TR-545 May 1977

A STUDY IN AUTOMATIC DEBUGGING OF COMPILERS

BY

Hanan Samet
Computer Science Department
University of Maryland
College Park, Maryland 20742

Abstract

Automatic debugging is examined in the context of compiler correctness. A system is described whose goal is to prove the correctness of translations involving heuristically optimized code. Use of the system in automatically pinpointing errors is demonstrated along with a discussion of the prospects of automatic debugging of a complex program that was incorrectly translated. The actual debugging procedure is seen to take several iterations at the end of which a correctly translated program is obtained.

Keywords and phrases: automatic programming, debugging, compilers, error detection, error correction, program verification, self repairing software

*This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-73-C-0435. The views expressed are those of the author.

A STUDY IN AUTOMATIC DEBUGGING OF COMPILERS*

by

Hanan Samet
Computer Science Department
University of Maryland
College Park, Maryland 20742

Abstract

Automatic debugging is examined in the context of compiler correctness. A system is described whose goal is to prove the correctness of translations involving heuristically optimized code. Use of the system in automatically pinpointing errors is demonstrated along with a discussion of the prospects of automatically correcting them. The viability of the approach is illustrated by the automatic debugging of a complex program that was incorrectly translated. The actual debugging procedure is seen to take several iterations at the end of which a correctly translated program is obtained.

Keywords and phrases: automatic programming, debugging, compilers, error detection, error correction, program verification, self repairing software

*This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-73-C-0435. The views expressed are those of the author.

1. INTRODUCTION

A significant amount of current research in automatic programming is devoted to the construction of more efficient programs. Researchers have basically taken two approaches to this problem. At one end, work is proceeding towards the automatic development of programs from task specifications. These efforts range from the automatic construction of manipulator programs [Taylor76] to more conventional programming tasks such as sorting [Green76]. The latter is driven by dialogues which explain the desired task. Other work makes use of examples [Summers75]. At the other end, progress is being made at rendering existing programs more efficient. Such work is typified by studies such as [Low74] which aim at automatically selecting the data structure thereby relieving the programmer from worrying about such issues. In the middle of the spectrum lies work in program verification [Waldinger69] and debugging [Sussman75].

Our work lies in the middle of the above spectrum. We describe the use of a compiler testing system [Samet75] in detecting errors in heuristically optimized code as well as the prospects for automatically correcting them. This work is motivated by the realization that often there is no a priori knowledge of how certain computer programs are to be optimized. In such a case, there may be a need to resort to heuristics. Such a paradigm embodies "hypothesis and test" techniques [Newell73] thereby necessitating a mechanism for verifying that the various attempts at optimization do indeed function properly. Currently, a system exists [Samet75] which proves that programs are correctly translated as well as pinpoints the mistakes in erroneous translations. The goal of our presentation is to illustrate the errors that can be detected and to demonstrate that often the error information is sufficient to indicate the necessary correction. Thus it will be shown that a significant number of errors, in addition to being detected, could also be corrected automatically.

Such an error detection and correction capability is attractive in the context of self repairing software. In many artificial intelligence applications, programs write other programs which they later execute. In such a case, efficiency considerations may lead to the invocation of a compiler to translate the newly created program. Use of techniques presented here can lead to a greater degree of reliability of compilers used in such an environment. In particular, if one is operating in a hostile environment, say Mars, then it would be difficult to debug a program such as a compiler from Earth. Thus just as self checking circuits find usefulness in hardware, we feel that at times a need exists for their software analogs, self repairing programs.

This paper is organized into several sections. First, we present a brief overview of the concept of compiler testing. This is followed by a short example to illustrate the type of programs our system can handle. Next, we discuss the error detection capabilities of the system. Finally, an erroneously encoded complex example is given and the reader is led through the errors that the system discovered as well as the necessary corrections. Often, the actual corrections are quite straightforward thereby justifying a conclusion that automatic error correction is feasible in a large number of situations.

2. COMPILER TESTING

Compiler testing is a term we use to describe a means of proving that given a compiler (or any translation procedure) and a program to be compiled, the translation has been correctly performed. The actual test consists of demonstrating a correspondence or equivalence between a program input to the compiler and the corresponding translated program. By equivalence we mean that the two programs must be capable of being proved to be structurally equivalent [Lee72], that is they have identical execution sequences except for certain valid rearrangements of computations. Note that this is a more stringent requirement than that posed by the conventional definition which holds that two programs are equivalent if they have a common domain and range and both produce the same output for any given input in their common domain. For example, using our techniques, we cannot prove that a high level insertion sort program is equivalent to a low level quicksort program.

The actual testing procedure relies on the existence of an intermediate representation common to both the source and object programs. This representation reflects all of the computations performed on all possible execution paths. Given the existence of such a representation, the testing procedure consists of three steps (see fig. 1). First, the high level language program is converted to the intermediate representation via the use of a suitable set of syntactic transformations. Second, the low level program must be converted to the intermediate representation. This is achieved by use of a process termed symbolic interpretation [Samet76] which interprets procedural descriptions of low level machine operations to build the intermediate representation. Third, a check must be performed of the equivalence of the two representations. This check is in the form of a procedure which applies equivalence preserving transformations to the results of the first two steps in attempting to reduce them to a common representation.

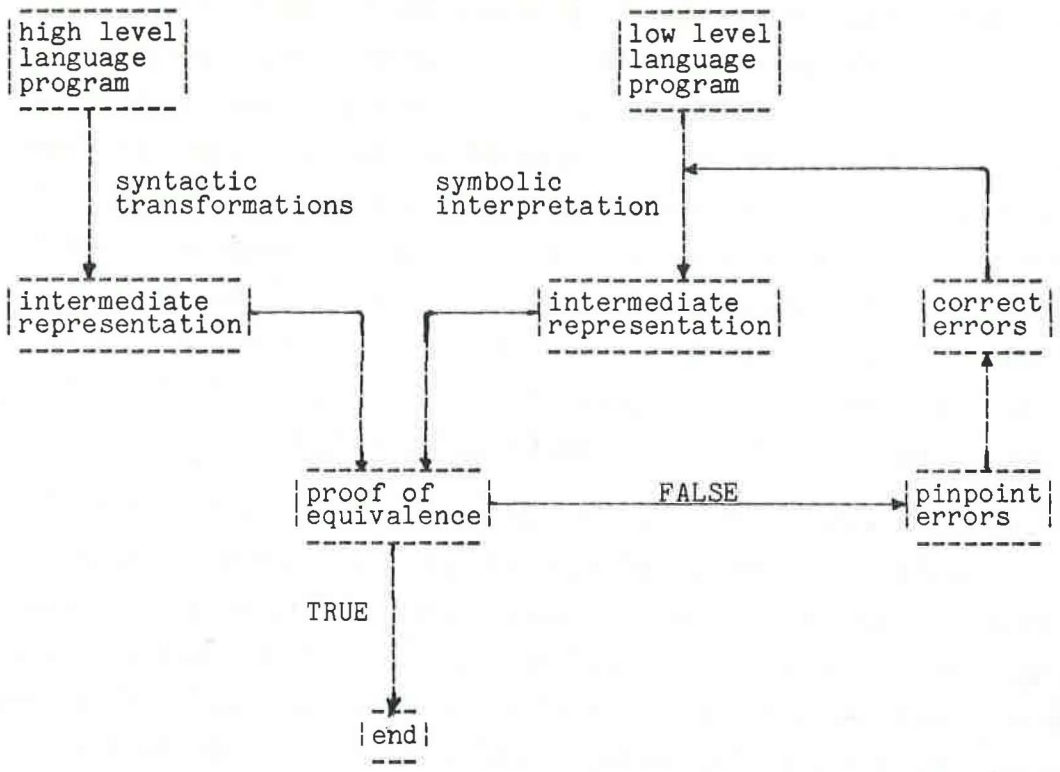

Fig. 1 - Compiler testing system diagram

In this paper we are primarily concerned with the error detection capabilities of such a technique and the implications it has for error correction. To this end we need a sample system. We use a subset of LISP 1.6 [Quam72] (a variant of LISP [McCarthy60]) as the high level language and LAP [Quam72] (a variant of the PDP-10 [DEC69] assembly language) as the low level language. A suitable intermediate representation for our subset of LISP in the form of a tree is shown to exist in [Samet77].

