this case, we recall that we proved

containspushes(FCOMPLIS(<b1 ... bn>,M,LOCTABLE),n).

We then apply axiom S2 to get the subgoal of

stackok(FCOMPEXP(exp,M-n,ADDIDS(LOCTABLE,<a1 ... an>,1-M)).

But this is TRUE by inductive hypothesis.

This proves the lambda case.

A.8.11 Function Definition Case

We here prove the statement of correctness of the compiler for the case of a function definition (ISFUNCTIONDEF(S) is TRUE):

$$\text{ISFUNCTIONDEF}(S) \rightarrow$$

$$(\text{Entry}(f) \begin{vmatrix}\text{NIL} & \text{a1} & \ldots & \text{an} \\ 0 & \text{R1'} & \ldots & \text{Rn'}\end{vmatrix}$$

$$\{\ \text{FCOMP}(S)\ \}\ \text{Exit}(f) \begin{vmatrix}\text{h} & \text{NIL} & \text{a1} & \ldots & \text{an} \\ \text{R1} & 0 & \text{R1'} & \ldots & \text{Rn'}\end{vmatrix})$$

$$\wedge\ \text{stackokreturns}(\text{FCOMP}(S))$$

where S = < 'DE f < a1 ... an > exp >, h is the designation used in Exit(f) for the function value returned by the function f, and Ri' is the initial value of register Ri.

For S in this form, we have:

```
FCOMP(S) = < < 'LAP f 'SUBR >
               ! FMKPUSH(n,1)
               ! FCOMPEXP(exp,-n,PRUP(<a1 ... an>,1))
               < 'SUB 'P <'C 0 0 n n> >
               < 'POPJ 'P >
               'NIL
             >
```

We will apply Hoare rules to the right-hand portion of the Hoare rule part of the statement of correctness for the statements of FCOMP(S) in order to prove it. The 'NIL is just an end marker for the compiled code, and so may be ignored. We apply the POPJ Hoare rule (essentially a RETURN rule):

$$\text{Exit}(f) \begin{vmatrix}\text{h} & \text{NIL} & \text{a1} & \ldots & \text{an} \\ \text{R1} & 0 & \text{R1'} & \ldots & \text{Rn'}\end{vmatrix}\ \{\ <\ \text{'POPJ 'P}\ >\ \}\ Q$$

where f is the function in which we find the POPJ instruction and h and the a's are as previously, to obtain the same result.

Using the SUB Hoare rule:

$$Q \begin{vmatrix}P \\ P\text{-}n\ \{\ <\ \text{'SUB 'P <'C 0 0 n n>}\ >\ \}\ Q\end{vmatrix}$$

gives us:

$$\text{Exit}(f) \begin{vmatrix}\text{h} & \text{NIL} & \text{a1} & \ldots & \text{an} & P \\ \text{R1} & 0 & \text{R1'} & \ldots & \text{Rn'} & P\text{-}n\end{vmatrix}$$

Since P $\neg\epsilon$ Exit(f), nor 0, nor any registers, we can apply subrule 3b to establish

$$P \neg\epsilon \text{Exit}(f) \begin{vmatrix}\text{h} & \text{NIL} & \text{a1} & \ldots & \text{an} \\ \text{R1} & 0 & \text{R1'} & \ldots & \text{Rn'}\end{vmatrix}$$

Subrule 4 allows us to then drop the P-n substitution.

We now use the FCOMPEXP part of the statement of correctness inductively. LOCTABLE will be PRUP(<a1 ... an>,1), M will be -n, and S will be exp in using the FCOMPEXP properties. From the code of PRUP we have:

PRUP(<a1 ... an>,k) = < <a1 . k> ! PRUP(<a2 ... an>,k+1) >

Induction on n gives us:

PRUP(<a1 ... an>,k) = < <a1 . k> <a2 . k+1> ... <an . k+n-1> >

For the case where k = 1, we have:

PRUP(<a1 ... an>,1) = < <a1 . 1> <a2 . 2> ... <an . n> >

Thus the v of the FCOMPEXP properties is <a1 ... an>, and w is <m[-n+P+1] ... m[P]>. Thus verification condition generation back through FCOMPEXP gives us:

$$\text{Pre(exp)} \begin{vmatrix} a1 & \dots & an & NIL \\ m[P-n+1] & \dots & m[P] & 0 \end{vmatrix} \wedge$$

$$(\text{Post(exp)} \begin{vmatrix} a1 & \dots & an & NIL \\ m[P-n+1] & \dots & m[P] & 0 \end{vmatrix} \rightarrow$$

$$\forall R2, \dots, RN(\text{exp})$$

$$(\text{Exit(f)} \begin{vmatrix} h & NIL & a1 & \dots & an & R1 & a1 & \dots & an & NIL \\ R1 & 0 & R1' & \dots & Rn' & exp & m[P-n+1] & \dots & m[P] & 0 \end{vmatrix}))$$

For all i, $1 \le i \le n$ we may apply subrule 12 at the point where we substitute for ai on exp. Then apply subrule 3b for the remaining substitutions on exp to get

$$ai \neg\epsilon \begin{vmatrix} R1 & & & \\ exp & a1 & \dots & an & NIL \\ & m[P-n+1] & \dots & m[P] & 0 \end{vmatrix}$$

Then we may apply subrule 7b to rearrange the Exit term to:

$$\text{Exit(f)} \begin{vmatrix} h & R1 & & & & NIL & a1 & \dots & an \\ R1 & exp & a1 & \dots & an & NIL & & & \\ & & m[P-n+1] & \dots & m[P] & 0 & 0 & R1' & \dots & Rn' \end{vmatrix}$$

Since R1 $\neg\epsilon$ Exit(f), we may apply subrule 5 to obtain:

$$\text{Exit}(f)\ \begin{vmatrix} h & & & & & \text{NIL} & a1 & \ldots & an \\ & a1 & \ldots & an & \text{NIL} & & & & \\ exp & m[P-n+1] & \ldots & m[P] & 0 & 0 & R1' & \ldots & Rn' \end{vmatrix}$$

For all Ri ($2 \le i \le N(exp)$) we may apply subrule 3b to show that

$$Ri\ \neg\epsilon\ exp\ \begin{vmatrix} a1 & \ldots & an & \text{NIL} \\ m[P-m+1] & \ldots & m[P] & 0 \end{vmatrix}$$

then again to show that Ri $\neg\epsilon$ the quantified expression. We can therefore drop the quantifier by subrule 16c.

Apply the FMKPUSH rule to the entire expression, then distribute those FMKPUSH substitutions over the three terms. On both the Pre and Post terms we may apply subrule 8 repeatedly to each of the FMKPUSH substitutions to move them inward to Pre(exp) or Post(exp). Since P $\neg\epsilon$ Pre(exp) or Post(exp) and m $\neg\epsilon$ Pre(exp) or Post(exp), we may drop all the FMKPUSH substitutions directly on Pre(exp) and Post(exp) by subrule 4. The explanation at the end of the FMKPUSH rule derivation shows us that the m's result in Ri's when the FMKPUSH substitutions are applied to them. Thus we have:

$$\text{Pre}(exp)\ \begin{vmatrix} a1 & \ldots & an & \text{NIL} \\ R1 & \ldots & Rn & 0 \end{vmatrix} \wedge$$

$$(\text{Post}(exp)\ \begin{vmatrix} a1 & \ldots & an & \text{NIL} \\ R1 & \ldots & Rn & 0 \end{vmatrix} \rightarrow$$

$$\text{Exit}(f)\ \begin{vmatrix} h & & & & & \text{NIL} & a1 & \ldots & an \\ & a1 & \ldots & an & \text{NIL} & & & & \\ exp & m[P-n+1] & \ldots & m[P] & 0 & 0 & R1' & \ldots & Rn' \end{vmatrix}$$

$$\begin{vmatrix} P & m & \ldots & m \\ & & & \\ P+n & \alpha(m,P+n,Rn) & \ldots & \alpha(m,P+1,R1) \end{vmatrix}$$

Use subrule 7b on each of the FMKPUSH substitutions to move them inward past the NIL substitution. Then use subrule 8 to move each FMKPUSH substitution in past the h substitution. Then apply subrule 4 to drop all FMKPUSH substitutions directly applied to Exit(f). Now we still have the FMKPUSH substitutions applied to the end of the exp phrase. We can apply subrule 8 to move the FMKPUSH substitutions inward to exp, at which point subrule 4 allows us to drop them. The action of subrule 8 just applied has resulted in the FMKPUSH substitutions being applied to the m's, and we have previously shown that changes the m's to Ri's. The result is:

$$\text{Pre(exp)} \begin{vmatrix} a1 & \ldots & an & NIL \\ R1 & \ldots & Rn & 0 \end{vmatrix} \wedge$$

$$\text{(Post(exp)} \begin{vmatrix} a1 & \ldots & an & NIL \\ R1 & \ldots & Rn & 0 \end{vmatrix} \rightarrow$$

$$\text{Exit(f)} \begin{vmatrix} h & & & & NIL & a1 & \ldots & an \\ exp & \begin{vmatrix} a1 & \ldots & an & NIL \\ R1 & \ldots & Rn & 0 \end{vmatrix} & & & 0 & R1' & \ldots & Rn' \end{vmatrix}$$