An an example, consider fig. 2 where a function, REVERSE, which reverses the links of a list, is encoded in MLISP [Smith70], an ALGOL-like [Naur60] version of LISP.

```
REVERSE(L) = if NULL(L) then L
             else *APPEND(REVERSE(CDR(L)),LIST(CAR(L)))
```

Fig. 2 - Definition of REVERSE

Prior to presenting a LAP encoding we describe our execution environment. A LISP cell is represented by a full word whose left and right halves point to CAR and CDR respectively. Addresses of atoms are represented by (QUOTE <atom-name>) and by zero in the case of NIL. A stack is used for control with accumulator 12 containing a stack pointer, and upon function entry the return address is found on the top of

the stack. A LAP program expects to find all of its arguments in the accumulators and returns its result in accumulator 1. The accumulators containing the parameters are always of such a form that a 0 is in the left half and the LISP pointer is in the right half. All parameters are assumed to be valid LISP pointers. Whenever recursion or a call to an external function occur, the contents of all of the accumulators (except 12) are assumed to be destroyed with the exception of CONS, XCONS, and NCONS, in which case all accumulators but 1 in the case of NCONS, and 1 and 2 in the case of CONS and XCONS, have the same values before and after the call. XCONS is the antisymmetric counterpart of CONS - i.e., CONS(A,B) = XCONS(B,A) while NCONS obeys the relation NCONS(A) = CONS(A,NIL) = LIST(A).

Fig. 3 contains a LAP encoding for the function given in fig. 2. The format of a LAP instruction is (OPCODE AC ADDR INDEX) where INDEX and ADDR are optional. OPCODE is a PDP-10 instruction optionally suffixed by @ which denotes indirect addressing. ADDR denotes the address field. AC and INDEX denote respectively the accumulator associated with the instruction and the accumulator to be used in case of indexing. These two fields contain a number between 0 and decimal 15. (CALL 1 (E NCONS)) denotes that NCONS is a recursive function of type EXPR (call by value) and is called with one argument. Similarly, JCALL corresponds to a non-recursive function call. JCALL is used to invoke *APPEND since once this function is exited nothing remains to be computed in REVERSE.

```
REVERSE (SKIPN 2 1)            load acc. 2 with L and skip if not NIL
        (POPJ 12)              return NIL
        (HLRZ 1 0 1)           load acc. 1 with CAR(L)
        (CALL 1 (E NCONS))     compute LIST(CAR(L))
        (PUSH 12 1)            push LIST(CAR(L)) on the stack
        (HRRZ 1 0 2)           load acc. 1 with CDR(L)
        (CALL 1 (E REVERSE))   compute REVERSE(CDR(L))
        (POP 12 2)             pop LIST(CAR(L)) from the stack
PC9     (JCALL 2 (E *APPEND))  compute *APPEND(REVERSE(CDR(L)),LIST(CAR(L)))
```

Fig. 3 - LAP encoding of REVERSE

The intermediate representation obtained by the symbolic interpretation procedure is given in fig. 4. Notice that we have a symbolic representation and a numeric representation. The numbers in the latter are unique to each computation and execution path and their purpose is to indicate a relative ordering for the sequence of computations. The numbers are used in a proof to enable us to prove that equivalence is preserved when certain functions are computed out of order. However, these numbers can also be used profitably in the process of error detection. Since the numbers are unique to each computation and execution path, we may determine from each computation where in the program it was computed and thus pinpoint the error. This is accomplished by maintaining a dictionary of computation

numbers where with each entry is stored an instruction address and the labels that were encountered along the execution path starting at function entry.

Fig. 4 - Intermediate representation of fig. 3

3. ERRORS

Errors in the translated program that are caused by the translation process can be detected. This is accomplished, in part, with the aid of the computation dictionary mentioned in the previous section. There are basically four classes of errors. Errors of the first class are detected by the symbolic interpretation procedure while the remaining three classes are detected during the proof procedure as computations are being matched in the two intermediate representations. Errors detected during the symbolic interpretation phase pertain to the well-formedness of the object program - i.e., violations of the rules set forth in the definition of the execution environment. Errors detected during the proof procedure are often the result of computations occuring in one intermediate representation and not in the other.

(1) Errors pertaining to the well-formedness of the program include improper calling sequences, illegal stack pointer formats, illegal operations on certain high level data structures, etc. For example, performing arithmetic on LISP pointers and possibly attempting to pass the result to another LISP function. Using a calling sequence which combines or replaces an accumulator with a stack location incorrectly. Storing data in locations which are off limits - i.e., certain accumulators and even unknown addresses. The stack also serves as a source of error due to confusion as to the status of the stack pointer. All of these errors are detected during the symbolic interpretation phase. Whenever such an error is encountered, the current execution path is abandoned and symbolic interpretation is continued on an alternate path so that a maximal number of errors can be detected.

(2) All of the computations in one of the intermediate representations were found to exist in the other representation, but the reverse is not true. Such an error may occur when certain side effect computations occur in one of the programs and not in the other. Alternatively, this may also occur when certain tests are performed in one program and not in the other.

(3) There are occasions when each of the intermediate representations reflects the performance of the same computations along each execution path, yet, the two representations are not identical. This occurs when the results of the execution paths are different. For example, consider the two representations given below. Notice that all computations performed on the left are also performed on the right. However, the results of the two right subtrees are not equivalent (i.e., (CDR A) is not equivalent to (CDR B)).

(4) The actual proof procedure may reach a point at which it cannot continue. This is the case when a function in the intermediate representation of the low level program can not be matched with a function in the intermediate representation corresponding to the original high level program. This is caused by such factors as invalid rearranging of computations, mistakes in the object program, invalid optimizations, etc. Some of the errors of this class that have been detected (see section 4) include use of wrong accumulators, misuse of antisymmetry, misspelling of operation codes and operands thereby causing the wrong instruction to be executed, and testing the wrong sense of a condition.

When errors of type (1)-(3) occur, the system will return a message indicating the error type. We also indicate the erroneous computation (somewhat meaningless for type (1) errors) as well as what should have been computed according to the intermediate representation corresponding to the original high level program. In addition, the values of the conditions in terms of truth values are given so that the offending path can be identified.

We are primarily interested in errors of type (4). When such errors occur, the system returns the invalid computation along with the computation dictionary entry corresponding to the computation number of the outermost function - i.e., the address of the instruction computing this function and the labels associated with the path. The actual error is caused by either the wrong function applied to a set of arguments or the function applied to the wrong set of arguments. For example, consider an error in *LESS(A,B). The error could be that we desire *GREAT(A,B) or possibly *LESS(A,C). The proof system indicates that an error has occurred when attempting to match the computation *LESS(A,B). In addition, it also returns the address of the instruction corresponding to the *LESS function which is denoted as the location of error as well as the path along which the error was detected. Thus

when debugging the program we must ascertain whether the error was in the function or in the arguments.

Error correction is a difficult area. Currently, we only have a limited set of heuristics to guide us. Nevertheless, it does seem to be a powerful one. As mentioned earlier, whenever an error occurs in a function, we must determine if the error is caused by the wrong function being applied to a set of arguments (e.g., error (13) in section 4) or the correct function being applied to the wrong set of arguments (e.g., error (4) in section 4). Our approach is first to attempt to correct the function. Next, an attempt is made to correct the arguments (e.g., errors (5) and (7) in section 4). When correcting arguments, we know the accumulators which must contain the arguments and thus we can work backwards to determine where and when the wrong values were computed and loaded into the accumulators (e.g., error (7) in section 4). Often the debugging process is aided by the presence of instructions that manipulate data that will no longer be referenced in the program (e.g., error (12) in section 4). Such instructions often serve as candidates for removal and replacement by the correct instruction. Errors also occur frequently in the sense of a condition - i.e., the wrong sense is being tested. This is especially common with arithmetic relations such as less than and greater than (e.g., errors (6),(8), and (9) in section 4). Such occurrences are signaled by the presence of errors in both subtrees of a condition in close proximity (in terms of the logical flow of the program) to the instruction at which the condition is tested. This can be corrected in the following manner. Reverse the sense of the test. If all of the errors disappear, then the diagnosis is clearly correct. If some of the errors disappear, then the diagnosis is quite likely to be valid. The previous is especially true if at least one error in each subtree disappears after making the change. Note that changing the sense of the test may lead to new errors. However, as long as some of the current errors disappear, the correction is likely to be valid.

4. EXAMPLE

In this section we examine the error detection capabilities of the system reported in [Samet75] as well as the potential for automatic error correction. We use a rather complex function known as HIER1 which is fairly typical of the type of functions found in artificial intelligence programs. The algorithm originated in the FOL [Weyhrauch74] system where it is used extensively. We will not dwell to any length on the actual effect of the function except for the following brief summary.

Application of the function results in the conversion of a list representing an expression with prefix and infix operators to a tree-like representation. The primary driving force in the determination of the operands corresponding to each of the operators is a set of binding powers (operator precedence). The second argument to the function denotes the binding power of the operator corresponding to the expression in question.

Fig. 5 contains an encoding of HIER1 in MLISP. Note the use of square brackets. This is an MLISP construct which is very useful in visualizing the structure of a list. Each index indicates a number, say num, which is interpreted as being equivalent to num-1 CDR operations followed by a CAR operation. The brackets can be likened to a function whose arguments indicate a sequence of CDR and CAR operations applied from left to right. For example L[2,1] is equivalent to (CAADDR L) - i.e., CAR(CAR(CDR(CDR(L)))). Angle brackets are used to indicate a list consisting of the elements separated by commas within the angled brackets. For example, <A,B,C> is equivalent to LIST(A,B,C). We also use the single quote symbol instead of the word QUOTE.

```
EXPR HIER1(L,RBP);
IF NULL(L[1]) & NULL(CDDR(L)) THEN L
ELSE IF NULL(CDDR(L)) THEN HIER1(<CDR(L[1]),CONS(L[1,1],L[2])>,RBP)
ELSE IF NULL(L[1]) THEN
    IF RBP GEQ BP1(L[3,1],'LEFT&) THEN L
    ELSE HIER1(CONS(NIL,
                    CONS(CONS(L[3,1],
                              CONS(L[2],
                                   (SETQ(L,
                                         HIER1(CONS(L[3,2],
                                                    CONS(L[3,3],CDDR(L))),
                                               BP1(L[3,1],'RIGHT&))))[2])),
                         CDDR(L))),
               RBP)
ELSE IF BP1(L[1,1],'PRIGHT&) GEQ BP1(L[3,1],'LEFT&) THEN
    HIER1(CONS(CDR(L[1]),CONS(CONS(L[1,1],L[2]),CDDR(L))),RBP)
ELSE HIER1(CONS(L[1],
                CONS((SETQ(L,
                           HIER1(CONS(NIL,CDR(L)),
                                 BP1(L[1,1],'PRIGHT&))))[2],
                     CDDR(L))),
           RBP);

EXPR BP1(X,Y);
GET(X,Y);
```

Fig. 5 - MLISP encoding of HIER1

Fig. 6 denotes the LAP encoding of HIER1 that is generated by the LISP 1.6 compiler. The meaning of the instructions should be clear from the adjoining comments. In addition, an encoding is given in fig. 7, obtained by a hand optimization process, containing a number of errors. These errors occurred during the optimization process and were not intentional. The remainder of the discussion focusses on these errors and demonstrates the error detection capability of the