Ignoring the LAP statement as simply an externally available label, we add the Entry hypothesis and drop the primes (initial value is same as present value here) to complete the verification condition generation, obtaining:

$$\text{Entry(f)} \begin{vmatrix} NIL & a1 & \ldots & an \\ 0 & R1 & \ldots & Rn \end{vmatrix} \rightarrow$$

$$\text{Pre(exp)} \begin{vmatrix} a1 & \ldots & an & NIL \\ R1 & \ldots & Rn & 0 \end{vmatrix} \wedge$$

$$\text{(Post(exp)} \begin{vmatrix} a1 & \ldots & an & NIL \\ R1 & \ldots & Rn & 0 \end{vmatrix} \rightarrow$$

$$\text{Exit(f)} \begin{vmatrix} h & & & & NIL & a1 & \ldots & an \\ exp & \begin{vmatrix} a1 & \ldots & an & NIL \\ R1 & \ldots & Rn & 0 \end{vmatrix} & & & 0 & R1 & \ldots & Rn \end{vmatrix}$$

Apply subrule 19 at the NIL substitutions in the Exit term. We may then use subrule 7b to rearrange the a1 substitution to the place before the NIL, and apply subrule 20 again. Similarly we can apply subrule 20 to all the substitutions on exp, then arrange the substitution of the a's back to their original order. Then use subrule 7b to place all NIL substitutions at the end of their terms. Subrule 6 applied then gives:

$$\text{(Entry(f)} \rightarrow \text{Pre(exp)} \wedge \text{(Post(exp)} \rightarrow \text{Exit(f)} \begin{vmatrix} h & a1 & \ldots & an & NIL \\ exp\ )) & R1 & \ldots & Rn & 0 \end{vmatrix}$$

But the quantity in the outermost parentheses must be TRUE in order to have a proof of the source program in the source language. Since TRUE with substitutions is still TRUE, we have proved the Hoare rule part of the case of a function definition.

To prove the stackokreturns property we use axiom S1, which reduces the problem to that of proving:

```
stackok( < ! FMKPUSH(n,1)
           ! FCOMPEXP(exp,-n,PRUP(<a1 ... an>,1))
           < 'SUB 'P <'C 0 0 n n> >
         > )
```

Note that we are still ignoring the LAP and NIL for the reasons given above. By induction on n it is easy to obtain from axioms C1 and C3:

containspushes(FMKPUSH(n,1), n)

We may now apply axiom S2, obtaining as the subgoal stackok(FCOMPEXP(exp,-n,PRUP(<a1 ... an>,1))). But this may be assumed by the inductive step.

This concludes the proof of the properties of FCOMP.

A.8.12 COMPLIS

We here derive a Hoare rule to describe the action of the instructions represented by FCOMPLIS. From the code of COMPLIS we have:

```
FCOMPLIS(<b1 ... bn>,M,LOCTABLE) =
          < ! FCOMPEXP(b1,M,LOCTABLE)
            < 'PUSH 'P 1 >
            ! FCOMPLIS(<b2 ... bn>,M-1,LOCTABLE)
          >
```

if n > 0, else an empty list is the result. Induction on n shows us that

```
FCOMPLIS(<b1 ... bn>,M,LOCTABLE) = < ! FCOMPEXP(b1,M,LOCTABLE)
                                     < 'PUSH 'P 1 >
                                     ! FCOMPEXP(b2,M-1,LOCTABLE)
                                     < 'PUSH 'P 1 >
                                     ...
                                     ! FCOMPEXP(bn,M-n+1,LOCTABLE)
                                     < 'PUSH 'P 1 >
                                   >
```

Since we are proving the compiler by induction on the syntactic structure of the source language, we may assume the FCOMPEXP result on smaller pieces of code, such as the b's. We apply this result and the following PUSH rule n times:

$$Q \begin{vmatrix} m \\ \alpha(m,P,Ri) \end{vmatrix} \begin{vmatrix} P \\ P+1 \end{vmatrix} \{ < \text{'PUSH 'P } i > \} \; Q$$

resulting in:

$$\text{Pre(b1)} \left|\left| \begin{matrix} v \\ W(0) \end{matrix} \right| \begin{matrix} NIL \\ 0 \end{matrix} \wedge (\text{Post(b1)} \left|\left| \begin{matrix} v \\ W(0) \end{matrix} \right| \begin{matrix} NIL \\ 0 \end{matrix} \rightarrow VR2,\ldots,RN(b1) \; ((\right.\right.$$

$$\text{Pre(b2)} \left|\left| \begin{matrix} v \\ W(-1) \end{matrix} \right| \begin{matrix} NIL \\ 0 \end{matrix} \wedge (\text{Post(b2)} \left|\left| \begin{matrix} v \\ W(-1) \end{matrix} \right| \begin{matrix} NIL \\ 0 \end{matrix} \rightarrow VR2,\ldots,RN(b2) \; ((\right.\right.$$

...

$$\text{Pre(bn-1)}\ \left|\left|\begin{matrix}v\\W(2-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \wedge\ (\text{Post(bn-1)}\ \left|\left|\begin{matrix}v\\W(2-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \rightarrow \forall R2,\ldots,RN(bn-1)(($$

$$\text{Pre(bn)}\ \left|\left|\begin{matrix}v\\W(1-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \wedge\ (\text{Post(bn)}\ \left|\left|\begin{matrix}v\\W(1-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \rightarrow \forall R2,\ldots,RN(bn)\ ($$

$$Q\ \begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\ \begin{matrix}P\\P+1\end{matrix}\ \begin{matrix}R1\\bn\end{matrix}\ \left|\left|\begin{matrix}v\\W(1-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ ))$$

$$)\ \begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\ \begin{matrix}P\\P+1\end{matrix}\ \begin{matrix}R1\\bn-1\end{matrix}\ \left|\left|\begin{matrix}v\\W(2-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ ))$$

...

$$)\ \begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\ \begin{matrix}P\\P+1\end{matrix}\ \begin{matrix}R1\\b2\end{matrix}\ \left|\left|\begin{matrix}v\\W(-1)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ ))$$

$$)\ \begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\ \begin{matrix}P\\P+1\end{matrix}\ \begin{matrix}R1\\b1\end{matrix}\ \left|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ ))$$

where $W(k) = < m[M+P+k+LOC1] \ldots m[M+P+k+LOCr] >$.

The W notation is necessitated by the fact that the M handed to COMPEXP is different at each call. We will eventually have only W(0) terms and be able to use the lower case w, which is equal to W(0) by definition.

Distribute the α, P+1, and bi etc. substitutions onto the Pre and Post terms. For all Ri ($i \geq 2$) we have that $Ri \neg\epsilon W(k)$, $Ri \neg\epsilon 0$, $Ri \neg\epsilon \alpha(m,P,R1)$, $Ri \neg\epsilon P+1$, $Ri \neg\epsilon bj$, $Ri \neg\epsilon Pre(bj)$, and $Ri \neg\epsilon Post(bj)$. Application of subrule 3b to first all the bj substitutions, then to the substitutions on all the Pre's and Post's results in $Ri \neg\epsilon$ any Pre or Post term with all substitutions indicated having been applied. We may now apply subrules 16a and 16b to obtain:

$$\text{Pre(b1)}\ \left|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \wedge\ (\text{Post(b1)}\ \left|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \rightarrow$$

$$\text{Pre(b2)}\ \left|\left|\begin{matrix}v\\W(-1)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\ \begin{matrix}P\\P+1\end{matrix}\ \begin{matrix}R1\\b1\end{matrix}\ \left|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \wedge$$

$$(\text{Post(b2)}\ \left|\left|\begin{matrix}v\\W(-1)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\ \begin{matrix}P\\P+1\end{matrix}\ \begin{matrix}R1\\b1\end{matrix}\ \left|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \rightarrow$$