system. We show how the errors were detected and how the available information can be used to correct them. All corrections are made relative to the encoding in fig. 7 and thus all instruction locations refer to fig. 7. During this process, we successively make the corrections deemed necessary by the error detection mechanism until a correct program results. Unfortunately, the errors preclude fig. 7 from containing a completely commented encoding. However, the meaning of the uncommented instructions will become clear as the corrections are being discussed. Note that the scenario presented, sans the automatic error correction, is essentially a transcript of a user session with our system. The only difference is that we have omitted the numeric representation from our discussion.

```
      (PUSH 12 1)              save L on the stack
      (PUSH 12 2)              save RBP on the stack
      (HLRZ@ 1 1)              load acc. 1 with L[1]
      (JUMPN 1 TAG2)           jump to TAG2 if L[1] is not NIL
      (HRRZ@ 1 -1 12)          load acc. 1 with CDR(L)
      (HRRZ@ 1 1)              load acc. 1 with CDDR(L)
      (JUMPN 1 TAG2)           jump to TAG2 if CDDR(L) is not NIL
      (MOVE 1 -1 12)           load acc. 1 with L
      (JRST 0 TAG1)            jump to TAG1
TAG2  (HRRZ@ 1 -1 12)          load acc. 1 with CDR(L)
      (HRRZ@ 1 1)              load acc. 1 with CDDR(L)
      (JUMPN 1 TAG4)           jump to TAG4 if CDDR(L) is not NIL
      (HRRZ@ 2 -1 12)          load acc. 2 with CDR(L)
      (HLRZ@ 2 2)              load acc. 2 with L[2]
      (HLRZ@ 1 -1 12)          load acc. 1 with L[1]
      (HLRZ@ 1 1)              load acc. 1 with L[1,1]
      (CALL 2 (E CONS))        compute CONS(L[1,1],L[2])
      (CALL 1 (E NCONS))       compute <CONS(L[1,1],L[2])>
      (HLRZ@ 2 -1 12)          load acc. 2 with L[1]
      (HRRZ@ 2 2)              load acc. 2 with CDR(L)
      (CALL 2 (E XCONS))       compute <CDR(L[1]),CONS(L[1,1],L[2])>
      (MOVE 2 0 12)            load acc. 2 with RBP
      (CALL 2 (E HIER1))       compute HIER1(<CDR(L[1]),CONS(L[1,1],L[2])>,RBP)
      (JRST 0 TAG1)            jump to TAG1
TAG4  (HLRZ@ 1 -1 12)          load acc.1 with L[1]
      (JUMPN 1 TAG5)           jump to TAG5 if L[1] is not NIL
      (MOVEI 2 (QUOTE LEFT&))  load acc. 2 with 'LEFT&
      (HRRZ@ 1 -1 12)          load acc. 1 with CDR(L)
      (CALL 1 (E CAADR))       891~[=+ c[3,1]
      (CALL 2 (E BP1))         compute BP1(L[3,1],'LEFT&)
      (MOVE 2 0 12)            load acc. 2 with RBP
      (CALL 2 (E *GREAT))      compute BP1(L[3,1],'LEFT&)>RBP
      (JUMPN 1 TAG7)           jump to TAG7 if BP1(L[3,1],'LEFT&)>RBP
      (MOVE 1 -1 12)           load acc. 1 with L
      (JRST 0 TAG6)            jump to TAG6
TAG7  (HRRZ@ 2 -1 12)          load acc. 2 with CDR(L)
      (HRRZ@ 2 2)              load acc. 2 with CDDR(L)
      (HRRZ@ 2 2)              load acc. 2 with CDDDR(L)
      (HRRZ@ 1 -1 12)          load acc. 1 with CDR(L)
      (CALL 1 (E CDADR))       compute CDR(L[3])
      (CALL 1 (E CADR))        compute L[3,3]
      (CALL 2 (E CONS))        compute CONS(L[3,3],CDDDR(L))
      (HRRZ@ 2 -1 12)          load acc. 2 with CDR(L)
      (HRRZ@ 2 2)              load acc. 2 with CDDR(L)
      (HLRZ@ 2 2)              load acc. 2 with L[3]
      (HRRZ@ 2 2)              load acc. 2 with CDR(L[3])
      (HLRZ@ 2 2)              load acc. 2 with L[3,2]
      (CALL 2 (E XCONS))       compute CONS(L[3,2],CONS(L[3,3],CDDDR(L)))
      (PUSH 12 1)              save CONS(L[3,2],CONS(L[3,3],CDDDR(L)))
                               on the stack
      (HRRZ@ 1 -2 12)          load acc. 1 with CDR(L)
      (HLRZ@ 1 1)              load acc. 1 with L[2]
      (PUSH 12 1)              save L[2] on the stack
      (HRRZ@ 1 -3 12)          load acc. 1 with CDR(L)
      (CALL 1 (E CAADR))       compute L[3,1]
      (MOVEI 2 (QUOTE RIGHT&)) load acc. 2 with 'RIGHT&
```

```
        (PUSH 12 1)             save L[3,1] on the stack
        (HRRZ@ 1 -4 12)         load acc. 1 with CDR(L)
        (CALL 1 (E CAADR))      compute L[3,1]
        (CALL 2 (E BP1))        compute BP1(L[3,1],'RIGHT&)
        (MOVE 2 1)              load acc. 2 with BP1(L[3,1],'RIGHT&)
        (EXCH 1 -2 12)          exchange acc. 1 with
                                CONS(L[3,2],CONS(L[3,3],CDDDR(L)))
        (CALL 2 (E HIER1))      compute
                                HIER1(CONS(L[3,2],CONS(L[3,3],CDDDR(L))),
                                      BP1(L[3,1],'RIGHT&))
        (HRRZ@ 2 1)             load acc. 2 with
                                CDR(HIER1(CONS(L[3,2],CONS(L[3,3],CDDDR(L))),
                                          BP1(L[3,1],'RIGHT&)))
        (HLRZ@ 2 2)             load acc. 2 with
                                HIER1(CONS(L[3,2],CONS(L[3,3],CDDDR(L))),
                                      BP1(L[3,1],'RIGHT&))[2]
        (EXCH 1 -1 12)          exchange acc. 1 with L[2]
        (CALL 2 (E CONS))       compute
                                CONS(L[2],
                                     HIER1(CONS(L[3,2],CONS(L[3,3],CDDDR(L))),
                                           BP1(L[3,1],'RIGHT&))[2])
        (POP 12 2)              load acc. 2 with L[3,1] from the stack
        (CALL 2 (E XCONS))      compute
                                CONS(L[3,1],
                                     CONS(L[2],
                                          HIER1(CONS(L[3,2],CONS(L[3,3],CDDDR(L))),
                                                BP1(L[3,1],'RIGHT&))[2]))
        (HRRZ@ 2 0 12)          load acc. 2 with                          TAG12
                                CDR(HIER1(CONS(L[3,2],CONS(L[3,3],CDDDR(L))),
                                          BP1(L[3,1],'RIGHT&)))
        (HRRZ@ 2 2)             load acc. 2 with
                                CDDR(HIER1(CONS(L[3,2],CONS(L[3,3],CDDDR(L))),
                                           BP1(L[3,1],'RIGHT&)))
        (CALL 2 (E CONS))       compute
                                CONS(CONS(L[3,1],
                                          CONS(L[2],
                                               HIER1(CONS(L[3,2],
                                                          CONS(L[3,3],CDDDR(L))),
                                                     BP1(L[3,1],'RIGHT&))[2])),
                                     CDDR(HIER1(CONS(L[3,2],CONS(L[3,3],CDDDR(L))),
                                                BP1(L[3,1],'RIGHT&))))
        (MOVEI 2 (QUOTE NIL))   load acc. 2 with NIL
        (CALL 2 (E XCONS))      compute
                                CONS(NIL,
                                     CONS(CONS(L[3,1],
                                               CONS(L[2],
                                                    HIER1(CONS(L[3,2],
                                                               CONS(L[3,3],
                                                                    CDDDR(L))),
                                                          BP1(L[3,1],'RIGHT&))[2])),
                                          CDDR(HIER1(CONS(L[3,2],
                                                          CONS(L[3,3],CDDDR(L))),
                                                     BP1(L[3,1],'RIGHT&)))))
        (MOVE 2 -2 12)          load acc. 2 with RBP from the stack
        (CALL 2 (E HIER1))      compute
                                HIER1(CONS(NIL,
                                           CONS(CONS(L[3,1],
                                                     CONS(L[2],
                                                          HIER1(CONS(L[3,2],
                                                                     CONS(L[3,3],
                                                                          CDDDR(L))),
                                                                BP1(L[3,1],
                                                                    'RIGHT&))[2])),
                                                CDDR(HIER1(CONS(L[3,2],
                                                                CONS(L[3,3],
                                                                     CDDDR(L))),
                                                           BP1(L[3,1],'RIGHT&))))),
                                      RBP)
        (POP 12 -3 12)          remove a stack entry
        (SUB 12 (C 0 0 1 1))    remove a stack entry
TAG6    (JRST 0 TAG1)           jump to TAG1                              TAG1
TAG5    (MOVEI 2 (QUOTE PRIGHT&)) load acc. 2 with 'PRIGHT&
        (HLRZ@ 1 -1 12)         load acc. 1 with L[1]
        (HLRZ@ 1 1)             load acc. 1 with L[1,1]
        (CALL 2 (E BP1))        compute BP1(L[1,1],'PRIGHT)
        (MOVEI 2 (QUOTE LEFT&)) load acc. 2 with 'LEFT&
```

```
        (PUSH 12 1)             save BP1(L[1,1],'PRIGHT) on the stack
        (HRRZ@ 1 -2 12)         load acc. 1 with CDR(L)
        (CALL 1 (E CAADR))      compute L[3,1]
        (CALL 2 (E BP1))        compute BP1(L[3,1],'LEFT&)
        (POP 12 2)              load acc. 2 with BP1(L[1,1],'PRIGHT)
        (CALL 2 (E *GREAT))     compute BP1(L[3,1],'LEFT)>BP1(L[1,1],'PRIGHT)
        (JUMPN 1 TAG12)         jump to TAG12 if
                                BP1(L[3,1],'LEFT)>BP1(L[1,1],'PRIGHT)
        (HRRZ@ 2 -1 12)         load acc. 2 with CDR(L)
        (HLRZ@ 2 2)             load acc. 2 with L[2]
        (HLRZ@ 1 -1 12)         load acc. 1 with L[1]
        (HLRZ@ 1 1)             load acc. 1 with L[1,1]
        (CALL 2 (E CONS))       compute CONS(L[1,1],L[2])
        (HRRZ@ 2 -1 12)         load acc. 2 with CDR(L)
        (HRRZ@ 2 2)             load acc. 2 with CDDR(L)
        (CALL 2 (E CONS))       compute CONS(CONS(L[1,1],L[2]),CDDR(L))
        (HLRZ@ 2 -1 12)         load acc. 2 with L[1]
        (HRRZ@ 2 2)             load acc. 2 with CDR(L[1])
        (CALL 2 (E XCONS))      compute
                                CONS(CDR(L[1]),
                                     CONS(CONS(L[1,1],L[2]),CDDR(L)))
        (MOVE 2 0 12)           load acc. 2 with RBP
        (CALL 2 (E HIER1))      compute
                                HIER1(CONS(CDR(L[1]),
                                           CONS(CONS(L[1,1],L[2]),CDDR(L))),
                                      RBP)
        (JRST 0 TAG1)           jump to TAG1
TAG12   (HRRZ@ 2 -1 12)         load acc. 2 with CDR(L)
        (MOVEI 1 (QUOTE NIL))   load acc. 1 with NIL
        (CALL 2 (E CONS))       compute CONS(NIL,CDR(L))
        (PUSH 12 1)             save CONS(NIL,CDR(L)) on the stack
        (HLRZ@ 1 -2 12)         load acc. 1 with L[1]
        (MOVEI 2 (QUOTE PRIGHT&)) load acc. 2 with 'PRIGHT&
        (PUSH 12 1)             save L[1] on the stack
        (HLRZ@ 1 -3 12)         load acc. 1 with L[1]
        (HLRZ@ 1 1)             load acc. 1 with L[1,1]
        (CALL 2 (E BP1))        compute BP1(L[1,1],'PRIGHT&)
        (MOVE 2 1)              load acc. 2 with BP1(L[1,1],'PRIGHT&)
        (EXCH 1 -1 12)          exchange acc. 1 with CONS(NIL,CDR(L))
        (CALL 2 (E HIER1))      compute
                                HIER1(CONS(NIL,CDR(L)),BP1(L[1,1],'PRIGHT&))
        (HRRZ@ 2 1)             compute
                                CDR(HIER1(CONS(NIL,CDR(L)),BP1(L[1,1],'PRIGHT&)))
        (HRRZ@ 2 2)             compute
                                CDDR(HIER1(CONS(NIL,CDR(L)),BP1(L[1,1],'PRIGHT&)))
        (MOVEM 1 -3 12)         replace the old value of L on the stack with
                                HIER1(CONS(NIL,CDR(L)),BP1(L[1,1],'PRIGHT&))
        (CALL 1 (E CADR))       compute
                                HIER1(CONS(NIL,CDR(L)),BP1(L[1,1],'PRIGHT&))[2]
        (CALL 2 (E CONS))       compute
                                CONS(HIER1(CONS(NIL,CDR(L)),
                                           BP1(L[1,1],'PRIGHT&))[2],
                                     CDDR(HIER1(CONS(NIL,CDR(L)),
                                                BP1(L[1,1],'PRIGHT&))))
        (POP 12 2)              load acc. 2 with L[1]
        (CALL 2 (E XCONS))      compute
                                CONS(L[1],
                                     CONS(HIER1(CONS(NIL,CDR(L)),
                                                BP1(L[1,1],'PRIGHT&))[2],
                                          CDDR(HIER1(CONS(NIL,CDR(L)),
                                                     BP1(L[1,1],'PRIGHT&)))))
        (MOVE 2 -1 12)          load acc. 2 with RBP
        (CALL 2 (E HIER1))      compute
                                HIER1(CONS(L[1],
                                           CONS(HIER1(CONS(NIL,CDR(L)),
                                                      BP1(L[1,1],'PRIGHT&))[2],
                                                CDDR(HIER1(CONS(NIL,CDR(L)),
                                                           BP1(L[1,1],'PRIGHT&))))),
                                      RBP)
        (SUB 12 (C 0 0 1 1))    remove one entry from the stack
TAG1    (SUB 12 (C 0 0 2 2))    undo the first two stack operations
        (POPJ 12)               return
```

Fig. 6 - LISP 1.6 compiler generated encoding for HIER1

The optimized encoding makes use of several optimizations which are briefly described. In some instances recursion is achieved by bypassing the start of the program via use of the label HIERA. This is motivated by the following observations. First, for the recursive calls the second argument need never be present in accumulator 2 because accumulator 2 is never being referenced prior to being overwritten. Second, observe that whenever recursion occurs, the second argument is already on the stack and thus there is no need to place it on the stack again. Hence, the first instruction may be bypassed and therefore for internal recursive calls there is no need to follow a calling sequence which makes use of accumulators. Instead, a calling sequence is used where one parameter is in accumulator 1 while the other parameter is on the stack. Other optimizations include common subexpression elimination and a wide use of accumulators to store temporary values across functions whose invocation does not result in the destruction of the contents of all of the accumulators (e.g., CONS, XCONS, and NCONS). Finally, conditions are compiled more efficiently so that redundant tests are avoided. This was a problem in the LISP 1.6 compiler generated LAP program due to the use of the AND operation in some of the conditions present in the original LISP function definition. The result of these optimizations, when chains of CAR-CDR operations are expanded in line, is an encoding containing 105 instructions instead of 145 instructions. Timing measurements indicated that the new encoding was about 40% faster and required 50% less stack space.

```
HIER1    1 (PUSH 12 2)                save RBP on the stack
HIERA    2 (HLRZ 5 0 1)               load acc. 5 with L[1]
         3 (JUMPN 5 TAG2)             jump to TAG2 if L[1] is not NIL
         4 (HRRZ 4 0 1)               load acc. 4 with CDR(L)
         5 (HRRZ 3 0 4)               load acc. 3 with CDDR(L)
         6 (JUMPE 3 TAGA)             jump to TAGA if CDDR(L) is NIL
         7 (JRST 0 TAGB)              jump to TAGB
TAG2     8 (HRRZ 4 0 1)               load acc. 2 with CDR(L)
         9 (HRRZ 3 0 4)               load acc. 3 with CDDR(L)
        10 (JUMPN 3 TAGC)             jump to TAGC if CDDR(L) is not NIL
        11 (HLRZ 1 0 5)               load acc. 1 with L[1,1]
        12 (HLRZ 2 0 4)               load acc. 2 with L[2]
        13 (CALL 2 (E CONS))          compute CONS(L[1,1],L[2])
        14 (CALL 1 (E NCONS))         compute <CONS(L[1,1],L[2]>
        15 (HRRZ 2 0 4)               load acc. 2 with CDDR(L)
        16 (CALL 2 (E XCONS))         compute <CDDR(L),CONS(L[1,1],L[2])>
        17 (JRST 0 HIERA)             compute HIER1(<CDDR(L),CONS(L[1,1],L[2])>,RBP)
TAGB    18 (PUSH 12 1)                save L on the stack
        19 (HLRZ 1 0 3)               load acc. 1 with L[3]
        20 (HLRZ 1 0 1)               load acc. 1 with L[3,1]
        21 (MOVEI 2 (QUOTE LEFT&))    load acc. 2 with 'LEFT&
        22 (CALL 2 (E BP1))           compute BP1(L[3,1],'LEFT&)
        23 (MOVE 2 -1 12)             load acc. 2 with RBP
        24 (CALL 2 (E *GREAT))        compute BP1(L[3,1],'LEFT&)>RBP
        25 (JUMPN 1 TAG7)             jump to TAG7 if BP1(L[3,1],'LEFT&)>RBP
        26 (POP 12 1)                 load acc. 1 with L
        27 (JRST 0 TAGA)              jump to TAGA
TAG7    28 (HRRZ@ 1 0 12)             load acc. 1 with CDR(L)
        29 (HRRZ 1 0 1)               load acc. 1 with CDDR(L)
        30 (HLRZ 1 0 1)               load acc. 1 with L[3]
        31 (HLRZ 1 0 1)               load acc. 1 with L[3,1]
        32 (MOVEI 2 (QUOTE RIGHT&))   load acc. 2 with 'RIGHT&
        33 (CALL 2 (E BP1))           compute BP1(L[3,1],'RIGHT&)
        34 (PUSH 12 1)                save BP1(L[3,1],'RIGHT&) on the stack
```

```
        35 (HRRZ@ 5 -1 12)          load acc. 5 with CDR(L)
        36 (HRRZ 4 0 5)             load acc. 4 with CDDR(L)
        37 (HLRZ 3 0 4)             load acc. 3 with L[3]
        38 (HLRZ 5 0 3)             load acc. 5 with L[3,1]
        39 (HRRZ 1 0 4)             load acc. 1 with CDDDR(L)
        40 (HLRZ 4 0 5)             load acc. 4 with L[3,1,1]
        41 (HLRZ 2 0 4)             load acc. 2 with L[3,1,1,1]
        42 (CALL 2 (E CONS))        compute CONS(CDDDR(L),L[3,1,1,1])
        43 (MOVE 2 4)               load acc. 2 with L[3,1,1]
        44 (CALL 2 (E XCONS))       compute CONS(L[3,1,1],CONS(CDDDR(L),L[3,1,1,1]))
        45 (PUSHJ 12 HIERA)
        46 (HRRZ 5 0 1)
TAGX    47 (HLRZ 2 0 1)
        48 (HRRZ@ 4 0 12)
        49 (HLRZ 1 0 4)
        50 (CALL 2 (E CONS))
        51 (HRRZ 3 0 4)
        52 (HLRZ 2 0 3)
        53 (HLRZ 2 0 2)
        54 (CALL 2 (E XCONS))
        55 (HRRZ 2 0 5)
        56 (CALL 2 (E CONS))
        57 (MOVEI 2 (QUOTE NIL))
        58 (CALL 2 (E XCONS))
        59 (SUB 12 (C 0 0 1 1))
        60 (JRST 0 HIERA)
TAGC    61 (PUSH 12 1)              save L on the stack
        62 (HLRZ 1 0 3)             load acc. 1 with L[3]
        63 (HLRZ 1 0 1)             load acc. 1 with L[3,1]
        64 (MOVEI 2 (QUOTE LEFT&))  load acc. 2 with 'LEFT&
        65 (CALL 2 (E BP1))         compute BP1(L[3,1],'LEFT&)
        66 (PUSH 12 1)              save BP1(L[3,1],'LEFT&) on the stack
        67 (HLRZ@ 1 -1 12)          load acc. 1 with L[1]
        68 (HLRZ 1 0 1)             load acc. 1 with L[1,1]
        69 (MOVEI 2 (QUOTE PRIGHT)) load acc. 2 with 'PRIGHT
        70 (CALL 2 (E BP1))         compute BP1(L[1,1],'PRIGHT)
        71 (POP 12 2)               load acc. 2 with BP1(L[3,1],'LEFT&)
        72 (CALL 2 (E *GREAT))      compute BP1(L[1,1],'PRIGHT)>BP1(L[3,1],'LEFT&)
        73 (JUMPE 1 TAG12)          jump to TAG12 if
                                    BP1(L[1,1],'PRIGHT) LEQ BP1(L[3,1],'LEFT&)
        74 (HLRZ@ 5 0 12)           load acc. 5 with L[1]
        75 (HLRZ 1 0 5)             load acc. 1 with L[1,1]
        76 (HRRZ@ 4 0 12)           load acc. 4 with CDR(L)
        77 (HLRZ 2 0 4)             load acc. 2 with L[2]
        78 (CALL 2 (E CONS))        compute CONS(L[1,1],L[2])
        79 (HRRZ 2 0 4)             load acc. 2 with CDDR(L)
        80 (CALL 2 (E CONS))        compute CONS(CONS(L[1,1],L[2]),CDDR(L))
        81 (HRRZ 2 0 5)             load acc. 2 with CDR(L[1])
        82 (CALL 2 (E XCONS))       compute
                                    CONS(CDR(L[1]),
                                         CONS(CONS(L[1,1],L[2]),CDDR(L)))
        83 (SUB 12 (C 0 0 1 1))     remove an entry from the stack
        84 (JRST 0 HIERA)           compute
                                    HIER1(CONS(CDR(L[1]),
                                               CONS(CONS(L[1,1],L[2]),CDDR(L))),
                                          RBP)
TAG12   85 (HLRZ@ 1 0 12)           load acc. 1 with L[1]
        86 (HLRZ 1 0 1)             load acc. 1 with L[1,1]
        87 (MOVEI 2 (QUOTE PRIGHT&)) load acc. 2 with 'PRIGHT&
        88 (CALL 2 (E BP1))         compute BP1(L[1,1],'PRIGHT&)
        89 (PUSH 12 1)              save BP1(L[1,1],'PRIGHT&) on the stack
        90 (HRRZ@ 2 -1 12)          load acc. 2 with CDR(L)
        91 (MOVEI 2 (QUOTE NIL))    load acc. 2 with NIL
        92 (CALL 2 (E CONS))        compute CONS(BP1(L[1,1],'PRIGHT&),NIL)
TAGY    93 (PUSHJ 12 HIERA)
        94 (HRRZ 5 0 1)
        95 (HLRZ 1 0 5)
        96 (HRRZ 2 0 5)
        97 (CALL 2 (E CONS))
        98 (HLRZ@ 2 0 12)
        99 (CALL 2 (E XCONS))
       100 (SUB 12 (C 0 0 1 1))
TAGA   101 (JRST 0 HIERA)
       102 (SUB 12 (C 0 0 1 1))
       103 (POPJ 12)
```