$$\forall R2,\ldots,RN(b1)\ ((\ \ldots$$

$$\mathrm{Pre}(bn\text{-}1)\ \left|\left|\begin{matrix}v\\W(2\text{-}n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\left|\begin{matrix}P\\P{+}1\end{matrix}\right|\begin{matrix}R1\\bn\text{-}2\end{matrix}\left|\left|\begin{matrix}v\\W(3\text{-}n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ \wedge$$

$$(\mathrm{Post}(bn\text{-}1)\ \left|\left|\begin{matrix}v\\W(2\text{-}n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\left|\begin{matrix}P\\P{+}1\end{matrix}\right|\begin{matrix}R1\\bn\text{-}2\end{matrix}\left|\left|\begin{matrix}v\\W(3\text{-}n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ \rightarrow$$

$$\forall R2,\ldots,RN(bn\text{-}2)\ (($$

$$\mathrm{Pre}(bn)\ \left|\left|\begin{matrix}v\\W(1\text{-}n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\left|\begin{matrix}P\\P{+}1\end{matrix}\right|\begin{matrix}R1\\bn\text{-}1\end{matrix}\left|\left|\begin{matrix}v\\W(2\text{-}n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ \wedge$$

$$(\mathrm{Post}(bn)\ \left|\left|\begin{matrix}v\\W(1\text{-}n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\left|\begin{matrix}P\\P{+}1\end{matrix}\right|\begin{matrix}R1\\bn\text{-}1\end{matrix}\left|\left|\begin{matrix}v\\W(2\text{-}n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.\ \rightarrow$$

$$\forall R2,\ldots,RN(bn\text{-}1)\ (($$

$$\forall R2,\ldots,RN(bn)\ (Q\ \left|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\right|\begin{matrix}P\\P{+}1\end{matrix}\left|\begin{matrix}R1\\bn\end{matrix}\right|\left|\begin{matrix}v\\W(1\text{-}n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ )$$

$$)\ \left|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\right|\begin{matrix}P\\P{+}1\end{matrix}\left|\begin{matrix}R1\\bn\text{-}1\end{matrix}\right|\left|\begin{matrix}v\\W(2\text{-}n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ ))$$

$$)\ \left|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\right|\begin{matrix}P\\P{+}1\end{matrix}\left|\begin{matrix}R1\\bn\text{-}2\end{matrix}\right|\left|\begin{matrix}v\\W(3\text{-}n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ ))$$

$$\ldots$$

$$)\ \left|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\right|\begin{matrix}P\\P{+}1\end{matrix}\left|\begin{matrix}R1\\b1\end{matrix}\right|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ )))$$

By alternating this process of distributing substitutions onto Pre and Post terms, then subrules 16a and 16b (with a similar justification involving subrule 9), we eventually arrive at:

$$\text{Pre}(b1)\ \left|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right. \wedge\ (\text{Post}(b1)\ \left|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right. \rightarrow$$

$$\text{Pre}(b2)\ \left|\left|\begin{matrix}v\\W(-1)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\left|\begin{matrix}P\\P+1\end{matrix}\right|\begin{matrix}R1\\b1\end{matrix}\left|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right. \wedge$$

$$(\text{Post}(b2)\ \left|\left|\begin{matrix}v\\W(-1)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\left|\begin{matrix}P\\P+1\end{matrix}\right|\begin{matrix}R1\\b1\end{matrix}\left|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right. \rightarrow$$

...

$$\text{Pre}(bn-1)\ \left|\left|\begin{matrix}v\\W(2-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\left|\begin{matrix}P\\P+1\end{matrix}\right|\begin{matrix}R1\\bn-2\end{matrix}\left|\left|\begin{matrix}v\\W(3-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.$$

...

$$\left|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\right|\begin{matrix}P\\P+1\end{matrix}\left|\begin{matrix}R1\\b1\end{matrix}\right|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \wedge$$

$$(\text{Post}(bn-1)\ \left|\left|\begin{matrix}v\\W(2-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\left|\begin{matrix}P\\P+1\end{matrix}\right|\begin{matrix}R1\\bn-2\end{matrix}\left|\left|\begin{matrix}v\\W(3-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.$$

...

$$\left|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\right|\begin{matrix}P\\P+1\end{matrix}\left|\begin{matrix}R1\\b1\end{matrix}\right|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \rightarrow$$

$$\text{Pre}(bn)\ \left|\left|\begin{matrix}v\\W(1-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\left|\begin{matrix}P\\P+1\end{matrix}\right|\begin{matrix}R1\\bn-1\end{matrix}\left|\left|\begin{matrix}v\\W(2-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.$$

...

$$\left|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\right|\begin{matrix}P\\P+1\end{matrix}\left|\begin{matrix}R1\\b1\end{matrix}\right|\left|\begin{matrix}v\\W(0)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\ \wedge$$

$$(\text{Post}(bn)\ \left|\left|\begin{matrix}v\\W(1-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right|\begin{matrix}m\\\alpha(m,P,R1)\end{matrix}\left|\begin{matrix}P\\P+1\end{matrix}\right|\begin{matrix}R1\\bn-1\end{matrix}\left|\left|\begin{matrix}v\\W(2-n)\end{matrix}\right|\begin{matrix}NIL\\0\end{matrix}\right.$$

...

$$\left|\begin{matrix}\text{m}\\ \alpha(\text{m,P,R1})\end{matrix}\right. \left|\begin{matrix}\text{P}\\ \text{P+1}\end{matrix}\right. \left|\begin{matrix}\text{R1}\\ \text{b1}\end{matrix}\right. \left|\left|\begin{matrix}\text{v}\\ \text{W(0)}\end{matrix}\right.\right. \left|\begin{matrix}\text{NIL}\\ \text{0}\end{matrix}\right. \rightarrow$$

$\forall$R2,...,RN(b1) ($\forall$R2,...,RN(b2) (...

$$\forall \text{R2},\ldots,\text{RN(bn-1)}\ (\forall \text{R2},\ldots,\text{RN(bn)}\ (\text{Q}\ \left|\begin{matrix}\text{m}\\ \alpha(\text{m,P,R1})\end{matrix}\right. \left|\begin{matrix}\text{P}\\ \text{P+1}\end{matrix}\right. \left|\begin{matrix}\text{R1}\\ \text{bn}\end{matrix}\right. \left|\left|\begin{matrix}\text{v}\\ \text{W(1-n)}\end{matrix}\right.\right. \left|\begin{matrix}\text{NIL}\\ \text{0}\end{matrix}\right.)$$

$$\left|\begin{matrix}\text{m}\\ \alpha(\text{m,P,R1})\end{matrix}\right. \left|\begin{matrix}\text{P}\\ \text{P+1}\end{matrix}\right. \left|\begin{matrix}\text{R1}\\ \text{bn-1}\end{matrix}\right. \left|\left|\begin{matrix}\text{v}\\ \text{W(2-n)}\end{matrix}\right.\right. \left|\begin{matrix}\text{NIL}\\ \text{0}\end{matrix}\right.)$$

...

$$\left|\begin{matrix}\text{m}\\ \alpha(\text{m,P,R1})\end{matrix}\right. \left|\begin{matrix}\text{P}\\ \text{P+1}\end{matrix}\right. \left|\begin{matrix}\text{R1}\\ \text{b2}\end{matrix}\right. \left|\left|\begin{matrix}\text{v}\\ \text{W(-1)}\end{matrix}\right.\right. \left|\begin{matrix}\text{NIL}\\ \text{0}\end{matrix}\right.)$$

$$\left|\begin{matrix}\text{m}\\ \alpha(\text{m,P,R1})\end{matrix}\right. \left|\begin{matrix}\text{P}\\ \text{P+1}\end{matrix}\right. \left|\begin{matrix}\text{R1}\\ \text{b1}\end{matrix}\right. \left|\left|\begin{matrix}\text{v}\\ \text{W(0)}\end{matrix}\right.\right. \left|\begin{matrix}\text{NIL}\\ \text{0}\end{matrix}\right.)$$

)) ...))

Use subrule 8 to place every P+1 substitution just inside the corresponding α substitution, resulting in all α's becoming α(m,P+1,R1). We must now make the assumption that R1 $\neg\epsilon$ Q. This assumption must be verified before applying the rule for COMPLIS that we are deriving here. Using the facts R1 $\neg\epsilon$ Pre's, R1 $\neg\epsilon$ Post's, R1 $\neg\epsilon$ W(k), and R1 $\neg\epsilon$ 0, we can apply subrules 12 and 3 to show that in all places of the form

$$\text{X}\ \left|\begin{matrix}\text{m}\\ \alpha(\text{m,P+1,R1})\end{matrix}\right. \left|\begin{matrix}\text{R1}\\ \text{bi}\end{matrix}\right. \left|\left|\begin{matrix}\text{v}\\ \text{W(1-i)}\end{matrix}\right.\right. \left|\begin{matrix}\text{NIL}\\ \text{0}\end{matrix}\right.$$

it holds that R1 $\neg\epsilon$ X. So we apply subrule 5 in all such places to get:

$$\text{X}\ \left|\begin{matrix}\text{m}\\ \alpha(\text{m,P+1,bi}\end{matrix}\right. \left|\left|\begin{matrix}\text{v}\\ \text{W(1-i)}\end{matrix}\right.\right. \left|\begin{matrix}\text{NIL}\\ \text{0}\end{matrix}\right.)$$

We may now apply subrule 8 repeatedly to all P+1 substitutions in Pre or Post terms until they have moved in next to the innermost P+1. This will require us to apply a P+j

substitution to W(k). We now show that this results in W(k+j). First recall that W(k) means < m[M+P+k+LOC1] ... m[M+P+k+LOCr] > . Application of the substitution gives us < m[M+P+j+k+LOC1] ... m[M+P+j+k+LOCr] > , which is W(k+j). Thus using subrule 8 repeatedly gives us:

$$X \left|\left|{}^{v}_{W(k)}\right. \left|{}^{P}_{P+j} \right. = X \left|{}^{P}_{P+j}\right. \left|\left|{}^{v}_{W(k+j)}\right.\right.$$

Additionally, if $P \neg\epsilon X$, we may drop the P+j substitution on the right, by use of subrule 4. We now use subrule 9 repeatedly in all places of the form

$$X \left|{}^{P}_{P+1}\right. \left|{}^{P}_{P+1}\right. \ldots \left|{}^{P}_{P+1}\right.$$

where the substitution occurs r times ($r \geq 2$) and simplify arithmetic to obtain

$$X \left|{}^{P}_{P+r}\right.$$

Subrule 7 allows us to interchange the order of all the NIL-0 and P+j substitutions. We then apply the P+j substitutions to the W(k) substitutions, as justified above. The P+j's get dropped then since $P \neg\epsilon$ Pre's or Post's. The result is:

$$\text{Pre(b1)} \left|\left|{}^{v}_{W(0)}\right. \left|{}^{NIL}_{0}\right. \wedge \text{(Post(b1)} \left|\left|{}^{v}_{W(0)}\right. \left|{}^{NIL}_{0}\right. \rightarrow$$

$$\text{Pre(b2)} \left|\left|{}^{v}_{W(0)}\right. \left|{}^{NIL}_{0}\right. \left|{}^{m}_{\alpha(m,P+1,b1}\right. \left|\left|{}^{v}_{W(0)}\right. \left|{}^{NIL}_{0}\right. \right) \wedge$$

$$\text{(Post(b2)} \left|\left|{}^{v}_{W(0)}\right. \left|{}^{NIL}_{0}\right. \left|{}^{m}_{\alpha(m,P+1,b1}\right. \left|\left|{}^{v}_{W(0)}\right. \left|{}^{NIL}_{0}\right. \right) \rightarrow$$

...

$$\text{Pre(bn-1)} \left|\left|{}^{v}_{W(0)}\right. \left|{}^{NIL}_{0}\right. \left|{}^{m}_{\alpha(m,P+n-2,bn-2}\right. \left|\left|{}^{v}_{W(0)}\right. \left|{}^{NIL}_{0}\right. \right)$$

...

$$\left|{}^{m}_{\alpha(m,P+1,b1}\right. \left|\left|{}^{v}_{W(0)}\right. \left|{}^{NIL}_{0}\right. \right) \wedge$$

$$\text{(Post(bn-1)} \left|\left|{}^{v}_{W(0)}\right. \left|{}^{NIL}_{0}\right. \left|{}^{m}_{\alpha(m,P+n-2,bn-2}\right. \left|\left|{}^{v}_{W(0)}\right. \left|{}^{NIL}_{0}\right. \right)$$

...

$$\left|\begin{matrix} m \\ \alpha(m,P+1,b1 \end{matrix}\right| \left|\left|\begin{matrix} v \\ W(0) \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \quad) \end{matrix}\right| \rightarrow$$

$$Pre(bn) \left|\left|\begin{matrix} v \\ W(0) \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \end{matrix}\right| \left|\begin{matrix} m \\ \alpha(m,P+n-1,bn-1 \end{matrix}\right| \left|\left|\begin{matrix} v \\ W(0) \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \quad) \end{matrix}\right|$$

$$\ldots$$

$$\left|\begin{matrix} m \\ \alpha(m,P+1,b1 \end{matrix}\right| \left|\left|\begin{matrix} v \\ W(0) \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \quad) \wedge \end{matrix}\right|$$

$$(Post(bn) \left|\left|\begin{matrix} v \\ W(0) \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \end{matrix}\right| \left|\begin{matrix} m \\ \alpha(m,P+n-1,bn-1 \end{matrix}\right| \left|\left|\begin{matrix} v \\ W(0) \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \quad) \end{matrix}\right|$$

$$\ldots$$

$$\left|\begin{matrix} m \\ \alpha(m,P+1,b1 \end{matrix}\right| \left|\left|\begin{matrix} v \\ W(0) \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \quad) \rightarrow \end{matrix}\right|$$

$$\forall R2,\ldots,RN(b1)\ (\ \forall R2,\ldots,RN(b2)\ (\ \ldots$$

$$\forall R2,\ldots,RN(bn-1)\ (\forall R2,\ldots,RN(bn)\ (Q \left|\begin{matrix} P \\ P+1 \end{matrix}\right| \left|\begin{matrix} m \\ \alpha(m,P+1,bn \end{matrix}\right| \left|\left|\begin{matrix} v \\ W(1-n) \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \quad) \ ) \end{matrix}\right|$$

$$\left|\begin{matrix} P \\ P+1 \end{matrix}\right| \left|\begin{matrix} m \\ \alpha(m,P+1,bn-1 \end{matrix}\right| \left|\left|\begin{matrix} v \\ W(2-n) \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \quad) \ ) \end{matrix}\right|$$

$$\ldots$$

$$\left|\begin{matrix} P \\ P+1 \end{matrix}\right| \left|\begin{matrix} m \\ \alpha(m,P+1,b2 \end{matrix}\right| \left|\left|\begin{matrix} v \\ W(-1) \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \quad) \ ) \end{matrix}\right|$$

$$\left|\begin{matrix} P \\ P+1 \end{matrix}\right| \left|\begin{matrix} m \\ \alpha(m,P+1,b1 \end{matrix}\right| \left|\left|\begin{matrix} v \\ W(0) \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \quad) \ ) \end{matrix}\right|$$

$$)) \ \ldots \ ))$$

Use of subrule 3 allows us to conclude for any i

$$Ri \neg\epsilon\ \alpha(m,P+1,bj \left|\left|\begin{matrix} v \\ W(1-j) \end{matrix}\right| \left|\begin{matrix} NIL \\ 0 \quad) \end{matrix}\right|$$

This allows us to apply subrule 18b repeatedly to reduce the part of the formula after the Post(bn) term to:

$$\forall R2,\ldots,RN(b1)\ (\ \forall R2,\ldots,RN(b2)\ (\ \ldots$$

$$\forall R2,\ldots,RN(bn-1)\ (\forall R2,\ldots,RN(bn)\ ($$

$$Q\ \begin{vmatrix} P \\ P+1 \end{vmatrix}\ \begin{vmatrix} m \\ \alpha(m,P+1,bn \end{vmatrix}\ \begin{vmatrix} v \\ W(1-n) \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

$$\begin{vmatrix} P \\ P+1 \end{vmatrix}\ \begin{vmatrix} m \\ \alpha(m,P+1,bn-1 \end{vmatrix}\ \begin{vmatrix} v \\ W(2-n) \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

$$\ldots$$

$$\begin{vmatrix} P \\ P+1 \end{vmatrix}\ \begin{vmatrix} m \\ \alpha(m,P+1,b2 \end{vmatrix}\ \begin{vmatrix} v \\ W(-1) \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

$$\begin{vmatrix} P \\ P+1 \end{vmatrix}\ \begin{vmatrix} m \\ \alpha(m,P+1,b1 \end{vmatrix}\ \begin{vmatrix} v \\ W(0) \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

$$))\ \ldots\ ))$$

We now apply subrule 17b repeatedly to the for all's. By the same method as was used on the Pre and Post terms, we can consolidate the P+1 substitutions. This will result in all the W's being W(0), so we may use lower case w in place of them. The above part becomes:

$$\forall R2,\ldots,Rmax(N(b1),\ldots,N(bn))\ ($$

$$Q\ \begin{vmatrix} P \\ P+n \end{vmatrix}\ \begin{vmatrix} m \\ \alpha(m,P+n,bn \end{vmatrix}\ \begin{vmatrix} v \\ w \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

$$\begin{vmatrix} m \\ \alpha(m,P+n-1,bn-1 \end{vmatrix}\ \begin{vmatrix} v \\ w \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

$$\ldots$$

$$\begin{vmatrix} m \\ \alpha(m,P+2,b2 \end{vmatrix}\ \begin{vmatrix} v \\ w \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

$$\begin{vmatrix} m \\ \alpha(m,P+1,b1 \end{vmatrix}\ \begin{vmatrix} v \\ w \end{vmatrix}\ \begin{vmatrix} NIL \\ 0 \end{vmatrix}\ ))$$