Fig. 7 - Erroneous hand optimized encoding of HIER1

When attempting to prove the equivalence of the encoding in fig. 7 and the original LISP function definition of HIER1, the following errors were detected. For type (1) errors, the error message indicates the location at which the error was detected. For type (4) errors, the error message indicates the location at which the function that could not be found to occur in the original program was computed. In both cases a set of instruction locations corresponding to the branches that were pursued is given. Note that in the interest of clarity we do not use brackets to express chains of CARs and CDRs in errors - i.e., we use CAR(CAR(L)) instead of CAAR(L) or L[1,1]. This is done in order to aid the reader in understanding when the various errors were detected.

(1) Return address on the stack must be a label.
Detected at instruction 45 along path 1,3,4,6,7,18,25,28.

(2) Return address on the stack must be a label.
Detected at instruction 93 along path 1,3,8,10,61,73,85.

(3) The following computation does not occur in the original LISP program:

```
CONS(NIL,
     CONS(CONS(CAR(CAR(L)),
               CAR(CDR(L))),
          NIL))
```

Computed at instruction 16 along path 1,3,8,10,11.

(4) The following computation does not occur in the original LISP program:

```
'PRIGHT
```

Computed at instruction 69 along path 1,3,8,10,61.

Errors (1) and (2) were detected by the symbolic interpretation procedure. They resulted from invalid return addresses on the stack at instructions 45 and 93 when recursion was implemented by bypassing the start of the program. In this case the stack is being used instead of accumulator 2 to contain the second argument. However, the contents of the stack are wrong. In particular, the return addresses (i.e., locations 46 and 94) appear in the stack at a position where the binding of the second argument is expected (i.e., the top of the stack). Thus when a return will be made from the recursive call, execution will not resume at locations 46 or 94. Also, all references to the top of the stack will fetch the return address rather than the binding of RBP. The solution is to place the return address on the stack before the binding of RBP. In the case of error (1), the binding of RBP is BP1(CAR(CAR(CDR(CDR(L)))),'RIGHT&) which is computed starting at location 28 and pushed on the stack at location 34. Thus the return address may be placed on the stack anywhere after location 27 and before location 34. We choose to do this between locations 27 and 28 (i.e., location 27A). In the case of error (2), the binding of RBP is BP1(CAR(CAR(L)),'PRIGHT&) which is computed starting at location

85 and pushed on the stack at location 89. Thus the return address may be placed on the stack anywhere after location 84 and before location 89. We choose to do this between locations 84 and 85 (i.e., location 84A). However, we are not yet through. First, we must insure that all references to labels TAG7 and TAG12 refer to locations 27A and 84A instead of locations 28 and 85 respectively. Second, since the return addresses are no longer placed on the stack at locations 45 and 93, we must only do a jump (i.e., JRST) rather than a push of a return address and a jump (i.e., PUSHJ) at locations 45 and 93. Third, placing the return address on the stack at locations 27A and 84A has caused the stack to contain an extra entry between locations 27 and 45 and 84 and 93. Thus all references to stack entries below the position holding the new return address must be incremented by one. Therefore, we make the following changes and additions:

```
location 27A:                    becomes  (PUSH 12 (C 0 0 TAGX 0))
location 28:  (HRRZ@ 1 0 12)    becomes  (HRRZ@ 1 -1 12)
location 35:  (HRRZ@ 5 -1 12)   becomes  (HRRZ@ 5 -2 12)
location 45:  (PUSHJ 12 HIERA)  becomes  (JRST 0 HIERA)
location 84A:                    becomes  (PUSH 12 (C 0 0 TAGY 0))
location 85:  (HLRZ@ 1 0 12)     becomes  (HLRZ@ 1 -1 12)
location 90:  (HRRZ@ 2 -1 12)    becomes  (HRRZ@ 2 -2 12)
location 93:  (PUSHJ 12 HIERA)   becomes  (JRST 0 HIERA)
```

Errors (3) and (4) were detected during the proof procedure. Error (3) was detected when CAR(L) was not NIL and CDDR(L) was NIL. Referring to the original function definition we see that at this point we want the following:

```
CONS(CDR(CAR(L))
     CONS(CONS(CAR(CAR(L)),
               CAR(CDR(L))),
          NIL))
```

Therefore, the error is in the arguments to the function being computed at location 16 (i.e., CONS). The first argument to this CONS operation is NIL which is identical to CDDR(L). Looking at the code we find that at location 15 we perform (HRRZ 2 0 4) which has the effect of loading accumulator 2 with CDR(CDR(L)) rather than the desired CDR(CAR(L)). However, CDR(CAR(L)) can be achieved by changing the instruction to refer to accumulator 5 instead of accumulator 4. Thus we see that there are several possible causes for the error. Among them are a confusion about the contents of certain accumulators, and mistyping of a 4 for a 5. We make the following modification:

```
location 15:  (HRRZ 2 0 4)     becomes  (HRRZ 2 0 5)
```

Error (4) was detected when both (CAR L) and CDDR(L) were not NIL. Referring to the original function definition we see that at this point we want the following:

```
'PRIGHT&
```

Therefore, the error is in the argument to the function being computed at location 69. This time there is no doubt that the cause of the error was misspelling of the atom PRIGHT&. We make the following modification:

location 69: (MOVEI 2 (QUOTE PRIGHT)) becomes (MOVEI 2 (QUOTE PRIGHT&))

Once the previous errors have been corrected in the LAP program, we input the resulting program to the proof system and obtain the following errors.