Note that the series of substitutions

$$\begin{vmatrix} m \\ \alpha(m,P+i,bi \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

...

$$\begin{vmatrix} m \\ \alpha(m,P+1,b1 \end{vmatrix} \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

is equivalent to replacing m by an array in which the P+1st element has been replaced by

$$b1 \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix},$$

the P+2nd element by

$$b2 \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix},$$

... , and the P+ith element by

$$bi \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix}$$

For the sake of smaller formulas, we will call such an array $\beta(i)$. Using this notation we have the COMPLIS rule:

$$Pre(b1) \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix} \wedge (Post(b1) \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL \\ 0 \end{vmatrix} \rightarrow$$

$$Pre(b2) \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL & m \\ 0 & \beta(1) \end{vmatrix} \wedge (Post(b2) \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL & m \\ 0 & \beta(1) \end{vmatrix} \rightarrow$$

...

$$Pre(bn-1) \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL & m \\ 0 & \beta(n-2) \end{vmatrix} \wedge (Post(bn-1) \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL & m \\ 0 & \beta(n-2) \end{vmatrix} \rightarrow$$

$$Pre(bn) \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL & m \\ 0 & \beta(n-1) \end{vmatrix} \wedge (Post(bn) \begin{vmatrix} v \\ w \end{vmatrix} \begin{vmatrix} NIL & m \\ 0 & \beta(n-1) \end{vmatrix} \rightarrow$$

$$\forall R2,\ldots,Rmax(N(b1),\ldots,N(bn)) \; (\; Q \begin{vmatrix} P \\ P+n \end{vmatrix} \begin{vmatrix} m \\ \beta(n) \end{vmatrix})$$

)) ...))

{ FCOMPLIS(<b1 ... bn>,M,LOCTABLE) } Q

if R1 $\neg\epsilon$ Q.

A.8.13 MKPUSH

Here we derive a Hoare rule to describe the action of the instructions represented by FMKPUSH. From the code of MKPUSH we have:

```
FMKPUSH(n,m) = < < 'PUSH 'P m >
                  ! FMKPUSH(n,m+1)
               >
```

if n ≥ m, else an empty list is the result. Induction on m shows us that

```
FMKPUSH(n,m) = < < 'PUSH 'P m   >
                  < 'PUSH 'P m+1 >
                  ...
                  < 'PUSH 'P n   >
               >
```

for n ≥ m. For the case of m = 1, we have:

```
FMKPUSH(n,1) = < < 'PUSH 'P 1 >
                  < 'PUSH 'P 2 >
                  ...
                  < 'PUSH 'P n >
               >
```

for n ≥ 1. Using the Hoare rule for PUSH:

$$Q \begin{vmatrix} m \\ \alpha(m,P,Ri) \end{vmatrix} \begin{vmatrix} P \\ P+1 \end{vmatrix} \{ < \text{'PUSH 'P i} > \} Q$$

we obtain the effect of MKPUSH as:

$$Q \begin{vmatrix} m \\ \alpha(m,P,Rn) \end{vmatrix} \begin{vmatrix} P \\ P+1 \end{vmatrix} \begin{vmatrix} m \\ \alpha(m,P,Rn-1) \end{vmatrix} \begin{vmatrix} P \\ P+1 \end{vmatrix} \ldots \begin{vmatrix} m \\ \alpha(m,P,R1) \end{vmatrix} \begin{vmatrix} P \\ P+1 \end{vmatrix}$$

$$\{ \text{FMKPUSH}(n,1) \} Q$$

We may apply subrule 8 repeatedly and simplify arithmetically to get:

$$Q \begin{vmatrix} P \\ P+1 \end{vmatrix} \ldots \begin{vmatrix} P \\ P+1 \end{vmatrix} \begin{vmatrix} m \\ \alpha(m,P+n,Rn) \end{vmatrix} \ldots \begin{vmatrix} m \\ \alpha(m,P+1,R1) \end{vmatrix} \{ \text{FMKPUSH}(n,1) \} Q$$

We apply subrule 9 repeatedly and simplify arithmetically to get:

$$Q \begin{vmatrix} P \\ P+n \end{vmatrix} \begin{vmatrix} m \\ \alpha(m,P+n,Rn) \end{vmatrix} \ldots \begin{vmatrix} m \\ \alpha(m,P+1,R1) \end{vmatrix} \{ \text{FMKPUSH}(n,1) \} Q$$

In applying this rule, we will be doing so on a Q containing the term m[P-n+i], where i is a constant. The first substitution of the MKPUSH rule will produce m[P+i]. Now for all of the α substitutions made before α(m,P+i,Ri), we will be able to apply the simplification rule:

$$\alpha(m,j,x)[k] = m[k] \quad \text{if} \quad j \neq k$$

This gives m[P+i] still. The substitution where j = k uses the simplification rule:

$$\alpha(m,j,x)[j] = x$$

This gives Ri. Further substitutions beyond the ith α may be ignored by virtue of subrule 4. Thus applying the MKPUSH rule to m[P-n+i] gives Ri.

A.8.14 LOADAC

Here we derive a Hoare rule to describe the action of the instructions represented by FLOADAC. From the code of LOADAC we have:

```
FLOADAC(m,k) = < < 'MOVE k m 'P >
                  ! FLOADAC(m+1,k+1)
                >
```

if m ≤ 0, else an empty list is the result. Induction on m shows us that

```
FLOADAC(m,k) = < < 'MOVE k    m    'P >
                  < 'MOVE k+1 m+1  'P >
                  ...
                  < 'MOVE k-m 0    'P >
                >
```

if m ≤ 0. For the case where m = 1-n (n > 0) and k = 1 we have:

```
FLOADAC(1-n,1) = < < 'MOVE 1 1-n 'P >
                    < 'MOVE 2 2-n 'P >
                    ...
                    < 'MOVE n 0   'P >
                  >
```

Using the Hoare rule for MOVE:

$$Q \begin{vmatrix} Ri \\ m[P+j] \end{vmatrix} \{ \langle \text{'MOVE i j 'P} \rangle \} Q$$

we obtain the effect of LOADAC *as:*

```
   |Rn     |Rn-1    ...  |R1
Q  |m[P]   |m[P-1]  ...  |m[P-n+1]  { FLOADAC(1-n,1) } Q
```

The R's are obviously distinct from each other, and $\neg\epsilon$ the stack references (the m's). We may therefore apply subrule 7b to reverse the order of substitution. The FLOADAC rule is then:

```
   |R1          |R2          ...  |Rn
Q  |m[P-n+1]    |m[P-n+2]    ...  |m[P]  { FLOADAC(1-n,1) } Q
```

A.9 Example Equivalence Proof

The following is a proof of the semantic equivalence of the output produced by the compiler C0 and that produced by the compiler C4 for the source language syntactic case of a function call. C4 is an optimizing version of C0 which was proved by London [London71]. MC0, the modification of C0 that is in this dissertation, produces the same code for this case as C0. The equivalence proof is presented here to demonstrate one approach to proving compilers with optimizations. Those interested in the details of this proof are referred to the above-referenced report for the listing of C4, explanations of how it works, and the lemmas later referenced here. The terminology used here continues in the vein used in the MC0 proof, though the methods more closely approximate London's.