(5) The following computation does not occur in the original LISP program:
CAR(CAR(CAR(CDR(CDR(L)))))
Computed at instruction 40 along path 1,3,4,6,7,18,25,27A.

(6) The following computation does not occur in the original LISP program:
*LESS(BP1(CAR(CAR(CDR(CDR(L)))),'LEFT&),
BP1(CAR(CAR(L)),'PRIGHT&))
Computed at instruction 72 along path 1,3,8,10,61.

Error (5) was detected when CAR(L) was NIL, CDDR(L) was not NIL, and RBP < BP1(L[3,1],'LEFT&). Referring to the original function definition, the argument to the CAR operation at location 40 has already been found to occur in the intermediate representation of the original program. Moreover, at this point a CDR operation is required as shown below:

CDR(CDR(CAR(CDR(CDR(L)))))

We temporarily disregard the fact that the argument to the outermost CDR operation is wrong - i.e., it has not yet been found to occur in the original program. The next sequence of debugging will find this error. Recall from section 3 that we first attempt to correct the function, and only once this is done do we attempt to correct the arguments. Inspection of the code reveals that at instruction 40 we perform (HLRZ 4 0 5) which has the wrong effect. Moreover, the result of this operation, i.e., CAR(CAR(CAR(CDR(CDR(L))))), was not matched and thus it can be changed to a (HRRZ 4 0 5) instruction. The cause for this error can be confusion as to the contents of a location or again misspelling. However, we lean towards the former since the error is of a compound nature as will be seen at the next stage of debugging. We make the following modification:

location 40: (HLRZ 4 0 5) becomes (HRRZ 4 0 5)

Error (6) was detected when both CAR(L) and CDDR(L) were not NIL. One of the characteristics of the intermediate representation is that operations known to be antisymmetric are always represented by only one of the two possible choices. Thus CONS and XCONS are represented by CONS and similarly, *LESS and *GREAT are represented by *LESS. Therefore, according to the original function definition, we want the computation:

*LESS(BP1(CAR(CAR(L)),'PRIGHT&),
BP1(CAR(CAR(CDR(CDR(L)))),'LEFT&))

In other words, the error is in the order of the arguments to the *LESS function. Looking at the LAP program we find that at location 72 (CALL 2 (E *GREAT))) is performed rather than the necessary (CALL 2 (E *LESS)). An equivalent interpretation of the error is that the contents of accumulators 1 and 2 (which must contain the arguments to the function) have been permuted. Nevertheless, we opt for the first interpretation since less code need be changed. Clearly, the source of this error is a misunderstanding by the programmer of the antisymmetric properties of the arithmetic relations less than, greater than, less than or equal, and greater than or equal. We make the following modification.

location 72: (CALL 2 (E *GREAT)) becomes (CALL 2 (E *LESS))

Once the previous errors have been corrected in the LAP program, we input the resulting program to the proof system and obtain the following errors.

(7) The following computation does not occur in the original LISP program:
CDR(CAR(CAR(CDR(CDR(L)))))
Computed at instruction 40 along path 1,3,4,6,7,18,25,27A.

(8) The following computation does not occur in the original LISP program:
CONS(BP1(CAR(CAR(L)),'PRIGHT&),NIL)
Computed at instruction 92 along path 1,3,8,10,61,73,84A.

(9) The following computation does not occur in the original LISP program:
CAR(CDR(L))
Computed at instruction 77 along path 1,3,8,10,61,73,74.

Error (7) was detected when CAR(L) was NIL, CDDR(L) was not NIL, and RBP < BP1(L[3,1],'LEFT&). Referring to the original function definition, we see that the function computed at location 40 is being applied to the wrong argument. Recall from the last debugging session that the desired computation was:

CDR(CDR(CAR(CDR(CDR(L)))))

Therefore, the error is in the argument to the function being computed at location 40. The instruction at location 40 is (HRRZ 4 0 5). Therefore its argument is in accumulator 5 which is set at location 38 by a (HLRZ 5 0 3) instruction. We note that accumulator 5 is not referenced with this value except at location 40, and thus it is quite reasonable to believe that an error occurred at location 38. The instruction at location 38 has the effect of loading accumulator 5 with CAR(CAR(CDR(CDR(L)))) rather than the desired CDR(CAR(CDR(CDR(L)))). However, this is a relatively easy modification since we merely need to replace the HLRZ operation at location 38 by a HRRZ operation. The cause of this error is confusion about the contents of accumulators or misspelling. We lean towards the latter in light of the remedy. Recall that this was part of a compound error as discussed in the analysis of the previous set of bugs. We make the following modification:

location 38: (HLRZ 5 0 3) becomes (HRRZ 5 0 3)

Errors (8) and (9) occured when both CAR(L) and CDDR(L) were not NIL. The difference is that error (8) occurs when BP1(L[1,1],'PRIGHT&) is greater than or equal to BP1(L[3,1],'LEFT&) and error (9) occurs when the latter condition is not true. If we were to proceed along lines proposed earlier, we would check if the functions computed at these locations are erroneous or if their arguments are not correct. Using this strategy, we would discover that we do not get a real idea as to the error. The problem is that we have branched on the wrong sense of the condition computed at location 72 and tested at location 73. Such errors are a possibility when there are two errors in the subtrees of the same condition. The error could be detected by the scheme discussed in section 3. In the case of this example, we did indeed test the wrong sense of the condition. We were aware of this fact during the last debugging session; however, we did not discuss it because we feel that the present setting is more enlightening. Nevertheless, the problem should have been fixed at that time since the error did occur in the computation of a function. Such problems in the context of multiple errors are quite difficult and an adequate method to dispose of them is a subject for future research. Therefore, change the sense of the test performed at location 73 by making the following modification:

location 73: (JUMPE 1 TAG12) becomes (JUMPN 1 TAG12)

Once the previous errors have been corrected in the LAP program, we input the resulting program to the proof system and obtain the following errors.

(10) The following computation does not occur in the original LISP program:
```
CONS(CDR(CDR(CDR(L))),
     CAR(CDR(CDR(CAR(CDR(CDR(L)))))))
```
Computed at instruction 42 along path 1,3,4,6,7,18,25,27A.

(11) The following computation does not occur in the original LISP program:
```
CONS(BP1(CAR(CAR(L)),'PRIGHT&),
     NIL)
```
Computed at instruction 92 along path 1,3,8,61,73,84A.

Error (10) was detected when CAR(L) was NIL, CDDR(L) was not NIL, and RBP < BP1(L[3,1],'LEFT&). Referring to the original function definition, we see that at this point we want the following:

```
CONS(CAR(CDR(CDR(CAR(CDR(CDR(L)))))),
     CDR(CDR(CDR(L))))
```

Clearly, the order of the arguments to the CONS operation has been reversed. Inspection of the code reveals that at location 42 we perform (CALL 2 (E CONS)) rather than the necessary (CALL 2 (E XCONS)). This conclusion is made on the basis of CONS being an antisymmetric function. An equivalent interpretation of the error is that the contents of accumulators 1 and 2 (which must contain the arguments to

the function) have been permuted. Nevertheless, we opt for the first interpretation since less code needs to be changed. Clearly, the cause of the error is a confusion as to the contents of accumulators 1 and 2. We make the following modification:

location 42: (CALL 2 (E CONS))becomes (CALL 2 (E XCONS))

Error (11) was detected when both CAR(L) and CDDR(L) were not NIL and BP1(L[1,1],'PRIGHT&) was not greater than or equal to BP1(L[3,1],'LEFT&). Referring to the original function definition we see that at this point we want the following:

CONS(NIL,CDR(L))

Therefore, the error is in the arguments to the function being computed at location 92. The desired arguments, NIL and CDR(L), have already been computed at locations 91 and 92 respectively and found to occur in the intermediate representation of the original program. Thus the correction is to simply make sure that they reside in the proper accumulators for the CONS operation at location 92 to be correct. This means that instead of loading accumulator 2 with NIL at location 91, we load accumulator 1 with this value. Notice that the error that was made was to load accumulator 2 with NIL at location 91 via (MOVEI 2 (QUOTE NIL)) thereby destroying the previous contents which was CDR(L). This error was detected, and quite easily corrected, because we always record all computations that have been performed whether or not they are referenced. This is useful because the proof procedure will make sure that the computation is performed. Thus when errors occur in arguments to functions we can easily make a correction since we know where and when the desired arguments are computed even though they may have been misused. The error in this case can be clearly attributed to an oversight by the programmer in typing a 2 instead of a 1. We make the following modification:

location 92: (MOVEI 2 (QUOTE NIL)) becomes (MOVEI 1 (QUOTE NIL))

Once the previous errors have been corrected in the LAP program, we input the resulting program to the proof system and obtain the following errors.