First we will write the result of C0 for a function call, as obtained during the MC0 part one proof. We will assume that the input is of form <f b1 b2 ... bn>.

```
FCOMPEXP(<f b1 b2 ... bn>,M,LOCTABLE) =
< ! FCOMPLIS(<b1 b2 ... bn>,M,LOCTABLE)
  ! FLOADAC(1-n,1)
  < 'SUB 'P < 'C 0 0 n n > >
  < 'CALL n < 'E f > >
>
```

We will now expand the FCOMPLIS and FLOADAC terms by the use of the forms we derived during the MC0 proof.

```
FCOMPEXP(<f b1 b2 ... bn>,M,LOCTABLE) =
< ! FCOMPEXP(b1,M,LOCTABLE)
  < 'PUSH 'P 1 >
  ! FCOMPEXP(b2,M,LOCTABLE)
  < 'PUSH 'P 1 >
  ...
  ! FCOMPEXP(bn,M,LOCTABLE)
  < 'PUSH 'P 1 >
  < 'MOVE 1 1-n 'P >
  < 'MOVE 2 2-n 'P >
  ...
  < 'MOVE n  0  'P >
  < 'SUB 'P < 'C 0 0 n n > >
  < 'CALL n < 'E f > >
>
```

We will now hand execute this code to determine its effects. The first item of code, the FCOMPEXP of b1, produces the value of argument b1 in register R1, at the possible expense of wiping out the other registers. We will represent this in London's trace notation, in which values are written to the right of target variable names, then "crossed out" by appending an asterisk when a new value takes its place. In this notation we have so far:

```
R1: b1
R2: undefined
...
RN: undefined
```

where N is the number of the last register available for target code use. We will now add the stack to our notation with a separate line for each location used in the stack. We now execute the first push to obtain:

```
R1: b1
R2: undefined
...
RN: undefined
stack: b1
```

After executing through the last push we have:

```
R1: b1* b2* ... bn
R2: undefined* undefined* ... undefined
...
RN: undefined* undefined* ... undefined
stack: b1
       b2
       ...
       bn
```

Executing the moves places the values in the stack into the registers, while the SUB removes n items from the stack, resulting in:

```
R1: b1* b2* ... bn* b1
R2: undefined* undefined* ... undefined* b2
...
Rn: undefined* undefined* ... undefined* bn
...
RN: undefined* undefined* ... undefined
stack:
```

Because the result of compiling on C4 ends with the identical call statement, we will not execute the call. We will now derive the form of the C4 result and execute it (except for the call) to show that the same state exists in the target machine as after executing the C0 results. This is the sense in which we consider the results semantically equivalent.

It is obvious from the code of C4 that its output for this case is:

```
FCOMPEXP(<f b1 b2 ... bn>,M,LOCTABLE) =
< ! FCOMPLISA(<b1 b2 ... bn>,M,LOCTABLE)
  < 'CALL n < 'E f > >
>
```

Because COMPEXP, the main compiling routine, has been redefined in C4, FCOMPEXP is of course redefined from that of C0. The same will be true of COMPLIS and LOADAC, so we must rederive forms output by those routines.

We may obtain an expression for COMPLISA directly from its code, and expand FCOMPLISA to obtain:

```
FCOMPEXP(<f b1 b2 ... bn>,M,LOCTABLE) =
< ! FCOMPLIS(CLASSIFY(<b1 b2 ... bn>),M,1,LOCTABLE)
  ! FLOADAC(CLASSIFY(<b1 b2 ... bn>),
            1-CCOUNT(CLASSIFY(<b1 b2 ... bn>)),
            1,
            M-CCOUNT(CLASSIFY(<b1 b2 ... bn>)),
            LOCTABLE)
  ! SUBSTACK(CCOUNT(CLASSIFY(<b1 b2 ... bn>)))
  < 'CALL n < 'E f > >
>
```

We know by London's Lemma 7 that classify(<b1 b2 ... bn>) = < <d1 . b1> <d2 . b2> ... <dn . bn> >, where di is given by the syntactic type of the corresponding bi according to this table:

di	bi type
0	T, NIL, numeric atom
1	other atom
2	quoted expression
3	CAR-CDR chain ending in an atom
4	other expression (except last)
5	last other expression

By Lemma 8 we know that CCOUNT(< <d1 . b1> <d1 . b2> ... <dn . bn> >) is the number of di's that are 4, which we shall call Z4. Let e be an array in which we place the subscripts of the d's that are 4. Then de[1] is the first 4, de[2] is the second, and de[Z4] is the last (if any d's are 4). *Note that the notation de[1] means di where i=e[1].* Let Z5 be 0 if no d's are 5, else Z4+1. Then let e[Z5] be the subscript of the d that is 5 (if it exists.) We may expand SUBSTACK by use of Lemma 9. With this terminology, the C4 result of compiling a function call is:

```
FCOMPEXP(<f b1 b2 ... bn>,M,LOCTABLE) =
< ! FCOMPLIS(< <d1 . b1> <d2 . b2> ... <dn . bn> >,M,1,LOCTABLE)
  ! FLOADAC(< <d1 . b1> <d2 . b2> ... <dn . bn> >,
            1-Z4,
            1,
            M-Z4,
            LOCTABLE)
  < 'SUB 'P < 'C 0 0 Z4 Z4 > >
  < 'CALL n < 'E f > >
>
```

unless Z4 is zero, in which case the SUB is omitted.

We must now derive expressions for FCOMPLIS and FLOADAC. From the code of COMPLIS we have:

```
FCOMPLIS(< <dK . bK> <dK+1 . bK+1> ... <dn . bn> >,M,K,LOCTABLE) =
  If null first argument then < >
  Else if dK = 4 then < ! FCOMPEXP(bK,M,LOCTABLE)
                        < 'PUSH 'P 1 >
                        ! FCOMPLIS(< <dK+1 . bK+1> ... <dn . bn> >,
                                   M-1,K+1,LOCTABLE)
                      >
  Else if dK = 5 then < ! FCOMPEXP(bK,M,LOCTABLE)
                        < 'MOVE K 1 >
                      >
  Else < ! FCOMPLIS(< <dK+1 . bK+1> ... <dn . bn> >,M,K+1,LOCTABLE) >
```

unless K is 1, in which case the move is omitted.

Theorem A: With the terminology given above, FCOMPLIS is given by:

```
FCOMPLIS(< <dK . bK> <dK+1 . bK+1> ... <dn . bn> >,M,K,LOCTABLE) =
< ! FCOMPEXP(be[1],M,LOCTABLE)
  < 'PUSH 'P 1 >
  ! FCOMPEXP(be[2],M-1,LOCTABLE)
  < 'PUSH 'P 1 >
  ...
  ! FCOMPEXP(be[Z4],M-Z4+1,LOCTABLE)
  < 'PUSH 'P 1 >
  ! FCOMPEXP(be[Z5],M-Z4,LOCTABLE)
  < 'MOVE e[Z5] 1 >
>
```

unless Z5 is 1, in which case the move is omitted. Of course if Z4 is zero, the first 2*Z4 terms are vacuous, and if Z5 is zero, the entire expression is vacuous.

The proof of theorem A is a straightforward application of induction on the structure of the first argument of COMPLIS. It is proved by the cases reflected in the code of COMPLIS. It might be pointed out that e[1] must mark the first occurrence of a type 4 in the range from K to n, not 1 to n.

The way in which LOADAC optimizes target code is that it generates all arguments possible in the register that they are to occupy at the time the function is called. The restriction is that the argument must not disturb other registers, since they may already have other arguments in them. It is exactly the types which have d's of 3 or less that may be generated with the use of only one register. Thus LOADAC will generate exactly the same code for each of these types that a recursion to the main compiling routine would except that the register to be used is changed according to which argument we are compiling. We will therefore define a new function to describe the code output by LOADAC for these types. We will call it KFCOMPEXP, and define it as follows.

```
KFCOMPEXP(K,EXP,M,LOCTABLE) =
  If DTYPE(EXP) = 1 then < <'MOVE K M+CDR(ASSOC(EXP,LOCTABLE)) 'P> >
  Else if DTYPE(EXP) = 0 then < <'MOVEI K <'QUOTE EXP>> >
  Else if DTYPE(EXP) = 2 then < <'MOVEI K EXP> >
  Else if DTYPE(EXP) = 3 then < ! REVERSE(COMPC(EXP,K,M,LOCTABLE)) >
```

where DTYPE(EXP) is the value of d that would result from applying classify to a list containing EXP.

Lemma A: KFCOMPEXP gives a result that is semantically equivalent to FCOMPEXP over the types for which the former is defined, except that KFCOMPEXP uses only register k instead of only register 1.