(12) The following computation does not occur in the original LISP program:

```
CONS(CDR(CDR(CAR(CDR(CDR(L))))),
     CONS(CAR(CDR(CDR(CAR(CDR(CDR(L)))))),
          CDR(CDR(CDR(L)))))
```

Computed at instruction 44 along path 1,3,4,6,7,18,25,27A.

Error (12) was detected when CAR(L) was NIL, CDDR(L) was not NIL, and RBP < BP1(L[3,1],'LEFT&). Referring to the original function definition we see that at this point we want the following:

```
CONS(CAR(CDR(CAR(CDR(CDR(L))))),
     CONS(CAR(CDR(CDR(CAR(CDR(CDR(L)))))),
          CDR(CDR(CDR(L)))))
```

Therefore, the error is in the arguments to the function being computed at location 44. The desired second argument is correct, but the first one is invalid. The instruction performed at location 44 is (CALL 2 (E XCONS)) and thus the argument in accumulator 2 is wrong. The desired contents of accumulator 2 is CAR(CDR(CAR(CDR(CDR(L))))). Inspection of the code reveals that accumulator 2 is last loaded at location 43 by the instruction (MOVE 2 4). However, this value is not necessary in the future and thus the instruction at this location may be removed. The validity of the previous removal is obvious when we recall the that the value in accumulator 2 is not referenced past location 44. An alternative reason is that the XCONS operation is assumed to destroy accumulators 1 and 2. In its place we need to compute CAR(CDR(CAR(CDR(CDR(L))))) since it has not yet been computed. This can be done quite easily since at this point accumulator 5 already contains CDR(CAR(CDR(CDR(L)))) and thus we need only obtain CAR of the contents of register 5. This is quite easily done by inserting (HLRZ 2 0 5) at location 43. The cause of this error is obviously confusion on the part of the programmer as to the contents of accumulator 2. We make the following modification:

location 43: (MOVE 2 4) becomes (HLRZ 2 0 5)

Once the previous errors have been corrected in the LAP program, we input the resulting program to the proof system and obtain the following error.

(13) The following computation does not occur in the original LISP program:

```
CAR(HIER1(CONS(CAR(CDR(CAR(CDR(CDR(L))))),
                    CONS(CAR(CDR(CDR(CAR(CDR(CDR(L)))))),
                         CDR(CDR(CDR(L))))),
              BP1(CAR(CAR(CDR(CDR(L)))),'RIGHT&)))
```

Computed at instruction 47 along path 1,3,4,6,7,18,25,27A.

Error (13) was detected when CAR(L) was NIL, CDDR(L) was not NIL, and RBP < BP1(L[3,1],'LEFT&). Referring to the original function definition we see that this computation is unnecessary. Moreover what is required at this point is the following:

```
CDR(HIER1(CONS(CAR(CDR(CAR(CDR(CDR(L))))),
          CONS(CAR(CDR(CDR(CAR(CDR(CDR(L)))))),
               CDR(CDR(CDR(L))))),
          BP1(CAR(CAR(CDR(CDR(L)))),'RIGHT&)))
```

Clearly, what happened here is that a CAR operation was computed rather than a CDR operation. In terms of machine instructions the previous is translated into the performance of a HLRZ rather than a HRRZ. By now we are rather adept at making such corrections and we simply replace the (HLRZ 2 0 1) instruction at location 47 by (HRRZ 2 0 1). Note that we made use of the fact that results of the previous instruction at location 47 were never referenced in the future. Clearly, the cause of this error is mistyping of HLRZ for HRRZ. We make the following modification:

location 47: (HLRZ 2 0 1) becomes (HRRZ 2 0 1)

At this point the proof system finds the corrected LAP program to be equivalent to the original LISP program. Thus we have seen how the system can aid the user in debugging his program. Our goal is to construct a system, employing similar reasoning as we have performed in this section, to debug and correct erroneous programs. Of course, not all errors could be caught by such a system. However, we feel that quite a reasonable number could be detected and corrected by such an automatic system.

5. CONCLUSION

We have demonstrated the performance of a semi-automatic debugging system. At the present, only the errors are detected and pinpointed automatically. It remains for the programmer to make use of this information to correct the program. In the future, we believe that the correction task can, in a large number of cases, be performed automatically. This is especially true for errors of class (4). Currently, we need to continue to exercise the system with erroneous encodings to determine if any more error-correction heuristics can be discovered. Such heuristics also provide an insight into the programming process. These insights might prove to be useful in future automatic programming systems.

6. REFERENCES

[DEC69] - "PDP-10 System Reference Manual," Digital Equipment Corporation, Maynard, Massachusetts, 1969.

[Green76] _ Green, C.C., "The Design of the PSI Program Synthesis System," Proceedings of the Second International Conference on Software Engineering, San Francisco, California, 1976.

[Lee72] - Lee, J.A.N., Computer Semantics, Van Nostrand Reinhold, New York, 1972, pp. 346-347.

[Low74] - Low, J.R., "Automatic Coding: Choice of Data Structures," Ph.D. Thesis, Stanford Artificial Intelligence Project Memo AIM-242, Computer Science Department, Stanford University, 1974.

[McCarthy60] - McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine," Communications of the ACM, April 1960, pp. 184-195.

[Naur60] - Naur, P., (Ed.), "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM, May 1960, pp. 299-314.

[Newell73] - Newell, A., "Artificial Intelligence and the Concept of Mind," in Computer Models of Thought and Language (Eds. Schank and Colby), W.H. Freeman, San Francisco, 1973, pp. 1-60.

[Quam72] - Quam, L.H., and Diffie, W., "Stanford LISP 1.6 Manual," Stanford Artificial Intelligence Project Operating Note 28.7, Computer Science Department, Stanford University, 1972.

[Samet75] - Samet, H., "Automatically Proving the Correctness of Translations Involving Optimized Code," Ph.D. Thesis, Stanford Artificial Intelligence Project Memo AIM-259, Computer Science Department, Stanford University, 1975.

[Samet76] - Samet, H., "Compiler Testing Via Symbolic Interpretation," Proceedings of the ACM 29th Annual Conference, 1976, pp. 492-497.

[Samet77] - Samet, H., "A Normal Form for Compiler Testing," Proceedings of the SIGART/SIGPLAN Symposium on Artificial Intelligence and Programming Languages, (also in SIGPLAN NOTICES, Vol. 12, No. 8, August 1977 and in SIGART Newsletter, No. 64, August 1977), Rochester, New York, August 1977, pp. 155-162.

[Smith70] - Smith, D.C., "MLISP," Stanford Artificial Intelligence Project Memo AIM-135, Computer Science Department, Stanford University, October 1970.

[Summers75] - Summers, P.D., "Program Construction from Examples," Ph.D. Thesis, Computer Science Department, Yale University, 1975.

[Sussman75] - Sussman, G.J., A Computer Model of Skill Acquisition, American Elsevier, New York, 1975.

[Taylor76] - Taylor, R.H., "A Synthesis of Manipulator Control Programs from Task-Level Specification," Ph.D. Thesis, Stanford Artificial Intelligence Project Memo AIM-282, Computer Science Department, Stanford University, 1976.

[Waldinger69] - Waldinger, R.J., "Constructing Programs Automatically Using Theorem Proving," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1969.

[Weyhrauch74] - Weyhrauch, R.W., and Thomas, A.J., "FOL: a Proof Checker for First-order Logic, Stanford Artificial Intelligence Project Memo AIM-235, Computer Science Department, Stanford University, September 1974.