The proof is by source language syntactic type cases. For ISNIL(EXP) we have DTYPE(EXP) = 0. KFCOMPEXP and FCOMPEXP are given by (respectively):

```
< <'MOVEI K <'QUOTE EXP>> >

< <'MOVEI 1 0> >
```

Because 0 and NIL (or quote NIL) are treated the same in the target language, this case is proved. For IST(EXP) we have DTYPE(EXP) = 0. KFCOMPEXP is the same as the previous case, while FCOMPEXP is:

```
< <'MOVEI 1 <'QUOTE EXP>> >
```

which is the same except for the register. For ISNUMBER(EXP) we have DTYPE(EXP) = 0. KFCOMPEXP and FCOMPEXP are both the same as in the previous case. For ISIDENTIFIER(EXP) we have DTYPE(EXP) = 1. The results are:

```
< <'MOVE K M+CDR(ASSOC(EXP,LOCTABLE)) 'P> >

< <'MOVE 1 M+CDR(ASSOC(EXP,LOCTABLE)) 'P> >
```

For ISQUOTE(EXP) we have DTYPE(EXP) = 2. The results are:

```
< <'MOVEI K EXP> >

< <'MOVEI 1 EXP> >
```

For ISCARORCDR(EXP) we have DTYPE(EXP) = 3 if EXP is a CAR-CDR chain ending in an atom, but DTYPE(EXP) > 3 otherwise. We may ignore cases where DTYPE(EXP) > 3 because KFCOMPEXP is not defined. For a CAR-CDR chain of form (Cβ1R (Cβ2R ...(CβNR α)...)) with each βi either an A or a D (to form CAR or CDR), we may use the expansion of Theorem 9 to get for KFCOMPEXP:

```
< <'HєNRZe K M+CDR(ASSOC(α,LOCTABLE)) 'P>
  <'HєN-1RZe K K>
  ...
  <'Hє2RZe K K>
  <'Hє1RZe K K>
>
```

where ϵi is L if βi is A and is R if βi is D (to form HLRZe or HRRZe). Examination of the code of C4 and a simple induction on the depth of nesting gives us the following expression for FCOMPEXP.

```
< <'HєNRZe 1 M+CDR(ASSOC(α,LOCTABLE)) 'P>
  <'HєN-1RZe 1 1>
  ...
  <'Hє2RZe 1 1>
  <'Hє1RZe 1 1>
>
```

Since all other syntactic types have DTYPE > 3, we have proved the lemma.

We now derive the expression for FLOADAC. From the code of LOADAC we have:

```
FLOADAC(< <dN2 . bN2> <dN2+1 . bN2+1> ... <dn . bn> >,
        M2,N2,M,LOCTABLE) =
  If null first argument then < >
  Else if dN2 < 3 then < ! KFCOMPEXP(N2,bN2,M,LOCTABLE)
                         ! FLOADAC(< <dN2+1 . bN2+1> ... <dn . bn> >,
                                   M2,N2+1,M,LOCTABLE)
                       >
  Else if dN2 = 5 then < ! FLOADAC(< <dN2+1 . bN2+1> ... <dn . bn> >,
                                   1,N2+1,M,LOCTABLE)
                       >
  Else < <'MOVE N2 M2 'P>
         ! FLOADAC(< <dN2+1 . bN2+1> ... <dn . bn> >,
                    M2+1,N2+1,M,LOCTABLE)
       >
```

Theorem B: With the terminology given above, FLOADAC is given by:

```
FLOADAC(< <dN2 . bN2> <dN2+1 . bN2+1> ... <dn . bn> >,
        M2,N2,M,LOCTABLE) =
< ! KFCOMPEXP(N2,bN2,M,LOCTABLE)
  ! KFCOMPEXP(N2+1,bN2+1,M,LOCTABLE)
  ...
  <'MOVE e[1] M2 'P>
  ...
  <'MOVE e[2] M2+1 'P>
  ...
  <'MOVE e[Z4] M2+Z4-1 'P>
  ...
  ! KFCOMPEXP(n,bn,M,LOCTABLE)
>
```

with the e[Z5] line (if any) missing. The interpretation of this is that the KFCOMPEXP forms occur for every value of N2 up to n except for the values e[1] ... e[Z4], at which the move form occurs instead. It should be understood that e[1] must mark the first occurrence of a type 4 in the range from N2 to n.

The proof of theorem B is a straightforward application of induction on the structure of the first argument of COMPLIS. It is proved by the cases reflected in the code of COMPLIS.

We may now substitute these forms into our expression for the result of compiling a function with C4. The result is:

```
FCOMPEXP(<f b1 b2 ... bn>,M,LOCTABLE) =
< ! FCOMPEXP(be[1],M,LOCTABLE)
  < 'PUSH 'P 1 >
  ! FCOMPEXP(be[2],M-1,LOCTABLE)
  < 'PUSH 'P 1 >
  ...
  ! FCOMPEXP(be[Z4],M-Z4+1,LOCTABLE)
  < 'PUSH 'P 1 >
  ! FCOMPEXP(be[z5],M-Z4,LOCTABLE)
  < 'MOVE e[Z5] 1 >
  ! KFCOMPEXP(1,b1,M-Z4,LOCTABLE)
  ! KFCOMPEXP(2,b2,M-Z4,LOCTABLE)
  ...
```

```
  < 'MOVE e[1] 1-Z4 'P >
  ...
  < 'MOVE e[2] 2-Z4 'P >
  ...
  < 'MOVE e[Z4] 0 'P >
  ...
  ! KFCOMPEXP(n,bn,M-Z4,LOCTABLE)
  < 'SUB 'P < 'C 0 0 Z4 Z4 > >
  < 'CALL n < 'E f > >
>
```

with the e[Z5] line missing from the kfcompexps, and if Z5 = 1 then the first move is omitted. The e's include all type 4s between 1 and n.

We will now execute this code (again, except for the final call) to determine if it has the same final values in the trace as the C0 code did. The trace just before executing the SUB instruction is:

```
R1: be[1]* be[2]* ... be[Z4]* be[Z5]* b1
R2: undefined* undefined* ... undefined* undefined* b2
...
Re[1]: undefined* undefined* ... undefined* undefined* be[1]
...
Re[2]: undefined* undefined* ... undefined* undefined* be[2]
...
Re[Z4]: undefined* undefined* ... undefined* undefined* be[Z4]
...
Re[Z5]: undefined* undefined* ... undefined* undefined* be[Z5]
...
Rn: undefined* undefined* ... undefined* undefined* bn
...
RN: undefined* undefined* ... undefined* undefined
stack: be[1]
       be[2]
       ...
       be[Z4]
```

The SUB instruction will remove Z4 items from the stack and will leave the machine state exactly as we saw for the C0 code.

A.10 Subrule Justification

In this section we will justify some of the sequential substitution simplification rules (subrules) by use of a few basic principles, and give the vein in which the proof of the others may be approached. The subrules are given in Section 3.6 in the forms found to be useful during the part two proofs.

The definition of the notation

$$Q\Big|_{H}^{D}$$

is the substitution of the expression H for all free occurrences of the identifier D in the formula Q, with the caution that H must not have free uses of any identifier that become bound when introduced into Q. As in the previous discussion of substitution, all D's and X's, numbered or not, will represent atomic identifiers.

Towards the goal of simplifying such substitutions, we wish to define D not-in A (denoted $D \neg\epsilon A$) as meaning that there are no free uses of the identifier D in the expression A. The complication that arises is that in all the part two proofs we express expressions such as A symbolically in terms of the compiler variables, not in the source or target language terms that we would wish to use in A. Thus we may not inspect A to see if indeed a target or source language variable (identifier) appears in A, but must use the subrules to prove that it does or does not.

The basic principle we will use to prove the not-in subrules is that an identifier D is not-in the whole if it is not-in the parts of a target language or source language expression. Formally stated, this is:

$$D \neg\epsilon Gi \ (for\ 1 \leq i \leq n) \rightarrow D \neg\epsilon (f\ G1 \ldots Gn)$$

for all source language functions f. Those familiar with Lisp will recognize the (f G1 ... Gn) as the source language (of MC0) notation for a function call. Since there is no translation of function names between source and target language, f is also a target language function. This is subrule 13b, and it forms the basis for proving most of the other not-in subrules. The converse of subrule 13b is also true and will occasionally be used under the name subrule 13a.

We should note that subrule 13b precludes the possibility that D can somehow be created by the combination of items that do not individually contain D. This is why D must be an atomic identifier, not an expression. Because we will use not-in frequently in our substitution rules, we require that the object which we substitute for is also atomic.

If H1 represents a target language or source language expression that is atomic, then the determination of $D \neg\epsilon H$ is equivalent to the observation that $D \neq H$. This follows directly from the definition of not-in, and may be expressed by:

$$D1 \neq D2 \rightarrow D1 \neg\epsilon D2$$

$$D1 \neg\epsilon D2 \rightarrow D1 \neq D2$$

The first is called subrule 11, and the second (its converse) will be called subrule 11b. It must be understood that a statement such as $P \neq A$ does not mean that P does not have the same value as A, but that the identifier P is not the identifier A for purposes of deciding if a substitution involving P need be made on an expression using the identifier A. The statement $P \neq A$ is the equivalent of the Lisp expression (NOT (EQ 'P 'A)).

Subrule 11 is usually applied to identifiers that are proved unequal by subrule 2, which is simply a recognition of the fact that source language variables are distinct objects from target language ones, and that various classes of target language variables are distinct from each other.

Before we proceed, we need to investigate quantification. It is introduced into the source language only by the assertions, but is also introduced into the resulting verification conditions by Hoare rules with quantification in them. By the definition of not-in, we can see that if an identifier is not-in an expression, it is not-in (as a free use, recall) the same expression but with an identifier bound by quantification. Formally expressed, we have:

$$D \neq X \vee D \neg\in G \rightarrow D \neg\in \forall X\ (G)$$

and

$$D \neq X \vee D \neg\in G \rightarrow D \neg\in \exists X\ (G)$$

These are the parallels of subrule 13b, but for quantification instead of functions. We will call them axioms A1 and A2, respectively. Note that the converse of the quantification rules is not true without further conditions. The converses with conditions will be called axioms B1 and B2, and may be expressed as:

$$D \neg\in \forall X\ (G) \wedge D \neq X \rightarrow D \neg\in G$$

$$D \neg\in \exists X\ (G) \wedge D \neq X \rightarrow D \neg\in G$$

We can now prove subrule 1b. It states that P (a specific target language variable) is not-in any source language expression. All source or target language expressions are built of nested function calls and quantification. The proof is then one of induction on the depth of nesting. The base step is established by using subrule 2b (P $\neq$ any source variable) and subrule 11. It might be pointed out that for purposes of this discussion we may treat source constants as if they were source variables. That is, we may use subrule 2b to show P $\neq$ 2, where 2 is a source language constant. The induction step simply requires the use of subrule 13b and the corresponding rules for quantification.

Exactly the same proof holds for subrules 1a and 1c, since they also involve showing that certain target language variables are not-in source language expressions. The proof of subrule 1d is the same except that the roles of target and source language have been reversed.

We will now return to substitution. The basic axiom we will assume about substitution is similar to the one for not-in. It may be characterized by stating that substitution distributes over the arguments of source language functions, and is expressed formally by:

$$(f\ G1\ \ldots\ Gn)\ \Big|_{H}^{D} = (f\ G1\ \Big|_{H}^{D}\ \ldots\ Gn\ \Big|_{H}^{D})$$

where f is any source language function. This is subrule 6.

By referring to the definition of substitution we can state the effects of applying a substitution to an identifier. There are two cases: one when it is the identifier to be substituted for, and the other when it is not.

$$D\ \Big|_{H}^{D} = H$$

$$D \neq D1 \rightarrow D1 \Big|_{H = D1}^{D}$$

The former is subrule 21, and the latter we shall call axiom C.

We will also need to give axioms for the interface between substitution and quantification. These may be stated in words as: substituting for the free occurrences of an identifier that is quantified has no effect, and substitutions may be done inside quantification if we do not substitute for the quantified identifier nor introduce a use of the quantified identifier that gets bound. The axioms, along with their names, are:

$$19: \forall X\ (G) \Big|_{H = \forall X\ (G)}^{X}$$

$$19b: \exists X\ (G) \Big|_{H = \exists X\ (G)}^{X}$$

$$18a: X \neq D \wedge X \neg\in H \rightarrow \forall X\ (G) \Big|_{H = \forall X\ (G\ \Big|_{H}^{D})}^{D}$$

$$18a2: X \neq D \wedge X \neg\in H \rightarrow \exists X\ (G) \Big|_{H = \exists X\ (G\ \Big|_{H}^{D})}^{D}$$

Subrule 18b is simply a generalization of 18a for use in cases having multiple substitutions or quantifications. It is easily derived from subrule 18a by induction on the number of quantifications and induction on the number of substitutions.

By the definition of substitution we are not allowed to introduce by means of substitution a free use of an identifier into a place where it becomes bound. The following axioms characterize this as giving an undefined result.

$$D1: X \neq D \wedge X \in H \rightarrow \forall X\ (G) \Big|_{H = \forall X\ (G\ \Big|_{\text{undefined}}^{D})}^{D}$$

$$D2: X \neq D \wedge X \in H \rightarrow \exists X\ (G) \Big|_{H = \exists X\ (G\ \Big|_{\text{undefined}}^{D})}^{D}$$

We will now prove subrule 4:

$$D \neg\in G \rightarrow G \Big|_{H = G}^{D}$$

By repeated distribution of the D-H substitution by subrule 6 we get an expression equal to G with the D-H substitution that has that substitution done on each atomic identifier, except in the case of identifiers within quantified expressions, the substitution is applied to the outermost quantified expression. For example, if G were (F1 A (F2 B $\forall$X ((F3 X C)))), the result of the D-H substitution upon G would become:

$$(F1\ A \begin{vmatrix} D \\ H \end{vmatrix} (F2\ B \begin{vmatrix} D \\ H \end{vmatrix} \forall X\ ((F3\ X\ C)) \begin{vmatrix} D \\ H \end{vmatrix}))$$

Now we repeatedly distribute D not-in G as far as possible into G by subrule 13a. Then we apply subrule 11b at all the atomic identifiers to which we have pushed the not-in. This allows us to use axiom C and drop all the D-H substitutions on the atomic identifiers. But we still have the D-H substitutions at all outermost quantifications. Let the quantified identifier under discussion be denoted X. Now two cases arise: either D = X, or D ≠ X. If D = X, apply subrule 19 or 19b (depending on which kind of quantification it is) to discard the substitution. If D ≠ X and X $\neg\epsilon$ H, we apply subrule 18a or 18a2. If D ≠ X but X $\neg\epsilon$ H, use D1 or D2 to move the substitution inside the quantification. We may also apply axioms B1 or B2 to move the not-in inside the quantification. By these means we will eventually move the substitution and the not-in down in all cases to either an atomic identifier upon which axiom C applies or a quantification upon which subrules 19 or 19b apply. In all cases the substitution will be discarded, resulting in the original form of G being left. Hence subrule 4 is proved.

We will now prove subrule 3a:

$$D\ \neg\epsilon\ G \wedge D\ \neg\epsilon\ H1 \rightarrow D\ \neg\epsilon\ G \begin{vmatrix} D1 \\ H1 \end{vmatrix}$$

In a manner similar to the proof of subrule 4 we will repeatedly distribute the D1-H1 substitution by subrule 6, and the D not-in G by subrule 13a. When the substitution and not-in distributions are forced as far as they can go, the object on which the substitution is performed is one of four cases. If the object is an atomic identifier, we will call it A. If it is a quantified expression, we will call the quantified identifier X. Either 1) X ≠ D1 and X ≠ D, 2) X ≠ D1 and X = D, 3) X = D1, or 4) it is atomic. In case 1) we may apply subrule 18a or 18a2 or axiom D1 or D2 according to whether X $\neg\epsilon$ H1 and according to the type of quantification. In applying any of the four rules we will move the substitution inside the quantification. We can apply B1 or B2 to move the D not-in to the inside of the quantification, and then continue to distribute the substitution and not-in. In case 2) we will apply axiom A1 or A2 to establish that D is not-in the quantified object with the substitution applied inside of it. But that is equal to the quantified object with the substitution outside by 18a, 18a2, D1, or D2. In case 3) we have established that d is not-in the quantified object without the substitution applied, but that is equal to the quantified object with the substitution applied by subrule 19 or 19b. In case 4) we have reached an atom that is not inside D or D1 quantification. However we may have applied axiom D1 or D2 sometime so that the substitution of D1-H1 may have become D1-undefined. In any case the result of the substitution will be A or H1 or undefined. We have distributed the not-in to show that D $\neg\epsilon$ A, a premise of the theorem is D $\neg\epsilon$ H1, and D $\neg\epsilon$ undefined. Therefore D is not-in the result of the substitution.

In all of the cases where the substitution stops propagating we have shown that D is not-in the substituted result. Therefore repeated application of subrule 13b and axioms A1 and A2 from all the parts of the substituted G back up to the whole will prove the conclusion of subrule 3a.

Subrule 3b is simply a generalization of 3a for the case of multiple substitution. It is easily proved by induction on the number of substitutions.

The proofs of subrules 5, 7, 8, 9, 12, 14, and 20 should follow in the same mold as those given here for 3 and 4 (though they have not been carried out). They all deal with properties of one, two, or three sequential substitutions, except the usual generalization to n substitutions in 7.

We now return to quantification for a moment. Subrule 16 gives three variations on moving irrelevant items outside the scope of quantification. We will take these properties as axioms. The corresponding axioms hold for existential quantification, but we have not encountered a need for them as subrules. Subrule 17a is a simple consequence of subrule 16b and axiom A1. Again we have a generalization in 17b for the case of multiple quantifications.